GRID-BASED REAL-TIME SIMULATION OF INCOMPRESSIBLE FLUIDS IN TWO DIMENSIONS ON THE GPU

Linus Mossberg
linmo400@student.liu.se

Thursday 6th May, 2021

Abstract

This project-report presents an implementation of a grid-based two-dimensional real-time simulator of incompressible fluids on the GPU using OpenGL. The report begins by giving an overview of the *Navier-Stokes equations* and the numerical methods required to solve them, which includes finite difference differential vector operators on a grid, numerical integration methods, the Helmhotz-Hodge decomposition and the iterative Jacobi method used to approximate Poisson's equation on a grid. The report then moves on to present the method used for the implementation. The method is mainly based on the *Stable Fluids* paper by Jos Stam [1], which presents a numerical method of solving the Navier-Stokes equations that is stable for all time steps. The implementation is also extended by adding the *Vorticity Confinement* method by Fedkiw et al. [2] to compensate for numerical diffusion. Several visualization methods that provides insights about the solution are also presented; passive ink fields, line integral convolution images (velocity streamlines), arrow plots and color maps. The report finally presents a few results from the program, which are analyzed to give examples of the fluid dynamics insights that the program can provide.

1 Introduction

A fluid is any substance that flows, which includes plasmas like solar winds, gasses like air, liquids such as water and even semi-solids like asphalt. As such, fluids exists all around us and they influence almost everything we do. The ability to understand and make predictions about fluids is therefore useful in a wide range of applications, such as in the fields of aerospace engineering, weather prediction and the movie and games industries.

This involves the development of mathematical models that describe fluids, numerical methods of simulating these models and graphical methods of visualizing the resulting solution. This report aims to address each of these areas to finally arrive at a program implementation that simulates and visualizes fluids in real-time on the GPU. For simplicity and to ensure that the simulation can run in real-time, the project is limited to incompressible fluids in two dimensions.

2 Theory

This section provides some background theory on the mathematical model of fluid dynamics, as well as some of the mathematical operators and their numerical approximations that are required to simulate the model on a grid. See appendix A for a few notes on the notation used in upcoming sections.

2.1 The Navier-Stokes Equations

Incompressible fluids are modelled with the *Navier-Stokes equations* given in (1),

$$\begin{aligned} \frac{\partial \mathbf{v}}{\partial t} &= -(\mathbf{v} \cdot \nabla)\mathbf{v} - \frac{1}{\rho}\nabla p + \frac{\mu}{\rho}\nabla^2 \mathbf{v} + \mathbf{F} \\ \nabla \cdot \mathbf{v} &= 0 \end{aligned} \tag{1}$$

where the constituent fields, variables and constants are given in Table 1 [1].

Table 1: Fields, variables and constants in the Navier-Stokes equations.

	Name	Type	Unit
$\mathbf{v}(\mathbf{x}, t)$	Velocity	Field	m/s
$p(\mathbf{x}, t)$	Pressure	Field	Pa
$\mathbf{F}(\mathbf{x}, t)$	External force	Field	N
$\mathbf{x}$	Position	Variable	m
t	Time	Variable	s
μ	Dynamic viscosity	Constant	$kg/(m \cdot s)$
ρ	Density	Constant	kg/m^3

The first Navier-Stokes equation is a partial differential equation that describes how the flow evolves over time, while the second equation is a condition that must be satisfied for incompressible fluids. The individual terms are [1]:

$(\mathbf{v} \cdot \nabla)\mathbf{v}$, the **self-advection** term, responsible for moving the flow with itself.

$\frac{1}{\rho}\nabla p$, the **pressure-gradient** term, responsible for moving the flow in the direction with the greatest change in pressure.

$\frac{\mu}{\rho}\nabla^2 \mathbf{v}$, the **viscosity** term, which models the friction between particles in the fluid.

F, the **external force** term, used to apply external forces such as gravity to the flow.

$\nabla \cdot \mathbf{v} = 0$, the **continuity equation**, which states that the flow must be divergence-free, and therefore free of sources and sinks where flow can be created or disappear.

2.2 Simulation Domain

The simulation domain is represented as a discrete $M \times N$ grid of square cells, where the corresponding field quantities such as velocity and pressure are defined to be mapped to the center of the grid cells [1]. The physical width and height of this domain in meters is represented by D_W and D_H respectively. The grid spacing Δx represents the distance between two adjacent cells in meters, and is given by $\Delta x = \frac{D_W}{M} = \frac{D_H}{N}$. This domain is visualized in figure 1.

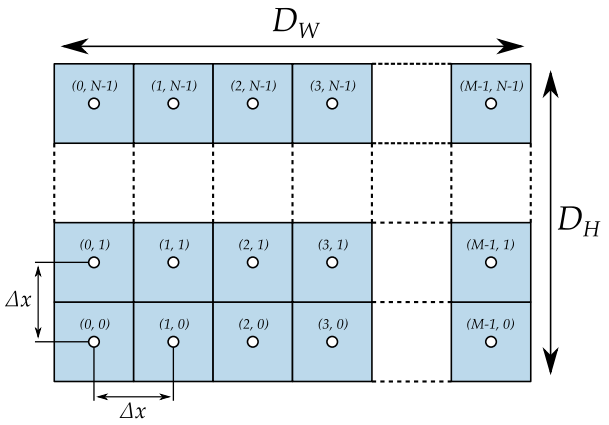

Figure 1: Simulation domain. The value-pairs in each cell represents the 2D-indices of the cells.

Using the lower left corner of the domain as origin, the center of a grid cell with indices (i, j) may be mapped to the position $\mathbf{x}$ in physical space using (2).

$$\mathbf{x} = \Delta x \cdot \left((i, j)^T + 0.5\right) \tag{2}$$

Another space that is important in practice is the *texture space*, which maps the coordinate $(0, 0)^T$ to the lower left corner and $(1, 1)^T$ to the upper right corner of the domain. A coordinate $\mathbf{x}$ in physical space may therefore be mapped to the corresponding coordinate $\mathbf{x}_t$ in texture space using (3).

$$\mathbf{x}_t = \frac{\mathbf{x}}{(D_W, D_H)^T} \tag{3}$$

2.2.1 Boundary

The fluid simulation is enclosed in a box that surrounds the simulation domain, which means that fluid is not able to exit or enter the simulation domain. The implicit boundary cells is any grid cell that lies outside of the simulation domain. This means that the edges of the simulation domain (any texture coordinate with any component equal to 0 or 1) is the boundary interface between the fluid and the solid wall that encloses the fluid.

2.3 Finite Difference Differential Vector Operators

Differential vector operators may be approximated numerically in the domain using *finite difference methods* [3]. This may be done by considering the four neighboring grid cells of the grid cell of interest as shown in figure 2.

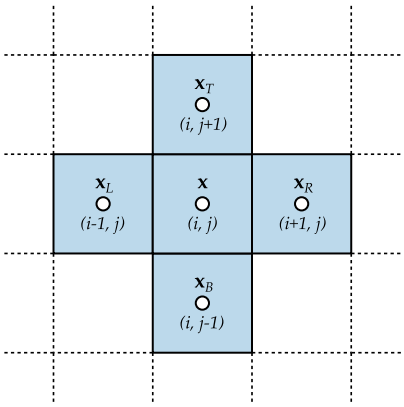

Figure 2: Neighboring grid cells of grid cell with 2D-indices (i, j) and center position $\mathbf{x}$.

To improve the readability of the following sections, the center positions of neighboring grid cells are defined as shown in (4).

$$\begin{aligned} \mathbf{x}_R &= \mathbf{x} + (\Delta x, 0)^T, \quad \mathbf{x}_L = \mathbf{x} - (\Delta x, 0)^T \\ \mathbf{x}_T &= \mathbf{x} + (0, \Delta x)^T, \quad \mathbf{x}_B = \mathbf{x} - (0, \Delta x)^T \end{aligned} \tag{4}$$

Using this definition, the partial derivative of a scalar field f at a position $\mathbf{x}$ with respect to y can for example be estimated with *central differences* using (5) [3].

$$\frac{\partial f}{\partial y}(\mathbf{x}) \approx \frac{f(\mathbf{x}_T) - f(\mathbf{x}_B)}{2\Delta x} \tag{5}$$

Differential vector operators are defined in terms of the *nabla* operator $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y})^T$, which produces different combinations of partial derivatives when combined with a field using some vector product [4]. These partial derivatives can be approximated using central differences as shown in (5) to create numerical approximations of differential vector operators.

The following sections details the differential vector operators needed for flow simulations and their numerical approximations. The analytical definition of these operators are available in vector calculus textbooks such as Ramgard [4].

2.3.1 Gradient

The gradient of a scalar field is a vector field that describes the direction and magnitude of the maximum local rate of change in the scalar field [4].

The analytical definition of the gradient of a scalar field q is given in (6) [4], and the numerical approximation is given in (7).

$$\nabla q = \left(\frac{\partial q}{\partial x}, \frac{\partial q}{\partial y}\right)^T \tag{6}$$

$$(\nabla q)(\mathbf{x}) \approx \frac{(q(\mathbf{x}_R) - q(\mathbf{x}_L),\, q(\mathbf{x}_T) - q(\mathbf{x}_B))^T}{2\Delta x} \tag{7}$$

2.3.2 Divergence

The divergence of a vector field is a scalar field that describes the degree to which the vector field locally flows out from a point, where a negative value means that the field flows in to the point (*sink*) and a positive value means that the field flows out from the point (*source*) [4].

The analytical definition of the divergence of a vector field $\mathbf{u}$ is given in (8) [4], and the numerical approximation is given in (9).

$$\nabla \cdot \mathbf{u} = \frac{\partial \mathbf{u}_x}{\partial x} + \frac{\partial \mathbf{u}_y}{\partial y} \tag{8}$$

$$(\nabla \cdot \mathbf{u})(\mathbf{x}) \approx \frac{\mathbf{u}_x(\mathbf{x}_R) - \mathbf{u}_x(\mathbf{x}_L) + \mathbf{u}_y(\mathbf{x}_T) - \mathbf{u}_y(\mathbf{x}_B)}{2\Delta x} \tag{9}$$

2.3.3 Laplacian

The laplacian of a scalar field is the divergence of the gradient of the scalar field [4]. The resulting scalar field therefore essentially describes the degree to which the gradient of the scalar field locally flows out from a point.

The analytical definition of the laplace operator of a scalar field q is given in (10) [4], and the numerical approximation is given in (11).

$$\nabla^2 q = \frac{\partial^2 q}{\partial x^2} + \frac{\partial^2 q}{\partial y^2} \tag{10}$$

$$(\nabla^2 q)(\mathbf{x}) \approx \frac{q(\mathbf{x}_R)+q(\mathbf{x}_L)+q(\mathbf{x}_T)+q(\mathbf{x}_B) - 4q(\mathbf{x})}{\Delta x^2} \tag{11}$$

2.3.4 Curl

The curl of a vector field is a vector field that describes the maximum local rate of rotation of the vector field, where the vector direction represents the rotation axis and the vector magnitude represents the rate of rotation about this axis (using the right-hand rule). The curl is defined as the cross product between the nabla operator and the vector field [4].

The cross product, and the curl operator by extension, is however only defined in three dimensions, while the domain is defined in two dimensions. To define the curl, a z-component that points out from the domain is therefore added, and this component is set to 0 in the vector field. The curl of a vector field $\mathbf{u}$ is then given by (12).

$$\begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} \end{pmatrix} \times \begin{pmatrix} \mathbf{u}_x \\ \mathbf{u}_y \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \frac{\partial \mathbf{u}_y}{\partial x} - \frac{\partial \mathbf{u}_x}{\partial y} \end{pmatrix} \tag{12}$$

Since the result only has rotation about the z-axis, the result from this operator is simplified to a scalar field, with the understanding that the scalar represents the rate of rotation about the z-axis. The analytical definition of the curl operator of a vector field $\mathbf{u}$ is then given in (13), and the numerical approximation is given in (14).

$$\nabla \times \mathbf{u} = \frac{\partial \mathbf{u}_y}{\partial x} - \frac{\partial \mathbf{u}_x}{\partial y} \tag{13}$$

$$(\nabla \times \mathbf{u})(\mathbf{x}) \approx \frac{\mathbf{u}_y(\mathbf{x}_R) - \mathbf{u}_y(\mathbf{x}_L) - \mathbf{u}_x(\mathbf{x}_T) + \mathbf{u}_x(\mathbf{x}_B)}{2\Delta x} \tag{14}$$

2.4 Numerical Integration

Differential equations on the form shown in (15) are common in flow simulation.

$$\frac{\partial \mathbf{y}}{\partial t} = \mathbf{f}(\mathbf{y}), \quad \mathbf{y}(t_0) = \mathbf{y}_0 \tag{15}$$

Here, $\mathbf{y}$ could for example be the position of a massless particle with a known initial position, and $\mathbf{f}$ a velocity field. The differential equation then describes how the particle moves with the flow over time.

Methods of numerically approximating integrals of such differential equations over a time step Δt is required in flow simulation. The following section details two such methods.

2.4.1 Euler Method

The simplest numerical integration method is the *Euler method*, which is listed in (16),

$$\mathbf{y}_1 = \mathbf{y}_0 + \mathbf{f}(\mathbf{y}_0) \cdot \Delta t \tag{16}$$

where $\mathbf{y}_1 \approx \mathbf{y}(t_0 + \Delta t)$ [3]. The *local truncation error*[1] of this method is in the order of $O(\Delta t^2)$ [3].

2.4.2 Runge-Kutta 4 Method

The accuracy of the numerical integration can be improved by iteratively sampling more vectors from $\mathbf{f}$ at different increments and forming a weighted average of these vectors as the final integration vector. The *Runge-Kutta 4 method* is a popular such method, which is listed in (17),

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f}(\mathbf{y}_0) \\ \mathbf{k}_2 &= \mathbf{f}(\mathbf{y}_0 + \tfrac{\mathbf{k}_1}{2} \cdot \Delta t) \\ \mathbf{k}_3 &= \mathbf{f}(\mathbf{y}_0 + \tfrac{\mathbf{k}_2}{2} \cdot \Delta t) \\ \mathbf{k}_4 &= \mathbf{f}(\mathbf{y}_0 + \mathbf{k}_3 \cdot \Delta t) \\ \mathbf{y}_1 &= \mathbf{y}_0 + \tfrac{1}{6} \cdot (\mathbf{k}_1 + 2 \cdot (\mathbf{k}_2 + \mathbf{k}_3) + \mathbf{k}_4) \cdot \Delta t \end{aligned} \tag{17}$$

where $\mathbf{y}_1 \approx \mathbf{y}(t_0 + \Delta t)$ [3]. The local truncation error of this method is in the order of $O(\Delta t^5)$ [3]. For short, the function notation in (18) is used in upcoming sections to reference this integration method.

$$\mathbf{y}_1 = \mathbb{RK}_4(\mathbf{f}, \mathbf{y}_0, \Delta t) \tag{18}$$

[1] Error (absolute difference from analytical solution) when performing one Δt step.

2.5 Boundary Conditions

When solving the Navier-Stokes equations in (1), two quantities must be solved for; velocity and pressure. Both of these have boundary conditions that must be satisfied at the boundary of the simulation domain [1].

By defining $\hat{\mathbf{n}}(\mathbf{x}_b)$ as the normal vector of the interface between the boundary and the fluid at any position $\mathbf{x}_b$ on the interface, the *no-slip condition* for the velocity is given in (19) [1].

$$\mathbf{v}(\mathbf{x}_b) \cdot \hat{\mathbf{n}}(\mathbf{x}_b) = 0 \tag{19}$$

That is, the normal component of the velocity at the interface should be equal to 0. However, since the velocity is defined at the center of grid cells while the interface lies between boundary and fluid cells, the velocity can not be directly specify the velocity at $\mathbf{x}_b$.

Instead, the velocity of boundary cells is set to be equal to the velocity of their adjacent fluid cells, but with a negated normal component [5]. On the discrete grid, the linearly interpolated velocity at the interface is then the average of these velocities. The normal component of this average velocity is equal to 0, which satisfies (19).

The *pure Neumann condition* for the pressure is given by (20) [1].

$$\frac{\partial p}{\partial \hat{\mathbf{n}}} = 0 \quad \text{on the boundary interface.} \tag{20}$$

That is, the rate of change in pressure along the normal component on the boundary interface should be equal to 0. This can be achieved by simply specifying the pressure at boundary cells to be equal the pressure of the adjacent fluid cell [5]. The central difference along the normal is then equal to 0, which satisfies (20).

2.6 Poisson's Equation

Poisson's equation is a partial differential equation on the form shown in (21),

$$\nabla^2 q = f \tag{21}$$

where f is a known scalar field and q is the sought scalar field [1]. Using the central differences Laplacian in (11) on the simulation domain to numerically approximate the solution to this equation results in the linear system in (21),

$$\mathbf{A}\mathbf{q} = \mathbf{f} \tag{22}$$

where $\mathbf{f}$ and $\mathbf{q}$ are $MN \times 1$ column vectors and A is a $MN \times MN$ matrix (see Hoffman Chapter 9 for details) [3]. This linear system is solved directly by finding the inverse of A, which potentially is a very large matrix depending on the size of the domain.

The matrix A is however a sparse and diagonally dominant matrix, which means that the iterative *Jacobi method* can be used to approximate this linear system with less computations [3]. This method works by first forming an initial guess q^0 of the discrete solution field, and then by iteratively improving this solution using (23) [5].

$$q^{k+1}(\mathbf{x}) = \frac{q^k(\mathbf{x}_R)+q^k(\mathbf{x}_L)+q^k(\mathbf{x}_T)+q^k(\mathbf{x}_B) - \Delta x^2 f(\mathbf{x})}{4} \tag{23}$$

2.7 Helmhotz-Hodge Decomposition

The Helmhotz-Hodge decomposition is a mathematical result that states that any vector field $\mathbf{w}$ can be decomposed into a divergence-free part $\mathbf{u}$ and a curl-free part ∇q as shown in (24),

$$\mathbf{w} = \mathbf{u} + \nabla q \tag{24}$$

where q is a scalar field and $\mathbf{u}$ is a vector field [1]. This result can be used to form a projection operator $\mathbb{P}$ that maps a vector field $\mathbf{w}$ onto its divergence free part $\mathbf{u}$ as shown in (25).

$$\mathbb{P}(\mathbf{w}) = \mathbf{w} - \nabla q \tag{25}$$

To perform this projection, the scalar field q must first be found. This can be done by taking the divergence on both sides of (24) as shown in (26) [1].

$$\nabla \cdot \mathbf{w} = \nabla \cdot \mathbf{u} + \nabla \cdot \nabla q \tag{26}$$

Since $\nabla \cdot u = 0$ and $\nabla \cdot \nabla = \nabla^2$, this equation simplifies to (21).

$$\nabla^2 q = \nabla \cdot \mathbf{w} \tag{27}$$

This is a Poisson equation on the form shown in (21), which can be approximated in the simulation domain using Jacobi iteration as shown in (23).

3 Method

The program was implemented in C++ and it uses OpenGL for both GPU accelerated simulation and visualization. The *graphical user interface* (GUI) was implemented using the cross-platform library *NanoGUI* [6], and the *OpenGL Mathematics* [7] (GLM) library was used to perform the few vector math operations required on the CPU.

3.1 Representing Fields on the GPU

The various fields used in the fluid simulation should ideally be represented as 2D-arrays of floating point data that is both write- and readable on the GPU. OpenGL is however made for writing to a framebuffer, which usually is the screen with 8-bit integer data, and to read from textures, which usually contain data that is specified on the CPU [8].

Framebuffer Objects (FBO's) are OpenGL objects that allow for the creation of framebuffers with custom attachments [8]. This enables the possibility to create a texture with 32-bit floating point data and use this as the color attachment of the framebuffer, which then makes it possible to write to the texture off-screen on the GPU. This texture can also be read from on the GPU as a normal texture when another framebuffer is bound. FBO's therefore makes it possible to alternate between reading from and writing to 2D-arrays of 32-bit floating point data entirely on the GPU [8]. Framebuffer objects with 32-bit floating point textures as color attachments were therefore chosen to represent the fields in the program.

3.2 Simulation Domain

The simulation domain that was used for the program in practice with OpenGL is represented as a quadrilateral that consists of four vertices $\mathbf{p}_n$ and two triangles. A texture coordinate $\mathbf{x}_{t,n}$ is also paired with each vertex. This quadrilateral is defined and visualized in figure 3.

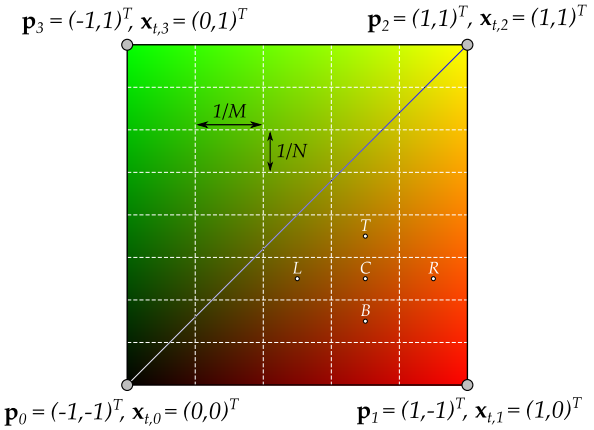

Figure 3: Visualization of GLSL simulation domain for $M = 5$ and $N = 8$. The x- and y-components of the interpolated texture coordinates $\mathbf{x}_t$ are mapped to the red and green channels respectively of this visualization.

The vertex positions are defined such that the quadrilateral spans the whole non-clipped part of the OpenGL *clip-space* in the xy-plane [8]. Using these vertex positions as output position for a vertex shader, along with the z- and $w-$component equal to 0 and 1 respectively, therefore makes the quadrilateral cover the whole framebuffer.

The texture coordinates are used to create five output variables `C`, `L`, `R`, `B` and `T` from the vertex shader that corresponds to the center and neighboring grid cell coordinates defined in (4). They are however defined in texture space rather than physical space, since the texture space is used to sample the fields in practice. OpenGL then interpolates these output variables in the rasterization process using *barycentric interpolation* across the triangles to create the correct texture coordinates for each fragment (grid cell) in the fragment shaders [8], where the actual simulation is carried out. The vertex shader used to create this simulation domain is listed in 1.

Listing 1 GLSL vertex shader that outputs the quad covering the whole framebuffer, along with texture coordinates for center and neighboring grid cells.

```
#version 430 core

layout (location = 0) in vec2 position;
layout (location = 1) in vec2 texcoord;

uniform vec2 tx_size; // (1/M, 1/N)

out vec2 T, L, C, R, B;

void main()
{
  C = texcoord;
  L = texcoord - vec2(tx_size.x, 0.0);
  R = texcoord + vec2(tx_size.x, 0.0);
  B = texcoord - vec2(0.0, tx_size.y);
  T = texcoord + vec2(0.0, tx_size.y);

  gl_Position = vec4(position, 0.0, 1.0);
}
```

3.2.1 Querying Velocity and Pressure with Respect to Boundary Conditions

The boundary conditions described in section 2.5 is satisfied in the program by first setting the texture wrapping for the pressure and velocity textures to `GL_CLAMP_TO_EDGE` [8]. Any queried texture coordinate outside of the simulation domain is then clamped to the texture coordinate of the closest grid cell within the simulation domain. This automatically satisfies the pure Neumann condition for the pressure since the central difference between neighboring fluid and implicit boundary cells becomes zero. The pressure is therefore queried in shaders using the function in listing 2.

The no-slip condition for the velocity is satisfied in combination with `GL_CLAMP_TO_EDGE` by simply negating the x- and/or y-component of the velocity if the corresponding component in the texture coordinate lies outside of the simulation domain. The average velocity of neighboring fluid and implicit boundary cells then becomes zero along

the normal component of the boundary interface. The velocity is therefore queried in shaders using the function in listing 3.

Listing 2 GLSL function to query the pressure at texture coordinate `X`.

```
float p(vec2 X)
{
  return texture(pressure, X).x;
}
```

Listing 3 GLSL function to query the velocity at texture coordinate `X`.

```
vec2 v(vec2 X)
{
  vec2 vel = texture(velocity, X).xy;

  if(X.x < 0.0 || X.x > 1.0) vel.x = -vel.x;
  if(X.y < 0.0 || X.y > 1.0) vel.y = -vel.y;

  return vel;
}
```

3.2.2 Workflow Example

The program has several shader programs and issues many OpenGL draw calls each simulation iteration. Instead of extensively listing all of these, this section provides a simple example of the workflow used to subtract the pressure gradient from the velocity. It should then be clear how this workflow was used to perform the rest of the computations using the provided equations.

The fragment shader used to subtract the pressure gradient from the velocity is listed in 4. This fragment shader is used with the vertex shader in listing 1 to create the complete shader program.

Listing 4 GLSL fragment shader that subtracts the pressure gradient from the velocity.

```
#version 430 core

layout(binding=0) uniform sampler2D pressure;
layout(binding=1) uniform sampler2D velocity;

uniform float half_inv_dx;

in vec2 C, L, R, T, B;

out vec4 new_velocity;

float p(vec2 X); // Listing 2
vec2 v(vec2 X); // Listing 3

void main()
{
  vec2 grad_p = vec2(p(R)-p(L),p(T)-p(B)) * half_inv_dx;
  new_velocity.xy = v(C) - grad_p;
}
```

Three framebuffer objects are needed to run this shader program; pressure, velocity and a temporary FBO to which the result is written. The textures of the pressure and velocity FBO's are bound to the `sampler2D` uniforms at bindings 0 and 1 respectively to provide read access to these textures in the fragment shader. The temporary FBO is bound as the framebuffer, to which the new velocity is written when the draw call is issued.

After the draw call has been issued to compute the result over the quadrilateral in figure 3, the velocity FBO still contains the old velocity while the temporary FBO contains the new velocity. The velocity is updated with the new velocity by simply swapping the temporary and velocity framebuffer objects. This was done by representing FBO's using memory pointers to instances of an utility class that encapsulates the handles and functionality of framebuffer objects, and then by simply swapping the memory pointers.

The resulting C++ member function that performs this operation is listed in 5. Note however that this function uses instances of OpenGL utility classes that were written for the project, which abtracts away most of the OpenGL calls for clarity and code re-usability. See the provided source code for implementation details of these classes.

Listing 5 C++ member function that subtracts the pressure gradient from the velocity by interfacing with OpenGL.

```
void FluidSolver::subtractPressureGradient()
{
  // Shader program using listing 1 and 4.
  static const Shader shader(vert, frag);

  Quad::bind();

  shader.use();

  glUniform2fv(shader.loc("tx_size"), 1, &tx_size[0]);
  glUniform1f(shader.loc("half_inv_dx"), 0.5f / dx);

  temp_fbo->bindFramebuffer();
  pressure->bindTexture(0);
  velocity->bindTexture(1);

  Quad::draw();

  std::swap(temp_fbo, velocity);
}
```

3.3 Solving Navier-Stokes Equations

The velocity terms of the differential equation in the Navier-Stokes equation in (1) may be approximated through numerical integration, but this leaves the pressure gradient term and continuity equation. To account for these, the projection operator defined in (25) is applied to both sides of the differential equation in (1) as shown in (28) to project both sides onto their divergence free parts [1].

$$\mathbb{P}\left(\frac{\partial \mathbf{v}}{\partial t}\right) = \mathbb{P}\left(-(\mathbf{v}\cdot\nabla)\mathbf{v} - \frac{1}{\rho}\nabla p + \frac{\mu}{\rho}\nabla^2\mathbf{v} + \mathbf{F}\right) \quad (28)$$

Since $\mathbf{v}$ is divergence free by definition, the left side is then unchanged, while the pressure gradient $\frac{1}{\rho}\nabla p$ on the right side is eliminated since the gradient of a scalar field

is guaranteed to be curl-free [4]. This results in the final equation to solve in (29) [1].

$$\frac{\partial \mathbf{v}}{\partial t} = \mathbb{P}\left(-(\mathbf{v} \cdot \nabla)\mathbf{v} + \frac{\mu}{\rho}\nabla^2\mathbf{v} + \mathbf{F}\right) \tag{29}$$

This equation is approximated by solving each velocity term one by one using numerical integration, where the solution to one term is used as the input velocity for the next term [1]. The resulting velocity is then projected onto its divergence free part to form the final velocity after one simulation iteration [1]. The outline of the fluid solver procedure to perform one simulation iteration is therfore given by algorithm 1. The order in which these terms are solved is based on the order suggested by Stam [1].

Algorithm 1 Fluid solver procedure used to perform one simulation iteration.

```
function SimulationStep(v, Δt)
    v = Force(v, Δt)
    v = Advect(v, Δt)
    v = Viscocity(v, Δt)
    v = Project(v)
    return v
end function
```

The following sections lists the methods that were implemented to solve each of these terms in the program. They are each expressed on the form in (30), where $\mathbb{F}$ is the operation the term uses to transform the input velocity $\mathbf{v}_i$ to the output velocity $\mathbf{v}_o$.

$$\mathbf{v}_o = \mathbb{F}(\mathbf{v}_i) \tag{30}$$

These sections uses the field and grid cell coordinates definitions in physical space from the theory section for clarity, but translating this to the texture space used in practice is straightforward as shown in section 3.2.2. Care must however be taken for the Runge-Kutta 4 function defined in (17), since each integration increment must be transformed from physical to texture space using (3) before sampling the texture.

3.3.1 External Force

The external force $\mathbf{F}$ used in the program is defined in (31),

$$\begin{aligned} \lambda(\mathbf{x}) &= \exp\left(-\frac{(\mathbf{x}-\mathbf{p})\cdot(\mathbf{x}-\mathbf{p})}{r}\right) \\ \mathbf{F}(\mathbf{x}) &= \mathbf{f}\cdot\frac{\lambda(\mathbf{x})}{\int_{\mathbb{D}}\lambda(\mathbf{y})d\mathbf{y}} = \mathbf{f}\cdot\frac{\lambda(\mathbf{x})}{\pi/r} \end{aligned} \tag{31}$$

where $\mathbf{p}$ is the position of the force, r is the radius extent around $\mathbf{p}$ where the force acts, $\mathbf{f}$ is the force vector in Newton and λ is the force falloff, which is normalized by dividing it by its integral over the simulation domain $\mathbb{D}$. This method was inspired by the one presented by Fernando [5], but their method does not normalize the falloff.

This force is applied to the velocity field using the Euler method as shown in (32) [1].

$$\mathbf{v}_o(\mathbf{x}) = \mathbf{v}_i(\mathbf{x}) + \mathbf{F}(\mathbf{x})\cdot\Delta t \tag{32}$$

The force position $\mathbf{p}$ can be moved around at run-time by using the mouse in the simulation window. The magnitude and angle of the force vector $\mathbf{f}$ can also be controlled via the GUI.

3.3.2 Self-Advection

The self-advection term is solved by moving the velocities from their original position to a new position found by integrating one time step along the velocity field [1]. This can be done using (33).

$$\mathbf{v}_o(\mathbb{RK}_4(\mathbf{v}_i, \mathbf{x}, \Delta t)) = \mathbf{v}_i(\mathbf{x}) \tag{33}$$

This explicit method is however potentially unstable if the integration step is larger than Δx [1]. It is also difficult to implement on the GPU, since it requires a *scatter* operation, in which the output velocity framebuffer is written to at computed memory locations.

Stam [1] instead suggests an implicit method that reverses this operation as shown in (34), which was used for the program.

$$\mathbf{v}_o(\mathbf{x}) = \mathbf{v}_i(\mathbb{RK}_4(\mathbf{v}_i, \mathbf{x}, -\Delta t)) \tag{34}$$

This method works by instead integrating in the opposite direction of the velocity field to find the velocity in the input texture that should be written to the current output velocity location.

The velocity texture filtering is set to `GL_LINEAR` when solving this term to perform bilinearly interpolated texture sampling [8]. This is because the integrated positions are likely to be located in between grid-cell centers, and the best approach is then to guess the velocity by interpolating surrounding velocities, rather than clamping the position to the nearest grid-cell center [5].

3.3.3 Viscosity

The viscosity term may be solved through explicit numerical integration using e.g. Euler as shown in (35),

$$\mathbf{v}_o(\mathbf{x}) = \mathbf{v}_i(\mathbf{x}) + \frac{\mu}{\rho}\left(\nabla^2\mathbf{v}_i\right)(\mathbf{x})\cdot\Delta t \tag{35}$$

where the scalar field $\nabla^2\mathbf{v}_i$ is approximated using the numerical approximation of the Laplacian in (11) [1]. This method is however potentially unstable for large integration steps, similar to the explicit method of solving the self-advection term [1].

Stam [1] therefore suggests an implicit method for this term as well which was used for the program. This method is given in (36),

$$\left(\mathrm{I} - \frac{\mu}{\rho}\nabla^2\Delta t\right)\mathbf{v}_o = \mathbf{v}_i \tag{36}$$

where I is the identity operator [1]. This is a Poisson-like equation which results in a large $MN \times MN$ linear system in the numerical case, which must be solved to find the new

velocity $\mathbf{v}_o$. This linear system is similarly approximated using the iterative Jacobi method in the program as shown in (37),

$$\mathbf{v}_o^{k+1}(\mathbf{x}) = \frac{\mathbf{v}_o^k(\mathbf{x}_R)+\mathbf{v}_o^k(\mathbf{x}_L)+\mathbf{v}_o^k(\mathbf{x}_T)+\mathbf{v}_o^k(\mathbf{x}_B) - \frac{\rho\Delta x^2}{\mu\Delta t}\mathbf{v}_i(\mathbf{x})}{4+\frac{\rho\Delta x^2}{\mu\Delta t}} \tag{37}$$

where $\mathbf{v}_o^0$ is the initial guess of the solution [5]. Again, see Hoffman [3] for a description of the linear system and how this translates to the expression used for the iterative Jacobi solver. The initial guess is set to $\mathbf{v}_i$ in the program, since the output velocity is likely to be similar to the input velocity.

The iterative Jacobi solver was implemented by using a loop on the CPU that issues one OpenGL draw call each iteration, using a shader program that performs one Jacobi iteration on the GPU using (37). The number of iterations of the loop therefore corresponds to the number of Jacobi iterations, which is specified via the GUI.

3.3.4 Projection

The projection step is performed using (38),

$$\mathbf{v}_o = \mathbb{P}(\mathbf{v}_i) \tag{38}$$

where $\mathbb{P}$ is the projection operator defined in (25) [1]. The scalar field q of the Poisson equation in (27) is then equal to $\frac{\Delta t}{\rho}p$, which is simply referred to as the pressure in the program since $\frac{\Delta t}{\rho}$ is baked in implicitly. This process is performed using the following steps:

1. Compute $\nabla \cdot \mathbf{v}_i$ using (9).
2. Solve $\nabla^2(\frac{\Delta t}{\rho}p) = \nabla \cdot \mathbf{v}_i$ using (23).
3. Compute the pressure gradient $\nabla(\frac{\Delta t}{\rho}p)$ using (7) and subtract it from $\mathbf{v}_i$.

Step 3 is the same process as the one detailed in section 3.2.2. Two approaches of selecting the initial guess for the Jacobi solver in step 2 was implemented; either use the pressure from the previous simulation iteration or set the initial guess to 0. The former method requires less solver iterations, but it can also produce oscillations in the pressure [5]. The number of Jacobi iterations used when solving the pressure is specified via the GUI.

3.4 Vorticity Confinement

A problem with the described method of performing the simulation on a discrete grid is that the solution suffers from numerical diffusion, which smooths out fine features in the flow over time [2]. This effect removes important features of the flow such as small vortices, which has a considerable effect on the appearance and the turbulence of the flow.

Fedkiw et al. [2] proposes a method called *vorticity confinement* that attempts to recover some of these features, which was implemented in the program. This method works by generating a rotational force vector field, which is applied to the velocity field using numerical integration. This step can be expressed as $\mathbf{v}_o = \mathbb{F}(\mathbf{v}_i)$ like the other velocity terms, and it is applied after the force term and before the advection term in algorithm 1.

This method works by first computing the curl ω of the vector field as shown in (39), using the numerical approximation in (14).

$$\omega = \nabla \times \mathbf{v}_i \tag{39}$$

A normalized curl location field $\hat{\mathbf{N}}$ is then computed using (40),

$$\hat{\mathbf{N}} = \frac{\nabla|\omega|}{\|(\nabla|\omega|)\|} \tag{40}$$

where $\nabla|\omega|$ is approximated using the numerical method in (9) [2]. This normalized vector field points in the direction of the greatest absolute change in curl.

In three dimensions, the rotational force vector field is then computed as the cross product between the normalized curl location field $\hat{\mathbf{N}}$ and the curl ω [2]. To do this in two dimensions, it is useful to expand the fields to three dimensions as described in section 2.3.4 again. The rotational force vector field is then given by (41),

$$\begin{pmatrix}\hat{\mathbf{N}}_x\\ \hat{\mathbf{N}}_y\\ 0\end{pmatrix} \times \begin{pmatrix}0\\0\\\omega\end{pmatrix} = \begin{pmatrix}\hat{\mathbf{N}}_y \cdot \omega\\ -\hat{\mathbf{N}}_x \cdot \omega\\ 0\end{pmatrix} \tag{41}$$

which reduces to only the x- and y-components in the two dimensional domain of the program. The output velocity is then finally found by applying this force to the input velocity using Euler integration as shown in (42),

$$\mathbf{v}_o = \mathbf{v}_i + \epsilon \cdot \begin{pmatrix}\hat{\mathbf{N}}_y\\ -\hat{\mathbf{N}}_x\end{pmatrix} \cdot \omega \cdot \Delta t \tag{42}$$

where ϵ is a scale factor used to control the strength of the vorticity confinement [2]. This scale factor is controlled via the GUI.

3.5 Visualization

This section describes the visualization methods that were implemented in the program to visualize the solution of the simulation. The visualization step is carried out after the simulation step in the program, and the various visualization modes and settings are specified via the GUI.

3.5.1 Ink

The ink visualization mode that was implemented in the program was inspired by the brief description by Fernando [5], and it works by maintaining a passive ink field $\boldsymbol{\psi}$ that is advected along the velocity field after each simulation iteration. This vector field has the form shown in (43),

$$\boldsymbol{\psi}(\mathbf{x}, t) = \begin{pmatrix} \boldsymbol{\psi}_R(\mathbf{x}, t) \\ \boldsymbol{\psi}_G(\mathbf{x}, t) \\ \boldsymbol{\psi}_B(\mathbf{x}, t) \end{pmatrix} \tag{43}$$

i.e. each vector component corresponds to red, green and blue in the *sRGB* color space. These components can however lie outside of the usual $[0, 1]$ sRGB range in the method that was implemented.

The initial state of this field is set to 0.5 for each component, and the field is updated after each simulation iteration by first adding new ink to the field. This is done using a method similar to the external force in section 3.3.1 as shown in (44),

$$\begin{aligned} \mathbf{C}(\mathbf{x}) &= \mathbf{c} \cdot \frac{\lambda(\mathbf{x})}{\pi / r} \\ \boldsymbol{\psi}_o(\mathbf{x}) &= \boldsymbol{\psi}_i(\mathbf{x}) + \mathbf{C}(\mathbf{x}) \cdot \Delta t \end{aligned} \tag{44}$$

where λ is the same falloff term used for the external force in (31) and $\mathbf{c}$ is a sRGB vector that specifies the color of the added ink. The value of $\mathbf{c}$ is determined by cyclically varying the hue and luminance of the color over time. The luminance can be both positive and negative, which means that the added ink can be both additive and subtractive. The luminance is also scaled by a user-defined value that controls the amount of added ink. The color is added in gamma compressed sRGB space, which based on testing appears to work better for ink mixing than the gamma expanded (linear) space that is usually used for additive and subtractive color mixing.

Once new ink has been added, the ink field is advected along the velocity field using the same implicit method used for the self-advection term as shown in (45).

$$\boldsymbol{\psi}_o(\mathbf{x}) = \boldsymbol{\psi}_i(\mathbb{RK}_4(\mathbf{v}, \mathbf{x}, -\Delta t)) \tag{45}$$

Before drawing the ink field to the screen, the field is finally tonemapped using an approximation of the ACES filmic tonemapping curve by Narkowicz [9]. This non-linear curve attempts to mimic film cameras by for example compressing color components above 1 and improving the contrast in darker areas. This was implemented to improve the dynamic range and the overall appearance/realism of the ink.

Another option to specify the ink distribution that was implemented in the program is to load an image from disk and use the colors from this to overwrite the ink field. This is simply done by creating an OpenGL 2D texture from the image, and then issuing a draw call with the ink framebuffer bound using a shader that samples the texture at each grid coordinate to overwrite the framebuffer. This makes it possible to load an image and then advect it along the velocity field.

3.5.2 Streamlines

The streamline visualization mode computes a *line integral convolution* (LIC) image along the velocity field. [10] This is performed in the program by first generating a 32-bit floating point noise image where each pixel contains a uniformly random value in the range $[0, 1]$. This image is then uploaded to the GPU as an OpenGL 2D texture.

The idea is then to compute a line integral over the time interval $[-\Delta T_c, \Delta T_c]$ along the velocity field, starting at each grid cell position. The noise texture is then sampled along the line integral and divided by the number of samples taken, to produce the average noise value along the line for the given grid cell [10].

In the numerical case, the coordinates $\mathbf{x}_i$ of the line integral starting at a point $\mathbf{x}$ is given by (46),

$$\mathbf{x}_i = \begin{cases} \mathbb{RK}_4(\mathbf{v}, \mathbf{x}_{i+1}, -\Delta t_c) & \text{if } i \in \{0, .., N - 1\} \\ \mathbf{x} & \text{if } i = N \\ \mathbb{RK}_4(\mathbf{v}, \mathbf{x}_{i-1}, \Delta t_c) & \text{if } i \in \{N+1, .., 2N\} \end{cases} \tag{46}$$

where $\Delta t_c = \frac{\Delta T_c}{N}$ and N is the number steps of the line integral in each direction. The average line integral noise $L(\mathbf{x})$ at a position $\mathbf{x}$ is then given by (47),

$$L(\mathbf{x}) = \frac{1}{2N+1} \sum_{i=0}^{2N} \eta(\mathbf{x}_i) \tag{47}$$

where η is the noise field/texture. This can be performed using the procedure in algorithm 2, which was implemented in a fragment shader.

Algorithm 2 Line integral convolution image procedure for a coordinate $\mathbf{x}$ in the domain.

$\eta_a = \eta(\mathbf{x})$
$\mathbf{x}_c = \mathbf{x}$
for all $i \in \{1, .., N\}$ **do**
  $\mathbf{x}_c = \mathbb{RK}_4(\mathbf{v}, \mathbf{x}_c, \Delta t_c)$
  $\eta_a = \eta_a + \eta(\mathbf{x}_c)$
end for
$\mathbf{x}_c = \mathbf{x}$
for all $i \in \{1, .., N\}$ **do**
  $\mathbf{x}_c = \mathbb{RK}_4(\mathbf{v}, \mathbf{x}_c, -\Delta t_c)$
  $\eta_a = \eta_a + \eta(\mathbf{x}_c)$
end for
$L(\mathbf{x}) = \frac{\eta_a}{2N+1}$

The resulting LIC image L is then visualized using the color map method described in section 3.5.3. Both the integration time ΔT_c and the number of integration steps N are specified via the GUI.

3.5.3 Color Maps

The fluid simulator maintains several scalar field, which can be visualized by mapping the scalar values to different colors using a transfer function. This was implemented in the program and the scalar fields that can be visualized using this method are:

- The curl of the velocity $\nabla \times \mathbf{v}$, computed in (39).
- The pressure $\frac{\Delta t}{\rho} p$, computed in (38).
- The LIC image L, computed in (47).
- The flow speed $\|\mathbf{v}\|$, computed at the end of each simulation iteration.

The transfer function is represented as an OpenGL 1D texture with the interpolation set to `GL_LINEAR` [8]. Several scientific color maps by Crameri [11] was added to the program, which are represented as arrays of sRGB color vectors on the CPU. These can be selected from the GUI, at which point they are uploaded to the transfer function texture buffer on the GPU.

Since 1D textures are sampled with scalar texture coordinates in the range $[0, 1]$ [8], the scalar values s in the visualized scalar field are mapped to texture coordinates s_t using (48),

$$s_t = \frac{s - s_{\min}}{s_{\max} - s_{\min}} \tag{48}$$

where $[s_{\min}, s_{\max}]$ is the range of the scalar field that should be mapped to the range $[0, 1]$ of the transfer function.

The range $[s_{\min}, s_{\max}]$ can be selected manually in the GUI, but an option to automatically select this range was also implemented. This is done by downloading the scalar field texture data from the GPU to the CPU using the OpenGL function `glGetTexImage` [8]. This data is then iterated through to find the minimum and maximum value of the scalar field, which is used for the range $[s_{\min}, s_{\max}]$ to map the transfer function to the whole range of the scalar field.

The wrap mode of the transfer function texture is set to `GL_MIRRORED_REPEAT`, which mirror-repeats the transfer function for texture coordinates outside of the range $[0, 1]$ [8]. This can be used as a simple way of visualizing *contour lines* of the scalar field by setting the range $[s_{\min}, s_{\max}]$ lower than the range of values in the scalar field.

3.5.4 Arrow Plot

The arrow plot visualization mode that was implemented in the program can be used in conjunction with all of the previously described visualization modes. This mode works by drawing regularly spaced arrows that points in the direction of the velocity field on top of the underlying visualization. This was done by first creating a simple arrow triangle mesh in the 3D-program *Blender* that consists of 9 vertices and 7 triangles.

The base of this arrow is located at the coordinate $(0, 0)^T$, and it points in the positive x-direction. The vertices of this arrow are processed using the vertex shader in listing 6.

Listing 6 GLSL vertex shader that processes the vertices of an arrow. This results in an arrow with the base located in texture coordinate `coord`, that points in the direction of the velocity field at this coordinate.

```
#version 430 core

layout(location = 0) in vec2 position;

layout(binding = 0) uniform sampler2D velocity;

uniform vec2 coord;
uniform vec2 scale;

void main()
{
  vec2 d = normalize(texture(velocity, coord).xy);
  mat2 rot = mat2(d.x, d.y, -d.y, d.x);
  gl_Position = vec4((coord - 0.5) * 2.0, 0.0, 1.0);
  gl_Position.xy += rot * position * scale;
}
```

This shader first samples the velocity at the texture coordinate `coord`, and then creates the rotation matrix that rotates the vector $(1, 0)^T$ to this velocity direction. The arrow vertices are then translated to the same coordinate (in clip space), rotated by the rotation matrix and scaled by the input `scale` vector. The scale vector used is defined as shown in (49),

$$(1, D_W/D_H)^T \cdot \kappa \tag{49}$$

where κ is the arrow scale specified via the GUI. Besides specifying the size of the arrow, this vector also accounts for the aspect ratio between the width and height of the domain to draw the arrow in the correct aspect ratio.

The fragment shader used for the complete arrow plot shader program simply outputs the arrow color, which is specified via the GUI. Multiple arrows are then drawn using this shader program by issuing multiple arrow draw calls with different arrow coordinates. The number of arrows in the horizontal direction is GUI-specified, and the program then determines the number of arrows to use in the vertical direction to distribute the arrows uniformly in the domain. Listing 3 shows the basic procedure that was implemented to do this.

Algorithm 3 Basic procedure to draw uniformly distributed arrows in the domain.

```
cols := GUI-specified
rows = ⌊cols · (D_H/D_W)⌋
c = 0
for all i ∈ {1, .., cols} do
    c_x = i / (cols + 1)
    for all j ∈ {1, .., rows} do
        c_y = j / (rows + 1)
        DrawArrowAt(c)
    end for
end for
```

4 Results

The source code for the fluid simulator and animated results can be found on github:

https://github.com/linusmossberg/fluid-simulator

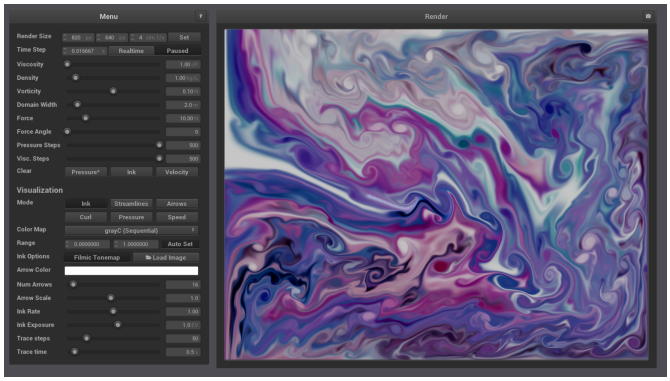

Figure 4: Screenshot of fluid simulator application.

See appendix B for instructions on how to download, build and run the program. A screenshot of the program is shown in figure 4.

The scientific color maps by Crameri [11] used for the results in this section are displayed in figure 5. These color maps are referenced as *color-map*(s_{min}, s_{max}), which makes it possible to use the mapping defined in (48) to map colors in the visualizations to the underlying scalar values in the visualized scalar field.

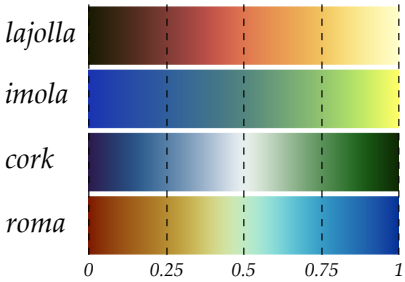

Figure 5: Scientific color maps by Crameri [11] used in results section.

Some of the images are pretty small in the upcoming section to fit them in the format of this document, but they are still high resolution which makes it possible to zoom in to get a better look.

4.1 Advecting an Image

Figure 6 shows an example where an image was loaded to overwrite the ink field, after which the image was advected along the velocity field. This was done by first running the simulation and moving around the force source in the domain with the mouse for a while to generate an interesting velocity field. The simulation was then paused and the ink rate and force magnitude was set to 0. The image was then loaded to overwrite the ink field, and the simulation was then un-paused and paused again every 0.5 seconds to capture an image after each step.

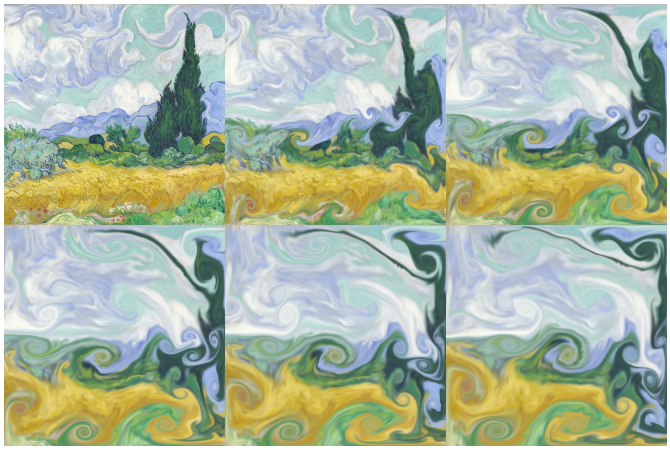

Figure 6: Example of ink field set to an image (*Wheat Field with Cypresses* by Vincent van Gogh). The sequence goes from left to right, top to bottom.

4.2 Contour Lines

Figure 7 shows and example where the pressure ($\frac{\Delta t}{\rho}p$) has been visualized using a color map range of [-15e-5, 15e-5]. This range is smaller than the range of the underlying pressure, which creates contour lines. Each cycle in the color map therefore represents a change in pressure of $|[\text{-}15\text{e-}5, 15\text{e-}5]| \cdot 2 \cdot \frac{\rho}{\Delta t} = 36\ Pa$ ($\frac{\rho}{\Delta t} = \frac{10^3\ kg/m^3}{1/60\ s}$), and the contour lines are orthogonal to the pressure gradient.

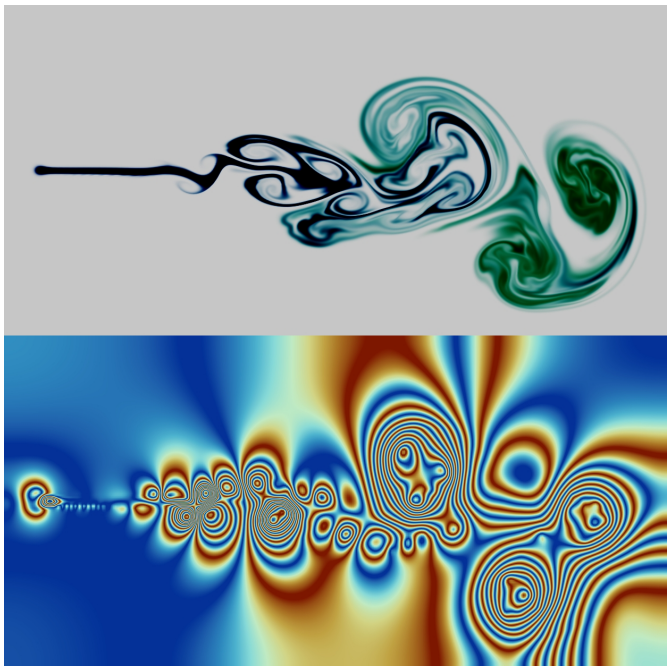

Figure 7: **Top**: Ink. **Bottom**: Pressure, *roma*(-15e-5, 15e-5). $\mu = 1\ cP$, $\rho = 1\ kg/L$, $\epsilon = 0.03$, $\Delta t = 1/60\ s$

4.3 Vorticity Confinement and Viscosity

This section lists comparisons of the effect of vorticity confinment and the viscosity. To get comparable results, all parameters except for the vorticity scale ϵ and the viscosity μ were kept the same, and all parameters were kept constant throughout the simulation. The simulation

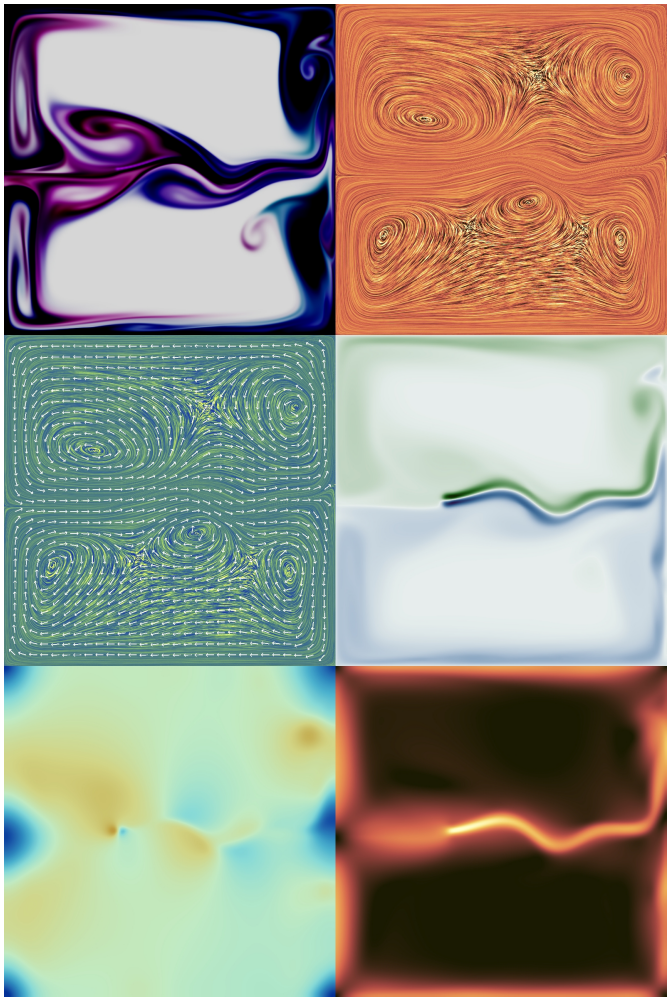

Figure 8: Visualizations of flow state with $\epsilon = 0$ and $\mu = 1\ cP$. Images from left to right, top to bottom:
1. Ink **2.** Streamlines, *lajolla*$(0, 1)$ **3.** Streamlines and Arrows, *imola*$(0, 1)$ **4.** Curl, *cork*$(-33.88, 33.88)$ **5.** Pressure, *roma*$(-0.0083, 0.0083)$ **6.** Speed, *lajolla*$(0.0012, 1.6372)$

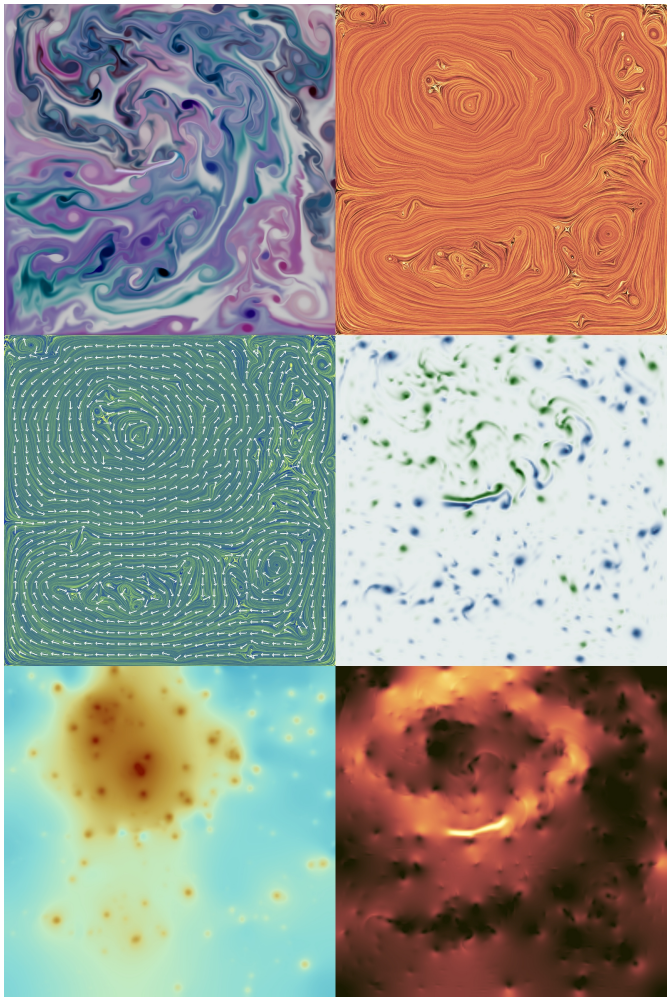

Figure 9: Visualizations of flow state with $\epsilon = 0.1$ and $\mu = 1\ cP$. Images from left to right, top to bottom:
1. Ink **2.** Streamlines, *lajolla*$(0, 1)$ **3.** Streamlines and Arrows, *imola*$(0, 1)$ **4.** Curl, *cork*$(-59.61, 59.61)$ **5.** Pressure, *roma*$(-0.0131, 0.0131)$ **6.** Speed, *lajolla*$(0.0025, 1.6785)$

ran until up until $t = 22$s in all cases, at which point the images were captured. The timestep Δt was also kept to a constant $\frac{1}{60}$s, since a dynamic time step can influence the results even though the simulation is stable for all time steps. The density ρ was set to 1 $kg/L = 1000\ kg/m^3$ and the simulation size to $D_W = D_H = 2$m. The force/ink source was placed at the texture coordinate $(\frac{1}{3}, \frac{1}{2})^T$, and the force vector was set to $(10, 0)^T\ N$.

Figures 8 and 9 show the difference between not using ($\epsilon = 0$) and using exaggerated ($\epsilon = 0.1$) vorticity confinement. The viscosity μ was set to 1 $cP = 0.001\ kg/(m \cdot s)$ in both of these simulations, which paired with $\rho = 1\ kg/L$ corresponds to water at 20° C [12].

The simulation in figure 10 used the same vorticity scale ($\epsilon = 0.1$) as the one in figure 9, but the viscosity μ was set to 3000 $cP = 3\ kg/(m \cdot s)$. Figures 9 and 10 therefore show the effect of the viscosity when all other parameters are kept the same. Note that the only way in which the dynamic viscosity μ and the density ρ are used in practice is on the form $\frac{\mu}{\rho}$, i.e. the *kinematic viscosity*. The results can therefore be translated to different liquids with the same ratio between μ and ρ, and there is no need to perform tests with varying ρ since the effect is inversely proportional to varying μ. The kinematic viscosity for $\mu = 3000\ cP$ and $\rho = 1\ kg/L$ roughly corresponds to the kinematic viscosity of cotton honey with 17% moisture at 30° C ($\mu = 4413\ cP$ [13] and $\rho = 1.47\ kg/L$ [14]).

Since the curl visualization uses the *cork* color map, blue and green corresponds to clock-wise (negative) and anti clock-wise (positive) rotation respectively. Similarly, because the pressure visualization uses the *roma* color map, blue and orange corresponds to regions of positive and negative pressures respectively. The color map range of the pressure visualizations can be multiplied with $\frac{\rho}{\Delta t} = 60000$ to get the pressure range in Pascal.

4.4 Performance

The program runs in real-time with $\sim \frac{1}{60}$ seconds per iteration even on the integrated graphics of a passively cooled *Intel Core m3-6Y30* processor from 2015, using 50 iterations for each Jacobi solver, a simulation grid size of 200×200 and a visualization size of 800×800 pixels. A dedicated graphics card like the *Nvidia GTX 1070* reduces the time per iteration to less than a millisecond when using the same settings.

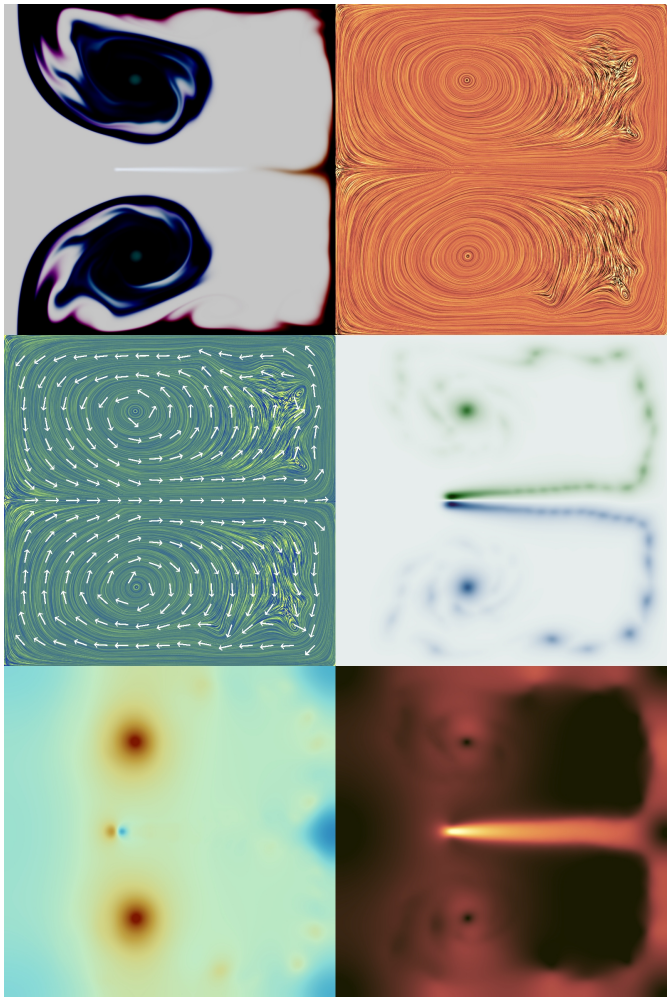

Figure 10: Visualizations of flow state with $\epsilon = 0.1$ and $\mu = 3000$ cP. Images from left to right, top to bottom:
1. Ink **2.** Streamlines, $lajolla(0, 1)$ **3.** Streamlines and Arrows, $imola(0, 1)$ **4.** Curl, $cork(-29.89, 29.89)$ **5.** Pressure, $roma(-0.042, 0.042)$ **6.** Speed, $lajolla(0.0012, 1.1460)$

5 Discussion

Several things can be interpreted by looking at the results in figures 8, 9 and 10 as a whole. By comparing the streamline and curl visualizations, we can see that regions with circular vortex flow corresponds to regions of high absolute curl, as expected based on the curl definition. By comparing these regions with the pressure visualization, we can also see that the pressure is lowest at the center of these vortices, and that the pressure gradually increases with distance to the vortex centers. We can also see that the force creates a region of low pressure behind it and a region of high pressure in front of it, relative to the force direction. This stands out the most in figure 10. Something that stands out in the pressure of figure 8 is also that increasing regions of alternating high and low pressure are created in front of the force, which curves the flow when it moves from regions of high to low pressure. This effect stands out more when looking at the simulation from the beginning in real-time, and it appears to be what causes the transition between laminar and turbulent flow.

As can be seen by comparing figure 8 and 9, vorticity confinement has a considerable effect on the fluid. The curl visualization in figure 9 shows that the vorticity confinement has created many small vortices scattered throughout the domain. The ranges of the color maps in the curl visualizations also show that the magnitude of the maximum rotational velocity is about twice as high in the simulation using vorticity confinement (33.99 vs. 59.61 rad/s). By looking at the streamlines and the pressure visualizations, we can also see that there are vortices at a wide range of scales. This creates chaotic behaviour where vortices of different scales interact with each other to create unpredictable turbulent flow. This turbulence has spread out the passive ink field over the whole domain in figure 9 to create a more intricate appearance of stirred ink/dye. Another interesting thing to note about the ink is that the total amount of added ink is the same in both images, but the ink in figure 9 appears brighter since it has been mixed more thoroughly with the brighter ink of the initial state. Turbulent flow therefore appears to be more diffusive than laminar flow.

As can be seen by comparing figure 9 and 10, the viscosity also has a considerable effect on the fluid. The increased viscosity has reduced the number of vortices in the domain, and therefore the turbulence of the flow. The flow has instead become much more predictable with two large mirrored vortices. Similarly to figure 8, the ink is not nearly as spread out in figure 10 when compared with figure 9, which enforces the observation that laminar flow is less diffuse than turbulent flow. Another interesting thing to note is that the ink in figure 10 has not been able to move as far in the 22 seconds as the ink in figures 8 or 10 due to the increased friction in the fluid. This is also reflected by comparing the ranges in the speed visualization, which shows that the maximum speed in the viscous fluid is 1.15 m/s, as opposed to 1.68 m/s in figure 9. This speed also dissipates faster with distance to the force in the viscous simulation, based on the speed color map visualizations.

6 Conclusions

This project turned out to be very interesting, and I had to learn a lot about fluid dynamics, numerical methods and general purpose GPU computing using OpenGL in the process.

The presented vorticity confinement method proved to work well in recovering some of the features lost due to numerical diffusion, as it greatly improved the visual appearance of the fluid. This method may not be very physically accurate however since the added force is scaled by an arbitrary factor that greatly influences the result. This method is therefore probably most suitable for applications where the fluid only needs to appear realistic, such as in movies or games. Slower particle-based methods or more accurate grid-based solvers with less numerical diffusion is therefore probably more suitable for engineering applications.

The implemented visualization methods proved to be very useful in interpreting the results, and they led to

insights about fluid dynamics which I previously did not have. Examples of this is how the pressure acts around vortices, and how the pressure acts around forces to transition the flow from laminar to turbulent. The fact that the simulation is interactive and runs in real-time thanks to the ability of the GPU and OpenGL to solve grid/fragment-based problems in parallel also proved to be a great way of gaining intuitive understanding about fluid dynamics.

Something I learned about numerical methods is that stable methods does not necessarily remove the need to be careful about simulation time steps. The solution is just guaranteed to not blow up to infinity, but the solver still becomes increasingly inaccurate for larger time steps, and increasingly numerically diffusing for smaller time steps.

References

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

[2] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, page 15–22. Association for Computing Machinery, New York, NY, USA, 2001. ISBN 158113374X.

[3] Joe D Hoffman. *Numerical Methods for Engineers and Scientists*. McGraw-Hill, New York, NY, 2:nd edition, 2001. ISBN 9780824704438.

[4] Anders Ramgard. *Vektoranalys*. Teknisk högskolelitteratur i Stockholm, Stockholm, 3:rd edition, 2000. ISBN 91-85484350.

[5] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Addison-Wesley Professional, 1:st edition, 2004. ISBN 0321228324.

[6] Wenzel Jakob and Contributors. Nanogui. https://github.com/mitsuba-renderer/nanogui. [Online; accessed 2021-05-06].

[7] G-Truc Creation and Contributors. Opengl mathematics. https://github.com/g-truc/glm. [Online; accessed 2021-05-06].

[8] Khronos group. Opengl® 4.5 reference pages. https://www.khronos.org/registry/OpenGL-Refpages/gl4/. [Online; accessed 2021-05-06].

[9] Krzysztof Narkowicz. Aces filmic tone mapping curve. https://knarkowicz.wordpress.com/2016/01/06/aces-filmic-tone-mapping-curve/, 2016. [Online; accessed 2021-05-06].

[10] Brian Cabral and Leith Casey Leedom. Imaging vector fields using line integral convolution. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, page 263–270. Association for Computing Machinery, New York, NY, USA, 1993. ISBN 0897916018.

[11] Fabio Crameri. Scientific colour maps, February 2021. [Software, version 7.0.0].

[12] Engineers Edge. Water - density viscosity specific weight. https://www.engineersedge.com/physics/water__density__viscosity__specific__weight__13146.htm. [Online; accessed 2021-05-06].

[13] S. Yanniotis, S. Skaltsi, and S. Karaburnioti. Effect of moisture content on the viscosity of honey at different temperatures. *Journal of Food Engineering*, 72(4):372–377, 2006. ISSN 0260-8774.

[14] Laleh Mehryar, Mohsen Esmaiili, and Ali Hassanzadeh. Evaluation of some physicochemical and rheological properties of iranian honeys and the effect of temperature on its viscosity. *American-Eurasian J. Agric. & Environ. Sci*, 13 (6):807–819, 2013.

A Notation

Vectors/coordinates $\mathbf{x} = (\mathbf{x}_x, \mathbf{x}_y)^T$ are denoted in upright bold, and the normalized version $\hat{\mathbf{x}} = \frac{\mathbf{x}}{\|\mathbf{x}\|}$ is denoted with a hat, where $\|\mathbf{x}\|$ denotes the length of $\mathbf{x}$.

The small dot symbol $\cdot$ denotes normal multiplication, while the large dot symbol $\bullet$ denotes the dot/scalar product between vectors.

To make equations more compact and readable, arguments of fields/vector functions $f(\mathbf{x}, t)$ are sometimes omitted, e.g. f or $f(\mathbf{x})$, depending on which arguments are most relevant.

B Cloning, Building and Running the Program

To clone and build the program and all external libraries, install git, CMake and a modern C++ compiler for your platform (e.g. MSVC on Windows or GCC on Linux) and run the commands in listing 7.

Listing 7 Commands to clone and create build files.

```
git clone --recursive
↪  https://github.com/linusmossberg/fluid-simulator
cd fluid-simulator
mkdir build
cd build
cmake ..
```

This will generate build files in the *fluid-simulator/build* directory, which can be built with your compiler (e.g. by opening the *.sln* file in Visual Studio for MSVC or using the `make` command for GCC). Additional packages are however required on certain platforms. These can be installed using listing 8 on Debian/Ubuntu, or listing 9 on Fedora.

Listing 8 Installing packages on Debian/Ubuntu.

```
apt-get install xorg-dev libglu1-mesa-dev
```

Listing 9 Installing packages on Fedora.

```
sudo dnf install mesa-libGLU-devel libXi-devel
↪  libXcursor-devel libXinerama-devel libXrandr-devel
↪  xorg-x11-server-devel
```

The build process generates a single self-contained executable file (with shaders embedded and libraries statically linked), which can simply be run anywhere.