Interactive 2D Particle-based Fluid Simulation for Mobile Devices

Daniel Månsson

073 508 1234

dmans@kth.se
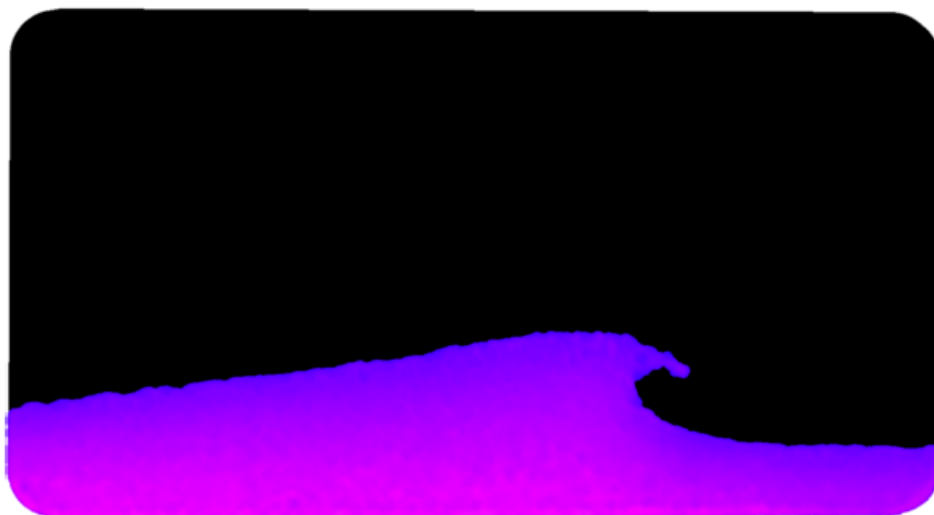
Stockholm, Sweden 2013

Abstract

This report presents an implementation of a fluid simulation algorithm, capable of running at 60 frames per second on a modern handheld tablet device. The goal was to create an interactive application, usable for helping students understand and visualize some fundamental fluid behaviours.

The algorithm was based on previous research for an algorithm, targeted to run on a desktop PC at 10 frames per seconds for 1000 particles. This algorithm was reduced by removing the capability to simulate complex elastic fluid properties and made run faster.

The resulting implementation is capable of simulating and visualizing: waves, splashes, fluid level equalization in a series of tubes and simple fluid flow through pipes. The implementation is capable of simulating 3000 particles at a steady 60 frames per second on the target hardware.

Referat

Interaktiv partikelbaserad vätskesimulation i 2D för mobila enheter

Den här rapporten presenterar en implementation av en vätskesimuleringsalgoritm, kapabel att simulera i 60 bilder per sekund på en modern surfplatta. Målet var att skapa en interaktiv applikations som är användbar för att hjälpa studenter förstå och visualisera några grundläggande vätskeegenskaper.

Algoritmen var baserat på tidigare forskning, som var riktad mot att fungera på en stationär PC med en uppdateringsfrekvens på 10 bilder per sekund för 1000 partiklar. Denna algoritm reducerades genom att ta bort möjligheten att simulera komplexa elastiska vätskor och kunde därmed fungera snabbare.

Den resulterande implementationen klarar av att simulera och visualisera: vågor, stänk, en vätskenivå som jämnar ut sig och enklare flöden genom rör. Implementationen har kapacitet att simulera 3000 partiklar i 60 bilder per sekund på målhårdvaran.

Contents

Chapter 1

Introduction

Computational fluid dynamics is an interesting and well researched field. There exists many algorithms for simulating and visualizing the motion of fluids for a range of different applications, especially in the field of special effects for movies and computer games. However, the calculations required for fluid simulations has always been computational expensive. Therefore, most algorithms either focus on offline rendering or require some kind of high power device, such as a GPU[1], for making the simulation run in real time.

In recent years, the power of mobile devices have increased enough to make them a viable choice for simulating fluids. Naturally, their computational power is still not on par with regular desktop computers, since they are required to run from a limited power source and have a greater size constraint. This puts an interesting constraint on the available previous research since the research to date has tended to focus either on algorithms for real-time simulation on the GPU, or algorithms for large scale offline simulations.

The target processing unit for this prototype will be the Tegra 3 mobile processor, released in 2011. It is a common high end processing unit that has been used in many devices. With the constant increase of processing power in mobile devices, the Tegra 3 should provide a good baseline for the performance benchmark. The Tegra 3 has got a quad- core processor and a GPU with 12 pixel shader units. The test device will be a Microsoft Surface RT tablet, using the Tegra 3 T30L(1.2 GHz) with 2 GB of RAM.

While the Tegra 3 is powerful by mobile standards, it does not compare to the performance of a desktop computer. This is extra noticeable when comparing the GPUs, where the Tegra 3 is noticeably weaker than a high end GPU. This will make the previous research using the GPU less suitable for this simulation, since they usually uses a high end desktop GPU. The Tegra 3 GPU will not have enough performance to perform a complex simulation on top of rendering the scene.

---

[1]GPU - Graphics Processing Unit. Specialized hardware used to perform graphics calculations with high performance. Since those calculations basically are floating point operations this kind of hardware can be used for other kind of calculations as well, such as fluid simulation.

1.1 Purpose

The purpose of this prototype is to explore the capability of running fluid simulation on mobile devices and to use the prototype to visualize some interesting fluid properties. The goal of the visualization is to make this prototype usable for educational purposes, where people getting started in fluid dynamics can get an idea of how fluids behave.

1.2 Problem statement

Performance is one of the main issues with simulating fluids on mobile devices. To get a usable simulation a high number of particles is needed, but increasing the number of particles also increases the computational load. This article aim to explore which parts of a existing algorithm can be left out, in order to increase performance, as well as the effect of threading on the Tegra 3 device.

Another issue to take into consideration is the actual fluid behaviour. While it is important that some of the most basic fluid dynamics features are plausibly reproduced, the simulation does not need to perfectly physically accurate. The goal is only to use this project as a visual aid and not to be used for actual measurements corresponding to the real world.

Chapter 2

Background

This chapter aims to give a brief introduction to computational fluid dynamics. There will not be an in-depth discussion of fluid simulation techniques, since the focus of this work is mainly implementation based and educational. The literature used in this chapter provides a good source for a more in-depth discussion.

2.1 Basic fluid simulation

There are many different variations of computational particle based fluid simulation algorithms available in literature. However, little work has been done on fluid simulation on lower power devices, since fluid dynamics is inherently computational expensive.

[HKK07] and [WLL04] describes two algorithms using the GPU for doing the computational work. In a few years, it is possible that mobile GPUs becomes powerful enough to simulate fluids and still being able to render a scene. This is not the case on the target device, which needs the GPU to work on rendering the frame.

Many of the CPU based algorithms are not designed to run in real time on a desktop computer. Even if they claim to be interactive, the frame rates usually ends up being 10-20 frames per seconds. This is not an acceptable frame rate for this implementation's target application, which requires a steady 60 frames per second.

The implementation described in this report will be based on an algorithm aiming to simulate viscoelastic fluids, written by Clavet, Beaudoin and Poulin [CBP05]. One problem with this algorithm is that it only runs at 10 frames per second on their target hardware and has support for much more complex fluid behavior than is needed for our implementation. Therefore, a reduced version of their algorithm will be used, avoiding the computationally expensive part which managed the elasticity. However, the algorithm is both robust and relatively simple to implement, making it suitible for the scale of this project.

The algorithm described in [CBP05] aims to solve the Navier- Stokes equation of fluid motion, which is the base equation for all fluid simulations. It is a precise mathematical model for natural fluid flow, which describes the fluid as a velocity field

for the fluid density. The Navier-Stokes equations precisely describes the evolution of this velocity field over time. There exists two main categories for integrating over the velocity field: eulerian grids and lagrangian particles, which will be given a quick overview in the following sections.

2.1.1 Eulerian grids

Eulerian grid use fixed points in a reference frame to track fluid motion. These points track the velocity of the flow passing through them, the current density at the point and how it changes.

This technique are mostly used for high quality offline fluid simulation, but are also in a few cases used for interactive fluid simulation, such as [WLL04].

While this technique can give great quality simulations, there are some possible problems. The conservation of mass has to be handled carefully, since it is tracked explicitly at the points. Even rounding errors can lead to mass disappearing from the simulation.

Fast moving particles can also be troublesome, since the maximum distance a unit of mass can travel during a simulation step is from one fixed point to the adjacent point.

This is not the technique used by Clavet, *et al*, but is a viable option in many cases.

2.1.2 Lagrangian particles

Lagrangian particles is an alternative approach to fluid simulation and the one used by Clavet, *et al*. Instead of tracking fixed points in space, small units of mass is tracked using particles. This approach is called Smoothed particle hydrodynamics (SPH) and was first developed by Lucy [Luc77] and by Gingold and Monaghan [GM77] to solve astrophysical problems.

The particles maintain certain field properties such as velocity and pressure. The state of the field can be evaluated at any position by sampling all particles by using a smoothing kernel function. This is a radially symmetric kernel function with combines the data from all nearby particles within the given radius. Particles closer to the center has more effect on the resulting value and particles outside of the radius does not effect the value at all.

With this approach conservation of mass is trivial, since the particles' mass values are constant. The fluid does not require a predefined grid and calculations are not needed where there are no fluid. It is also more suitable for fast moving fluids behavior, such as splashes, since there is no implicit limit on how far an unit of mass can travel during one simulation step. These properties makes this technique suitable for our implementation.

Chapter 3

Implementation

This chapter will describe the implementation of the simulation. This implementation is a two-dimensional fluid simulation, based on previous research with adjustments to make it more suitable for real time simulation.

Section 3.1 will describe the algorithm used by the simulation; the algorithm is based on previous research and here we will present the previous research along with the changes made to it.

The following sections will describe implementation specific algorithms and techniques which are not part of the core fluid simulation algorithm. Section 3.2 and 3.3 will discuss techniques for increasing the computational performance, section 3.4 and 3.5 will discuss issues related to user interaction and section 3.6 provides an overview of the rendering techniques used.

3.1 Algorithm

The algorithm used in this implementation is based on an algorithm for particle-based viscoelastic fluid simulation, presented by Clavet, *et. al*, mentioned in the background chapter.

The algorithm presented by [CBP05] aims to handle fluids of different viscosity and elasticity, making it able to simulate splashing water and more robust clay. The method used is robust, stable and able to animate simpler scenarios at interactive frame rates, making it suitable for a simulation with user interaction.

However, their performance requirements are based on a PC (2005) with no power limitations and their definition of interactive frame rate is 10 frames per second. Our implementation requires a frame rate of 60 frames per second and needs to be able to run on a low power handheld device. Therefore, some modifications are needed.

First of, instead of implementing the algorithm in three-dimensional space we will implement it in two-dimensional space. This does not modify the algorithm but will decrease the average number of neighbor- pairs per particle by a factor of 3 and make the rendering simpler. Also, the computation of the fluid elasticity is one of

the more computational expensive part of their algorithm, therefore it was left out of our algorithm.

We will describe the algorithm we used and which parts are based on their algorithm in section 3.1.3.

3.1.1 Numerical integration

The algorithm described by [CBP05] advances the simulation by using a prediction-relaxation approach. They argue that to avoid the stability issues inherited by an explicit integration scheme, using an implicit integration scheme is more suitable.

Their method avoids explicit force integration by going though the following steps, each simulation update:

1: Get the new position of a particle by advancing it based on its velocity.
2: This position is then changed (relaxed) during the simulation step by positional constraints, such as density adjustment and collision response.
3: At the end of the simulation step, the new velocity are calculated by subtracting the previous position from the new relaxed position.

3.1.2 Pseudo code clarification

The pseudo code in the following sections uses a combination of mathematical notation and programming conventions. This is to allow a more implementation oriented description of the algorithms used, since some techniques used are dependant on those details.

The concepts in need of clarification are the following:

1: $index$ $\triangleright$ Integer
2: $timeStep$ $\triangleright$ Scalar
3: **pos** $\triangleright$ Vector
4: **pos**.x $\triangleright$ X component of vector
5: Particle p $\triangleright$ p is of type Particle
6: $p.$**pos**.x $\triangleright$ Accessing members
7: DoStuff($timeStep, p$) $\triangleright$ Function call

3.1.3 Algorithm overview

This section aims to describe the algorithm used in such a way that performing a similar implementation should be clear. The pseudo code will therefore be more detailed than usual. The algorithm is based on [CBP05] and we will there not discuss the concepts in depth, and ask the reader to consult their paper for a deeper discussion.

Algorithm 1 Simulation step

1: **function** UPDATE(*timeStep*)
2:      ApplyExternalForces(*timeStep*)
3:      ApplyViscosity(*timeStep*)
4:      AdvanceParticles(*timeStep*)
5:      UpdateNeighbors()
6:      DoubleDensityRelaxation(*timeStep*)
7:      ResolveCollisions()
8:      UpdateVelocity(*timeStep*)

Algorithm 1 shows the algorithm at a high level. This function is called once per simulation step with the time elapsed since last step. To increase the numerical accuracy the time step is fixed at $1/60$, since that is the target frame rate. This avoids a few issues, such as instabilities caused by large time steps and nondeterministic behaviour.

Data structures and context

The algorithms described in section 3.1 are part of a particle manager component. Analogous to object oriented programming they can be seen as methods to the particle manager class. This means they have access to a collection of parameters and other components, available in the particle manager component.

The type 'Particle' contains the following data:

1: **pos**   ▷ The particle's position
2: **pos**$_{prev}$ ▷ The particle's previous position
3: **vel**   ▷ The particle's velocity
4: *index*   ▷ An index value used by the grid component

This is a list the parameters accessible by the algorithms:

1: *radius*   ▷ Maximum distance particles effect each other.
2: *collisionRadius*   ▷ The distance from a wall that counts as a collision.
3: p_0 ▷ Rest density
4: σ ▷ The viscosity's linear dependence on the velocity
5: β ▷ The viscosity's quadratic dependence on the velocity
6: k ▷ Stiffness used in DoubleDensityRelaxation
7: k_{near} ▷ Near-stiffness used in DoubleDensityRelaxation
8: **gravity**   ▷ The global gravity acceleration

This is a list the data accessible by the algorithms:

1: List<Particle> *particles*	
2: List<List<Particle>> *neighbors*	
3: Grid *grid*	▷ See section 3.2
4: DistanceField *distanceField*	▷ See section 3.4

1. List<Particle> *particles*

   This is the main list of particles. All algorithms in section 3.1.3 are based around iterating through this list. Since this is such a common operation all 'for each' loops over all particles in this list are simply listed as: **for each** Particle p

   This list is created and initialized in the beginning of the simulation.

2. List<List<Particle>> *neighbors*

   This is a collection of each particle's list of neighbors. These lists are generated in the function 'UpdateNeighbors' and used when a neighbor lookup is required. Only includes particles which are within each others radii. A neighbor list for particle p is accessed as: $neighbors_p$

3. Grid *grid*

   This component hashes the particles based on their position to a grid, to allow faster neighbor search. Particles register their movement to this component and the neighbor lists gets their data from this component. More information about this component is available in section 3.2.

4. DistanceField *distanceField*

   This component manages the world representation. The world is represented by a distance field, where each data point contains the distance to the closest edge and the normal away from that edge. The distance field is discussed in more depth in section 3.4.

ApplyExternalForces

Algorithm 2 Applying external forces

1: **function** APPLYEXTERNALFORCES(*timeStep*)
2:     **for each** Particle p
3:         $p.\textbf{vel} \leftarrow p.\textbf{vel} + \textbf{gravity}$
4:         $p.\textbf{vel} \leftarrow p.\textbf{vel} + \text{ForcesFromTouchInput}(p)$

This step applies external forces to all particles. There are two sources of external forces, both can be manipulated by the user. Section 3.5 describes the user interaction in more detail.

3. Gravity

   The gravity is the same for all particles.

4. ForcesFromTouchInput(p)

   This is an abstraction of the direct user interaction. These forces are different for each particle and are dependant on which type of behaviour is required by the touch controls. One possible case can be that each particle within a certain distance from a touch point receives a force towards that point.

ApplyViscosity

Algorithm 3 Applying viscosity

1: **function** APPLYVISCOSITY($timeStep$)
2:     **for each** Particle p
3:         **for each** Particle $n \in neighbors_p$
4:             $\mathbf{v}_{p,n} \leftarrow n.\mathbf{pos} - p.\mathbf{pos}$
5:             $vel_{inward} \leftarrow (p.\mathbf{vel} - n.\mathbf{vel}) \cdot \mathbf{v}_{p,n}$
6:             **if** $vel_{inward} > 0$
7:                 $length \leftarrow | \mathbf{v}_{p,n}|$
8:                 $vel_{inward} \leftarrow vel_{inward}/length$
9:                 $\mathbf{v}_{p,n} \leftarrow v_{p,n}/length$
10:                $q \leftarrow length/radius$
11:                $\mathbf{I} \leftarrow 0.5 * timeStep * (1 - q) * (\sigma * vel_{inward} + \beta * vel_{inward}^2) * \mathbf{v}_{p,n}$
12:                $p.\mathbf{vel} \leftarrow p.\mathbf{vel} - \mathbf{I}$

This step is heavily based on the viscosity step in the algorithm by [CBP05]. Viscosity has the effect of smoothing the velocities of the particles. It is an impulse applied radially between neighboring particles. In relation to the real world, water is a fluid with low viscosity, while scrap has higher viscosity and clay would have even higher viscosity.

For non-viscous fluids, which is the target behavior of this simulation, viscosity is used to handle collisions between two particles by decreasing the inwards velocity between them. The impulse is dependant on the factor $(1 - q)$ which increases with proximity and the factor $(\sigma * vel_{inward} + \beta * vel_{inward}^2)$, where σ and β stands for the viscosity's linear and quadratic dependence on the velocity.

[CBP05] recommends that only β should be set to a non-zero value for less viscous fluids. This is because the quadratic term only removes high inwards velocities, but leaves the interesting features of the particle's velocity unchanged.

Note that $\mathbf{v}_{p,n}$ is not normalized at line (5); the normalization of both the vector and the inwards velocity takes place at the lines (7 - 9).

AdvanceParticles

Algorithm 4 Advancing particles to predicted position

1: **function** ADVANCEPARTICLES(*timeStep*)
2:     **for each** Particle p
3:         $p.\mathbf{pos}_{prev} \leftarrow p.\mathbf{pos}$
4:         $p.\mathbf{pos} \leftarrow timeStep * p.\mathbf{vel}$
5:         $grid.\text{MoveParticle}(p)$ ▷ Section 3.2

This step advances the particles to their new positions, based on their velocities and stores their previous positions. At line (5) the particle notifies the grid about its movement, which updates the particles' position in the grid.

UpdateNeighbors

Algorithm 5 Update neighbor lists

1: **function** UPDATENEIGHBORS( )
2:     **for each** Particle p
3:         $neighbors_p.\text{clear}()$
4:         **for each** particle $n \in grid.\text{PossibleNeighbors}(p)$ ▷ Does not include p
5:             **if** $|p.\mathbf{pos} - n.\mathbf{pos}| < radius$
6:                 $neighbors_p.\text{Add}(n)$

This step updates the neighbor lists with the correct neighbor information. The function 'PossibleNeighbors' returns the particles in the 9 grid cells closest to p, excluding p itself. If p and the possible neighbor n is within each other's radii, n is added to $neighbors_p$.

DoubleDensityRelaxation

Algorithm 6 Double Density Relaxation

1: **function** DoubleDensityRelaxation($timeStep$)
2:     **for each** Particle p
3:         $p \leftarrow 0$
4:         $p_{near} \leftarrow 0$
5:         **for each** Particle $n \in neighbors_p$
6:             $temp_n \leftarrow |p.pos - n.pos|$
7:             $q \leftarrow 1.0 - temp_n/radius$
8:             $p \leftarrow p + q^2$
9:             $p_{near} \leftarrow p_{near} + q^3$
10:        $P \leftarrow k * (p - p_0)$
11:        $P_{near} \leftarrow k_{near} * p_{near}$
12:        $delta \leftarrow 0$
13:        **for each** Particle $n \in neighbors_p$
14:            $q \leftarrow 1.0 - temp_n/radius$ ▷ $temp_n$ from line 6
15:            $v_{p,n} \leftarrow (p.pos - n.pos)/temp_n$
16:            $D \leftarrow 0.5 * timeStep^2 * (P * q + P_{near} * q^2) * v_{p,n}$
17:            $n.pos \leftarrow n.pos + D$
18:            $delta \leftarrow delta - D$
19:        $p.pos \leftarrow p.pos + D$

This step is implemented using the double density relaxation step described in [CBP05]. It is a simplified and extended formulation of the SPH paradigm and aims to ensure volume conservation. Two different measures of the particles' neighbor density is used to decide the impulses between the particles.

The first one is the density p and the second is the near density p_{near}:

$$p = \sum_{n \in neighbors_p} (1 - |p.pos - n.pos|/radius)^2$$

$$p_{near} = \sum_{n \in neighbors_p} (1 - |p.pos - n.pos|/radius)^3$$

Clavet *et al.* writes that these forms of densities are not a true physical property, but rather a number quantifying how the particle relates to its neighbors. They also mention that they tested various other kernel shapes, but arrived to the conclusion that these gave the best results. These densities are calculated at the lines (3- 9) and are then used to calculate a pseudo-pressure P and a near pressure P_{near}.

Pseudo-pressure is a term defined by [CBP05] as $P = k(p - p_0)$ where p is the density calculated above, k is a stiffness constant and p_0 is the rest density. This will give particles with a density lower than rest density a negative pressure and a particle with a density greater than rest density a positive pressure. This pressure will be used at line (16) to calculate if the particle should pull in or push away its neighbors.

The near pressure is defined as $P_{near} = k_{near}p_{near}$, where k_{near} is stiffness constant for the near pressure and p_{near} is the near density calculated above. The near pressure was introduced by [CBP05] to solve a clustering problem. Without the near pressure, the particles would reach rest density by strongly pulling a small number of neighbors close enough, which made the fluid separate into independent clusters. With the introduction of near pressure particles will kept apart to avoid small clusters. Note that the near pressure kernel function is cubic instead of square, making it less influential on neighbors far apart.

ResolveCollisions

Algorithm 7 Resolving collisions

1: **function** RESOLVECOLLISIONS($timeStep$)
2:     **for each** Particle p
3:         $index \leftarrow distanceField$.GetIndex($p.pos$)
4:       **if** $index \neq -1$
5:           $distance \leftarrow distanceField$.GetDistance($index$)
6:         **if** $distance > -collisionRadius$
7:             $v_{p,n} \leftarrow (p.pos - n.pos)/temp_n$
8:             $normal \leftarrow distanceField$.GetNormal($index$)
9:             $tangent \leftarrow PerpendicularCCW(normal)$
10:            $tangent \leftarrow timeStep * friction * (v_{p,n} \cdot tangent) * tangent$
11:            $p.pos \leftarrow p.pos - tangent$
12:            $p.pos \leftarrow p.pos - collisionSoftness * (dist + r) * normal$

This step resolves the collisions between the particles and the world. The distance field is queried for the distance from the particle's position to the closest edge. If that distance is less than the collision radius, move the particle away from the edge by an value dependant on the penetration depth. Also particles in contact with the edge will receive a small friction force in the tangential direction of the edge.

Since the distance field have limited resolution and have a noticeable amount of error in some cases, correcting strictly after the distance field will give jerky results. Therefore a *collisionSoftness* factor is introduced to let the correction take place over multiple simulation steps. This will make the particles slightly penetrate the wall a couple of frames, but will result in a smoother movement. In our implementation we use values in the range of $0.4 - 0.8$, depending on world properties.

UpdateVelocity

Algorithm 8 Updating velocities

1: **function** UPDATEVELOCITY(*timeStep*)
2:    **for each** Particle p
3:       $p.vel \leftarrow (p.pos - p.pos_{prev})/timeStep$

This step updates the velocities to be used in the beginning of next simulation step. The velocity is defined as the change in position during this step, from the previous position to the predicted, and then relaxed, new position.

3.2   Neighbor search

Since all particle interaction will be between pairs of neighbors, where all values are equal to zero outside of the particle's kernel radius, there is no need for each particle to consider all other particles potential neighbors. Therefore, a spatial hashing grid were implemented.

We make the assumption that the radius of the particles are constant and that the world is static with fixed boundaries. These assumptions can be used to decrease the computational cost, in comparison to using a sparse grid which can adjust for varying radius and world size.

The grid consists of $N * M$ square cells with the width and height of *radius*. Each cell contains a list of particles belonging to the cell. When a particle has moved enough to change cell, it is removed from the old cell and inserted in the new cell.

When a particle requests its possible neighbors, the grid find which cell it belongs to and returns the particles in that cell and the 8 cells around it. All neighbors are guaranteed to lie within those 9 cells, due to the cell edge size of *radius*, but it is not guaranteed that all those particles are actual neighbors. Therefore, the neighbors are calculated once per simulation step and stored into lists. This is to avoid the overhead of fetching the potential neighbors from the 9 cells and to a distance test every time a list of neighbors are needed. This calculation is described by the function UpdateNeighbors in section 3.1.3, algorithm 5.

3.3   Threading

A simple approach was used to make the algorithm usable for multiple threads. Most of the steps in simulation step algorithm (Algorithm 1) consists of an outer loop iterating over all particles and performing some kind of calculation. This outer loop is a simple for loop with a fixed order and there are no rearrangements or other special operations during any step. This made the following adjustment possible.

Each thread is given an integer id *thread_id*, from 0 up to the maximum number of threads - 1 *thread_count*. Each loop is then changed to start at *thread_id* instead of 0 and increment the index with *thread_count* instead of 1.

Algorithm 9 Update loops with and without threading

1: **for** (int $i = 0$; $i < particle\_count$; $++i$)
2: **for** (int $i = thread\_id$; $i < particle\_count$; $i\ += thread\_count$)

This approach does not seem to very cache efficient, in comparison with giving each thread a range of indices to iterate over. However, the neighbor access can be seen as completely random due to the nature of the particle movement, most likely changing enough of the cache to make the next particle miss in any case.

A test was made to see if changing the access order by grid cell would increase performance, since a member of the same cell is more likely to already be in the cache. However, the only difference were a very small increase of computational time, most likely caused by the overhead from accessing cells. Since there were no visible benefits from one method over the next, the simplest method was chosen.

3.4 World collision

The world is defined as a distance field where each data point contains the distance to the closest edge an the normal away from that edge. This gives a fast and robust way of managing collisions in the world. Each particle only has to sample the data field once to find a collision, get the collision depth and the normal along which it should be displaced.

The downside of this method is that it is memory consuming and slow to generate. The memory consumption is not a problem on any modern handheld device and the slow generation was solved by generating most of the distance fields offline. The distances still have to be adjusted to the world size during runtime, but this only take a fraction of the whole generation time.

The world is generated by using a black and white image, where white represents a solid material and black represents air. The image is used as input in the following algorithm and will generate a manhattan distance field. An example of the result is shown in figure 3.1.

Algorithm 10 Generate distance field
___
1: //Upwards pass
2: $s \leftarrow$ Set of all edges, white cells with black neighbors.
3: $v \leftarrow 0$
4: **while** s not empty
5:      Set all cells in s to v
6:      $s \leftarrow$ All white neighbors to s which has not been given a value
7:      $v \leftarrow v + 1$
8: //Downwards pass
9: $s \leftarrow$ Set of all edges, white cells with black neighbors.
10: $v \leftarrow 0$
11: **while** s not empty
12:      Set all cells in s to v
13:      $s \leftarrow$ All black neighbors to s which has not been given a value
14:      $v \leftarrow v - 1$
___

Figure 3.1: An example of the first step of the distance field generation. The numbers represent the distance from the edge, where positive numbers are inside the wall and negative numbers are outside.

To calculate the normals a kernel of uneven size is used. A larger kernel will result in a smoother but less detailed normal field. In our implementation a kernel size of 9 was used, which requires sampling of 81 data points to generate each normal. The normal is calculated by:

$\hat{\mathbf{n}}_i = normalize(\sum_{j \in N(i)} d_j \hat{\mathbf{u}}_{j,i})$, where $N(i)$ is all data points of the distance field in the kernel, d_j is the distance value at j and $\hat{\mathbf{u}}_{j,i}$ is a normalized vector from the data point j to i.

These normals are then used to make an approximate transformation of the manhattan distances[1] to euclidean distances[2], to make some diagonal cases more accurate.

3.5   User interaction

The gravity is controlled by the device accelerometer. This allows the user to tilt the device back and forth to create waves and let the fluid flow through the world in any direction.

In addition to controlling the gravity, the user is able to pick up chunks of fluid using the touch controls or the mouse. The pick up interaction consists of three forces acting on the nearby particles: an attraction force with pulls the particles towards the touch point, a small friction force to make the picked up fluid come to rest earlier and a force corresponding to the movement of the touch point. Figure 3.2 shows the simulation a short while after a user picked up two chunks of fluid.
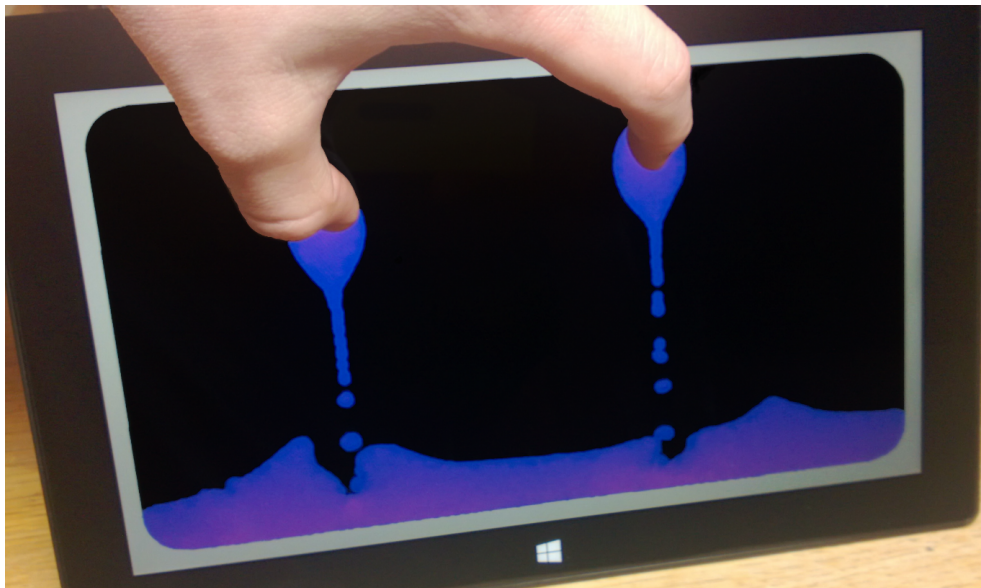

Figure 3.2:   An user picking up two chunks of the fluid. The force used to pick up the fluid quickly weakens with distance and the force keeping most of the fluid in place is the force applied by the double density relaxation(Section 3.1.3), which in this case behaves like surface tension. The downward streams of particles quickly form into droplets.

---

[1]Manhattan distance: The distance between to points using only a strictly vertical and horizontal path. $distance(\mathbf{p}_1, \mathbf{p}_2) = |\mathbf{p}_1.x - \mathbf{p}_2.x| + |\mathbf{p}_1.y - \mathbf{p}_2.y|$

[2]Euclidean distance: The distance between two points using a straight path. The distance is given by pythagoras theorem: $distance(\mathbf{p}_1, \mathbf{p}_2) = \sqrt{\mathbf{p}_1 \cdot \mathbf{p}_2}$
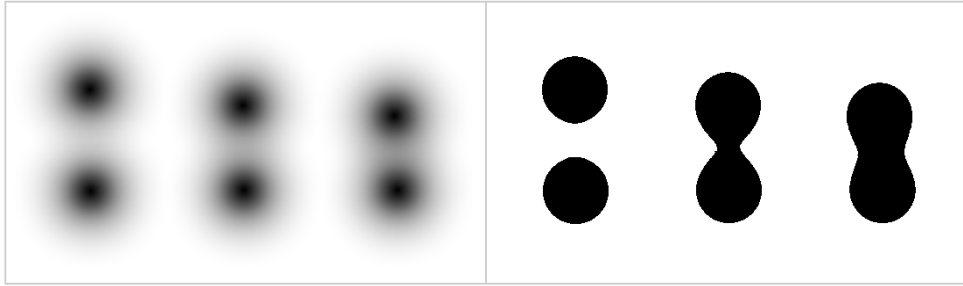
Figure 3.3: A visualization of the fluid rendering process. The alpha value is used as a threshold value in a second rendering pass to get a smooth surface. This is natively supported on most graphics hardware and therefore a very cheap effect. Left: the (inverted for clearer image) alpha channel in the first pass. Right: The alpha after threshold comparison in the second pass.

3.6 Rendering

This implementation does not focus on the aspect of rendering fluids, but a short description of the techniques used will be given.

The fluid rendering consists of two passes. The first pass render each particle to an off-screen buffer with additive blending. The alpha[3] values in the sprite are based on the distance from the center of the sprite, where the alpha is 1.0 in the center and 0.0 at the edges. The left part of figure 3.3 shows this. In this implementation, the dimensions of the off-screen buffer is scaled down by a factor of 4, compared to the screen resolution.

In the second pass, the background is first drawn, then the off-screen buffer is sampled with a linear sampler to produce the final image on the screen. The alpha value is read and compared to a threshold cutoff value. If the alpha is above the threshold it renders the fluid color with alpha 1.0, if the alpha is below the threshold the pixel is ignored, leaving the background unchanged. This produces a smooth fluid-like surface as shown in the right side of figure 3.3.

The color of a particle is dependant on the amount of pressure there is at the particle. A simple palette is used where the colors goes from: low pressure → high pressure, blue → purple → white. Figure 3.2 shows this effect slightly; the fluid is blue at the top of the surface and more purple at the bottom.

As an option for the user, some particles can be turned white to visualize flow better. This will simply make 10% of the particles ignore their pressure color value and always use white. This will, however, only visualize flow when the simulation is running and not in captured images.

---

[3]Alpha channel: A common channel in images that is being rendered, along with red, green and blue. Alpha is most commonly used for transparency.

3.7   Testing method

Fluid behavior

The fluid behavior tests are mostly performed and verified in a subjective and graphical manner. If the exhibited behaviour looks as expected, it is good enough for this implementation's target use case.

Performance

The performance test will consists of two different types of tests.

The first type of test will be a static test where all the particles has come to a rest in the world. The lack of motion will make this test more suitable for showing differences between different implementations, since the input will be very similar and stable.

The second type of test will be a dynamic test where the particles are in motion. This type of test is more suitable for measure real usage performance, where there are occasional performance spikes.

Chapter 4

Results

This chapter will present the results from the implementation. Section 4.1 will describe the goals with this implementation in detail, followed by section 3.7 which describes the testing method. The tests are split into two sections: section 4.2 contains the fluid behaviour tests and section 4.3the performance tests.

At the end of the section the visualization of the fluid is also discussed.

4.1   Goal

Fluid behavior

As mentioned in the introduction, the aim of this implementation is to provide an interactive learning platform for people interested in basic fluid dynamics. Therefore, the local numerical accuracy of the simulation is not critical as the results will only be used for visualization. However, it is important that the fundamental fluid behavior is plausible reproduced in the simulation. To make the requirements fit the scale of this project a couple of scenarios were selected as a goal: waves, splashes, simple pressure equalization and flow through narrow passages. These scenarios should provide a good start for an interested student.

More advanced scenarios were out of scope for this project, such as: simulating gases as particles, allowing more advanced pressure simulation, bubbles and drag; elasticity, allowing for simulation of clay or other non-Newtonian fluids; or fluids with different properties, allowing a simulation of mixing oil and water.

Performance

The performance requirements are quite strict: the maximum time per frame is $16.7ms$ since the screen on the target device updates at 60 Hz. This includes all work, such as: reading input, updating the fluid simulation and render the result to the screen. Occasional bad spots will occur and are to a minor extent accepted. The goal of the performance test is to find a suitable number of particles to use in the simulation and to measure the effects of threading.

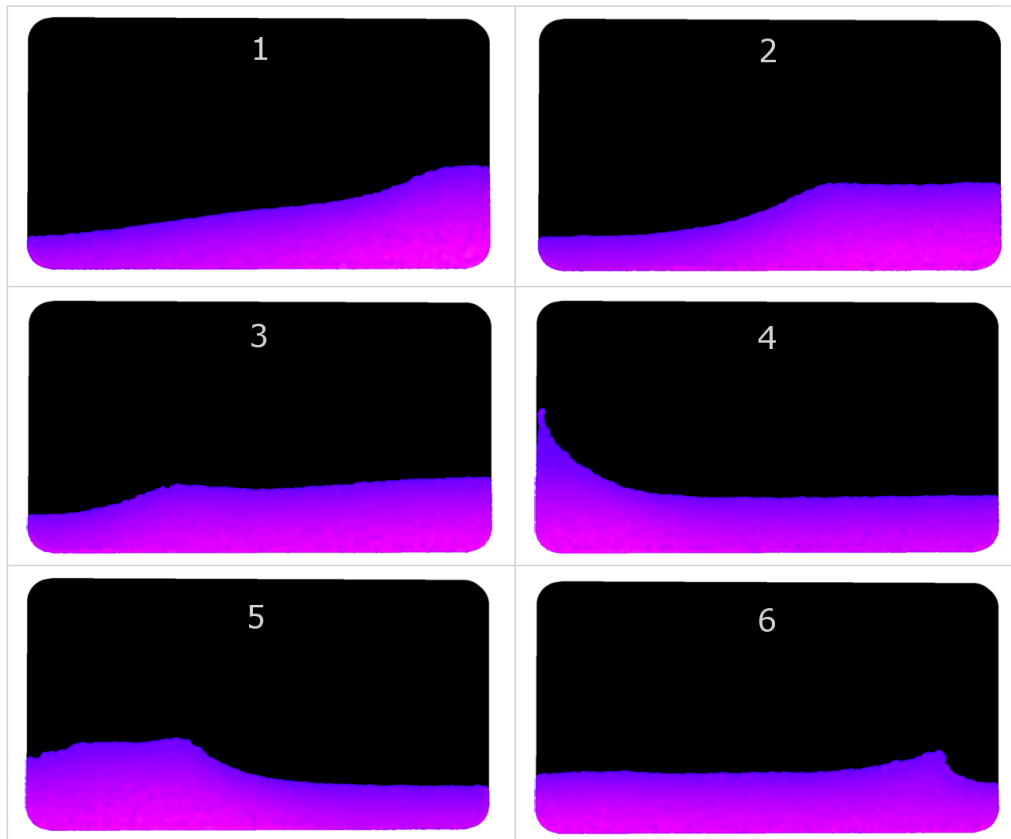## 4.2 Fluid behavior

4.2.1 Waves

Figure 4.1: A wave propagating from the right to the left and back again over a time frame of approximately three seconds.

The simulation is capable of handling waves. If there is a strong wave without interruptions it will propagate back and forth across the screen, slowly losing momentum. This behaviour is shown in figure 4.1.

Creating an acceptable breaking wave was harder, but the result is showed in figure 4.2. It required parameters changes which reduced the surface tension effect enough to make medium sized droplets break apart. Dropping a droplet as in figure 4.2.3 would not be possible since the droplet would break apart into multiple small droplets before hitting the surface.
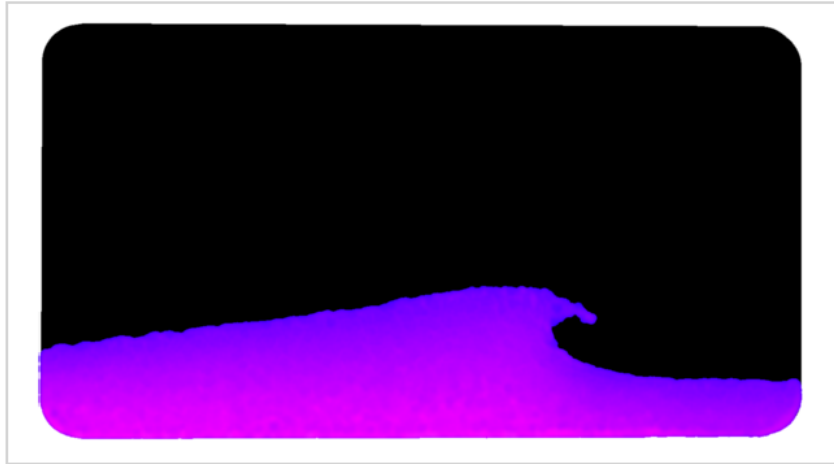
Figure 4.2: A breaking wave.

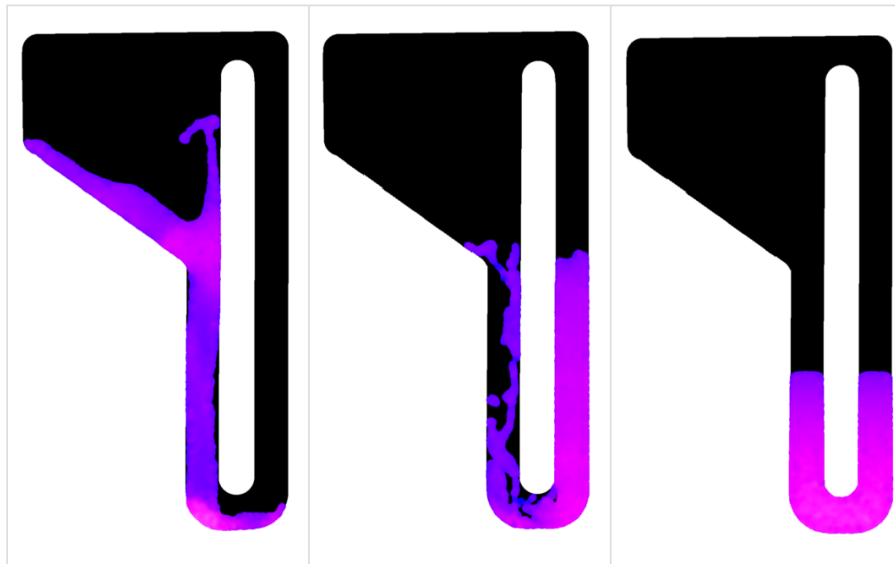### 4.2.2 Pressure equalization in an U-tube

Figure 4.3: A sequence leading to pressure equalization in an U-pipe.

The simulation is able to handle pressure equalization remarkably well. It performs well even in more complicated cases than the one showed in figure 4.3. One downside

is that it can take several seconds before the particle system comes to rest at the equalized state.
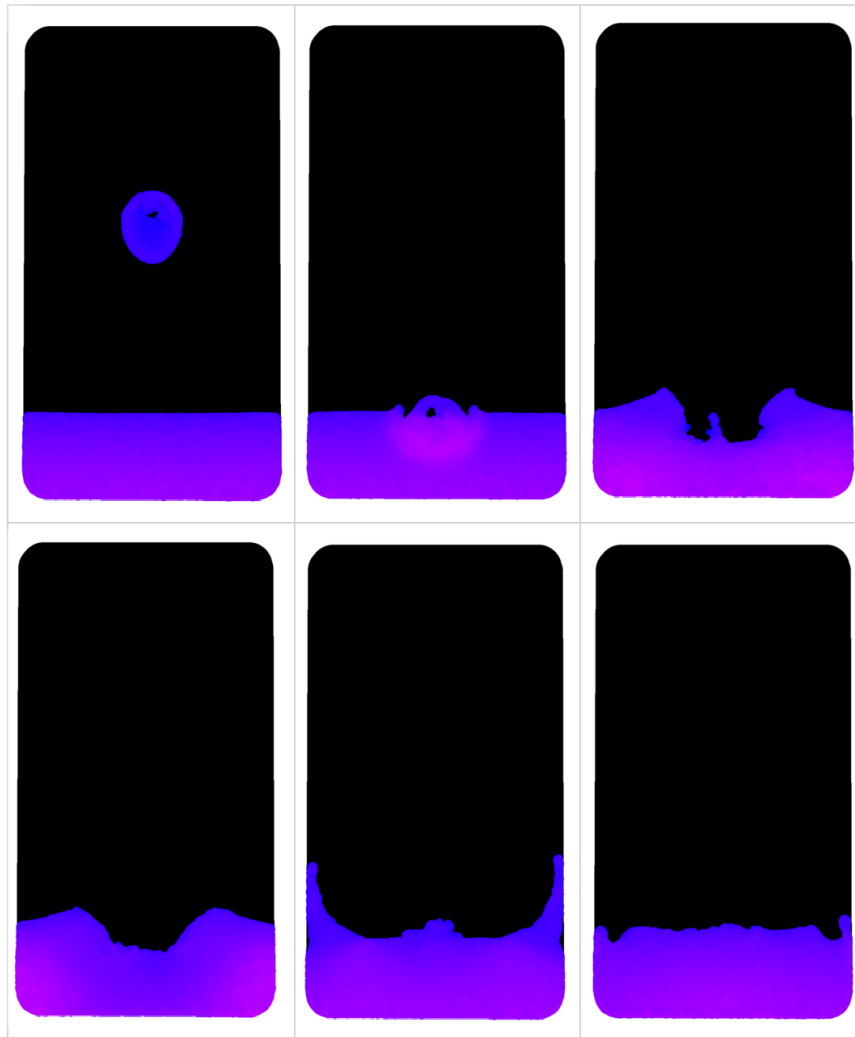
4.2.3 Splash

Figure 4.4: Dropping a droplet into a body of fluid. Notice the changes in pressure, indicated by the purple color, in the frames after the collision, especially frame 2 and 4.

Splashes were hard to get right in this simulation. The forces keeping the particles together are working against the splash behaviour, but if those forces are decreased the droplet, as shown in the first image of figure 4.4, will not form naturally.

An interesting effect to notice in figure 4.4 is how the pressure behaves after the collision. Purple represents high pressure and a shockwave can be seen (2), bouncing on the bottom of the screen (3), out to the two sides of the container (4), making the fluid rise (5) at the sides and finally come to rest (6).
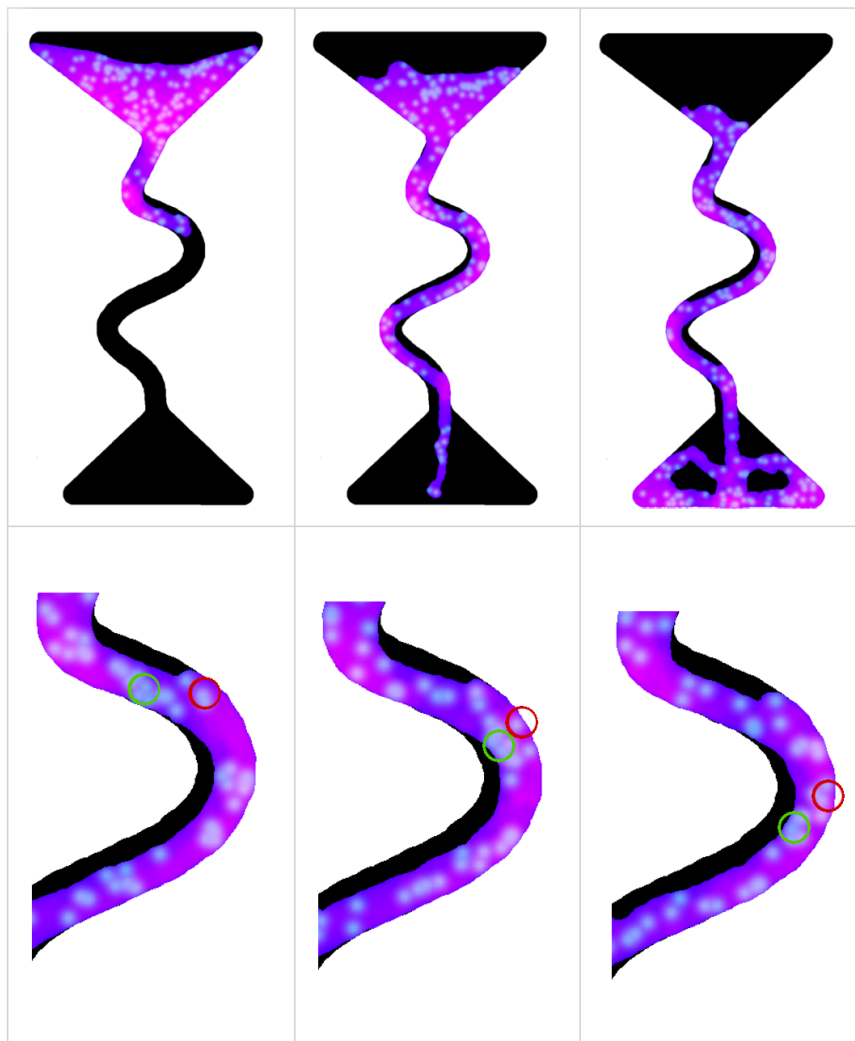
4.2.4 Flow

Figure 4.5:   Top: overview of the flow test. Bottom: zoomed in, showing the difference in velocity for particles adjacent to walls (red) and particles not adjacent to walls (green). The friction force from the wall slows down collection of particles tracked by the red circle in comparison to the ones tracked by the green circle.

The fluid flow in the simulation turned out to work better than expected. Narrow openings limit the flow and pressure in the expected way. The small friction force of the wall collisions also makes the fluid flow slower closer to walls.

4.3 Performance

This section will discuss performance related subjects, such as the performance impact of threading and which number of particles is suitable for a fluent and interactive experience. Two different types of performance tests are used, one static and one dynamic. The static test is a test were the fluid has come to a rest and is not making any significant movement and the dynamic test is a test where the fluid is constantly moving.

The testing device was a Windows RT tablet of the model: Microsoft Surface RT 64 GB. It features a quad-core ARM based CPU and 2 GB of RAM.

4.3.1 Impact from threading

The following table lists the frame times during multiple simple tests made. The times are an approximation of the average of the last 3 seconds frame times. In the tests, 3000 particles was used with four different set of parameters, per type of testing.

	1 thread	4 threads	improvement factor
Static test 1	36 ms	11 ms	3,3
Static test 2	34 ms	10 ms	3,4
Static test 3	24 ms	8,1 ms	3,0
Static test 4	50 ms	15 ms	3,3
Dynamic test 1	32 ms	11 ms	2,9
Dynamic test 2	32 ms	10 ms	3,2
Dynamic test 3	23 ms	8,5 ms	2,7
Dynamic test 4	42 ms	14 ms	3,0

Table 4.1: A comparison between the average frame time with and without using threading. This shows that using our technique for threading, the performance was increased by approximately a factor of 3.

As shown in table 4.1, running our implementation on four hardware threads, approximately yields a speed up by a factor of 3. These values seem to be correct even outside of the testing environment, where user interaction is part of the simulation.

4.3.2 Particle count

The following test were made to find a reasonable particle count for use in an average user case. The two testing scenarios were used with different particle count, as shown

in table 4.2

count	static	dynamic
2000	6,9 ms	6,3 ms
2500	8,7 ms	8,3 ms
3000	11 ms	10 ms
3500	13,5 ms	13 ms
4000	16 ms	15 ms
4500	18,5 ms	17 ms
5000	21 ms	21 ms

Table 4.2: A comparison between the average frame time in two different scenarios, with different amount of particles.

An interesting observation from this result is that the dynamic test receives slightly better results. Intuitionally, this can be unexpected results since moving particles have to change their position in the grid more often. However, a moving fluid is more likely to spread better across the available space, reducing the average number of neighbor pairs which in turn reduces the number of computations required for each particle.

It is important to note that these time figures are measurements of only the fluid simulation step; other parts of the implementation is not included in this time, such as input management and rendering. The rendering used in our implementation is quite simple and not very time consuming, in an average case the rendering uses 1 - 2 ms per frame to set up all draw calls and other information required by the GPU to render the frame.

The main loop is set up in the order: render, update, present; where present waits for vertical sync and flips the screen back buffers. This allows the GPU to work on rendering during the simulation step update. Since this implementation is heavily CPU bound, by the time the simulation step is complete, the GPU has a frame ready to be presented.

4.4 Visualization

To give an interesting user experience it is important to visualize the fluid simulation in an interesting way. In this implementation, the rendering is kept as simple as possible, while still creating convincing results. The only effect used is the smoothing of the edges, as described in section sec:sec:Rendering, otherwise the rendering is basically just colored balls.

There are two aspects which the visualization focuses on: pressure changes and flow. Pressure is visualized by giving each particle a color from a palette, based on their pressure in the simulation step DoubleDensityRelaxation in section 3.1.3. This makes the user able to visualize how the pressure changes during different events;

figure 4.5 shows increased pressure where the fluid are pressed against a wall and the second image of figure 4.4 shows the beginning of a shockwave when the droplet hit the surface.

Flow are visualized by simply highlighting 10% of the particles. When the particles are colored by pressure, it can be hard to follow the flow of particles since the pressure changes rapidly. By highlighting a number of particles it gives the user some reference points to track, making the flow easier to visualize. Figure 4.5 shows how this highlighting looks, but an actual moving simulation is required to fully appreciate the effect.

Chapter 5

Conclusion

In conclusion, this implementation works reasonably well on the target device. The implementation is able to recreate the fluid behaviours we aimed to implement, such as waves, pressure equalization and interesting flow behaviour. Adjusting the parameters to get the desired result turned out to be difficult and we did not succeed in creating a breaking wave in an environment, without the side effect of making the surface tension too low.

The computational performance of the simulation allows the simulation to handle about 3000 particles. This is more than enough to simulate the scenarios targeted by this report.

5.1 Future work

This implementation is mostly meant as an introductory work to fluid dynamics and can therefore be seen as relatively simple and naïve. There are many aspects of which this simulation could be improved.

The targeted fluid behavior can be extended by using one of the many variations available in literature, such as the full algorithm by Clavet [CBP05] to allow viscoelastic behavior.

Many of the components used in the implementation are focused on simplicity instead of performance, making them not optimal and suitable for change. The neighbor search can be implemented in a number of different ways. A solution described by [Gre08] seems like a good possible choice.

With the recent advancement of mobile processors, the GPU capabilities are starting to become powerful. In a few years, the GPU based techniques such as the ones described in [WLL04] and [HKK07] might be more suitable for mobile fluid simulation.

Also, this implementation did not use the ARM SIMD[1] instruction set, NEON, to speed up calculations. By utilizing this the calculations would most likely have been considerably sped up.

---

[1]SIMD: Single Instruction, Multiple Data. Performs the same operation on multiple data elements, at the cost of one operation.

Bibliography

[CBP05] Simon Clavet, Philippe Beaudoin, and Pierre Poulin. Particle-based viscoelastic fluid simulation. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 219–228. ACM, 2005.

[GM77] Robert A Gingold and Joseph J Monaghan. Smoothed particle hydrodynamics-theory and application to non-spherical stars. *Monthly notices of the royal astronomical society*, 181:375–389, 1977.

[Gre08] Simin Green. Particle-based Fluid Simunlation. Presentation by NVIDIA Corporation, 2008.

[HKK07] Takahiro Harada, Seiichi Koshizuka, and Yoichiro Kawaguchi. Smoothed particle hydrodynamics on gpus. In *Computer Graphics International*, pages 63–70, 2007.

[Luc77] Leon B Lucy. A numerical approach to the testing of the fission hypothesis. *The astronomical journal*, 82:1013–1024, 1977.

[WLL04] Enhua Wu, Youquan Liu, and Xuehui Liu. An improved study of real-time fluid simulation on gpu. *Computer Animation and Virtual Worlds*, 15(3-4):139–146, 2004.