

Institutionen för systemteknik

Department of Electrical Engineering

Examensarbete

Water simulation for cell based sandbox games

Examensarbete utfört i Datateknik
vid Tekniska högskolan vid Linköpings universitet
av

Christian Lundell

LiTH-ISY-EX--14/4761--SE

Linköping 2014



Linköpings universitet
TEKNISKA HÖGSKOLAN

Water simulation for cell based sandbox games

Examensarbete utfört i Datateknik
vid Tekniska högskolan vid Linköpings universitet
av


Christian Lundell

LiTH-ISY-EX--14/4761--SE

Handledare: **Fredrik Viksten**
ISY, Linköpings universitet

Examinator: **Ingemar Ragnemalm**
ISY, Linköpings universitet

Linköping, 7 juni 2014

	Avdelning, Institution Division, Department Information Coding Department of Electrical Engineering SE-581 83 Linköping	Datum Date 2014-06-07				
Språk Language <input type="checkbox"/> Svenska/Swedish <input checked="" type="checkbox"/> Engelska/English <input type="checkbox"/> _____	Rapporttyp Report category <input type="checkbox"/> Licentiatavhandling <input checked="" type="checkbox"/> Examensarbete <input type="checkbox"/> C-uppsats <input type="checkbox"/> D-uppsats <input type="checkbox"/> Övrig rapport <input type="checkbox"/> _____	ISBN _____ ISRN LiTH-ISY-EX--14/4761--SE Serietitel och serienummer ISSN Title of series, numbering _____				
URL för elektronisk version http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-XXXXX						
<table border="0"> <tr> <td style="vertical-align: top;">Titel Title</td> <td>Vattensimulering för cellbaserade sandboxspel Water simulation for cell based sandbox games</td> </tr> <tr> <td style="vertical-align: top;">Författare Author</td> <td>Christian Lundell</td> </tr> </table>			Titel Title	Vattensimulering för cellbaserade sandboxspel Water simulation for cell based sandbox games	Författare Author	Christian Lundell
Titel Title	Vattensimulering för cellbaserade sandboxspel Water simulation for cell based sandbox games					
Författare Author	Christian Lundell					
Sammanfattning Abstract <p>This thesis work presents a new algorithm for simulating fluid based on the Navier-Stokes equations. The algorithm is designed for cell based sandbox games where interactivity and performance are the main priorities. The algorithm enforces mass conservation conservatively instead of enforcing a divergence free velocity field. A global scale pressure model that simulates hydrostatic pressure is used where the pressure propagates between neighboring cells. A prefix sum algorithm is used to only compute work areas that contain fluid.</p>						
Nyckelord Keywords Fluid Simulation, Navier-Stokes, Computer Games, GPGPU						

Abstract

This thesis work presents a new algorithm for simulating fluid based on the Navier-Stokes equations. The algorithm is designed for cell based sandbox games where interactivity and performance are the main priorities. The algorithm enforces mass conservation conservatively instead of enforcing a divergence free velocity field. A global scale pressure model that simulates hydrostatic pressure is used where the pressure propagates between neighboring cells. A prefix sum algorithm is used to only compute work areas that contain fluid.

Contents

1	Introduction	1
1.1	Related work	2
2	Theory	5
2.1	Navier-Stokes	5
2.2	Incompressibility	6
2.3	Velocity	8
2.4	Pressure	9
2.4.1	Hydrostatic pressure	10
2.5	Advection	10
2.6	OpenGL	11
2.6.1	Work group	12
2.6.2	Memory	12
2.6.3	Memory alignment	13
3	Implementation	15
3.1	Parallelization	15
3.2	Fluid cell data structure	16
3.3	Pseudo Code	16
3.4	Resting work areas	18
4	Result	21
4.1	Performance	21
4.2	Behavior	22
4.3	Overall Evaluation	26
4.3.1	Is the algorithm faster than real-time performance?	26
4.3.2	Is the algorithm giving consistent performance with no lag spikes?	28
4.3.3	Is the algorithm cell based?	28
4.3.4	Is the algorithm mass conserving?	28
4.3.5	Is the algorithm calculating pressure on a macroscopic scale?	28
4.3.6	Is the algorithm running on the GPU?	28

5	Summary	29
5.1	Future work	29
	Bibliography	31

1

Introduction

Cell based sandbox games is a genre of computer games where the player can interact with the environment and shape it by mining and placing blocks. The player can build structures, machines, and traps and is generally free to be creative and find his own solution to challenges. The cells in these games are often rectangular blocks in 3D. Games such as Dwarf Fortress, Minecraft and Terraria have made the genre very prominent. The goal of this work is to develop a fluid simulation algorithm that the player can interact with and that runs in realtime.

The goal of this thesis work is to present a fluid simulation algorithm that is:

1. Faster than real-time performance
2. Giving consistent performance with no lag spikes
3. Cell based
4. Mass conserving
5. Calculating pressure on a macroscopic scale
6. Running on the GPU

Computer games are often very performance intensive and any component in a computer game has to run at interactive framerates alongside every other component of the game. To achieve this the fluid simulation must run faster than real-time on its own.

Most fluid simulation research to this date has been on accurate physical simulations or visual focused animation. When developing a fluid simulation algorithm for real-time applications it is common to relax the mass conserving property of the fluid. This is a good way to speed up the simulation in applications where

fluid is added and removed from the system regularly such as with smoke, fire, rivers, or ocean. In cell based sandbox games the user has more direct control over the fluid and can build contraptions that depend on fluid that does not disappear. Another property that is often relaxed to speed up the simulation is pressure on a large scale. In cell based sandbox games it is useful if the pressure propagates across large distances for example so that the user can build reservoirs that supplies water to the users base.

Unlike many other applications realism is not important in this work. Looking at popular cell based sandbox games we see that the realism of the fluid does not necessarily correlate with player enjoyment. It is more important that the players understand how the fluid behaves and can interact with it. A good visual representation of the fluid can help build immersion but is not something that is covered in this thesis work.

The cells in cell based sandbox games are often of the size $1m^2$ or $1m^3$. Dwarf Fortress has cells that are 2 m high and 1 m wide, Minecraft has $1m^3$ cells, and Terraria has $0.7 m^2$ cells. A lot of fluid behavior such as surface tension, turbulence, and splashes will be lost with fluids simulated at this scale. The target world size in this project is $256 * 128 * 256$ cells which is a typical size for a cell based sandbox game.

1.1 Related work

Fluid simulation has been researched for a lot of different purposes. Weather forecasts use fluid simulation to predict weather. Movies and other visual media use fluid simulation to make realistic looking water, smoke, and fire.

The most common way of simulating fluids is by using the Navier-Stokes equation [1]. By solving this equation you get a velocity field that is used to advect the fluid. Advection is the process where the mass and other properties of a fluid is transported with regards to the fluids velocity field. Of particular interest in this thesis work is the simulation of incompressible newtonian fluids. That a fluid is incompressible means that the density of the fluid remains constant. A newtonian fluid has a constant viscosity. Viscosity can be thought of as the fluids thickness. An example of an incompressible newtonian fluids is water.

Jos Stam [2] proposes an unconditionally stable way to solve Navier Stokes equation. This allows for longer time steps which makes real-time interaction possible. The method works by representing the fluid in an isotropic grid while representing advection as particles, which is called semi-Lagrangian advection.

Lentine, Aanjaneya, and Fedkiw [3] proposes an improvement to the semi-Lagrangian method which unlike the traditional implementation of the semi-Lagrangian method is fully momentum conserving. Their improvements come from a new method in which the order of the advection, diffusion and clamping of mass has been modified.

The shallow water equations can be used to describe the flow below a pressure surface in a fluid. The equations are derived by integrating the Navier Stokes equation along an axis, which reduces the complexity. The shallow water equations are useful in situations where the horizontal size is much bigger than the vertical size of the fluid. It has problems in more unpredictable scenarios such as waterfalls, underground rivers, and eruptions. Kass and Miller [4] presents a fast and stable height-field approach to fluid simulation based on an approximation of the shallow water equations. The approximation consists of making the velocity of the waves proportional to the square root of the depth of the water. The stability comes from their use of an alternating-direction implicit integration technique.

A version of the height-field approach is the column-based height-field approach where the fluid is represented as columns that lie directly on the terrain. The method works by calculating the hydrostatic pressure in the columns and the flow between the columns that result from pressure differences. Marcelo, Fujimoto, and Norishige [5] presents an optimization of the column-based height-field approach that is designed to work with fluids on irregular terrain such as rivers. The optimization in their implementation is based upon removing redundant interactions between columns which reduces the amount of memory and calculations necessary. They also show an implementation that is parallelized for the GPU.

Chentanez and Müller [6] present an algorithm that combines the height-field approach with an isotropic grid. The height-field part of the algorithm makes up the bottom part of the body of fluid while the isotropic grid is on top of the height-field and provides the fluid with surface detail. By doing this they get visually rich fluid that runs with real-time performance. They also present a multigrid algorithm for solving the Poisson equation that represents the fluids viscosity. The algorithm runs on the GPU.

Smoothed particle hydrodynamics is a method of fluid simulation where the fluid is represented as particles. Each particle represent a mass of the fluid which makes the method inherently mass conserving. One drawback is that the method requires a large number of particles to achieve the same quality of simulation as a grid-based approach. Another method is needed to generate geometry in order to render the fluid. Hegeman, Carr, and Miller [7] shows an implementation of smoothed particle hydrodynamics that runs on the GPU. They use a dynamic quadtree structure to accelerate the lookup of nearest neighbor. The GPU implementation is nearly an order of magnitude faster than previous CPU versions.

Guay, Colin, and Egli [8] present an algorithm that solve the Navier Stokes equation in a single pass by temporary relaxing the incompressibility condition. This gives very good performance but is not mass conserving.

Chen, Lobo, Hughes, and Moshell [9] present an algorithm that first solves the Navier Stokes equation in 2D and then raises the surface according to the fluids pressure field. This reduces the complexity but is unstable.

2

Theory

This chapter will cover the theory behind the methods used in this project. The Navier-Stokes equation will be used as a baseline and is covered in the next section. The goal parameters of this project are slightly unusual so some parts of the Navier-Stokes equation has been modified to better fit the goal. In order to simulate a fluid we need three more parts that works together with the Navier-Stokes equation. We need to make the fluid incompressible and the different methods of achieving this will be discussed in section 2.2 Incompressibility. The Navier-Stokes equation uses a pressure gradient but how this pressure is calculated is not covered in the equation. In section 2.4 Pressure we will discuss how the pressure can be calculated. The final part that needs to be determined in the simulation is the method of advection, which is discussed in section 2.5 Advection. The algorithm is implemented in OpenGLs compute shaders. Section 2.6 OpenGL describes how to develop for OpenGL.

2.1 Navier-Stokes

Incompressible flows of Newtonian fluids can be described using the Navier-Stokes equation [1]:

$$\rho\left(\frac{\partial v}{\partial t} + v \cdot \nabla v\right) = -\nabla p + \mu \nabla^2 v + f \quad (2.1)$$

where v represents the fluid velocity vector, ρ is the fluid density, p is the pressure, μ is the viscosity, and f is external force vectors. The left side of the equation represent the inertia of the fluid while the right side of the equation represent stress factors and external forces. The equation is derived from Newton's second

law together with the assumption that the viscosity is proportional to the gradient of the velocity field.

The Navier-Stokes equation is a nonlinear partial differential equation which makes it very expensive to solve with high accuracy. By solving the Navier-Stokes equation we get a velocity field that describes the flow of the fluid at a given point in space. This velocity field is then used to advect the fluid.

As mentioned in the introduction to this chapter, the Navier-Stokes equation does not specify how the pressure and external forces are calculated. If we disregard the pressure and external forces we get the part of the equation that handles the velocity. How this part is calculated is discussed in section 2.3 Velocity.

2.2 Incompressibility

The fluids covered in this work are all incompressible. The most common way to enforce fluid incompressibility is to make the velocity field of the fluid divergence free, as seen in equation 2.2. This means that at least two kinds fluids need to be simulated in the model. The two fluids are usually air and the fluid that is the focus of the simulation, such as water, smoke or fire. Note that air is compressible but a common simplification in simulations where the pressure is relatively low is to consider air to be incompressible. There are two popular ways of enforcing a divergence free velocity field.

$$\nabla \cdot v = 0 \quad (2.2)$$

The first way consists of two passes. In the first pass we solve equation 2.1 and get a velocity field w that is not divergence free. In a second pass we project w onto its divergence free component to get v which is divergence free. To get the divergence free part you decompose the vector field into $w = u + \nabla q$ where u is a divergence free vector field and q is a scalar field. The projection operation will take the form of a Poisson equation and to solve it with high accuracy is expensive. This is the method used by Jos Stam in Stable Fluids [2]. A Poisson equation is a partial differential equation that is often written as $\nabla^2 \varphi = f$ where φ is the potential function to be determined and f is a known function. In a 3D grid the equation takes the form of $\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right) \varphi(x, y, z) = f(x, y, z)$.

The second way revolves around trying to enforce density invariance. This is done by solving the Navier Stokes equation for density to get the relation between the density variation and the divergence of the velocity vector field. This is used to create a corrective pressure field $P = K(\rho^n - \rho_0)$ where K is a constant chosen according to the fluid properties and ρ_0 is the initial density. This pressure field is used to make the vector field divergence free. It should be noted that the corrective pressure field does not replace the pressure field in equation 2.1 but could be seen as an internal pressure field. This is the method used Guay, Colin,

and Egli in Simple and Fast Fluids [8].

Neither of these methods give completely incompressible fluids, only approximations of it. To get completely incompressible fluids in real time alternative methods need to be used. One method is to relax the incompressibility condition in the first pass, and in a second pass move fluid from cells with compressed fluid into cells that are not full. This can be done with a flood-fill algorithm that traces backwards in the velocity field with the help of an influence map. This method suffers from inconsistent performance and poor performance in deep fluids. It also produces a “bubbling” effect on the surface of the fluid.

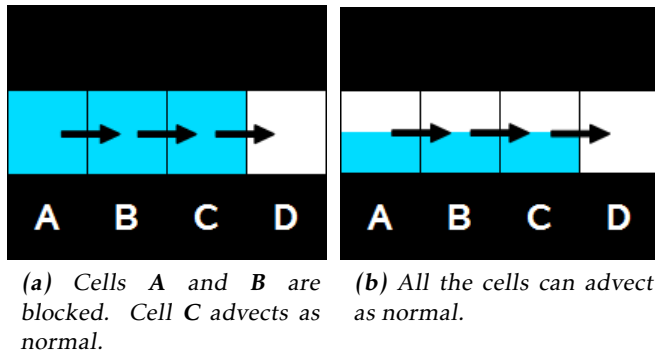


Fig. 2.1: The arrows represent fluid velocity which in this case is equal to the maximum fluid velocity.

It is also possible to enforce fluid incompressibility without making the velocity field divergence free and using equation 2.2. This means that only one fluid can be simulated at once and interactions between different fluids need to be handled separately. Some realism is also lost from the lack of bubbles and the lack of resistance from the second fluid. A very important side effect of this is that cells that do not contain the simulated fluid do not need to be simulated, which is discussed in greater depth in section 3.4 Resting work areas. The terrain in cell based sandbox games is often unpredictable and may change at any time from things like floodgates closing, placing blocks, or starting pumps. A change in the terrain can create a chain reaction where the velocity field for a large body of fluid becomes affected. This means that it is expensive to predict the density of a cell in the next iteration. To enforce incompressibility we have to conservatively advect. This works by assuming that no other fluid than the current cell will advect and then only advect the amount of fluid that can currently fit in the destination cell. If more than one cell tries to move fluid into the same cell then the fluid in that cell can temporarily be compressed. If the system compensates for this by increasing the pressure then the system can become unstable. Another side effect from this method is that the maximum advect amount is limited by the density of the destination cell which means that cells with fluids at maximum velocity can only be half full. Figure 2.1 illustrates this effect. The effect can be reduced by

decreasing the time step length without increasing the maximum velocity. This method is very cheap to execute and is easy to make stable.

2.3 Velocity

In this section I will discuss how to solve the velocity part of the Navier-Stokes equation. I will do this by breaknig down the equation and looking at what effect each part contributes with and how to solve it. The velocity part of the Navier-Stokes equation can be written as:

$$\frac{\partial v}{\partial t} = -(v \cdot \nabla v) + \mu \nabla^2 v \quad (2.3)$$

The first term, $-(v \cdot \nabla v)$, represent convective acceleration which is the change in velocity over position. The convective acceleration make the cells spread their velocity by averaging the velocity of the transported fluid with the velocity of the fluid in the destination cell. When enforcing a divergence free velocity field this will result in effects such as fluids accelerating through narrower parts of a pipe. Since we are not enforcing a divergence free velocity field these effects will be lost. In this work we solve the convective acceleration as:

$$v_a^{n+1} = \frac{v_a^n * m_a^n + v_b^n * m_b^n}{m_a^n + m_b^n} \quad (2.4)$$

where v_a and m_a is the velocity and mass of the current cell, while v_b and m_b is the velocity and mass of the fluid transported to the current cell.

The second term, $\mu \nabla^2 v$, represent viscosity which can be thought of as the fluids thickness. Lava has a very high viscosity, water has a low viscosity, and air has a very low viscosity. The viscosity term can be solved efficiently using a second order central finite difference scheme as shown by Guay, Colin, and Egli [8]. Like the vast majority of real time applications of the Navier-Stokes equation their method is made for a divergence free velocity field. To make the viscosity work without a divergence free velocity field is beyond the scope for this thesis work so the viscosity term is ignored. This can be seen as setting the viscosity to zero which gives the fluid a thickness closer to air than water. To emulate the effect of viscosity I use a down scaling of the velocity as: $v^{n+1} = K * v^n$ where K is the scaling factor.

Another side effect of a divergence free velocity field is that the velocity in a cell will decrease if the velocity in the neighboring cell decreases. If a body of moving fluid becomes blocked then the fluid velocity in the body will decrease. I emulate this by decreasing the velocity in a cell proportionally to how much of the fluid that could not fit in the neighboring cell. I solve this as:

$$v^{n+1} = v^n * (1 - m^n) \quad (2.5)$$

where v is the velocity and m is the mass of the fluid that became blocked.

2.4 Pressure

The pressure field can be derived from $p = \frac{F}{A}$ where F is the normal force and A is the contact area. If we disregard external forces the equation can be written as $p = \frac{ma}{A}$ where m is the fluid mass and a is the fluid acceleration. The large time steps needed for real time simulation makes it hard to calculate the pressure from fluid acceleration. The accuracy becomes poor and the system becomes unstable for large time steps. Instead of using the fluid acceleration we limit the algorithm to only handle fluid that is at rest.

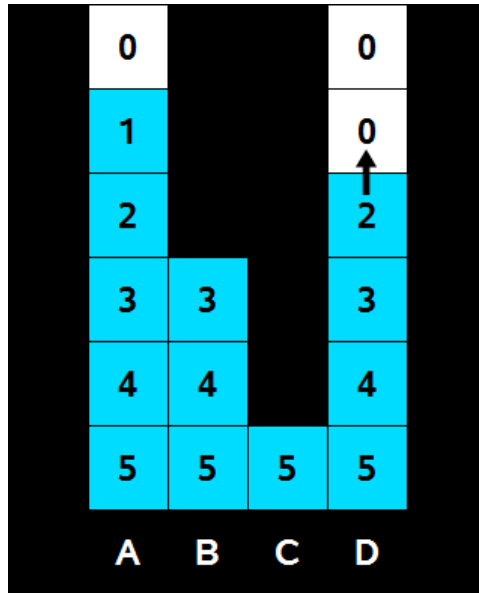


Fig. 2.2: A deformed U-pipe seen from the side. The numbers represent the pressure values and the letters are for column reference. Note how the pressure is calculated as normal according to $p = \rho gh$ in column A. In column B and C the pressure has been shared from the neighboring cells. In column D the pressure is being shared upwards. The arrow to the top right represent the change in the velocity field that will result from the pressure field and gravity.

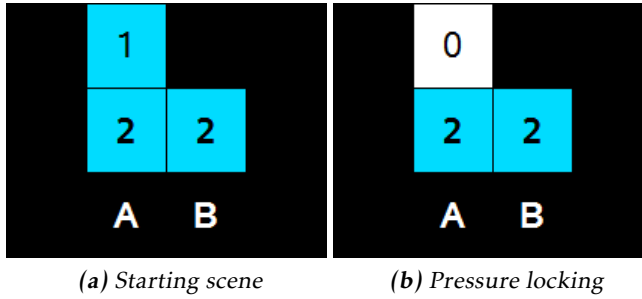


Fig. 2.3: A scene where the pressure is locked between two cells after the original pressure source is removed. The numbers represent the pressure values.

2.4.1 Hydrostatic pressure

When the fluid is at rest and is only affected by gravity we can derive the hydrostatic pressure as $p = \rho gh$ where g is the gravitational acceleration and h is the height of the fluid column above the active cell. This assumes that the fluid is incompressible and that g is constant. With this limitation we loose the interaction of momentum to pressure but we get a more responsive system that is stable. The pressure is the same for fluid in the same system at the same height which means that cells need to share their pressure with neighboring cells in the xz -plane. To allow the fluid to flow upwards the pressure needs to be shared with neighboring cells upwards. This is illustrated in figure 2.2. When the pressure is shared between neighboring cells a new problem is introduced. Two neighboring cells can “lock” a pressure value by continuously sharing the pressure between them after the source of the pressure is removed. This is illustrated in figure 2.3. To solve this we introduce a dampening effect on the pressure sharing which will make the pressure locked cells loose their locked pressure over a few iterations. A side effect is that the accuracy of the pressure is slightly reduced.

2.5 Advection

Advection is the process where the mass and other properties of a fluid is transported with regards to the fluids velocity field. There are two popular methods of advection that works in a cell based simulation; semi-Lagrangian advection and eulerian advection.

In lagrangian advection the fluid is represented as particles and each particle has its own velocity. This technique is difficult to make stable with the large time steps required in real-time applications. A far more popular technique for advecting a fluid in real-time is semi-Lagrangian advection [2]. In this technique the fluid is represented in a grid of cells that contain the mass and velocity of the fluid, but the advection of the fluid is done with particles. A particle is cre-

ated in the center of each cell and is then traced through the velocity field for Δt time. The fluid is advected between the source cell and the cell that the particle ends up in. This is illustrated in figure 2.4. Semi-Lagrangian advection can be made stable for any time step length but requires that the velocity field is divergence free which can be a problem as discussed in section 2.2 Incompressibility. The most common version of semi-Lagrangian advection traces the particle backwards against the velocity field.

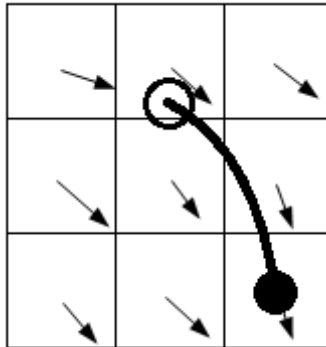


Fig. 2.4: semi-Lagrangian advection. The filled circle is the starting point of the particle and the non-filled circle is the destination.

The technique used in this work is eulerian advection where the fluid and the advection is represented in a grid of cells. Between every neighboring cell is a virtual pipe that the fluid advects through. How many neighbors each cell interacts with will be discussed in section 3.1 Parallelization. Eulerian advection does not require a divergence free velocity field and is very easy to parallelize.

2.6 OpenGL

Compute shaders [10] has been a part of OpenGL since version 4.3 and is a new shader stage used entirely for general-purpose computing on graphics processing units (GPGPU). Compute shaders are separate from the traditional rendering pipeline and does not have any defined input or output variables except for the compute space defining constants. It is up to the user to read and write to buffers and to determine how to interpret the data in the buffers.

Shader storage buffer object (SSBO) [11] is a new type of buffer that was introduced in OpenGL version 4.3. SSBOs have similar functionality as texture objects using the image load/store functions but are easier to work with since the data in an SSBO is generic while the data format in a texture is generally more restrictive. SSBOs can typically contain an amount of data on the order of magnitude of GPU memory and can read and write atomically.

2.6.1 Work group

GPUs consists of a number of compute cores. The GPU I used when testing the implementation of this work is a GeForce GTX 670 which has 1344 cores[12]. The cores are grouped into a number of work groups which shares a work group specific shared memory[10]. It is generally a good idea to have as large work groups as possible while still having enough shared memory for each work item. In this implementation I use a work group size of $16 \times 8 \times 8$ work items which is equal to the required maximum work group size of 1024 in the OpenGL specification. As will be discussed in section 3.2 Fluid cell data structure this work group size also makes good use of all the shared memory.

2.6.2 Memory

GPUs have a number of different memory types and using the correct memory type for each situation can have a big performance impact [13]. In this work I will only use three of them but I will briefly describe them all. Note that the memory type names can be different between different platforms such as CUDA and OpenGL.

Global memory can be read from and written to by the host program and all threads in all workgroups. Global memory is cached but is still very slow. Accessing the global memory can be a major bottleneck so it is recommended to limit its use. The content in the global memory is preserved between shader invocations which allows us to limit the expensive data transfer between the host program and the GPU. In this work we use the global memory for SSBOs containing the fluid cell data structure arrays.

Shared memory can be read from and written to by all threads in a workgroup. Each work group has its own shared memory which can not be accessed by the other workgroups. Shared memory is very fast and often used as a manual cache for the global memory. A common pattern which is used in this work is to assign each element in the data structure array to its own thread and let that thread copy the content of the data structure from the global memory to the shared memory. This way other threads in the workgroup can access the data from the shared memory instead of the global memory. This is the way I use shared memory in this work. The content in the shared memory will not be preserved between shader invocations.

Registers are where the local variables in a thread are stored. Registers are very fast. In most applications the programmer does not have to consider the use of registers. In applications that have very much local variable data the data will be stored in local memory. Local memory is an abstraction of the global memory and as such is very slow. In this work we do not have that much local variable data so that we have to worry about local memory.

Texture memory can be written to by the host program and read from by all threads in all workgroups. Texture memory is of comparable speed of the global memory but the cache is optimized for operations common for textures. Texture

memory provides free linear interpolation. I do not use texture memory in this work.

Constant memory can be written to by the host program and read from by all threads in all workgroups. Constant memory is as fast as registers but is very small. Constant memory could be used if the host program wants to change settings in runtime but is not used in this work.

2.6.3 Memory alignment

GPUs are optimized to read chunks of 32, 64, or 128 bytes at a time that is aligned to their respective size from the global memory [14]. Lets do an example where we read an array with elements of size 20 bytes from the global memory. This is illustrated in figure 2.5. When we access the element A we would read the first 32 bytes and get element A and parts of element B. When we access element B the GPU will read two chunks of 32 bytes and then merge data from the two chunks to get element B. As we can see the unaligned memory forces the GPU to access twice as much data as aligned memory would. To fix this we add padding to our elements so their size becomes 2^x for some positive integer x . Figure 2.6 shows a case where the memory is aligned.

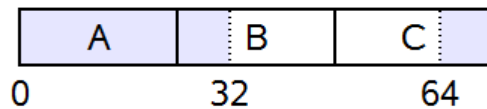


Fig. 2.5: *Unaligned memory. The numbers represent byte indices in the memory. A, B, and C are the elements.*

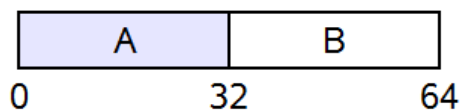


Fig. 2.6: *Aligned memory. The numbers represent byte indices in the memory. A, B, and C are the elements.*

3

Implementation

This thesis work has been implemented in C++ with the main algorithm implemented in OpenGLs compute shaders. A basic rendering code was written for debugging and demonstration purposes.

3.1 Parallelization

The volume to be simulated is discretized into an isotropic grid of cells where each cell is meant to represent 1 m^3 . Each cell can be calculated using only the previous state of the grid. This means that the order the cells are calculated does not matter as long as the grid is completely calculated before the next iteration starts. In other words, the cells can be calculated in parallel. If each cell writes the result of the calculations into the same buffer that it read from we introduce race conditions. The cells might access the data of the neighboring cells from this iteration or the previous iteration. To solve this we use the ping-pong technique [15]. The cells read data from an input buffer and writes data to an output buffer. After each iteration the buffers are swapped so the cells always read the data from the previous iteration. By using this technique the data in the input buffer will remain constant during an iteration which removes the race conditions.

How far each cell looks is also an important consideration. In this project cells have 6 direct neighbors in 3D, which is called 6-connectivity. If the cells look to their closest neighbors they have to access 7 cells, if they look one cell further they have to access 25 cells, and if they look 3 cells away they have to access 63 cells. If the cells access cells further away we can allow a faster maximum fluid velocity and a faster pressure spread but the calculations will take considerably longer to execute. If the cells only access their closest neighbors we achieve a

fluid velocity and pressure spread speed of 30 m/s if the algorithm runs at 30 ticks per second with 1 m^3 cells. The terminal velocity of rain is 9 m/s [16] and a typical river flows at about 2 m/s [17] so a maximum fluid velocity of 30 m/s is sufficient for most scenarios. In real life the pressure in water spreads at the speed of sound which is about 1500 m/s in water [18]. This is far faster than 30 m/s but on a small scale it is not noticeable. In this thesis work I prioritize performance which is why the cells only access their closest neighbors.

3.2 Fluid cell data structure

The fluid simulation works with 4 variables per cell; *volume*, *pressure*, *solid*, and *velocity*. The variables and their respective size can be seen in table 3.1. If all variables have a precision of 4 bytes per float then the cell data structure is 21 bytes large. In order to get memory alignment we can add padding to the fluid cell data structure so it becomes 32 bytes large. If needed, the 11 bytes of padding could be used to store some other property such as temperature or to increase the precision of some variable. It is also possible to achieve memory alignment by making the data structure 16 bytes large. The first way to do this is to lower the precision of volume to 1 byte and the precision of pressure to 2 bytes. The other way is to lower the precision of velocity to 6 bytes. Both ways can be made to work but will noticeably lower the simulation quality. One of the reasons why one would want to have a cell data structure size of 16 bytes instead of 32 is to fit twice as many cells in the shared memory of a work group. In the OpenGL specification, the required maximum shared memory size is 32 KB, and the required maximum work group size is 1024. This makes for a ratio of 32 bytes per cell which is as much as we have in the full precision version of the cell data structure, which means that we can not have larger work groups anyway. The other reason to have data cell structures of 16 bytes is that a lower amount of data is quicker to read from the global memory than a larger amount of data.

Name	Full precision size	Alternative size A	Alternative size B
Volume	4 bytes	1 bytes	4 bytes
Pressure	4 bytes	2 byte	4 bytes
Solid	1 bit	1 bit	1 bit
Velocity	12 bytes	12 bytes	6 bytes
Total with padding	32 bytes	16 bytes	16 bytes

Table 3.1: Fluid cell data structure with 3 precision options.

3.3 Pseudo Code

The algorithm is implemented in an OpenGL compute shader. Compute shaders do not have input or output variables but instead buffers in the global memory that it can read and write. This algorithm uses one SSBO for input and one SSBO

for output.

```

copy data into shared memory
wait for the rest of the work group
for each axis

    find the neighbor cell that this.velocity points to
    unclampedTransport = this.mass * this.velocity * dt
    clampedTransport = clamp unclampedTransport
                        between 0 and neighbor.spaceLeft
    blockedFluid = unclampedTransport - clampedTransport
    this.mass = this.mass - clampedTransport
    this.pressure = blockedFluid / dt
    this.velocity = this.velocity * (1 - blockedFluid)
end

```

The cell looks forward through the velocity field to find the neighboring cell that this cell will advect to. It calculates how much fluid can be transported to the neighboring cell conservatively as discussed in section 2.2 Incompressibility. The reason pressure depends on the amount of blocked fluid is to make cells contribute with pressure only if something prevents it from transporting all the fluid it can. This prevents falling and other non-static fluids from adding pressure. Finally we decrease the velocity proportionally with the amount of blocked fluid as discussed in section 2.3 Velocity.

```

for each neighbor

    if the neighbor.velocity points to this cell
        unclampedTransport = neighbor.mass *
                               neighbor.velocity * dt
        clampedTransport = clamp unclampedTransport
                               between 0 and this.spaceLeft
        this.mass = this.mass + clampedTransport
        this.velocity =
            (neighbor.velocity * clampedTransport +
             this.velocity * this.mass) /
            (clampedTransport + this.mass)
    end
end

```

In this section the cell looks backwards through the velocity field to find each neighboring cell that wants to advect to this cell. Like with forward step the cell advects conservatively as discussed in section 2.2 Incompressibility. The other thing we do in this section is the convective acceleration part that is described in section 2.3 Velocity.

```

add pressure from the cell above
if cell(below).pressure - maxWater > this.pressure

```

```

        copy the pressure from the cell(below)
    end
    for each neighbor at the same height
        if the neighbor has a higher pressure
            copy the pressure from the neighbor
        end
    end
end

```

Here we spread the pressure from neighboring cells to this cell as described in section 2.4 Pressure.

```

    add gravity acceleration
    reduce the velocity by a scaling factor
    clamp the velocity to be within the maximum velocity

```

Here we add external forces which in this case is gravity. We then scale the velocity down to simulate the viscosity part of the Navier-Stokes equation as discussed in section 2.3 Velocity. Finally we make sure that the velocity stays within the maximum velocity.

3.4 Resting work areas

In this section the group of cells that a work group computes will be called *work area*. Because of the decision to not enforce a divergence free velocity field we don't have to compute work areas where no cells currently contain any fluid or will contain fluid in the next iteration. To know which work areas should be computed we create an array where each element represent whether or not a work area should be active or resting. We call that array the *work area array*. An active work area is a work area that contains fluid and will be computed. A resting or inactive work area is a work area that does not contain fluid and will not be computed. The elements in the work area array are stored in an SSBO in the global memory and are reset to 'inactive' before each solver iteration. When a cell detects that it contains fluid or will advect fluid to an adjacent cell it sets the corresponding element in the work area array to 'active'. By doing this a lot of cells will overwrite each others data in the work area array but since the data can only change from 'inactive' to 'active' this does not matter.

In an implementation on the CPU we could just query the work area array with an if-statement in the beginning of the solver to skip work areas that are empty, but because of the SIMD architecture of a GPU that solution will not work. In this implementation I solve this by using a parallel prefix sum algorithm. A prefix sum algorithm will take an array of data as input and output an array where each element is the sum of all previous elements. We call this array the *prefix sum array*. For example, if we input [2, 5, 1, 0, 9] we would get [0, 2, 7, 8, 8]. Note how the output is shifted one step and the last element in the input is ignored. This version of a prefix sum algorithm is called *exclusive*. The implementation of the

prefix sum algorithm used in this thesis work is the one described in chapter 39 of GPU Gems 3[19].

Thread index	0	1	2	3	4	5	6
Work area array	[0	1	0	1	1	0	1]
Prefix sum array	[0	0	1	1	2	3	3]
Compact array	[1	3	4	6]			

Fig. 3.1: A compact array and a prefix sum array that is created from a work area array. The elements that thread #3 reads and writes has been circled.

What we want in order to skip empty work areas is an array where the elements are indices for work areas that are active. We call that array the *compact array*. In the normal case each work group would compute its corresponding work area, but now the work groups will instead compute the work area linked by the corresponding element in the compact array. For example, if the compact array is [7, 31, 36, 42] then the first work group would compute work area 7 and the second work group would compute work area 31 and so on. The compact array can be created in parallel. Each thread looks at its corresponding element in the work area array. If that element is 'active' then the thread looks at its corresponding element in the prefix sum array. The value in the prefix sum array is the index in the compact array that the thread will write the index of the thread. This is illustrated in figure 3.1. For example, thread #3 in the figure will first check index 3 in the work area array. Since that value is 'active' it will check index 3 in the prefix sum array. That value is 1 which is the index in the compact array that the thread will write its thread index to, which is 3. The threads #0, #2, and #5 where its corresponding element in the work area array is 'inactive' will not write to the compact array.

4

Result

To evaluate this project I will look at the performance of the algorithm and the behavior of the fluid. The performance is easy to objectively measure but the quality of the behavior is a subjective quality.

4.1 Performance

All of the measurements are done on a GeForce GTX 670 running OpenGL 4.3. Each test is run for one minute and the documented result is the average performance during this time. The performance is measured both with and without the prefix sum algorithm running. When testing without the prefix sum algorithm we are interested in the overall performance as well as how the performance scales with the number of cells. When testing with the prefix sum algorithm we are interested how the performance scales with different ratios of active to inactive cells as well as any overhead for running the prefix sum algorithm.

Table 4.1 shows the performance of the algorithm when running at different grid size without using the prefix sum algorithm. The performance of 91 frames per second for our target grid size of $256 \times 128 \times 256$ satisfies our requirement of faster than real-time performance. An interesting observation is that the performance scales superlinearly with the grid size. A superlinear speedup is when the algorithm runs faster on a larger data set than on a smaller data set [20]. The two most likely reasons for the superlinear speedup in this project is that the GPU use its work groups more efficiently, and that the data read from the global memory becomes cached which makes subsequent memory accesses more efficient. Figure 4.1 is a chart that shows the number of cells computed per second for different sized grid. The superlinear speedup is clearly visible in this chart.

Table 4.2 shows the performance of the algorithm using the prefix sum algorithm. All the tests are run on the target grid size of $256*128*256$ with varying amount of cells active. When the prefix sum is used but all cells are active we get a performance of 72 frames per second. Compared with the performance of 91 frames per second that the algorithm achieves without the prefix sum algorithm on the same grid size, we see that the overhead of using the prefix sum algorithm is about 19 frames per second in this case. Figure 4.2 is a chart that compares the performance when using and not using the prefix sum algorithm. The prefix sum algorithm plot uses the target grid size of $256*128*256$ but has a variable amount of cells active to be computed.

Number of cells	Frames per second
$256*128*256$	91
$192*128*256$	98
$128*128*256$	144
$128*128*128$	231
$64*64*64$	1,418
$32*32*32$	9,698

Table 4.1: Performance without the prefix sum algorithm

Number of cells	Percent active	Frames per second
$256*128*256$	100 %	72
$256*128*256$	75 %	87
$256*128*256$	50 %	115
$256*128*256$	25 %	201
$256*128*256$	0 %	23,422

Table 4.2: Performance with the prefix sum algorithm

4.2 Behavior

Please note that visualization of the fluid is not a part of this thesis work. A simple renderer was implemented for debugging and demonstration purposes. The cells are colored in such a way that it is easy to see the height of the cells. The cells are colored from blue at the lowest height, to green after 10 cells difference in height. The color then wraps to blue again. The internal height in the cells is rendered from not red at the bottom of the cells, to red at the top of the cells.

The effect of hydrostatic pressure can be seen in figure 4.3. The pressure spreads to neighboring cells and pushes the water up the right side of the pipe. In this case I'm reducing the pressure spread by 5% in order to prevent pressure locking. In the figure the pressure spreads 13 cells and the reduced pressure spread can be seen in the bottom-right image in figure 4.3 where the water is at rest. The

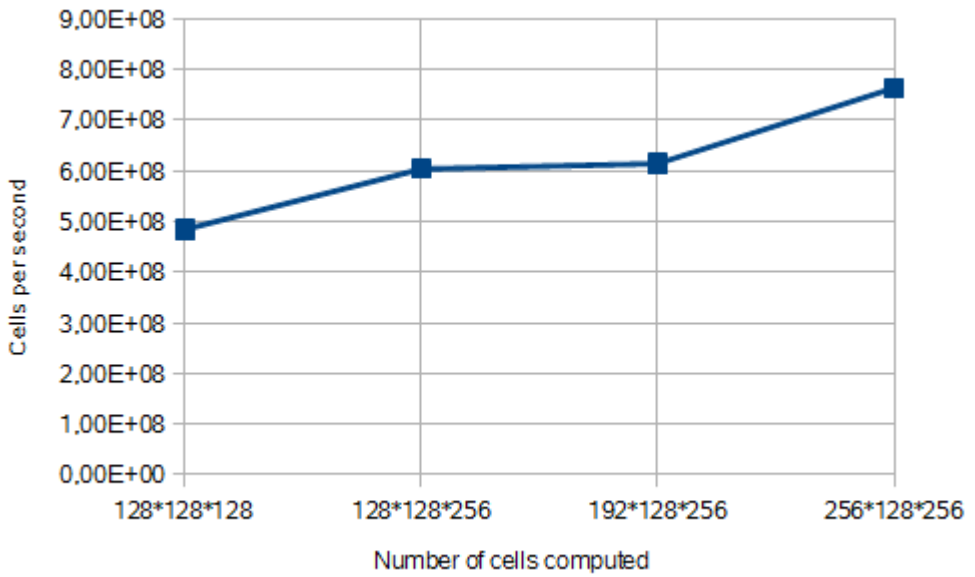


Fig. 4.1: Number of cells computed per second for different sized grid.

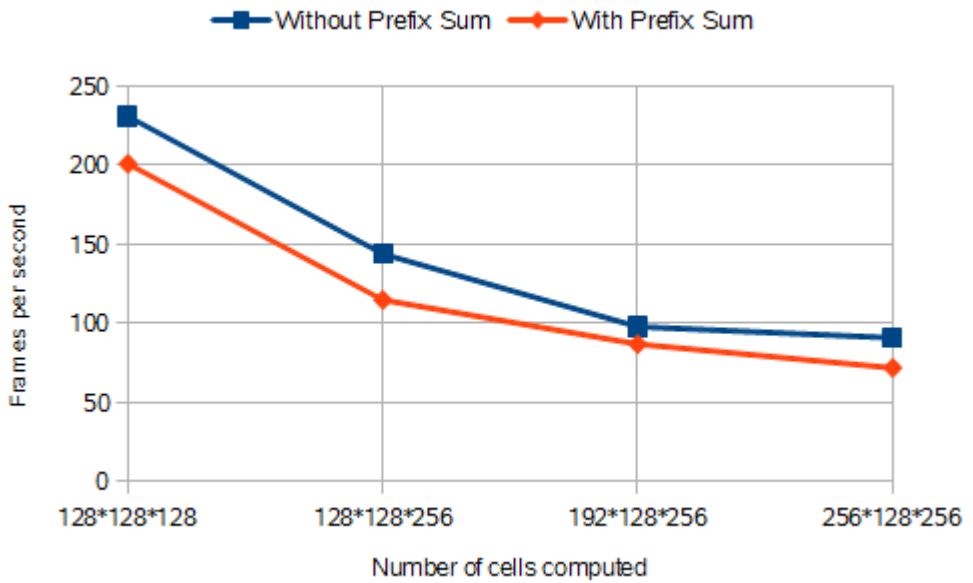


Fig. 4.2: Performance comparison between using and not using the prefix sum algorithm.

top-right image in the figure shows the momentum created from the pressure difference pushes the water against the right wall.

Figure 4.4 shows a scene where the hydrostatic pressure from a water pillar ejects the water through a pipe. The different height of water pillar creates the expected difference in arcs. A flaw in the pressure spread can also be seen in the figure. The pressure spreads from the water pillar out into the ejected water and thereby spreads the water.

In figure 4.5 we can see a wave created from an initially stationary body of water. This wave is the result of the higher pressure that spreads from below the smaller body of water. When the pressure spreads the water will accelerate towards the lower pressure which is upwards and outwards.

Figure 4.6 shows a river that is in the process of filling with water from an eternal water source.

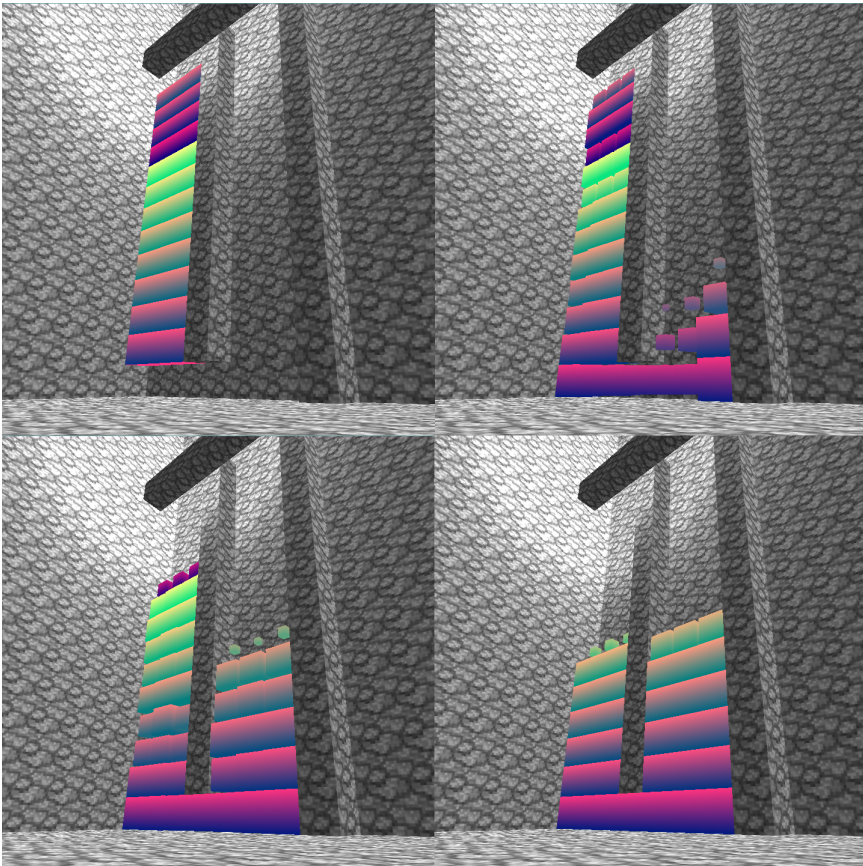


Fig. 4.3: *Hydrostatic pressure makes the fluid level equalize in the U-pipe. The imbalance in the final figure is from pressure lock prevention.*

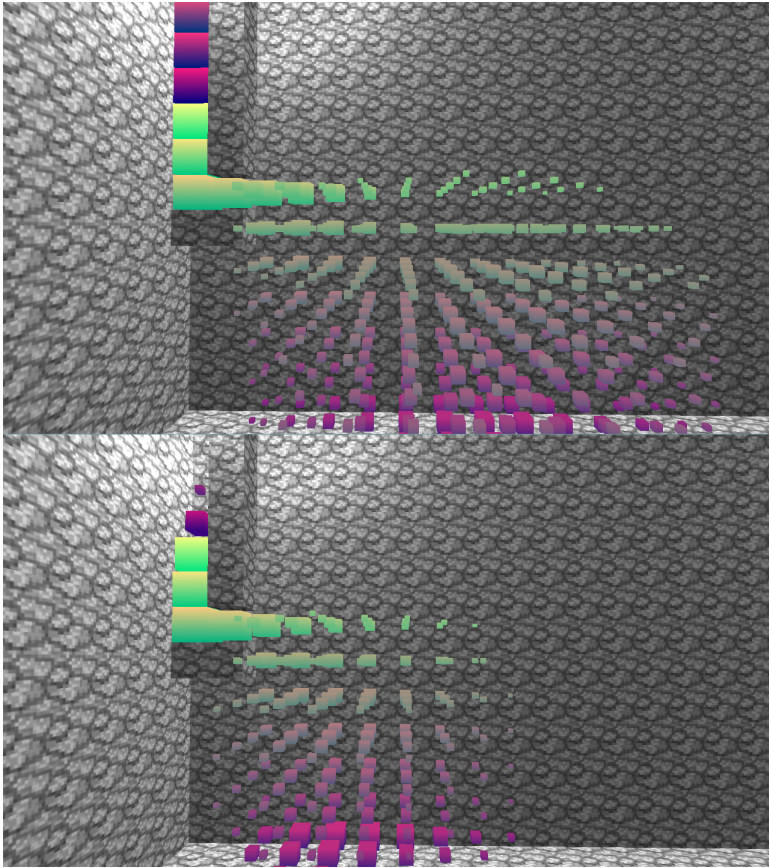


Fig. 4.4: Different height of water pillar ejecting water at different speed. The spread at the nozzle is because the pressure spreads out to the ejected water.

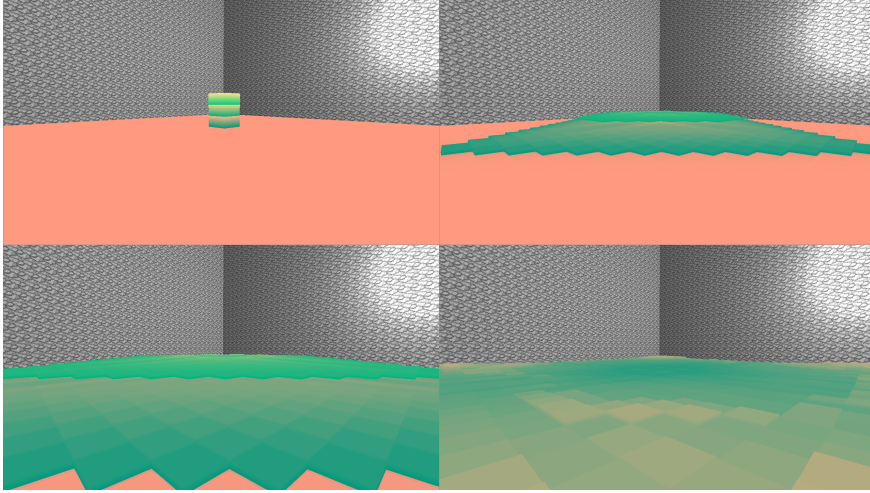


Fig. 4.5: A wave that is created from an initially stationary body of water. This effect is from hydrostatic pressure.

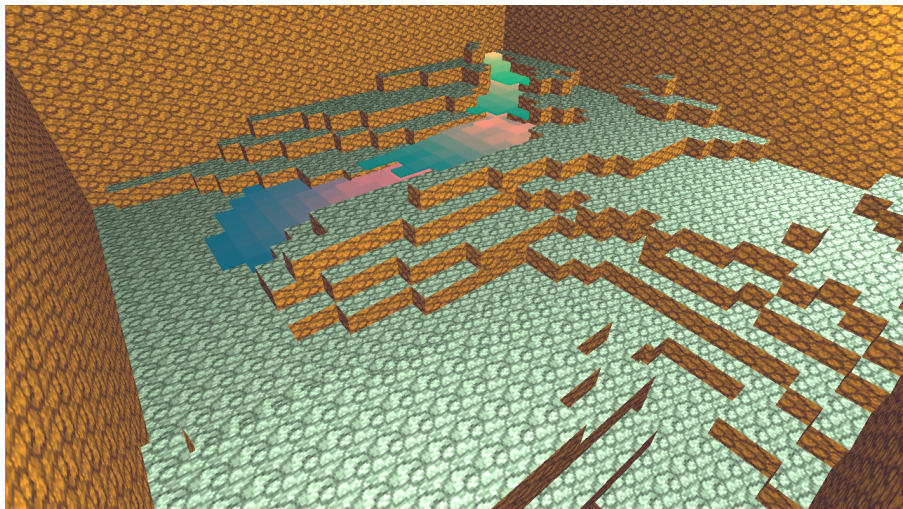
4.3 Overall Evaluation

In this section I will evaluate the result of this project compared with the goals set up in chapter 1:

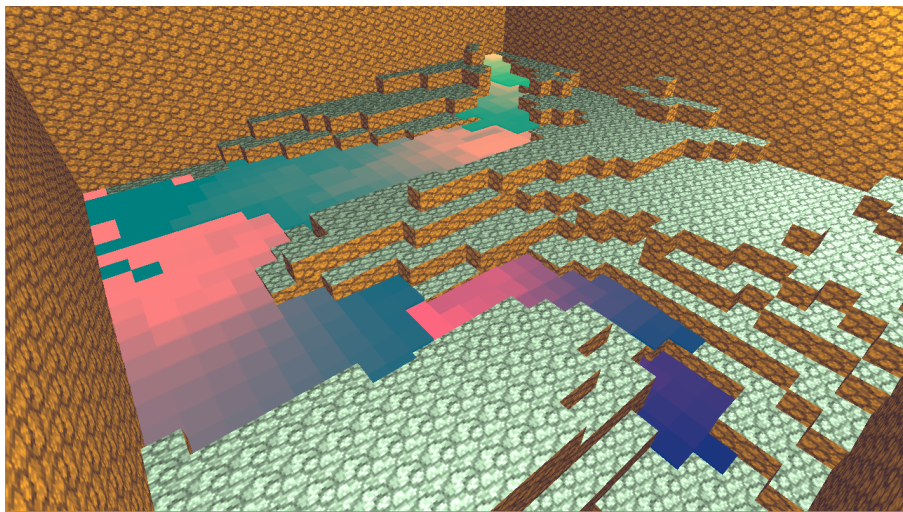
1. Faster than real-time performance
2. Giving consistent performance with no lag spikes
3. Cell based
4. Mass conserving
5. Calculating pressure on a macroscopic scale
6. Running on the GPU

4.3.1 Is the algorithm faster than real-time performance?

When running the algorithm on a grid of size $256 \times 128 \times 256$ cells we get a performance of 91 frames per second when the prefix sum algorithm is not used. If the prefix sum algorithm is used we can get a performance of hundreds of frames per second depending on how many cells are active. This is well within the criteria for faster than real-time performance. Rendering of the fluid can be done very efficiently since the fluid data is already located on the GPU which eliminates the need for most data transfers between the CPU and the GPU.



(a)



(b)

Fig. 4.6: A new river flowing down the landscape.

4.3.2 Is the algorithm giving consistent performance with no lag spikes?

The algorithm does the same amount of computation regardless of the data it works with. A cell where the fluid is at rest will take as long to compute as a cell where the fluid flows at high speed with multiple external forces present. When testing the algorithm the frame rate was consistent with variations of up to 5% and no lag spikes. This is good enough to be considered consistent performance. When using the prefix sum algorithm the performance can vary drastically. The performance is consistent while the same amount of cells are active. This means that the performance inconsistencies are only when the algorithm runs faster due to a lower amount of active cells. This can decrease the value of the prefix sum algorithm so whether or not it is to be used depends on the application.

4.3.3 Is the algorithm cell based?

The algorithm is cell based. In a cell based sandbox game the world already consists of cells which makes the interaction between the fluid simulation and the rest of the application seamless. If the cells in the application are large then it is hard to render thin bodies of fluid such as small streams or thin jets of water. If such details are important in the application then the cells in the fluid simulation can be made smaller than the cells in the rest of the application, and can then be stored in an octree or a similar data structure.

4.3.4 Is the algorithm mass conserving?

The conservative advection model used in this project makes the algorithm completely mass conserving. If mass conservation is of lower importance then the conservative advection can be relaxed in order to make the system stable for longer time steps or lower the limitation on velocity from cell mass as discussed in section 2.2 Incompressibility. The algorithm was completely mass conserving when testing.

4.3.5 Is the algorithm calculating pressure on a macroscopic scale?

The pressure calculated in this project is the hydrostatic pressure. The pressure is propagated to neighboring cells in order to get pressure on a macroscopic scale. To avoid pressure locking we have to decrease the propagation of the pressure which creates the artifacts that can be seen in figure 4.3. By only calculating the hydrostatic pressure we miss some fluid behavior and the interaction from momentum to pressure. The pressure model presented in this project is sufficient for some applications but is something that can be improved.

4.3.6 Is the algorithm running on the GPU?

The algorithm runs on OpenGLs compute shaders on the GPU. The algorithm is created to be run in parallel and different kinds of memory is used on the GPU to improve the performance.

5

Summary

In this report I have presented a new algorithm for simulating fluids in a cell based world. The algorithm is based on Navier-Stokes equation but is heavily modified to work without a divergence free velocity field. This has allowed the algorithm to be completely mass conserving and has made it easier to skip the computation of empty work areas which significantly speeds up the algorithm. In order to skip empty work areas a prefix sum algorithm was used. The algorithm also features a global scale pressure model that simulated hydrostatic pressure. This pressure spreads to neighboring cells which creates phenomena such as high pressured water flowing up a pipe, and waves created from an impact with a body of water.

5.1 Future work

By not enforcing a divergence free velocity field we miss some fluid like behavior like whirls and bubbles. These behaviors can be achieved by loosely enforcing a divergence free velocity field while still using the conservative advection described in this report. By doing this it might be possible to relax the restrictions on the conservative advection in order to lower the impact to maximum velocity that mass have. The limiting factor in such an algorithm would be stability as cells would often have more fluid what is allowed. If a divergence free velocity field were to be enforced then it would be hard to skip computing cells that are empty with a prefix sum algorithm. An alternative solution would be to go from an isotropic grid to an adaptive octree or some similar data structure.

In this project we simulate hydrostatic pressure. It might be possible to extend this to include both hydrostatic pressure and dynamic pressure. The biggest prob-

lems with this would be that the relatively long time steps required for real-time applications would create oscillations and instability. This might be solvable by having a separate data structure containing the pressure that dynamically changes so each element is a body of fluid that has the same pressure.

The focus of this project was to create an algorithm for simulating fluid but did not include any visualization of the fluid. Compared to most other fluid simulation algorithms, this algorithm is designed to work with relatively large cells. This might make it hard to create a convincing rendering that disguises the cells. Mist, foam, and splash particles can help with disguising the cells. Since this algorithm is designed to work in OpenGLs compute shaders it would be very efficient to render the fluid since all the fluid data is already on the GPU. The only time the fluid data has to be transferred between the CPU and the GPU is when the user interacts with the fluid which is something that happens comparatively rarely.

Bibliography

- [1] D. J Acheson. Elementary fluid dynamics, 1990. Cited on pages 2 and 5.
- [2] Jos Stam. Stable fluids. Cited on pages 2, 6, and 10.
- [3] Mridul Aanjaneya Michael Lentine and Ronald Fedkiw. Mass and momentum conservation for fluid simulation. Cited on page 2.
- [4] Michael Kass and Gavin Miller. Rapid, stable fluid dynamics for computer graphics, August 1990. Cited on page 3.
- [5] Tadahiro Fujimoto Marcelo M. Maes and Norishige Chiba. Efficient animation of water flow on irregular terrains. Cited on page 3.
- [6] Nuttapong Chentanez and Matthias Müller. Real-time eulerian water simulation using a restricted tall cell grid. Cited on page 3.
- [7] Nathan A. Carr Kyle Hegeman and Gavin S.P. Miller. Particle-based fluid simulation on the gpu. Cited on page 3.
- [8] Fabrice Colin Martin Guay and Richard Egli. Simple and fast fluids. Cited on pages 3, 7, and 8.
- [9] Hughes Chen, Lobo and Moshell. Real-time fluid simulation in a dynamic virtual environment, May-June 1997. Cited on page 3.
- [10] Opengl 4.3 core profile specification. Cited on pages 11 and 12.
- [11] Barthold Lichtenbelt. Opengl 4.3 overview. https://www.khronos.org/assets/uploads/developers/library/2012-siggraph-opengl-bof/OpenGL-4.3-Overview-SIGGRAPH_Aug12.pdf, May 2014. Cited on page 11.
- [12] Geforce gtx 670 specification. <http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-670/specifications>, May 2014. Cited on page 12.
- [13] Ashu Rege. An introduction to modern gpu architecture. [http:](http://)

- [//http.download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf](http://http.download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf), May 2014. Cited on page 12.
- [14] NVIDIA Corporation. Advanced cuda webinar, memory optimizations. http://on-demand.gputechconf.com/gtc-express/2011/presentations/NVIDIA_GPU_Computing_Webinars_CUDA_Memory_Optimization.pdf, 2009. Cited on page 13.
- [15] Daniel Weiskopf. Gpu-based interactive visualization techniques. Cited on page 15.
- [16] John H Corbet. *Physical Geography Manual*. 2013. Cited on page 16.
- [17] Marie Parsons. <http://www.touregypt.net/egypt-info/magazine-mag05012001-magf4a.htm>, May 2014. Cited on page 16.
- [18] Carl Nordling and Jonny Österman. *Physics Handbook for Science and Engineering*. 2006. Cited on page 16.
- [19] Hubert Nguyen. *GPU Gems 3*. 2007. Cited on page 19.
- [20] David R. Emerson Ismail H. Tuncer, Ülgen Gülcat and Kenichi Matsuno. *Parallel Computational Fluid Dynamics*. Springer, 2007. Cited on page 21.

Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: <http://www.ep.liu.se/>