Real-Time Water Simulation on the CPU and GPU

Anna Wästling
Edvin Nordin

May 15, 2023

Contents

1 Introduction

In this report we will go into detail regarding solving the Navier-Stokes equations in theory as well as our implementations on both the CPU and the GPU. The implementation of heightfield approximations will also be presented and discussed. The report is written as a compliment to a project conducted in the course TSBK03, Techniques for Advanced Computer Games, at Linköpings University.

1.1 Offline Rendering vs Real Time Rendering

The natural phenomenon of fluids has always been appealing to simulate in computer graphics. The famous Navier-Stokes equations successfully creates physically realistic fluids but often to the cost of high computational cost. When doing offline rendering for movies this is often not a big issue. However in games when the simulation needs to be done in real time it becomes a larger issue. Furthermore, in games the most important features would not always be for the fluid to be physically correct but rather that it looks convincing and is cheap to compute. This has inspired multiple simpler methods for simulating water. One popular method of simulating water is scrolling textures while another is to simply use noise textures to deform the surface. These kinds of method are often sufficient, however they can not be interacted with. An interesting interactive method is to use a 2D implemented water simulation, using the Navier-Stokes equations, with heightfield approximations[2] to create the illusion of waves but to a lower computational cost.

1.2 Grid System vs Particle System

In the project a grid-based system was used where the velocity, density and pressure was tracked at each point in the grid. When using a particle system these quantities are instead stored in a particle that moves. The two systems have different advantages. For instance, when using a particle system the breaking of waves can be simulated and the system often has a faster computational time. While using of a grid system, the visual quality is often higher.

2 Background

The Navier-Stokes equations describes how the flow of a fluid changes over time and is described by a vector field **V** [3]. The Navier-Stokes equations are

$$\frac{\delta V}{\delta t} = F + v\nabla^2 V - (V \cdot \nabla)V - \frac{p}{\rho} \tag{1}$$

$$\nabla \cdot V = 0 \tag{2}$$

The equations consist of multiple terms that will be presented in this section.

2.1 External Force

F is the external force that affect the fluid. This can be gravity, wind, or a character in a game etc.

2.2 Viscosity

$v\nabla^2 V$ is the viscosity and corresponds to how resistive a fluid is to flow. This has a strong correspondence to how thick a fluid is. For example, water has a low viscosity while oil has a higher viscosity. This is also connected to diffusion which is how a fluid spreads and mixes with itself.

2.3 Advection

$(V \cdot \nabla)V$ is the advection term and can be described as the fluids flow inside itself. Since V is time dependent the following differential equation can be used

$$\frac{\delta V_1}{\delta t} = -(V_0 \cdot \nabla)V_0 \tag{3}$$

This calculates V_1 by backwards tracing in time.

2.4 Projection

The pressure field is p and is calculated using the Poisson Pressure equation. Poisson Pressure equation is calculated by using an iterative method called Jacobi iteration. ρ is the constant density of the fluid. $\nabla \cdot V = 0$ is an important constraint that together with the pressure field and density is responsible for remaining the fluid incompressible since a divergence free fluid is volume conserving. In the report this is referred to as the projection step. This is done by subtracting the gradient of p from the velocity field V.

2.5 Boundary Conditions

Two boundary condition are important to consider when solving the projection step. The first condition is the Dirichlet boundary that states that there can be no flow, in or out, of the boundary surface to which n is normal. Meaning it forbids fluid to flow into solid objects. It is defined mathematically as

$$V \cdot n = 0 \tag{4}$$

The second condition is the Neumann boundary condition:

$$\frac{\delta V}{\delta n} = 0 \tag{5}$$

Which forbids any flow along the normal direction of a solid surface.

3 Method

An 2D implementation of water simulation using Navier-Stokes equations was created first on the CPU and later on the GPU. The implementation was done in the game engine Unity[1].

3.1 Implementation on the CPU

The implementation of the water simulation follows the examples and explanations presented in [5] and [1].

The implementation was made in a grid based system with each square corresponding to a point which is able to change color depending on the density. These were made by two nested for-loops instantiating a 1x1 square, N * N times, in a square grid. Multiple arrays were created for calculating the velocity and density of each square and how they change over time. These arrays being the current velocity, the previous velocity, the external force, and the density, all being N * N big with each element corresponding to a point in the grid.

The steps in the simulation are as follows: Calculate the diffusion of the edges, set the points to equilibrium with projection, calculate the advection of the edges, set the points to equilibrium again, calculate the diffusion of non-edge points, and lastly calculate the advection of the non-edge points. When calculating the boundaries the velocity is set into the opposite direction to the point next to it and the corners pointing towards the center of the grid to ensure the fluid doesn't "leak out".

3.1.1 Gauss-Seidel Relaxation

The system uses the Gauss-Seidel Relaxation method to solve linear equations that gets created. Gauss-Seidel uses values that gets closer and closer to the correct result for each iteration.

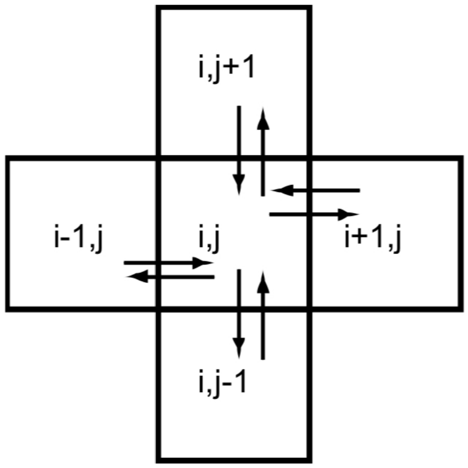


Figure 1: Linear interpolation of surrounding points.

3.1.2 Diffusion

To create a stable solution to the diffusion a backtrack solution is necessary to ensure the system is stable. This is done by using equation 6.

$$d(x, y) = dPrev(x, y) + \frac{a * (d(x+1, y) + d(x-1, y) + d(x, y+1) + d(x, y-1)}{1 + a} \tag{6}$$

Where the function d is the density, $dPrev$ is the density of the previous time step, and a is the amount of change. The density of the same position in the previous time step is calculated and then the linear interpolation of the surrounding points as seen in figure 1. The numerator is solved using the Gauss-Seidel relaxation method.

3.1.3 Advection

The advection is also done by tracing back in time to the previous time step, we find the points that after a time step ends up exactly at the center of each square and then calculate the density that this point carries. The backtracking is done by reversing the direction of the velocity and ending

[1]https://unity.com/

up at the point the velocity now points at in the previous time step. The density of the found point is in the previous time step is used, see equation 7, using the density and velocity (d and v), size (N), and time step (dt).

$$result(x, y) = d(x, y) - v(x, y) * N * dt \tag{7}$$

By using a similar interpolation as in figure 1 but weighting the surrounding points, of the newly found point, by how close they are we can achieve the correct value for advection.

3.1.4 Projection

The projection was done by first calculating the divergence of each point in the grid, using equation 8, taking the difference of the x-velocity of the points that's to the left and right and the difference in y-velocity of the points that's above and below.

$$diver(i, j) = vX(i+1, j) - vX(i-1, j) + vY(i, j+1) - vY(i, j-1) * \frac{-1}{2 * N} \tag{8}$$

By again using the Gauss-Seidel relaxation method the Poisson equation can be calculated and the output of the height field is gained, as discussed in 2.4. The last step is to subtract the gradient from the height field from the velocity which creates a mass conserving field, this is done using equation 9 and 10.

$$vX(i, j) -= 0.5f * n * (p(i + 1, j) - p(i - 1, j)) \tag{9}$$

$$vY(i, j) -= 0.5f * n * (p(i, j + 1) - p(i, j - 1)) \tag{10}$$

3.1.5 Adding Density and External Force

After implementing a working simulation additional functionality was added. A functionality to add density and input an external force was then implemented. The add density function was done by obtaining the correct point which the user mouse position is on, and thereafter increasing the value at that point in the density array when the user presses the left mouse button. The external force function was done by saving the mouse position of the previous frame and calculating the difference between the current mouse position as well as the previous. The difference was then added to the value in the velocity array corresponding to the previous mouse position. In this way, the user affects the velocity field by holding down the right mouse button while dragging the mouse around.

3.1.6 2.5D

By increasing the points position in the z-axis, depending on the density value the two dimensional grid turns into 2.5 dimensions. This creates a real-life wave effect when viewing the simulation from the side or with a non-orthographic camera.

3.1.7 Debug-mode

To ensure the velocity array behaviour was correct a ”debug-mode” was created. To each square a triangle was added which is rotated to point in the direction of the velocity array.

3.2 Implementation on the GPU

With knowledge about the Navier-Stokes equations a further implementation of the simulation on the GPU was done, by following the approach discussed in GPU Gems [4]. Since the simulation represented a 2D grid, textures could be used to store the velocity field and pressure field as discussed in section 2. To replace the for-loops, used in the CPU implementation for iteration, the frame buffer was used. Two textures were implemented for each of the field textures, one for reading off and one for writing to. When rendering to the texture it is used as the frame buffer to be able to write to it, then the texture is copied from the frame buffer to the texture. A swap is thereby made, making the writing texture the reading texture, this is necessary since the frame buffer can not write to a texture that its currently reading [4]. The overall order of computations for one time step can be expressed by the following pseudo code:

```
Update(){
    AddBoundaries();
    Advection(src_tex[READ], dest_tex[WRITE]);
    Swap(src_tex[READ], dest_tex[WRITE]);

    Convection(scr_tex[READ], dest_tex[WRITE]);
    Swap(scr_tex[READ], dest_tex[WRITE]);

    if(clicked){
        AddExternalForce(scr_tex[READ], dest_tex[WRITE]);
        Swap(scr_tex[READ], dest_tex[WRITE]);
    }
    ComputeDivergence();
    for (int i = 0; i < iterations; ++i)
        {
            LinearSolver(scr_tex[READ], dest_tex[WRITE]);
            Swap(scr_tex[READ], dest_tex[WRITE]);
        }
    SubtractGradient(scr_tex[READ], dest_tex[WRITE]);
    Swap(scr_tex[READ], dest_tex[WRITE]);
}
```

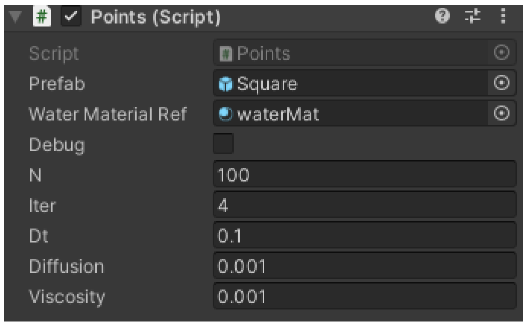
Where the *src-tex* and *dest-tex* refers to the velocity field texture, the pressure field texture or the density field texture. Each function corresponds to a shader program doing the calculations as discussed in section 2 and saving it in the write texture. After each step the swap of the textures is performed.

4 Result

The result of the implementation was a fluid simulation on the CPU with height approximation as well as a fluid simulation on the GPU without height approximation.

The implementation on the CPU resulted in the ability to add color and move it around using left and right mouse click respectively. The user has the ability to change settings that affects the fluid simulation. These settings can be seen in figure 2 and are:

- Prefab: what prefab to use to span the grid
- Water Material Ref: what material is placed on each prefab
- Debug: To enable or disable debug-mode
- N: Size of one axis in the grid, the grid is always N * N big
- Iter: Amount of iterations in the linear solver
- Dt: The time step
- Diffusion: What diffusion coefficient the simulation will use
- Viscosity: What viscosity coefficient the simulation will use



Points (Script)

Script	Points
Prefab	Square
Water Material Ref	waterMat
Debug	
N	100
Iter	4
Dt	0.1
Diffusion	0.001
Viscosity	0.001

Figure 2: The starting settings the user can change.

Still images of the resulting simulation can be seen in the figures 3, 4 and 5 below, using different views to show the 2.5D effect of the CPU implementation. The result of the GPU implementation was a similar simulation where color was added with a left mouse click and then moved through a constant force towards the upper border. The result can be seen in figure 6.

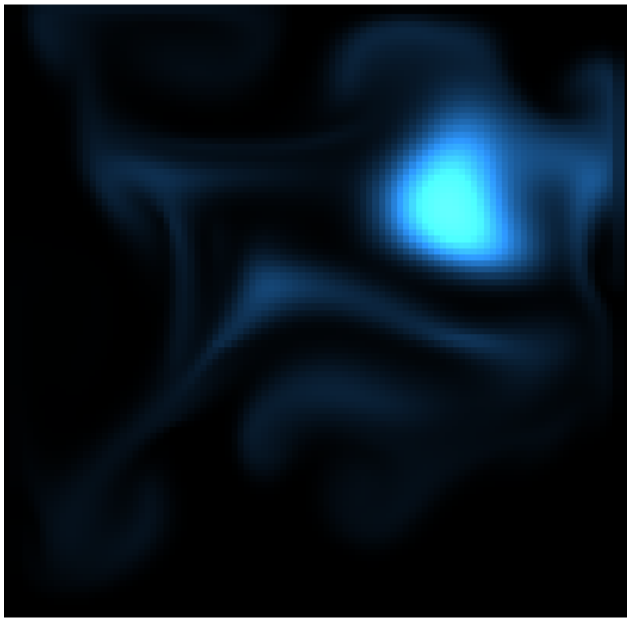


Figure 3: Result of the CPU implementation with an orthographic view

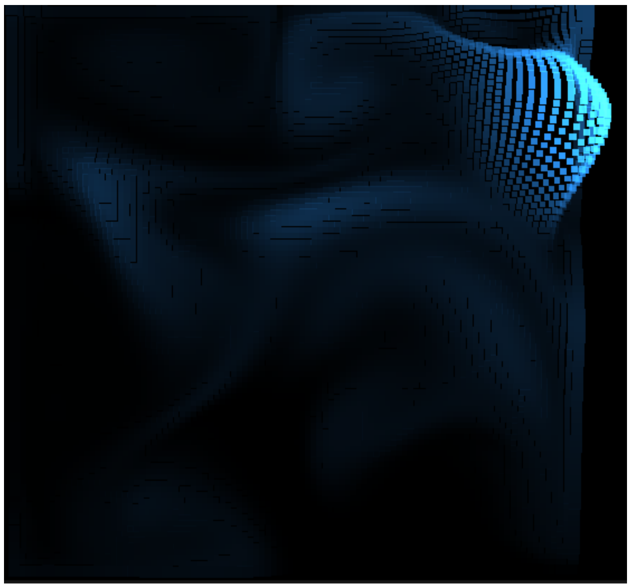


Figure 4: Result of the CPU implementation with a perspective view

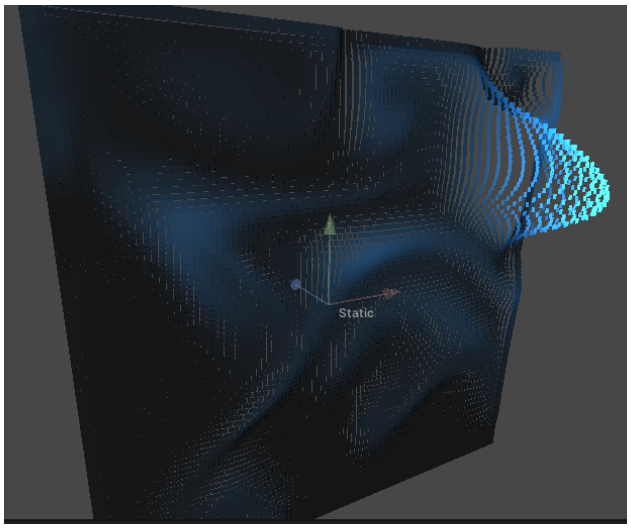


Figure 5: Result of the CPU implementation with a perspective view from side

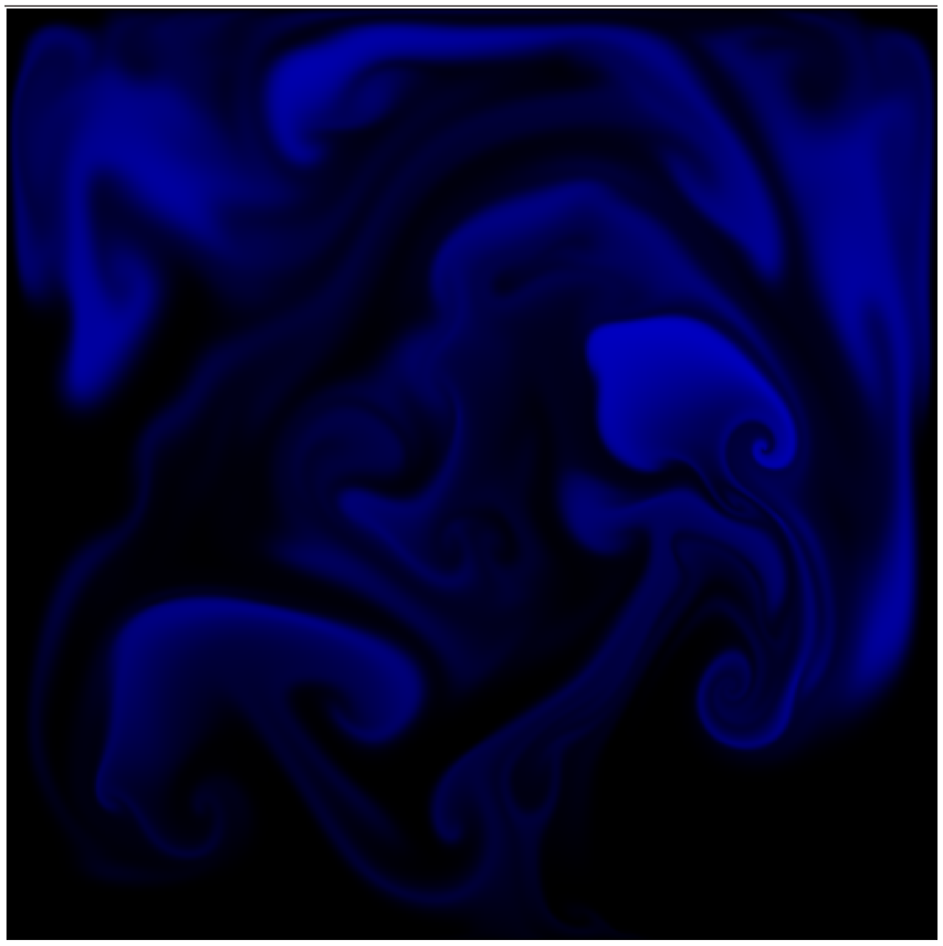


Figure 6: Result of the GPU implementation

5 Discussion

The resulting simulations both had their advantages as well as their disadvantages. The water simulation in 2D on the CPU used heightfield approximations depending on the pressure which yielded more realistic result. However the simulation had poor performance. The GPU simulation was implemented without heightfield approximations but with a better performance. Heightfield approximations made a big difference in the appearance of the simulation and seem like a sufficient method to use for a more realistic result.

The complexity of the code was greater with the GPU since the language is often harder to understand and more of the functionality is done out of sight. For some games this technique can seem unnecessarily complicated and simpler tricks as rolling textures can be sufficient.

The results have no values proving that the implementation on the GPU is better than the our implementation on the CPU. But from what we see we can say that the results have a higher frame rate at larger sizes. This is because allocating each thread of the GPU to each point in grid is a much more time efficient method than calculating each point separately using nested loops.

5.1 Further Work

With more time it would have been interesting to investigate how the same fluid simulation could have been implemented using Unity compute shaders[2] to achieve better computation time with heightfield approximations implemented. It would also be interesting to make the simulation more realistic by changing the shape of the boundary shape to a lake or river etc. with a constant flow and external forces. Integration with other obstacles such as branches and other fluids, would also be interesting to implement if more time had been allocated to the project.

5.2 Conclusion

Implementing a fluid simulation with Navier-Stokes on the CPU works well in smaller scales. The exponential compute power that is needed for creating

[2]https://docs.unity3d.com/ScriptReference/ComputeShader.html

a larger scale realistic simulation is unavoidable. Although it is often harder to understand and implement, the implementation on the GPU provided a better result than the one on the CPU.

References

[1] Mike Ash. Fluid simulation for dummies, 2006.

[2] Robert Bridson. *Fluid simulation for computer graphics.* A K Peters, 2008.

[3] Mark Eric Dieckmann, Robin Skånberg, and Emma Broman. Fluid simulation, 2021.

[4] Mark J. Harris. Gpu gems, chapter 38. fast fluid dynamics simulation on the gpu.

[5] Jos Stam. Real-time fluid dynamics for games. In *Proceedings of the game developer conference*, volume 18, page 25, 2003.