Stockholms universitet

Simulating Shallow Bodies of Water in Video Game Environment

Simulering av Grunda Vatten i Datorspelsmiljö

Susanna Lind

Handledare: Lukas Lundgren, Niklas Terörde

Examinator: Lars Arvestad

Inlämningsdatum: 2024-08-20

Kandidatuppsats i datalogi
Bachelor Thesis in Computer Science

Abstract

In computer graphics in general, and in video game development specifically, achieving realistic and interactive water simulation is a significant challenge. The computational complexity and hardware constraints make real-time, physically-based fluid dynamics difficult to implement. This thesis investigates the techniques used to simulate water in video games, focusing on shallow water bodies. It explores the historical evolution of fluid animation, modern commercial implementations, and surveys how realism is balanced with computational efficiency. By analyzing the current methods used in computer graphics to simulate water in real-time video game environments, this study aims to identify the challenges and advantages surrounding the different methods. A practical implementation is conducted using the Unity game engine, demonstrating the application of selected techniques. The findings highlight the trade-offs between perceived realism and performance, offering insights into creating visually convincing water simulations under the constraints of current gaming technology.

Sammanfattning

Inom datagrafik i allmänhet, och inom utveckling av datorspel i synnerhet, är det en utmaning att uppnå realistisk, interaktiv vattensimulering. Den beräkningsmässiga komplexiteten och hårdvarubegränsningarna gör det svårt att implementera fysikaliskt baserad vätskedynamik i realtid. Denna uppsats ämnar att undersöka de tekniker som används för att simulera vatten i datorspel med fokus på grunda vattenmiljöer. Den utforskar den historiska utvecklingen av vätskeanimation, moderna kommersiella implementeringar och undersöker hur realism balanseras med beräkningseffektivitet. Genom att analysera de nuvarande metoderna som används inom datagrafik för att simulera vatten i realtidsmiljöer i datorspel, syftar denna studie till att identifiera de utmaningar och fördelar som är förknippade med de olika metoderna. En praktisk implementation genomförs med hjälp av Unity-spelmotorn, där tillämpningen av utvalda tekniker demonstreras. Resultaten belyser de kompromisser som görs mellan upplevd realism och prestanda, och ger insikter i hur man kan skapa visuellt övertygande vattensimuleringar inom ramen för den nuvarande spelteknologins begränsningar.

Contents

1 Introduction

The behavior of fluids, including water, is chaotic. Any change to the water will ripple out onto itself, creating a chain of cause and effect that echoes out through the body, causing waves, ripples, droplets and perhaps separation of the body into many more bodies, a fine mist, or being poured into another container. The very nature of this phenomena, since it is so adaptable to change, makes it complex to simulate. Every part of the water affects the next, and the surface of the water can be divided in one instant and be rejoined in the next.

Simulating complex physical phenomena like water flow is a complicated problem. The governing equations of the motion of fluids have been widely studied in modern science, but analytic solutions are only available for very specific configurations [12]. Physically based water simulation requires the use of numerical methods to solve systems of partial differential equations, and these methods are computationally costly [5]. The restraints of hardware and software therefore directly impacts the resolution of the simulation. By writing more efficient code, time and memory complexity can be reduced, larger datasets can be computed, and a higher resolution can be achieved. A method also needs to be stable and accurate, as to properly mimic the phenomena it seeks to imitate and not diverge from the solution. However, when creating a virtual world for a video game the most important aspect of the simulation is not the accuracy of the simulation but whether the simulation appears to be accurate. Therefore, different aspects of the water can be simulated differently, creating an illusion of water rather than a physically correct model.

Video games have the unique restrictions that any simulation must take place under a fixed window of time, since the game is updated in "real-time". The difference between real-time rendering and pre-rendered graphics is the time of which the rendering process takes [37]. To achieve a photo-realistic effect the minimum rendering speed is 24 frames per second [37]. Anything that is rendered in more than this is considered pre-rendered graphics. A scene in a game can contain numerous different objects that need to be simulated and rendered at the same time, some pre-rendered and some rendered in real-time. Given the fixed frame rate, a simulation algorithm must maintain stability across diverse scenarios to ensure a consistent and reliable experience for players.

To render a simulation algorithm suitable for video games, the algorithm therefore needs to be stable and cost-effective in computation. Another aspect is low memory consumption, as it directly influences computational complexity.

1.1 Problem Statement

This essay aims to survey the different methods to simulate water, specifically shallow bodies of water, in video games. The essay aims to answer what the potential drawbacks are and what the positive aspects for the different methods are, answering these questions:

- How are fluid dynamics applied in video games for shallow bodies of water?
- What are the challenges in simulating water in a video game environment?
- What are the techniques used for water simulation?
- How can one implement a water simulation with available methods and techniques?

The main source for this essay is the book "Fluid Simulation for Computer Graphics 2nd Edition" by Robert Bridson [5]. Various different papers on simulating water in computer graphics have been used to research the application of techniques in video games, as well as contemporary news sources. Furthermore, to research and inquire how to implement the methods, a fluid simulation is implemented in Unity game engine with code written in C#. It is common for developers to implement their simulations in game engines, where many companies implement their own internal game engine. The most common game engine available for download are Unity and Unreal engine. While it's possible to perform the computation for the simulation in parts or even fully leveraging the parallelisation opportunity of the Graphics Processing Unit, "GPU" (via compute shaders written in High Level Shading Language, "HLSL"), this was deemed beyond the scope of this thesis. A light-weight graphics shader was written and simulated in HLSL to illustrate a cost efficient non-physically based fluid simulation.

This thesis is organized as follows. In Section 2, we give a brief historical overview of fluid animation and the current techniques used for simulating shallow bodies of water. In Section 3 we investigate the current techniques used for simulating shallow bodies of water in more detail. In Section 4, we implement a selection of the current techniques in a game engine. Finally, in section 5, we give concluding remarks.

2 Simulating Fluids in Video Games

2.1 A Historical Overview of Fluid Animation

In order simulate a physical phenomenon, such as fluid motion, it is important to start with an appropriate mathematical model. Modern fluid dynamics describing incompressible flow are based on the partial differential equations written in the mid 1800-s by Claude-Louis Navier and George Gabriel Stokes known as the as the incompressible Navier-Stokes equations:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p + \nu \nabla \cdot \nabla \vec{u} = \vec{F} \tag{2.1}$$

$$\nabla \cdot \vec{u} = 0 \tag{2.2}$$

where $\vec{u}$ is the velocity field with three components (u, v, w), ρ is the density of the fluid (for water, $\rho \approx 1000 \frac{kg}{m^3}$), p is pressure force per unit area, ν is viscosity and $\vec{F}$ is the force acted on the fluid [5]. Equation 2.1 is called the **momentum equation**, and describes how a fluid accelerates from forces acting on it. The equation is derived from applying Newtons second law, $\mathbf{F} = \mathbf{ma}$; the force is equal to the mass times the acceleration, to fluid motion. On the left-hand side the first two terms make up an advection equation; how the velocity field $\vec{u}$ moves. The following terms describe the forces acting on the fluid. The third term is the gradient of the pressure field. The forth term determines how much or little the fluid acts force on itself due to its viscosity. Lastly on the right hand side are the body forces acting on the fluid, such as gravity. Equation 2.2 is called the **incompressibility condition** and states that the divergence of the velocity field must be zero. This means that for any point in the velocity field the inflow to that point must be equal to the outflow; there cannot be a point in which the fluid collapses on itself or generates more fluid from itself. The two equations combined describe the nature of incompressible flow, and serve as the mathematical model for a fluid such as water.

There are several different techniques to solving this partial differential equation, or PDE, numerically so that the solution is stable and accurate up to a certain degree [21]. However, from a visual point of view in graphics, the most important aspect of the method is not how small the error is, but how stable and fast the method is and whether or not the solution looks good [33]. Therefore, given the limitation

from computational hardware, historically, the solutions to render water were achieved by simulating something that can be perceived as water rather than simulating a physically accurate fluid.

In the 80's, perturbating the surface normal of a plane was introduced to make the appearance of a moving fluid [17]. In the animated short "Carla's Island" by Nelson Max [26], released in 1981 [17], the effect of waves is created by adding up a series of cosines to mimic wave trains:

$$f(x, y, t) = -h + \sum_{i=1}^{m} a_i cos(r_i x + s_i y - w_i t), \tag{2.3}$$

where a_i represents the amplitude of the wave train specified by a wave vector (r_i, s_i) such that $r_i^2 + s_i^2 = k_i^2$, w_i is the angular frequency in radians/s, h is the distance of the mean sea level below the eye at $z = 0$, and m is the number of wave trains [17].

However, viewing figure 2.1, the waves appear to cut into the shoreline. Later this model was extended to take into the height of the sea floor to accurately model waves near beaches [17]. While these methods are cheap to compute and can simulate a believable water surface, they cannot handle more complex three dimensional phenomena as flow around objects and dynamically changing boundaries [11].

Figure 2.1: Waves cutting into island shore, Carla's Island, 1981 [26].

To ensure this more complex dynamic behaviour, research began exploring solving equations (2.1) and (2.2) over a grid [17]. In 1990 Kass and Miller [19] introduce a water simulation based on the linearized form of the shallow water equations; a simplified form of equations (2.1) and (2.2), where the vertical velocity is considered to be negligible due to the shallowness of the body of water. The horizontal velocities with their corresponding height are then mapped onto a height field representing the surface [17]. For a height field, every point (x, y) is mapped onto one point (z) [11]. This makes it impossible to simulate breaking waves, foam or bubbles which requires multiple z-coordinates for each water interface. Also, this model cannot easily incorporate floating objects and buoyancy effects since the velocity is only calculated on the surface [14].

Further exploring simulations based on solving equations (2.1) and (2.2) numerically over a grid, Metaxas and Foster [14] in 1996 simulate 3D velocities throughout a volume. The approach used a Marker-and-Cell (MAC) method where marker particles mark cells in the grid as fluid, surface or empty. However, the discretization scheme

used is stable only when the time step is sufficiently small. Therefore, for small separations and/or large velocities, very small time steps have to be taken [33]. An unconditionally stable solver for fluid dynamics in computer graphics is introduced by Jos Stam [33] in 2001.

Instead of solving equations (2.1) and (2.2) over a grid, particle systems can be implemented. The Particle-In-Cell (PIC) method is introduced by Harlow in 1957 [5] where particles are used to track the movement of the fluid over a grid. The method is further developed by Brackbill and Ruppel in 1985 [4] with Fluid-Implicit-Particle, "FLIP", implementing an adaptable grid. A completely gridless approach to simulate compressible flow is the Smoothed Particle Hydrodynamics method, "SPH", introduced in 1977 [29]. This method is extended to incorporate incompressible flow in 1996 in the method Moving Particle Semi-Implicit, "MPS" [29].

Continuing into the millennia, physically based fluid simulation methods are labelled either Lagrangian methods or Eularian methods, depending on how the solutions to the equations are modeled [5]. In Lagrangian methods velocities and densities are stored on particles moving in space, while in Eulerian methods the quantities are stored on a grid. There are also hybrid methods, where both particles and grids are implemented [17]. In modern times, the methods, hardware and software have evolved to where we now see these numerical solutions to the PDE:s applied in pre-rendered graphics but also in real-time rendering for video games, both in the form of the simplified shallow water equations but also as solutions to the full Navier-Stokes equations, in three dimensions as well as two. However, these physically based methods are often paired with extra effects to model different parts of the water, such as the foam and droplets on the surface.

2.2 Modern Commercial Fluid Implementations

In this section, some examples of the current state-of-the-art implementations of fluid simulations are given.

2.2.1 Shallow Water Equations

One of the latest releases in simulating water for shallow bodies is the plug-in "Fluid Flux"[20], see Figure 2.2, for Unreal Engine, which utilises the shallow water equations coupled with predefined wave profile animation to simulate wave breaks. The waves in the open world game "Horizon: Forbidden West" were modeled in a similar fashion. [32]

Lightspeed Studios "Photon Water System" for Unreal Engine also uses the shallow water equations to simulate water together with a grid-based foam solution that can be generated and moved based on the velocity field from the water. [32].

Figure 2.2: Water flow on rock formations, filling up towards sea shore, Fluid Flux [20].

2.2.2 Eulerian and Lagrangian Methods

Figure 2.3: Particle based simulation of a waterfall falling on top of a zebra, Zibra Liquids by Zibra AI [1].

Zibra Liquids, by Zibra AI [1], a plug-in for Unity and Unreal engine, includes a particle based three dimensional water simulation in real time, also capable of simulating fluids of different viscosity, see Figure 2.3.

Similarly, Niagara Fluids by Epic Games [39] supports three-dimensional real-time liquid effects, including shallow water simulations and the simulation of various liquids such as gas, fire, and more viscous fluids. Epic Games encourages the use of the three-dimensional effects for pre-rendered "hero effects and cinematics" [39], rather than real-time use, due to the overwhelming computational cost. Instead, because of higher efficiency, the two-dimensional models are recommended for in-game use [39]. The gases are simulated on a grid, while the liquids are simulated using FLIP [39].

2.3 The Role of Water and Perceived Realism

Concerning perceived realism one can look at graphic realism, inferential or imaginative realism and enactive realism [22]. Graphic realism concerns how realistic or lifelike objects look, inferential or imaginative realism concerns the accuracy and plausibility of events, characters and environment, and lastly enactive realism concerns the accuracy and plausibility of the interaction with the interface, controller and other characters [22]. However, it should be mentioned that given the subjective nature of the topic, there has been issues to both accurately define this categorical division and in extension to empirically research how these different types of realism affects players enjoyment in games [31].

Figure 2.4: Water fall texture rendered on plane, Red Dead Redemption 2 [36].

In this essay it will be assumed that graphical realism concerns the look of the water; the lighting refraction, the colour, and level of detail of the surface of the water with foam and other effects. The imaginative and inferential realism is assumed to handle the accuracy and plausibility of the movement of the water, and the enactive realism to be how the engagement with the water is plausible and accurate.

There is a fine balance between perceived realism and computational cost [5], and since there are many different aspects to a scene in a game that all have to be rendered in a fixed window of time, this balance decides what level of graphical, inferential and enactive realism can be achieved. Since graphical realism is assumed to pertain to the look of the water, it is the level of detail, the lighting and the colour of the water that assesses the realism. More complex lighting refraction is more computationally costly, as well as rendering a surface with a higher level of detail.

Imaginative and inferential realism is not apparent until the water engages with other objects. For example, a waterfall, since it falls downward, can still behave accurately and have a level of imaginative and inferential realism if it is a more or less motionless object rendered with a graphically realistic surface that moves, an example of which can be seen in the game "Red Dead Redemption", see Figure 2.4.

However, if the water is moving such as waves on a lake, to achieve inferential and imaginative realism a more complex physically accurate simulation method needs to be applied. Simulating water with the shallow water equations and a height map can simulate water engaging with solid boundaries with a low computational complexity [5]. The game "Hydrophobia", released in 2010 by developers Dark Energy Digital, features a three dimensional real-time water simulation moving through doorways, flooding corridors and adjusting the water level from one room to another, see Figure 2.5. The water effects were an important aspect to the game-play and a main selling point of the game [18]. Another game where water is a main feature is the game "Breakwaters", an upcoming release from Soaring Pixels Games [27]. The ocean is the main aspect of the game where players can manipulate the flow of the water. The real-time three dimensional water simulation flows through openings, occasionally swallows islands, and can be tossed by the player to create a swell [28].

Figure 2.5: Waves from water interacting with the surroundings, filling up a corridor as part of game-play, Hydrophobia [18].

Figure 2.6: Body of water moved around the screen as part of game-play, Fluidity/ Hydraventure [15].

Regarding the enactive realism it is not apparent until the player engages with the water whether it behaves plausibly and accurately. If the water is moved in such a way that a height field is impossible, an Eulerian or Lagrangian simulation method is needed for the water simulation to behave plausibly and accurately. Examples of this are puzzle game "Hydroventure" or "Fluidity" by Wiiware [9], that features a real-time simulation of a two-dimensional body of water moved around by the player to solve different puzzles, see Figure 2.6. The game was released for Nintendo Wii in 2010, and the player uses a hand-motion based controller to tilt the screen and pour the water into different spaces. Similarly, in the game "Puddle" released 2012 on various platforms by developer Neko Entertainment, players guide a fluid through different environments by tilting the screen[10].

3 Methods

In this section we will be looking at the different methods mentioned earlier in more detail. We will mostly consider the techniques and methods presented by Robert Bridson in "Fluid Simulation for Computer Graphics, Second Edition" [5], including the shallow water equations, grid based methods, and particle based methods. The grid based methods, particle methods and hybrid methods, as seen in the three dimensional simulations of Niagara and in Zibra Liquids, see Figure 2.3, are more computationally expensive as well as more difficult to implement. The most expensive part of the physically based methods solving the full Navier-Stokes equations is the calculation of the pressure, which requires solving a large linear system of equations. The shallow water equations, mentioned earlier as implemented in many simulations of small to medium size bodies of water such as the ocean front, smaller puddles and rivers as seen in Fluid Flux and Niagara, are the least computationally expensive. This is because the shallow water equations are a simplified version of the Navier-Stokes equations, where the velocities are only solved horizontally, neglecting the vertical velocity, and the pressure is a function of the height, effectively solving a two-dimensional partial differential equation rather than a three-dimensional. We start by an overview of the grid-based method, continue with the particle- and hybrid methods and end with the shallow water equations.

3.1 Eulerian / Grid-Based Simulation

The Eulerian, or Grid-Based simulation, is a physically based simulation. Quantities such as velocity and pressure are modeled onto a grid and updated for each time-step. The grid cells are marked as either fluid, empty, or solid. If a fluid cell is marked as empty, the velocity for that cell is marked as unknown. For each simulation step all of the velocities on the grid are updated by moving the fluid around and correcting the velocities to make the fluid incompressible. The surface is tracked as the isocontour $\phi = 0$ of an implicit function ϕ. There are many numerical methods for the Navier-Stokes equations [24], but the methods presented in this chapter are the ones recommended by Bridson in "Fluid Simulation for Computer Graphics" [5].

Notation

In a grid based simulation the velocity field $\vec{u}$ is stored as a discretely sampled vector field. Grid element at position (i, j, k) is referred to as $\vec{u}_{i,j,k}$. Time-step n is referred as $\vec{u}^n_{i,j,k}$.

Splitting the equation

The grid based methods are based on the Navier-Stokes equations (2.1) and (2.2), solving the velocities of the velocity field $\vec{u}$ over a grid of discrete points. Since the viscosity of water is very small, the viscosity can be neglected and we are left with the incompressible Euler equations:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{F}. \tag{3.1}$$

The equation can now be split into three parts; advection, body forces and pressure/incompressibility:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = 0 \qquad \text{(advection)}$$

$$\frac{\partial \vec{u}}{\partial t} = \vec{F} \qquad \text{(body forces)}$$

$$\frac{\partial \vec{u}}{\partial t} + \frac{1}{\rho} \nabla p = 0 \text{ s.t } \nabla \cdot \vec{u} = 0, \qquad \text{(pressure/imcompressibility)}$$

as presented in "Fluid Simulation for Computer Graphics" [5]. For the **advection** an algorithm is implemented that takes an arbitrary quantity and advects it through the velocity field $\vec{u}$ for a time interval Δt using a semi-lagrangian method. Both velocity as well as different quantities such as dye are advected.

The **body forces** are added using a forward Euler discretization.

The **pressure/incompressibility** is implemented using a projection algorithm that projects a velocity field to a divergence-free velocity field by subtracting the pressure scalar-field.

The algorithm for a grid based solver first advects the velocities through the velocity field, updates the velocity field, then applies the body forces to the velocities. The last step is the pressure/incompressibility step, ensuring that the velocity is divergence free.

Grid

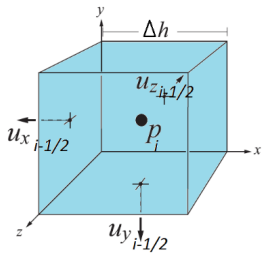

Figure 3.1: Staggered grid with velocities u_x, u_y, u_z on cell faces and pressure p in cell centre[5].

The grid type used commonly in fluid simulations is the staggered grid used in the marker-and-cell, MAC, scheme introduced by Harlow and Welsch in 1965 [16], see Figure 3.1. In this type of grid the velocity field is not stored as a vector in the centre of each cell, but is divided into its components that are stored in the centre of each cell face. Other quantities are sampled in the centre of each cell [5]. The staggered grid is used to avoid stability issues arising from the connection between pressure and velocity.

By sampling the velocity at the cell faces the calculation of the pressure gradient and the divergence is facilitated in the projection step, while also using an unbiased central difference scheme that is accurate to $O(\Delta x^2)$, with Δx as the length of the cell face [5]. The staggered grid however has the disadvantage that the velocity vector at any given point must be interpolated.

At grid locations the interpolation is done by averaging, here showed in 2D:

$$\vec{u}_{i,j} = \left(\frac{u_{i-\frac{1}{2},j} + u_{i+\frac{1}{2},j}}{2}, \frac{v_{i,j-\frac{1}{2}} + v_{i,j+\frac{1}{2}}}{2} \right) \tag{3.2}$$

$$\vec{u}_{i-\frac{1}{2},j} = \left(u_{i-\frac{1}{2},j}, \frac{v_{i,j-\frac{1}{2}} + v_{i,j+\frac{1}{2}} + v_{i+\frac{1}{2},j+\frac{1}{2}} + v_{i+\frac{1}{2},j-\frac{1}{2}}}{4} \right) \tag{3.3}$$

For three dimensions, with grid size N, the velocity components are stored in an array of size $[N \cdot N \cdot (N+1)]$, and pressure, density and other quantities that are sampled at the center of cells are stored in arrays of size $[N \cdot N \cdot N]$ Since half indices can not be used in implementation, Bridson recommends using this formulae:

$$p[i][k][j] = p_{i,j,k} \tag{3.4}$$

$$u_x[i][j][k] = u_{i-\frac{1}{2},j,k} \tag{3.5}$$

$$u_y[i][j][k] = u_{i,j-\frac{1}{2},k} \tag{3.6}$$

$$u_z[i][j][k] = u_{i,j,k-\frac{1}{2}} \tag{3.7}$$

3.1.1 Advection

The advection equation relates to the first two terms in equation 2.1, namely

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = 0. \tag{3.8}$$

Bridson recommends the use of a semi-lagrangian method[35] to solve equation (3.8) [5]. The method was developed to overcome the time-step restrictions associated with explicit time discretizations and allows for unconditionally large time-step [5, 34].

The semi-lagrangian advection method introduces an imaginary particle that is traced backwards through the velocity field, finds the previous location for the particle, and sets the new quantity at the new location to that of the previous. For a given quantity q at position $\vec{x}$, an imaginary particle is introduced at $\vec{x}$. The ordinary differential equation for a particle at $\vec{x}$ moving through the velocity field is defined as:

$$\frac{d\vec{x}}{dt} = \vec{u}.$$

Since the particle is being moved from its previous position $\vec{x}_{prev}$ to its current position $\vec{x}_{current}$ the previous position is found by moving in reverse:

$$\vec{x}_{prev} = \vec{x}_{current} - \Delta t \, \vec{u}_{current}$$

The current quantity $q_{current}$ is updated by setting it to the quantity of the previous position q_{prev}. Since the field is discrete, the values of the surrounding quantities must be interpolated.

Algorithm:

1. Find velocity $\vec{u}$ at $\vec{x}$.
2. Find previous position for particle: $\vec{x}_{prev} = \vec{x} - \vec{u} \cdot \Delta t$.
3. Interpolate the nearest quantities to approximate q^n_{prev} at $\vec{x}_{prev}$.
4. Set $q^{n+1}_{current}$ to q^n_{prev}.

Dissipation because of numerical errors from step 2 and 3 can be improved using different methods. In step 2, the discretization of the differential equation can be made more accurate using a higher-order method such as high-order Runge-Kutta methods.

In step 3, for the interpolation in one dimension, the particle $\vec{x}_{prev}$ is assumed to

be on the interval $[x_i, x_{i+1}]$, with $\alpha = \frac{x_{prev} - x_i}{\Delta x}$, s giving rise to the following linear interpolation:

$$q^{n+1} = (1 - \alpha)q_i^n + \alpha q_{i+1}^n.$$

Because of this averaging, dissipation from numerical errors can occur. Other types of interpolation can be used for higher-order accuracy.

3.1.2 Projection/Pressure Solve

For solving the pressure update, the algorithm that is implemented is called **project**. This algorithm is based on the **Helmholtz-Hodge Decomposition** by Hermann von Helmholtz in 1958, introduced for fluid mechanics by Chorin in 1968 [6] and Temam in 1969 [38] and later incorporated in computer graphics in the late 1990's and early 2000's [2]. The Helmholtz-Hodge Decomposition asserts that any vector field $\mathbf{w}$ can be decomposed uniquely into two components:

$$\mathbf{w} = \mathbf{u} + \nabla q \tag{3.9}$$

where $\mathbf{u}$ has zero divergence ($\nabla \cdot \mathbf{u} = 0$) and q is a scalar field. Any vector field is the sum of a mass conserving field and a gradient field. [33]

With this, the operator $\mathbf{P}$ is defined which projects a vector field $\mathbf{w}$ onto its divergence free part $\mathbf{u} = \mathbf{P}\mathbf{w}$. Applying the divergence operator on (3.9) yields:

$$\nabla \cdot \mathbf{w} = \nabla^2 q \tag{3.10}$$

This equation represents a Poisson equation for the scalar field q, subject to the Neumann boundary condition $\frac{\partial q}{\partial n} = 0$ on ∂D. The solution to this equation is then used to compute the projection $\mathbf{u}$:

$$\mathbf{u} = \mathbf{P}\mathbf{w} = \mathbf{w} - \nabla q \tag{3.11}$$

The projection step involves solving the linear system for the Poisson equation, which is expanded further by Bridson in the text "Fluid simulation for Computer Graphics" to include other boundary conditions than the Neumann condition, namely that the projection routine subtracts the pressure gradient from the velocity field $\vec{u}$:

$$\vec{u}^{n+1} = \vec{u}^n - \Delta t \frac{1}{\rho} \nabla p, \tag{3.12}$$

such that the result satisfies incompressibility (3.13) inside the fluid as well as solid

boundary conditions (3.14):

$$\nabla \cdot \vec{u}^{n+1} = 0 \tag{3.13}$$

$$\vec{u}^{n+1} \cdot \hat{n} = \vec{u}_{solid} \cdot \hat{n}. \tag{3.14}$$

The idea of the update is simple. After the pressure update, the vector field $\vec{u}^{n+1}$ is supposed to be divergence free, and therefore satisfies (3.13). To find the values of pressure and velocity that ensure this we first discretize the divergence using finite differences; here in two dimensions with $\vec{u} = (u, v)$, for ease of reading:

$$\nabla \cdot \vec{u}^{n+1} \approx \Big(\frac{u^{n+1}_{i+\frac{1}{2},j} - u^{n+1}_{i-\frac{1}{2},j}}{\Delta x}\Big) + \Big(\frac{v^{n+1}_{i,j+\frac{1}{2}} - v^{n+1}_{i,j-\frac{1}{2}}}{\Delta x}\Big). \tag{3.15}$$

Now, we find the different velocities written as the pressure update (3.12):

$$\begin{aligned}
u^{n+1}_{i+\frac{1}{2},j} &= u_{i+\frac{1}{2},j} - \Delta t \frac{1}{\rho} \frac{p_{i+1,j} - p_{i,j}}{\Delta x} \\
u^{n+1}_{i-\frac{1}{2},j} &= u_{i-\frac{1}{2},j} - \Delta t \frac{1}{\rho} \frac{p_{i+1,j} - p_{i,j}}{\Delta x} \\
v^{n+1}_{i,j+\frac{1}{2}} &= v_{i,j+\frac{1}{2}} - \Delta t \frac{1}{\rho} \frac{p_{i,j+1} - p_{i,j}}{\Delta x} \\
v^{n+1}_{i,j-\frac{1}{2}} &= v_{i,j-\frac{1}{2}} - \Delta t \frac{1}{\rho} \frac{p_{i,j+1} - p_{i,j}}{\Delta x}.
\end{aligned}$$

We can now substitute the velocity components inside equation (3.15) with the discretized pressure updates. Simplifying we arrive at the following equation:

$$\begin{aligned}
\frac{\Delta t}{\rho}\left(\frac{4p_{i,j} - p_{i+1,j} - p_{i-1,j} - p_{i,j+1} - p_{i,j-1}}{\Delta x^2}\right) = \\
- \left(\Big(\frac{u_{i+\frac{1}{2},j} - u_{i-\frac{1}{2},j}}{\Delta x}\Big) + \Big(\frac{v_{i,j+\frac{1}{2}} - v_{i,j-\frac{1}{2}}}{\Delta x}\Big)\right).
\end{aligned} \tag{3.16}$$

Which is exactly the equation (3.10) derived earlier. Now comes the question of enforcing the solid boundary conditions (3.14) as well as the free boundary condition. By enforcing the free boundary condition we simply set the pressure of the adjacent air cell to 0.

To update the solid boundary, we must solve for the pressure outside of the fluid domain. For a a fluid cell (i, j), to find the value for pressure at $p_{i+1,j}$, we can look at the pressure update for the velocity at the border:

$$u^{n+1}_{i+\frac{1}{2},j} = u^{n}_{i+\frac{1}{2},j} - \Delta t \frac{1}{\rho} \frac{p_{i+1,j} - p_{i,j}}{\Delta x}. \tag{3.17}$$

We can substitute $u^{n+1}_{i+\frac{1}{2},j}$ with u_{solid}, and rearranging we get the following linear equation:

$$p_{i+1,j} = p_{i,j} + \frac{\rho \Delta x}{\Delta t}(u^{n}_{i+\frac{1}{2},j} - u_{solid}). \tag{3.18}$$

To enforce these conditions we simply insert them into the pressure equation (3.16):

$$\frac{\Delta t}{\rho}\left(\frac{4p_{i,j} - \left[p_{i,j} + \frac{\rho\Delta x}{\Delta t}(u^{n}_{i+\frac{1}{2},j} - u_{solid})\right] - p_{i-1,j} - p_{i,j+1} - p_{i,j-1}}{\Delta x^2}\right) = - \left(\left(\frac{u_{i+\frac{1}{2},j} - u_{i-\frac{1}{2},j}}{\Delta x}\right) + \left(\frac{v_{i,j+\frac{1}{2}} - v_{i,j-\frac{1}{2}}}{\Delta x}\right)\right)$$

$$\frac{\Delta t}{\rho}\left(\frac{3p_{i,j} - p_{i-1,j} - p_{i,j+1} - p_{i,j-1}}{\Delta x^2}\right) = -\left(\left(\frac{u_{solid} - u_{i-\frac{1}{2},j}}{\Delta x}\right) + \left(\frac{v_{i,j+\frac{1}{2}} - v_{i,j-\frac{1}{2}}}{\Delta x}\right)\right).$$

Simply, the coefficients in front of the pressure index (i, j) is the same as the sum of non-solid neighbours, for a solid wall the velocity is set to the velocity of the solid, and pressure from neighbouring air cells disappear. We now have a system of linear equations for each fluid cell that can be written as a matrix of coefficients A for a pressure vector p that equals the divergence d, written in matrix form $\mathbf{Ap} = \mathbf{d}$. The system is only solved for fluid cells and velocities in air cells are left unknown.

There are several different ways of solving this type of linear system, Bridson recommends using a preconditioned conjugate gradient solver, but multigrid can also be used. For a smaller grid, a simple Jacobian preconditioning can be implemented [34].

3.1.3 Tracking the Surface

Marker Particles

To track what cells are fluid or air, Bridson recommends using marker particles. The particles are emitted in all fluid cells in a jittered pattern such that the density of particles per fluid cell is at least 4 particles per cell. The particles can then be advected using the neighbouring velocities in the velocity field. By entering a cell, the fluid particles then mark the cell as fluid to be included in the pressure update step.

Implicit Surface Function

Because of the inherent square properties of a grid, the surface of the fluid will exhibit voxelized jagged edges. To make a smooth surface, there have been many different approaches; Blinn introduces "Blobbies" [3], constructing a smooth implicit surface $F(\vec{x}) = \tau$, around the particles, with $F(\vec{x})$ defined as

$$F(\vec{x}) = \sum_i k\Big(\frac{||\vec{x} - \vec{x}_i||}{h}\Big), \tag{3.19}$$

where k is a smooth kernel function of choice, $\vec{x}_i$ is a particle, h is a parameter for the extent of every particle [5]. A kernel smoother usually defines weights that decrease in a smooth fashion as one moves away from the target point [30]. The surface is created where $F(x) = \tau$, the τ-isocontour for the function [5].

This approach leads to a surface with round protruding shapes, which to some extent can be smoothed out by increasing the h-parameter, however by increasing it too much details might disappear. For a smoother look Bridson argues for an approach where the implicit surface function is defined as

$$\phi(\vec{x}) = ||\vec{x} - \vec{X}|| - \vec{r}$$

with $\vec{X}$ as a weighted average of nearby particle locations:

$$\vec{X} = \frac{\sum_i k\Big(\frac{||\vec{x}-\vec{x_i}||}{h}\Big)\vec{x}_i}{\sum_i k\Big(\frac{||\vec{x}-\vec{x_i}||}{h}\Big)}$$

and $\vec{r}$ a weighted average of nearby particle radii:

$$\vec{X} = \frac{\sum_i k\Big(\frac{||\vec{x}-\vec{x_i}||}{h}\Big)r_i}{\sum_i k\Big(\frac{||\vec{x}-\vec{x_i}||}{h}\Big)}.$$

The surface is then defined as the set of coordinates $\vec{x}$ such that $\phi(\vec{x}) = 0$. This is usually referred to as the 0-isocontour, or level set, of ϕ.

To render the surface, a level set can be directly ray traced, but it is also common to construct a mesh approximating the surface [5].

3.2 Lagrangian and Hybrid Simulation

3.2.1 Particle-In-Cell

The Particle-In-Cell method, PIC, involves having all of the quantities stored on particles in stead of on a grid [5]. For each step, the particles are moved according to the advection equation. The advection used for Lagrangian advection are bounded by stricter timesteps since the errors accumulate over time, and are not reset like for the semi-lagrangian advection. Bridson recommends using this scheme for the minimization of errors:

$$\vec{k}_1 = \vec{u}(\vec{x}_n) \tag{3.20}$$
$$\vec{k}_2 = \vec{u}(\vec{x}_n + \tfrac{1}{2}\Delta t\vec{k}_1) \tag{3.21}$$
$$\vec{k}_3 = \vec{u}(\vec{x}_n + \tfrac{3}{2}\Delta t\vec{k}_2) \tag{3.22}$$
$$\vec{x}_{n+1} = \vec{x}_n + \tfrac{2}{9}\Delta t\vec{k}_1 + \tfrac{3}{9}\Delta t\vec{k}_2 + \tfrac{4}{9}\Delta t\vec{k}_3 \tag{3.23}$$

The quantities of the advected particles are then transferred to a grid, and the rest of the computations are performed as for a Eulerian grid simulation. The quantities are then transferred back to the particles. The fluid implicit particle method, FLIP, works just like PIC, but instead of replacing the quantities, the change in quantities are interpolated and used to increment the values stored at the particles.

3.3 Shallow Water Equations

Moving on from the more computationally complex methods to a simplified version of a physical based simulation, the idea of the shallow water equations stems from assuming that if a body of water is shallow, one can ignore vertical variations in the velocity field and only solve for the depth-averaged horizontal velocities.[5] This method effectively makes a three dimensional shape from a two dimensional velocity field.

The water surface is represented as a height field $y = h(x, z)$, and the bottom region as $y = b(x, z)$ with the following region defined as water:

$$b(x, z) < y < h(x, z),$$

with water depth d is defined as $d(x, z) = h(x, z) - b(x, z)$. Since the height of the water is constructed in this way the model can only portray one height per surface region, eliminating effects like breaking waves and splashes. Different techniques have been introduced to emulate these features, such as the pre-rendered wave technology seen in Fluid Flux and Horizon Zero Dawn, but also by introducing particle systems to generate foam and splashes [5].

Deriving the Equations

The simplified momentum equation (3.1), used in the previous section, is further reduced to make the shallow water equations. To fully understand how these simplifications are made, we start by looking at the pressure term.

To clarify, in this context we assume that for the spatial coordinates (x, y, z), the y axis is the vertical axis in the simulation. Different game engines use different coordinate systems, where the alternative is to use the z-axis as the vertical component.

From the assumption that the horizontal velocities are larger than the vertical, hydrostatic pressure is assumed:

$$\frac{1}{\rho}\frac{\partial p}{\partial y} = -g.$$

Combined with the free surface boundary condition $p = 0$ for $y = h$, this gives rise to the pressure approximation

$$p(x, y, z, t) = \rho g(h(x, z, t) - y). \tag{3.24}$$

The pressure can now be written as a function of the height h, and does not need to be solved in a large linear system of equations. This significantly speeds up the simulation.

Continuing from the pressure, expanding equation 3.1 and disregarding the vertical velocity component, we are left with the two horizontal velocity components. Since we assume that the vertical velocities are constant, the derivative of the vertical component $v\frac{\partial v}{\partial y}$ equals to zero, reducing the horizontal velocities to

$$\begin{aligned}
\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + w\frac{\partial u}{\partial z} + \frac{1}{\rho}\frac{\partial p}{\partial y} = 0 \\
\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + w\frac{\partial w}{\partial z} + \frac{1}{\rho}\frac{\partial p}{\partial y} = 0
\end{aligned}$$

which consist of advection in two dimensions and the pressure gradient. However, since we earlier deduced that the pressure can be written as equation (3.24), and the horizontal components of the pressure are constant in y we can replace the pressure term in the velocities, arriving at

$$\begin{aligned}
\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + w\frac{\partial u}{\partial z} + g\frac{\partial h}{\partial x} = 0 \\
\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + w\frac{\partial w}{\partial z} + g\frac{\partial h}{\partial z} = 0,
\end{aligned} \tag{3.25}$$

where the velocities are determined by horizontal advection and a gravitational force pulling higher regions of the water towards the bottom.

We can write the change of height as the following equation:

$$\frac{\partial d}{\partial t} + u\frac{\partial d}{\partial x} + w\frac{\partial d}{\partial z} + = -d\left(\frac{\partial u}{\partial x} + \frac{\partial w}{\partial z}\right), \tag{3.26}$$

which is derived from the advection equation in three dimensions for the height of the surface together with an expression for the vertical velocity. In the equation we can see that the depth is moved by the horizontal velocities, and changed proportional to the depth times the divergence.

For the boundary, there are many different approaches to be made. For a solid wall, given the 2 dimensional normal to a wall $\hat{n}$, it is required that

$$(u, w) \cdot \hat{n} = 0. \tag{3.27}$$

This ensures that no water is flowing outside of the domain, and pushed back into the body of water. To handle an open edge, such as matching the shallow water to the ocean, the usual method is to gradually blend out velocities [5] or use a characteristic boundary condition that allows for waves to exit the computational domain [23].

Implementing the Shallow Water Equations

To implement this method, Bridson recommends storing the the horizontal velocities on a 2D MAC-grid, on the faces of each cell. The depth d is stored in the centre of each cell. The height h is constructed as $h = (b + d)$, where b is the height of the bottom.

First the velocities and the depth are advected with a semi-lagrangian advection method, to u^{advect}, w^{advect}, d^{advect}. The next step involves setting the new heights according to the advected velocities and depths, with $h^{\text{advect}} = (b + d^{\text{advect}})$. The height is then extrapolated to non-fluid cells. After this, the velocities are updated with the pressure acceleration as follows:

$$u^{n+1}_{i+1/2,k} = u^{\text{advect}}_{i+1/2,k} - \Delta t g\left(\frac{h^{\text{advect}_{i+1,k}} - h^{\text{advect}}_{i,k}}{\Delta x}\right)$$
$$w^{n+1}_{i+1/2,k} = w^{\text{advect}}_{i+1/2,k} - \Delta t g\left(\frac{h^{\text{advect}_{i+1,k}} - h^{\text{advect}}_{i,k}}{\Delta x}\right),$$

before they also are extrapolated to non-fluid cells. The last step involves updating the depth with the divergence terms, as follows:

$$d_{i,k}^{n+1} = d_{i,k}^{advect} - \Delta t d_{i,k}^{advect} \left(\frac{u_{i+1/2,k}^{n+1} - u_{i-1/2,k}^{n+1}}{\Delta x} + \frac{w_{i,k+1/2}^{n+1} - w_{i,k-1/2}^{n+1}}{\Delta x} \right).$$

4 Implementation

The objective of this implementation was to explore how a water simulation could be achieved using available methods and techniques. The selected methods include a lightweight graphics shader solution and a more complex physically-based simulation.

In Figure 4.1, two sections are highlighted: the programmable vertex processor and the programmable pixel processor. This is a generalized visualization of the graphics pipeline performed on the GPU to render an image or video in computer graphics. The vertex shader manipulates the vertices of a mesh, while the pixel shader modifies rasterized fragments. The graphics shader implemented in this section includes both a vertex shader and a pixel shader, both written in High-Level Shading Language (HLSL).

To harness the computing power of the Graphics Processing Unit (GPU) beyond the rendering pipeline, another type of shader can be utilized; compute shaders. This type of shader can be used to write an arbitrary algorithm that is performed on the GPU completely separate of the graphics rendering pipeline. These shaders are essential for real-time fluid simulations as they enable multiple calculations to occur simultaneously.

For the physically-based simulation, the algorithms were initially implemented on the Central Processing Unit (CPU) before being translated to code that runs on the GPU. This approach allowed for the algorithms to be refined and debugged in a more familiar environment before tackling the more challenging task of writing the code in the shader language associated with the game engine. However, due to time constraints, the implementation of compute shaders was deemed beyond the scope of this thesis. The physically-based simulation implemented here is a two-dimensional Eulerian simulation written for the CPU.

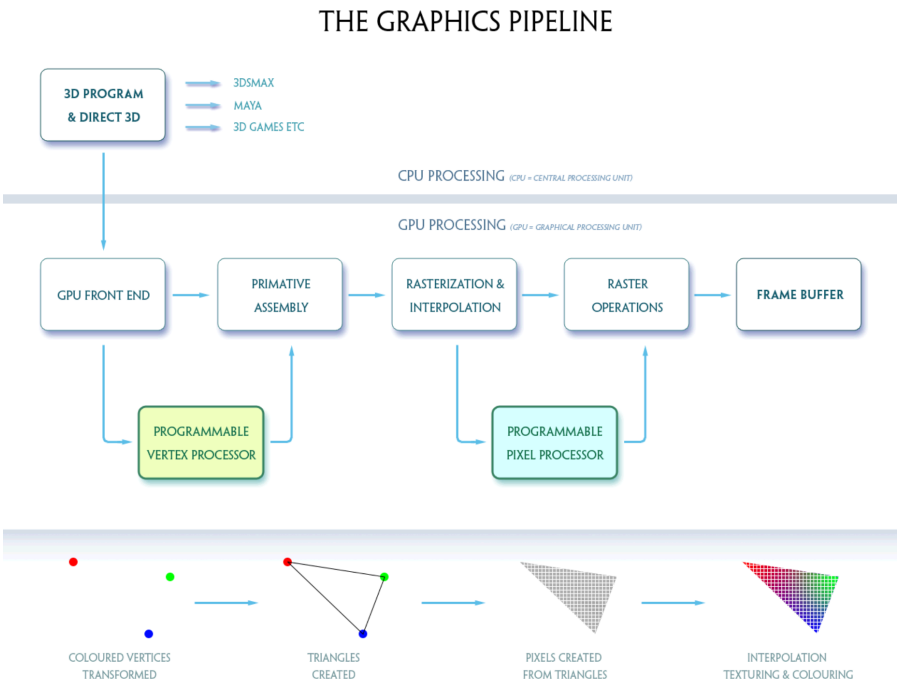

Figure 4.1: The Graphics Pipeline: step by step GPU processing ©Ben Cloward 2019 [7].

4.1 Selecting Game Engine

When selecting the game engine for the project different aspects were considered. Availability of resources and material for learning, previous knowledge of coding language and familiarity with the engine was considered. There are several game engines available such as Unreal Engine, Unity and Godot. When comparing the amount of resources available, Unity arguably had the most material accessible, and was familiar to the author. Scripts in Unity are written in C# and shaders are written in HLSL.

4.2 Rendering a Plane; a Light-Weight Graphics Shader

In Unity, a texture shader is analogous to a vertex shader, and a surface shader is analogous to a pixel shader. An introduction to coding shaders for Unity can be found on the website catlikecoding.com [13], that presents a section on programming shaders for fluid motion.

Arguably the most efficient algorithm to simulate a water like effect is the method of adding sine functions introduced in "Carlas Island", mentioned in Chapter 2 (See equation (2.3)). Any type of periodic function will generate a wave pattern, however, a popular function to be used is the Gerstner Wave, which makes a sharp pointed wave. The Gerstner wave, introduced by Franz Joseph von Gerstner in 1802, was the first exact nonlinear solution for waves of finite amplitude on deep water [8], and are also called periodic surface gravity waves [13].

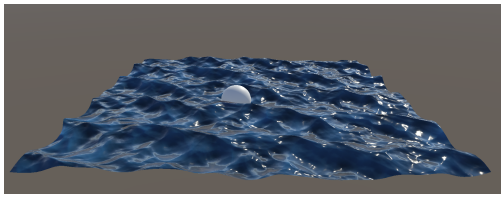

Figure 4.2: Gerstner wave texture shader, rendered in Unity.

Based on a code presented on catlikecoding.com [13], an implementation of a shader for Unity with a series of Gerstner waves was used to create a texture shader. Another shader, a surface shader, combines a noise filter to generate the perception of ripples in a somewhat still water. These two were combined in a shader to create more realistic water effects, see Figure 4.2.

The code provides the user with input parameters in setting the wave directions in (x, y) coordinates, steepness of the wave and wavelength. The simulation currently allows for 6 waves, however, there is no upper bound for the amount of waves that could be added. Since the function is analytical and the derivative is easily found through simple derivation, a normal map to reflect light is easily generated which further adds to the simplicity of the model.

4.3 Grid-Based Approach

Besides Bridson [5] different sources were researched to further understand the nature of the implementation of the theory. An implementation of a grid-based simulation of a wind tunnel by Matthias Müller [25] written for a web browser in HTML and Javascript, used a 2-dimensional staggered grid and a boolean field to mark grid cells as either fluid or solid. Another implementation of a grid-based simulation, "Real-Time Fluid Dynamics For Games" by Jos Stam [34], does not implement a staggered grid but instead samples all quantities in the centre of each cell. The method does not simulate the fluid vertically with a surface but instead implements a less complex horizontal 2-dimensional fluid simulation where the entire domain is fluid.

Both an implementation of Müllers, Bridsons and Stams approach were performed. The implementation of Müllers method was written in Javascript and HTML to run in a web browser, see Figure 4.3. The Bridson and Stam approaches were made in Unity, written in C#.

4.3.1 Wind Tunnel

To illustrate the flow in the tunnel Müller implements a dye-field, an array of floats the same size as the domain ranging from zero to one, that is advected by the surrounding velocities. However, in the code, the projection step is different; Müller updates the velocities by implementing what seems to be a Jacobian solver, but structures the pressure equation a bit differently to the Bridson implementation. For each iteration in the solver, a variable **p** is set to **-div**/**s**, where **div** is divergence and **s** is the number of solid neighbours. The variable **p** is then multiplied by a factor of 1.9 which Müller refers to as a relaxation factor to increase convergence. The velocities on the edges of the cell are updated by subtracting **p** from the velocities on the lower fluid cell edge and left fluid cell edge and adding **p** to the velocities on the right and upper cell edge. The number of iterations is set to 100. Müller seems to disregard gravity and assumes constant density, effectively making a visually successful subsonic gas simulation. The simulation implementation performed can be seen in Figure 4.3.

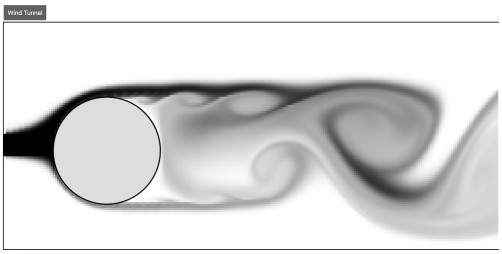

Figure 4.3: Eulerian grid simulation, written in Javascript and HTML.

4.3.2 2-Dimensional Fluid Container

Bridson Approach

When implementing the Bridson approach, the problem was divided into parts; visualisation, force input, advection and projection. The fluid was simulated in a square container, where the fluid grid cells were marked as either fluid or solid. A color was introduced, as in the Müller method, to illustrate the flow. For simplicity, the entire region was assumed to be liquid, without air cells. The goal was to make a vertical simulation with air cells and a fluid with a surface.

Implementing the semi-lagrangian advection in the Bridson approach was similar to the implementation made by Müller. Another aspect driving the velocities into neighbouring cells was the application of forces, such as gravity. By adding gravity, the velocities were pushed downwards towards the bottom of the container, without any boundary. The projection algorithm was needed to enforce boundary conditions. However, when introducing the gravity, the notion of a surface to the fluid became more important.

To achieve a surface, grid cells were meant to be marked as air. Only fluid velocities that were inside the fluid domain were meant to be advected, and velocities outside of the fluid were to be marked as unknown. To be able to simulate velocities entering empty cells, velocities need to be extrapolated from fluid cells to empty cells. The extrapolation of the air boundary would entail implementing marker particles to mark cells as fluid or air.

In Bridson's approach, the projection algorithm is meant to be solved only for the cells that are marked as fluid, not for the entire domain. The task of tackling not only writing a solver for a linear system of equations, where Bridson recommended implementing a preconditioned conjugate gradient solver, but also solving the complex task of constructing the matrix of fluid cells, as well as implementing an implicit function to track the surface, was simply not possible in the time frame for this project.

Stam Approach

Instead, the less complex 2-dimensional fluid simulation without a surface by Stam, where instead a horizontal fluid is solved over the entire grid, was implemented. In this approach, the use of the MAC-grid is discarded for a simpler grid where all quantities are stored in the center of a grid cell. To illustrate a flow dye is added to the simulation such as in the Müller simulation. Stam introduces a density field, however this density field is not used to solve for pressure but merely to illustrate the diffusion of density in the fluid.

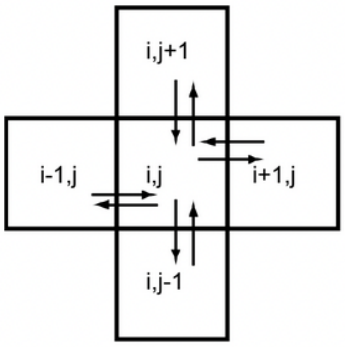

Figure 4.4: Grid cell structure for diffusion solver, Jos Stam [34]

Three fields are updated each timestep, the x velocity component field **u**, the y velocity component field **v**, and the density/dye field **d**.

To propagate the density and the velocities, besides the advection algorithm and the projection algorithm, Stam includes a diffusion algorithm. This diffusion algorithm is an iterative method to solve the quantity of a cell, comparing it to the quantities of its 4 neighbouring cells, see Figure 4.4. For every frame, the cell will lose quantity to its neighbors, while also being added quantity from its neighbours, resulting in a difference of

$$x^n_{i+1,j} + x^n_{i-i,j} + x^n_{i,j+1} + x^n_{i,j-1} - 4 \cdot x^n_{i,j}. \tag{4.1}$$

The rate of diffusion is determined by a factor $a = dt \cdot \mathbf{diff} \cdot N \cdot N$, where **diff** is a diffusion parameter to be set by the user. For a stable method, Stam uses a backwards propagating scheme, where the quantity for cell $x^{n-1}_{i,j}$ is defined by:

$$x^{n-1}_{i,j} = x^n_{i,j} - a(x^n_{i+1,j} + x^n_{i-i,j} + x^n_{i,j+1} + x^n_{i,j-1} - 4 \cdot x^n_{i,j}). \tag{4.2}$$

To find the quantity for cell $x^n_{i,j}$, the terms are rearranged to the following equation:

$$x^n_{i,j} = (x^{n-1}_{i,j} - a(x^n_{i+1,j} + x^n_{i-i,j} + x^n_{i,j+1} + x^n_{i,j-1})/(1 - 4a) \tag{4.3}$$

and solved with an iterative solver. Both velocities and densities are diffused using this algorithm with different diffusion parameters, using the term diffusion for the density field and viscosity for the velocity field.

The solver is divided into two steps; velocity step and density step, where the velocity step updates the velocity fields **u** and **v** and the density step updates the density/dye field **d**. The velocity step begins with the adding of forces, followed by diffusion, then projection and lastly advection. The density step begins with adding of density, followed by diffusion and lastly advection.

The advection is solved by using a semi-lagrangian advection scheme as in the Bridson method. The projection step follows the same algorithm as Bridson, however since the entire domain is a fluid, the projection is solved over the entire domain. Stam implements an iterative Jacobian solver to solve the pressure equation, which is arguably less complex to implement in code than the preconditioned conjugate gradient method recommended by Bridson, and was replicated in the implementation.

The diffusion algorithm is not included in the Bridson implementation nor the Müller implementation. As mentioned in Chapter 3, numerical dissipation arises from interpolation in the advection routine. Tests were conducted by comparing the visual effects in the different simulations where different values for diffusion were set. The initial velocity for the entire domain is set to 0, with an opening in the right hand side were velocities and white dye are injected into the domain. The velocity of the side input is set to 20. Since the fluid is modeled as water, viscosity was set to 0. The rate of diffusion was set to 0.001, 0.0001, and 0.

The top two rows in Figure 4.5 show how the diffusion routine adds diffusion in the dye field, at 1, 3 and 5 seconds of elapsed time. The third row shows a simulation where the diffusion is set to 0.

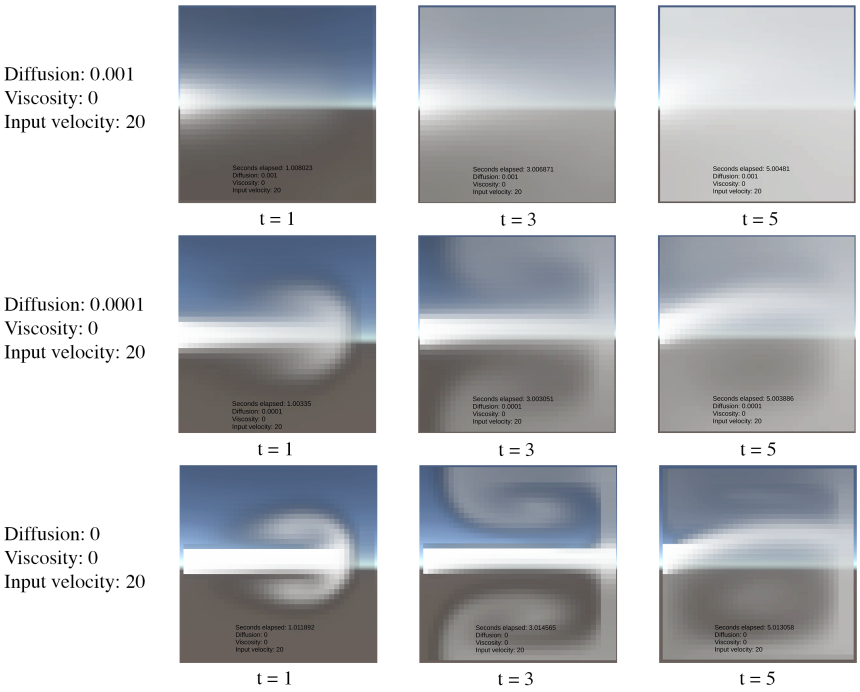

Figure 4.5: 2-Dimensional horizontal fluid simulation implementation of a Eulerian approach at different viscosities. White colour illustrate the incoming flow from the outside perimeter. Background colours from default background in Unity Game Engine.

5 Discussion

5.1 Advantages and Disadvantages of Water Simulation Methods

Arguably there are many ways to asses the quality of a method to simulate shallow bodies of water over another. The complexity of the method as well as the actual visual result and the perceived realism of the method are important factors to take in to account.

The challenges in simulating water in video game environments are partly the actual work of implementing the functions and integrating them into an environment, but also the task of compromising the level of realism to the complexity.

Looking at the current implementations presented in Chapter 2, the shallow water equations is the more popular choice to simulate a shallow body of water in game play, because of the low computational complexity of the methods. As stated in Chapter 2, the more complicated three dimensional simulations were recommended by Epic Games [39] to be used in cinematic episodes or hero effects rather than real-time simulations, because of time management and budget in developing the game as well as the computational complexity when running the game.

When simulating rivers, creeks, puddles or pools for a character to interact in, since the water is usually level, the advantages of the shallow water equations in terms of computational complexity exceeds the added perception of reality that would come from implementing a three dimensional simulation with a surface rendered from an implicit function. Of course, this is a subjective topic.

For something like a waterfall, where the height map is impossible, the only methods available would be the grid- or particle based methods. However, since a waterfall is usually quite stable in its movement, as seen in the example in Chapter 2, a believable effect can be generated by simply animating a surface. Again, one could argue that the added level of realism of interacting with a real-time simulated waterfall does not exceed the added work load and computational complexity.

One could argue that depending on the level of interaction a player has with a body of water, the more or less is the realism or lack thereof evident to the player. Working from this logic, the less interaction that takes place with the water, the less apparent is

the inferential and enactive realism and a less physically accurate and computationally cheaper method can be used. On the other hand, the more a player engages with a body of water the more apparent are these aspects. Since all objects in the game have to be simulated and rendered in every frame in such a way that the game is stable, the amount of memory and time that is awarded to rendering each object of the game is limited, and objects of less importance are thus rewarded.

With this logic, if a scene is constructed to entail water moving from one vessel to another or being moulded and shaped by the player as a part of the game play, such as the examples shown in Chapter 2, naturally the water needs to be simulated in real time using a two- or three dimensional simulation with a surface from an implicit function.

When considering the level of realism versus the complexity of the algorithm, one could therefore conclude that it is the game-play itself that sets these parameters.

The aspect besides computational complexity to consider is the level of difficulty in the actual implementations of the methods. Though this might not be an issue for a larger company looking to include a water simulation, however, for a smaller company or for a single creator this would in fact be a factor to consider, if the available add-ons are not compatible with the game-play wished to be achieved.

When attempting to implement a grid-based method, it became apparent that the level of difficulty to implementing this type of method is high and the implementation takes a long time to realise. To answer the question of how one could implement a water simulation with available methods and techniques; for an introductory programmer with little experience in developing code for video games, perhaps the initial attempt to implement a simulation of shallow body of water would be the shallow water equations. The perturbation of normals in a plane, as the example in Figure (4.2), albeit not a very realistic simulation of shallow water, could also serve as a viable option for a more stylized game.

5.2 Conclusion

There are a wide range of different methods to apply fluid dynamics for shallow bodies of water in video games. The particular setting of a real-time virtual reality in video games produces its specific requirements of a set time frame, but also provides with creative opportunities where a physical phenomena like water can be simulated with greater reigns of freedom. When abandoning physical accuracy, the simulation methods can be combined and puzzled together creating a realistic looking effect.

The challenges in simulating water in a video game environment lies in finding the delicate balance between perceived realism and computational cost. The constraints of hardware and software dictates to a degree how realistic game-play can become,

but the improvements of different aspects in the methods, for example improving the convergence for the projection solver, also pushes the envelope on how water can be realised in a video game. Extending a cheap method such as the shallow water equations with pre-defined wave breaks as mentioned in Chapter 2, is another example of this.

The different techniques of simulation are readily available to study and implement, however, for an independent developer albeit a physically based solver of the full Navier-Stokes equations might seem most enticing, the less computationally complex methods are not only easier to implement but also the industry standard. To get a water simulation up and running, the most efficient method to implement is a light-weight graphics shader such as the one implemented in Section 4.2. Secondly, a bit more computationally costly but with a much higher level of imaginative/inferential and enactive realism are the shallow water equations. Lastly, for an independent developer integrating a two- or three dimensional physically based solver is only worth the extra computational complexity and developmental difficulty if the game-play is centered around the body of water.

Bibliography

[1] Zibra AI. Zibra Liquid — assetstore.unity.com. https://assetstore.unity.com/packages/tools/physics/zibra-liquid-266451. [Accessed 10-05-2024].

[2] H. Bhatia, G. Norgard, V. Pascucci, and P. T. Bremer. The Helmholtz-Hodge Decomposition — a Survey. *IEEE Transactions On Visualizations And Computer Graphics*, 2013.

[3] James F. Blinn. *A Generalization of Algebraic Surface Drawing.* ACM Transactions on Graphics Volume 1 Issue 3, 1982.

[4] J. U. Brackbill and H. M. Ruppel. FLIP: A Method for Adaptively Zoned, Particle-in-Cell Calculations of Fluid Flows in Two Dimensions. *Journal of Computational Physics*, 1986.

[5] Robert Bridson. *Fluid Simulation for Computer Graphics 2nd Edition.* CRC Press, Taylor Francis Group, 2016.

[6] Alexandre Joel Chorin. Numerical Solution of the Navier-Stokes Equations. *Mathematics of computation*, 22(104):745–762, 1968.

[7] Ben Cloward. The Graphical Pipeline - HLSL Shader Creation 1 - HLSL Shader Fundamentals — youtube.com. https://www.youtube.com/watch?v=uabOsiYO2Mlist=PL78XDi0TS4lEDHfahG4ddRwZ3AUrOIYcqindex=3, 2019. [Accessed 19-08-2024].

[8] A. Craik. The Origins of Water Wave Theory. *Annual Review of Fluid Mechanics*, 2004.

[9] Patrick Elliot. Review: Hydroventure — nintendolife.com. https://www.nintendolife.com/reviews/wiiware/hydroventure, 2010. [Accessed 30-07-2024].

[10] Neko Entertainment. Puddle on Steam — store.steampowered.com. https://store.steampowered.com/app/222140/Puddle/. [Accessed 30-07-2024].

[11] R. Fedkiw and S. Marschner. *Animation and Rendering of Complex Water Surfaces.* Stanford, 2002.

[12] Charles L. Fefferman. Existence and Smoothness of the Navier–Stokes Equations.

https://www.claymath.org/wp-content/uploads/2022/06/navierstokes.pdf, 2006. [Accessed 13-08-2024].

[13] Jasper Flick. Waves — catlikecoding.com. https://catlikecoding.com/unity/tutorials/flow/waves/, 2018. [Accessed 15-05-2024].

[14] N. Foster and D. Metaxas. *Realistic Animation of Liquids.* University of Pennsylvania, 1996.

[15] John GodGames. Hydroventure / Fluidity - Wiiware Wii Gameplay — youtube.com. https://www.youtube.com/watch?v=Q2400xtzA-E, 2015. [Accessed 19-08-2024].

[16] F. Harlow and J.E. Welch. Numerical calculations of time-dependent viscous incompressible flow with free surface. *The Physics of Fluids*, 1965.

[17] A. Iglesias. *Computer Graphics for water modelling and rendering: a survey.* University of Cantabria, 2004.

[18] IGN. Hydrophobia — ign.com. https://www.ign.com/games/hydrophobia. [Accessed 01-08-2024].

[19] M. Kass and G. Miller. Rapid, stable fluid dynamics for computer graphics. *Computer Graphics, Volume24, Number4*, August, 1990.

[20] Krystian Komisarek. Fluid Flux — unrealengine.com. https://www.unrealengine.com/marketplace/en-US/product/fluid-flux, 2022. [Accessed 03-05-2024].

[21] S. Larsson and V. Thomée. *Partial Differential Equations With Numerical Methods.* Springer, 2009.

[22] J.H Lin and W Peng. The contributions of perceived graphic and enactive realism to enjoyment and engagement in active video games. *International Journal of Technology and Human Interaction*, 2015.

[23] L. Lundgren and K. Mattsson. An Efficient Finite Difference Method for the Shallow Water Equations. *Journal of Computational Physics*, 2020.

[24] Riccardo Milani. *Compatible Discrete Operator Schemes for the Unsteady Incompressible Navier–Stokes Equations.* Theses, Université Paris-Est, December 2020.

[25] Matthias Müller. Ten Minute Physics: Eulerian Fluid Simulation — github.com. https://github.com/matthias-research/pages/blob/master/tenMinutePhysics/17-fluidSim.html, 2022. [Accessed 16-05-2024].

[26] Nelson L. Max. Carla's Island — www.siggraph.org. https://digitalartarchive.siggraph.org/artwork/nelson-l-max-carlas-island/. [Accessed 09-05-2024].

[27] Soaring Pixels Games. Breakwaters — breakwatersgame.com. https://breakwatersgame.com. [Accessed 01-08-2024].

[28] Soaring Pixels Games. Breakwaters on Steam — store.steampowered.com. https://store.steampowered.com/app/1203180/Breakwaters/. [Accessed 01-08-2024].

[29] S. Premože1, T. Tasdizen, J. Bigler, A. Lefohn, and R. T. Whitaker. Particle-Based Simulation of Fluids. *Eurographics Volume 22 Number 3*, 2003.

[30] Rafael A. Irizarry. Kernel Methods — rafalab.dfci.harvard.edu. http://rafalab.dfci.harvard.edu/pages/649/. [Accessed 15-08-2024].

[31] K. Rogers, S. Karaosmanoglu, M. Altmeyer, A. Suarez, and L. E. Nacke. Much realistic, such wow! a systematic literature review of realism in digital games. https://dl.acm.org/doi/fullHtml/10.1145/3491102.3501875, 2022. [Accessed 30-07-2024].

[32] Arti Sergeev. Developing a Next-Gen Water Rendering Solution for Games — 80.lv. https://80.lv/articles/developing-a-next-gen-water-rendering-solution-for-games/, 2023. [Accessed 03-05-2024].

[33] Jos Stam. *Stable Fluids.* ACM Transactions on Graphics, 2001.

[34] Jos Stam. *Real-Time Fluid Dynamics for Games.* Autodesk, 2003.

[35] Andrew Staniforth and Jean Côté. Semi-Lagrangian Integration Schemes for Atmospheric Models — A Review. *Monthly weather review*, 119(9):2206–2223, 1991.

[36] T00N. Red Dead Redemption 2 - Waterfall Ragdolls Vol.1 — youtube.com. https://www.youtube.com/watch?v=zmksKrKqNMk, 2019. [Accessed 19-08-2024].

[37] Unity Technologies. Real-time rendering in 3D — unity.com. https://unity.com/how-to/real-time-rendering-3d. [Accessed 09-05-2024].

[38] Roger Temam. Sur l'approximation de la solution des équations de navier-stokes par la méthode des pas fractionnaires (ii). *Archive for rational mechanics and analysis*, 33:377–385, 1969.

[39] Unreal Engine 5.2 Documentation. Niagara Fluids Reference In Unreal Engine — dev.epicgames.com. https://dev.epicgames.com/documentation/en-us/unreal-engine/niagara-fluids-reference-in-unreal-engine. [Accessed 10-05-2024].

List of Figures

Datalogi
www.math.su.se

Beräkningsmatematik
Matematiska institutionen
Stockholms universitet
106 91 Stockholm