Improving the Realism of Real-Time Simulation of Fluids in Computer Games

Jens Christian Morten Laursen

School of Information and Communication Technology
Aalborg University

A thesis submitted for

10th Semester
Medialogy Group mta131037

30st May 2013

For a long time it puzzled me how something so expensive, so leading edge, could be so useless, and then it occurred to me that a computer is a stupid machine with the ability to do incredibly smart things, while computer programmers are smart people with the ability to do incredibly stupid things. They are, in short, a perfect match.

Bill Bryson

AALBORG UNIVERSITET

School of Information and Communication Technology

Selma Lagerlöfsvej 300
9220 Aalborg Øst
Danmark
Telefon: 99 40 72 28
http://www.sict.aau.dk

Titel: Improving the Realism of Real-Time Simulation of Fluids in Computer Games

Projektperiode: 04.02.13-30.05.13

Semester tema: Speciale

Vejleder: Martin Kraus

Projektgruppe nr.: mta131037

Deltagere:

Jens Christian Morten Laursen

____

Kopier: 2

Sider: 66

appendiks: 1

Synopsis:

Dette speciale dokumenterer udviklingen af en væske simulering til Unity3D spil motoren, baseret på Navier-Stokes ligningerne implementeret på en 2D Eulerian gitterstruktur kombineret med et højde felt. Fokus for denne simulering er en real time og grafisk realistisk væske simulator der kan håndtere forskellige typer væsker, samt interaktionen mellem væske og objekter med varierende massefylde. Dette speciale giver en introduktion til fysikken bag grafisk væske dynamik, en kort beskrivelse af forskellene på eksisterende simuleringer, en beskrivelse af emner, der er relevante for dette speciale, samt en beskrivelse af de ligninger der som er blevet brugt til at skabe en væske simulering.
Den resulterende simulering kan håndtere mindre mængder væske med justérbar tyktflydenhed, bruger-tilført kraftpåvirkning, to-vejs interaktion mellem væske og objekter med varierende massefylde ved interaktive hastigheder ($>$ 60 FPS).

AALBORG UNIVERSITET

School of Information and Communication Technology

Selma Lagerlöfsvej 300
9220 Aalborg Øst
Denmark
Telephone: 99 40 72 28
http://www.sict.aau.dk

Title: Improving the Realism of Real-Time Simulation of Fluids in Computer Games

Project period: 04.02.13-30.05.13

Semester theme: Master Thesis

Supervisor: Martin Kraus

Projectgroup no.: mta131037

Members:

Jens Christian Morten Laursen

Abstract:

This thesis documents the development of a fluid simulation for the Unity3D game engine, based on the Navier-Stokes equations implemented on a 2D Eulerian grid combined with a height field. The focus of the simulation is a real-time and graphically realistic simulation, capable of handling different types of fluid as well as the interaction between fluid and solids with various densities. The thesis includes an introduction to the physics of fluid dynamics for graphics, a short description of the differences between existing simulation techniques, a description of aspects relevant to the simulation as well as a description of the equations used in the creation of a fluid simulation.
The resulting simulation can handle small bodies of fluids with adjustable viscosity, user-applied force and two-way coupling with solids of adjustable densities at interactive rates ($>$ 60 FPS).

Copies: 2

Pages: 66

Appendices: 1

Acknowledgments

I would like to thank my supervisor, Martin Kraus, for making this project possible.

A special thanks is given to my family and friends for supporting me throughout this project, even though they by now must have forgotten what I look like.

Contents

List of Figures

Acronyms

Acronyms

CFD	Computational Fluid Dynamics
FPS	Frames Per Second
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
MAC	Marker-and-Cell
SPH	Smoothed Particle Hydrodynamics
SWE	Shallow Water Equations

Mathematical Symbols

$\mathbf{u}$	A vector $\mathbf{u} = \left[u_x \; u_y \; u_z\right]^T$
$\boldsymbol{\nabla}$	Gradient
$\boldsymbol{\nabla}\cdot$	Divergence
$\boldsymbol{\nabla}\cdot\boldsymbol{\nabla}$	Laplacian. In other literature often denoted $\boldsymbol{\nabla}^2$ or Δ

Preface

This thesis documents the analysis of existing relevant research within the area of fluid dynamics for graphics, the design and development of a fluid simulation to be used in the Unity3D game engine, as well as a performance test of the product and a demo showing its potential, made by Jens Christian M. Laursen during the spring of 2013 at Aalborg University. The included DVD contains a demo of the product, the thesis, referred webpages, referred videos as well as an AV-production documenting the process.

In the literature, *liquid* and *fluid* is often used synonymously, however the term fluid can technically be either liquid of gas. In this thesis the term fluid will be used to describe liquids.

Resumé

Dette speciale dokumenterer udviklingen af en væske simulering til Unity3D spil motoren, baseret på en kombination af et højde felt og en 2D Eulerian gitter metode. Fokus for denne simulering er en real time grafisk realistisk væske simulator der kan håndtere forskellige typer væsker, samt interaktionen mellem væske og objekter. Dette speciale giver en introduktion til fysikken bag grafisk væske dynamik, en kort beskrivelse af forskellene på eksisterende simuleringer samt en beskrivelse af emner, der er relevante for dette speciale. Der er også en beskrivelse af de ligninger der som er blevet brugt til at skabe en væske simulering.

I bøger, teaterstykker, film etc. er en del af målet af sikre at den individuelle læser/tilskuer er fordybet i historien i en sådan grad at tid og sted glemmes. Uanset hvor fordybet læseren/tilskueren er i historien, så er der dog omstændigheder der kan gøre at denne fordybelse brydes. Dette sker specielt i film hvis der på én eller anden måde er noget grafisk der ikke passer ind. Vær opmærksom på at her ikke menes forældet grafik; mange ældre film er lige så nemme at fordybe sig i som nye, på trods af at grafikken tydeligvis er fra to forskellige tidsaldre. Hvad der derimod menes, er grafik der på én eller anden måde ikke passer ind i universet.

Det samme fænomen gør sig gældende for computerspil, men i modsætning til film, hvor en hær af computere kan bruge timer, dage eller ligefrem uger på at rendere en sekvens billede for billede, skal computerspil være i stand til at rendere simuleringer i real time på forbruger hardware. Simuleringerne må endda kun optage en brøkdel af processor kraften, da det meste skal bruges til at holde spillet kørende.

Ydelses mæssigt er simuleringer af væsker meget krævende, hvilket er grundet til at de kun eksisterer i ganske få spil. I stedet er en række metoder igennem tiden blevet brugt til at lave simplificerede simuleringer der visuelt ligner og/eller opfører sig som væske.

Målet for dette speciale var at lave en interaktiv væske simulering til brug i Unity3D spil motoren. Simuleringen skulle være i stand til at simulere forskellige væsker samt interaktionen mellem væske og objekter. Det var et krav at simuleringen skulle være stabil, virke realistisk og være i stand til at køre med minimum 60FPS på forbruger hardware for at være brugbar i computerspil. For at gøre dette var Navier-Stokes ligningerne implementeret i en Eulerian 2D gitterstruktur, kombineret med et højdefelt. Konsekvensen af at have valgt denne implementeringsform er at detaljegraden for det endelige produkt er i den lavere ende; produktet kan ikke håndtere store gitterstrukturer (større end 48x48) og samtidig holde sig under 60 FPS grænsen. Dette betyder effektivt set at produktet kan bruges til at simulere mindre vandområder, men ikke søer, floder og hav. Produktet kan simulere mindre vandmængder, tyktflydenhed, bruger-tilført kraftpåvirkning samt to-vejs interaktionen mellem væske og primitive objekter (kugler og kuber) for 40+ objekter ad gangen med justerbare massefylde.

1 Introduction and Previous Work

There are two ways to write error-free programs;
only the third one works.

Alan J. Perlis

This thesis documents the development of a fluid simulation for the Unity3D game engine. The focus of the simulation is a real-time and graphically realistic interaction between fluid and solids. This chapter gives an introduction to fluid dynamics, a short description of the differences between existing simulation techniques and a description of aspects relevant to this thesis.

For those new to fluid simulation for graphics, I recommend the following articles:

- **Fast Fluid Dynamics Simulation on the GPU** [1], for a short, well-written introduction on the subject.
- **Fluid Simulation, SIGGRAPH 2007 Course Notes** [2], for an in-depth explanation on various aspects of fluid simulation and various ways of implementation.

1.1 Fluids in Computer Games

In books, plays in at theaters, movies etc., part of the goal is to ensure that the reader/viewer is immersed into the story, to the exclusion of everything else. However, no matter how immersed the reader/viewer is in the story, certain events can cause the reader/viewer to lose this connection.

Especially for movies, unfitting graphics can sometimes expel the viewer from immersion. Mind that here is not meant outdated graphics; many older movies heavy on graphics are as immersive as modern ones even though the graphics are obviously of two different ages. What *is* meant is graphics that in some ways simply do not fit the universe.

The above mentioned phenomena are no less valid in computer games. However, unlike movies where complicated simulations can be made by offline rendering where farms of computers can spend hours, days or even weeks building a sequence frame by frame, computer games must be able to render the simulations in real-time on consumer hardware while only taking up a fraction of the available processing power, since the majority of the latter is needed elsewhere.

Performance wise, real simulations of fluids are very expensive, which is why only few modern games contain them. Instead a variety of methods have been used to fake fluid-like behavior. The following paragraphs each describe fluids in various forms in computer games, going from the simpler ones, to the more advanced. The included DVD contains in-game videos of the effects.

- **Mass Effect 2**: One of the simplest kinds seen in computer games is from Mass Effect 2 [3], wherein the character can choose to order a drink from the bar. The drink in the glass is in this case simply a translucent container formed after the glass, and when the character empties the glass, the flat top of the contained fluid is simply moved locally towards the bottom of the glass.
- **Skylander: Spyro's Adventure** [4]: Features a common solution to render water surface in a lake; the surface of the lake is a translucent plane with a bluish color. A shader is then used to give the impression of ripples and other fluid-like movement on the surface.
- **Uru: Ages Beyond Myst** [5]: Uses procedural water, see Section 1.4.3.
- **Portal 2** [6]: This game uses an approach called Metaballs.
- **From Dust** [7]: This game uses a height-field method described later in Section 1.4.4.
- **Borderlands 2** [8]: Uses PhysX's Smoothed Particle Hydrodynamics (SPH) method to simulate various small puddles of blood, poison and more, and allows the user to interact with it. SPH is described in Section 1.4.2.3.

1.2 Navier-Stokes Equations

The majority of Computational Fluid Dynamics (CFD) for graphics methods are based on the Navier-Stokes equations for incompressible and homogeneous flow. That a fluid is homogeneous means the density does not vary across the fluid body, and that it is incompressible means that it does not vary over time, either. The first equation, the Momentum Equation, describes how forces acting on a fluid cause it to accelerate:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \boldsymbol{\nabla})\mathbf{u} - \frac{1}{\rho}\boldsymbol{\nabla} p + \nu \boldsymbol{\nabla} \cdot \boldsymbol{\nabla} \mathbf{u} + \mathbf{f} \tag{1.1}$$

$\mathbf{u}$ **Velocity of the fluid**. Is a vector velocity field. The velocity of a particle at position $\mathbf{x} = \begin{bmatrix} x_x \; x_y \; x_z \end{bmatrix}^T$ at time t is given by $\mathbf{u} = \mathbf{u}(t, \mathbf{x}) = \begin{bmatrix} u(t, \mathbf{x}) \; v(t, \mathbf{x}) \; w(t, \mathbf{x}) \end{bmatrix}^T$.

ρ **Density of the fluid at a point.** Remember that the density is constant and that $\rho = \frac{m}{V}$, where m is mass and V is volume.

- For syrup, this is roughly $1500 kg/m^3$
- For water, this is roughly $1000 kg/m^3$
- For air, this is roughly $1.3 kg/m^3$

p **Pressure** Is a scalar field, indicating the force per unit area that the fluid exerts on anything. The pressure of a particle at a position $\mathbf{x}$ at time t is given by $p = p(t, \mathbf{x})$

ν **Kinematic viscosity of the fluid**. It measures in m^2/s how viscous the fluid is, that is, how much the fluid will resist deformation. Remember that $\nu = \frac{\mu}{\rho}$, where μ is the dynamic viscosity of the fluid, measured in $Pa \cdot s$.

$\mathbf{f}$ **External forces** per unit volume. Often called *body forces*, since these forces affect the entire body of fluid, not just the surface. Often this is equal $\rho \mathbf{g}$, where $\mathbf{g}$ is acceleration due to gravity. Usually $(0, -9.81, 0) m/s^2$.

The second equation is the equation for conservation of mass, given by Equation 1.2. Most CFD methods focusing on visual effects consider fluids to be incompressible, which is also the assumption made in this thesis. This assumption leads to the simplified equation for the conservation of mass which is given by Equation 1.3.

It is important to note that the consequence of this assumption is that it effectively prevents the simulation of sound and shock waves within the fluid.

$$\frac{\partial \rho}{\partial t} = -\boldsymbol{\nabla} \cdot (\rho \mathbf{u}) \tag{1.2}$$

$$\boldsymbol{\nabla} \cdot \mathbf{u} = 0 \tag{1.3}$$

The $\boldsymbol{\nabla}$ operator, called *nabla*, in the Navier-Stokes equations, has three meanings, depending on how it is used. Appendix A goes into further details, but here is a short description:

$\boldsymbol{\nabla}$	is called the ***gradient***, and when coupled with a scalar field (e.g. with a pressure field: $\boldsymbol{\nabla} p$) results in a vector field.
$\boldsymbol{\nabla}\cdot$	in the Incompressibility Equation 1.3 is called the ***divergence*** and results in a scalar field when coupled with a vector field (e.g. with a velocity field $\boldsymbol{\nabla} \cdot \mathbf{u}$). The resulting scalar field is a measurement how much a vector quantity is either entering or exiting a given region of the fluid. The incompressibility equation states that the sum of all changes of fluid in the entire body must equal zero.
$\boldsymbol{\nabla} \cdot \boldsymbol{\nabla}$	is called the ***Laplacian***, and it refers to taking the divergence of a gradient. If the right hand side of the Laplacian is non-zero, e.g. $\boldsymbol{\nabla} \cdot \boldsymbol{\nabla} x = b$, it is called a *Poisson* equation.

1.2.1 Momentum Equation in Cartesian Coordinates

Since $\mathbf{u}$ is a vector, the momentum equation is in reality multiple equations. This is the momentum equation written explicitly in Cartesian Coordinates in three dimensions:

$$\begin{aligned}
\frac{\partial u}{\partial t} &= -\left(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z}\right) - \frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) + f_x \\
\frac{\partial v}{\partial t} &= -\left(u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z}\right) - \frac{1}{\rho}\frac{\partial p}{\partial y} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) + f_y \\
\frac{\partial w}{\partial t} &= -\left(u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z}\right) - \frac{1}{\rho}\frac{\partial p}{\partial z} + \nu\left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right) + f_z,
\end{aligned}$$

where f_x, f_y and f_z is the external force in the direction, specified by the implementation. The equation of mass, Equation 1.2 becomes:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho}{\partial x}u + \frac{\partial \rho}{\partial y}v + \frac{\partial \rho}{\partial z}w = 0,$$

but since the density is homogeneous, ρ does not change across the body of fluid and it becomes:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0$$

1.2.2 Explaining the equations

To solve the Navier-Stokes equations, they are first split into separate parts.

1.2.2.1 Velocity of a location $\frac{\partial \mathbf{u}}{\partial t}$

Starting from the left, the $\frac{\partial \mathbf{u}}{\partial t}$ part describes how the fluid velocity at a fixed location changes over time, see Figure 1.1.

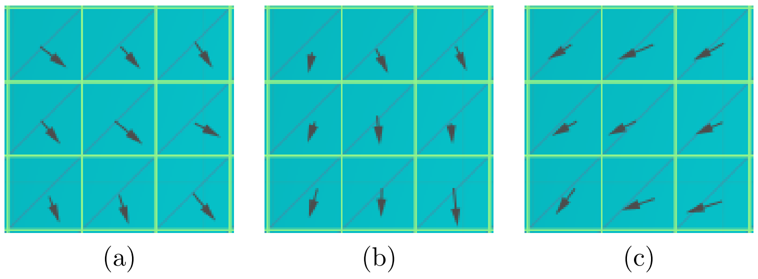


Figure 1.1: Velocity at a fixed location in a grid changing over time.

1.2.2.2 Advection $(\mathbf{u} \cdot \boldsymbol{\nabla})\mathbf{u}$

The first part on the right side, $(\mathbf{u} \cdot \boldsymbol{\nabla})\mathbf{u}$, is often called *Transport*, *Convective Acceleration*, *Self-Advection* or simply *Advection* in the literature. It is effectively a (velocity) vector (flow) field of the fluid motion, expressing how the velocity of a body of fluid changes as it moves around. In other words; it expresses how a quantity (which can be velocity, density or even solids) accelerates both as it follows the velocity field and as a result of the velocity field itself changing in time, and it is this that makes the motion of fluids quadratic instead of linear.

1.2.2.3 Pressure $\frac{1}{\rho}\boldsymbol{\nabla} p$

Pressure is *force per unit area.* When applied to a point in space, this part of the Navier-Stokes equations measures the net difference in pressure force, and any difference in pressure leads to acceleration, e.g. if a position in a fluid has low pressure compared to a neighbor position, then the fluid will accelerate from the high-pressure position toward the low-pressure position. The pressure is closely related to the equation for conservation of mass, described below.

1.2.2.4 Viscosity/Diffusion $\nu\nabla \cdot \nabla\mathbf{u}$

Viscosity, or diffusion, is a term that defines how much a given fluid will resist deformation. E.g. water has low viscosity, so if a solid is dropped into a relatively small body water, it will quickly affect the rest of the fluid. Syrup, on the other hand, has a high viscosity and will to a much larger degree resist the deformation caused by a dropping solid. In relation to a grid-based fluid simulation, viscosity defines how a quantity (e.g. velocity) in a cell interacts with its neighbors. The viscous fluid is achieved by applying diffusion to the velocity field.

1.2.2.5 External Force $\rho\mathbf{g}$ or $\mathbf{f}$

Often called *Other Forces* or *Body Force* and denoted $\mathbf{f}$. Typically, only gravity is contained within this variable, however, it can also contain electromagnetic- and/or centrifugal force.

1.2.3 Conservation of Mass $\nabla \cdot \mathbf{u} = 0$

For every time step, the Navier-Stokes equations are solved for the velocity field for a body of fluid: Advection, diffusion and force application. The result of these computations is a velocity field with non-zero divergence. Since Equation 1.3 demands a divergence-free velocity field, further calculations have to be done. The equation for conservation of mass is also called *Pressure Projection*, or simply *Projection* in literature, and is a term for the calculations that ensure incompressibility.

1.3 Boundary Conditions

Fluids interact with their containers and other fluids, stream around objects embedded in the fluids and carry them along if the density of the solid is less than that of the fluid. These interactions are called the *boundary conditions*.

Each equation has its own boundary conditions: The momentum equations have one set of boundary conditions, the pressure another, density yet another and so on.

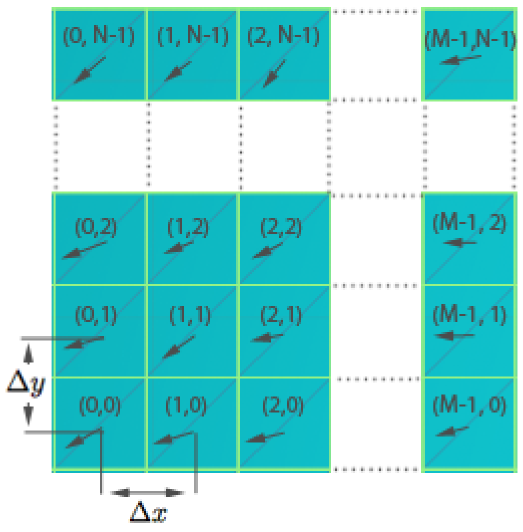


Figure 1.2: Eulerian Fixed Grid of size M-1×N-1. The distance between two grids is Δx and Δy, respectively. Usually $\Delta x = \Delta y$.

Free Surfaces: Free Surfaces is a term used for the part of the fluid boundary that is not in touch with walls, other solids or other fluids (air is typically ignored).

1.4 Fluid Simulation Techniques

There are a number of different techniques generally used to make real fluid simulations including Eulerian (grid-based) and Lagrangian (particle-based). Other techniques that focus on looking realistic, rather than being realistic, include procedural simulation and height field techniques. Each have their pros and cons and will be described in this section.

1.4.1 Grid Based - Eulerian

A fluid simulation based on the Eulerian view tracks the fluid properties at fixed (discrete) points in space, as seen in Figure 1.2. These properties are given by either scalar or vector fields, which are often defined in the center of individual grid cells. Grid based simulations come in various forms: Fixed Grid, Adaptable Grid and Tall Cell Grid among others.

The simplest form is the uniform fixed grid, depicted in Figure 1.3a: Herein a space is divided into grid cells, typically of equal size. This version has the advantage that it allows fast lookups, since the grid can be loaded into memory when the simulation is initiated [9]. The disadvantage is that it is also wasteful, since it is probable that a large number of cells will never be used. Also, unless the size of the cells are very small, some of the finer details will be lost in regions of great activity.

An adaptable grid, as the name implies, has a non-uniform grid, where regions with little activity will be given large cells, whereas regions with much activity (e.g. a region with vorticity), will be given smaller cells, as depicted in Figure 1.3b. While this version has obvious advantages over the uniform fixed grid, it is complicated to build a stable grid with fast lookups.

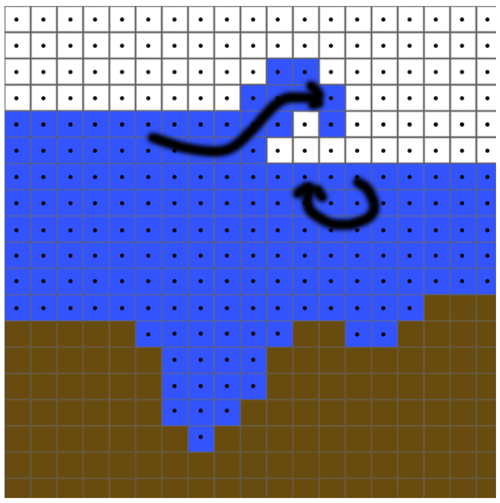
The last version of grid that will be described here is the Tall Cell grid, introduced by (Irving et al., 2006) in 2006 [10] and used again by (Chantanex and Müller, 2011) [11] in 2011. Similar to the adaptable grid, this version will focus processing power on regions with much activity, which is typically near the surface and near boundaries. The difference being that all cells beneath a certain distance to the surface will be converted into one tall cell in each column, as depicted in Figure 1.3c. While this version suffers from the same disadvantages as the adaptable grid, the great advantage is that processing power is focused only on the region near the surface while everything else is ignored.

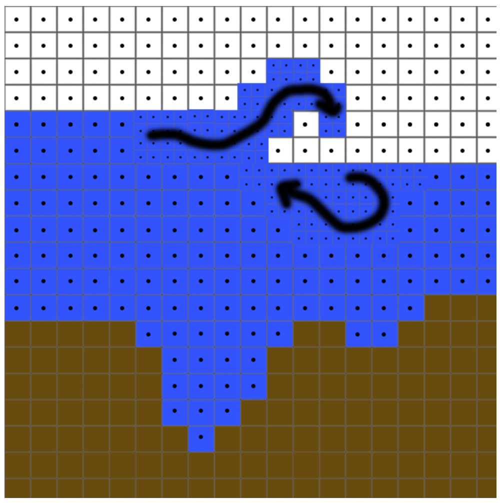
1.4.1.1 Basic Grid Structure

A relatively heavy part, performance wise, of Eulerian fluid simulations is to ensure that the incompressibility of the fluid, Equation 1.3, is maintained. A way of doing this is to estimate how much the amount of fluid in a cell is changing, and in which direction it is flowing. To make this easier, (Harlow and Welch, 1965) [12] introduced the Marker-and-Cell (MAC) grid structure in 1965. It is a so-called "Staggered" grid [2], meaning that different variables are stored different places. In 2D, the scalar quantities, such as density, are stored at the center of the grid, depicted as $p_{i,j}$ on Figure 1.4, whereas vector quantities, such as velocities, are stored at the center of the vertical and horizontal cell edges, depicted as $u_{i\pm1/2,j}$ and $v_{i,j\pm1/2}$, respectively. Similarly in 3D, the scalar quantities are stored at the center of the cell, while vector quantities are stored at the center of the faces.

1.4.2 Particle Based - Lagrangian

Instead of looking for change at fixed points in space, a Lagrangian based fluid simulation follows particles, each of which has a position $\mathbf{x} = \begin{bmatrix} x_x \; x_y \; x_z \end{bmatrix}^T$ and a velocity $\mathbf{u}$, see Figure 1.5. Many Lagrangian based simulations use two versions of particle systems; one to simulate spray and foam; and one to simulate fluid.


(a)


(b)

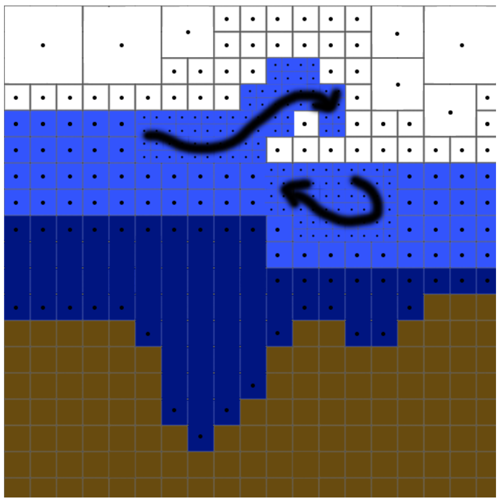

(c)

Figure 1.3: Eulerian Grids seen from the side, where the black lines indicate regions with much activity. (a) is a fixed grid, where every grid cell has the same size, (b) is an adaptable grid where the amount of grid cells increase in regions of great activity and decrease in regions without, (c) is a tall cell grid which is similar to the adaptable grid, only it converts every fluid grid cell beneath x grid cells to one tall cell, thereby focusing the processing power onto the surface.

Particles are generated before the program begins and/or by one/multiple emitters. These emitters can have any shape, but the simplest ones are point emitters and rectangle emitters. An emitter will typically generate particles with a number of parameters, including velocity, mass, lifetime etc. The emitter itself

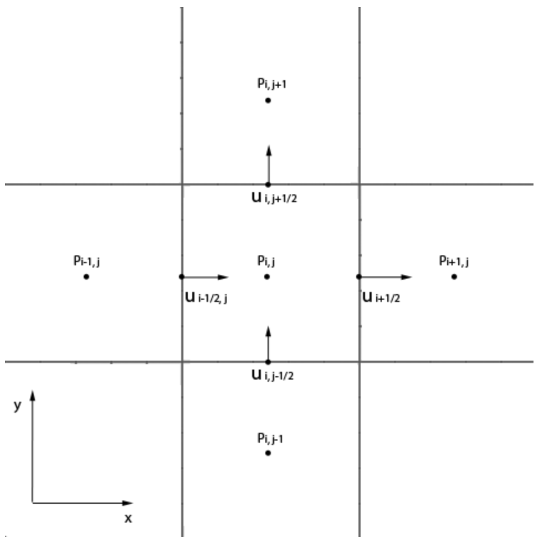


Figure 1.4: Eulerian Grid Structures. The image shows is a 2D MAC grid structure, where the scalar field values are stored in the center of the grid cells, and vector quantities are stored at the center of the edges.

also has a number of parameters, including spawn rate, spawn impulse force etc.

1.4.2.1 Non-Interacting Particles

Also called a *Simple Particle System.* This is the most common kind of particle system and, as the name implies, the system does not calculate collisions for the particles. Since it is not essential that foam and spray particles interact, they are usually simulated using this kind of particle system.

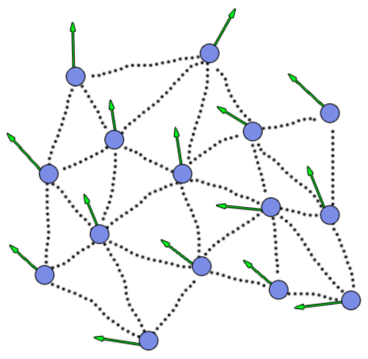


Figure 1.5: Lagrangian Particle Method. Consists of a number of particles which, unlike the grid cells in an Eulerian based simulation, moves around.

1.4.2.2 Interacting Particles

In fluid simulation, a change in one part of the fluid will affect part of, if not the entire, body of fluid. It is therefore essential that the particles can interact.

However, having every particle in the simulation calculate its distance to every other particle gives a quadratic complexity ($O(N^2)$) for N particles. The complexity can be decreased if each particle only takes nearby particles into account; each particle only calculate its distance to particles within a distance d. In optimal situations, this will reduce the expected complexity to linear time, ($O(N)$).

1.4.2.3 Smoothed Particle Hydrodynamics

The Smoothed Particle Hydrodynamics (SPH) method was invented within the field of computational astrophysics by (Lucy, 1977) [13] and (Gingold and Monaghan, 1977) [14] but has been used extensively in the field of fluid simulation. SPH is basically an interpolation method, used to approximate fluid properties at any position from particles in space. In other words, SPH is used to approximate continuous fluid properties by interpolating values between discrete samples. The process of interpolating values between particles is called *smoothing* [15], and it is done using so-called smoothing kernels[2].

1.4.3 Procedural Water

Procedural water is a term often used for fluid simulations wherein only the visual effect is important; that the fluid seems fluid-like. This kind of simulation is rarely based on any kind of physics, instead it is up to the individual programmer to find a creative method; an often used method is using superimposed sine waves with varying amplitudes and directions, sometimes with a dampening factor to make the waves flatten after a while. Among others, (Mark Flinch, 2004) [16] used superimposed sine waves to simulate water surface in lakes in the game *Uru: Ages Beyond Myst* [5].

Since this kind of simulation is not based on physics, it is typically very cheap and is therefore often used to simulate larger basins of fluid, such as lakes and oceans.

1.4.4 Height Field

A height field based simulation typically consists of a number of tall cubes with equal width and breadth, formed in a square, see Figure 1.6. It is relatively simple to set up, and is often used to implement physical aspects of fluid simulation; waves, solid-to-fluid and fluid-to-solid coupling etc. Another advantage of a height

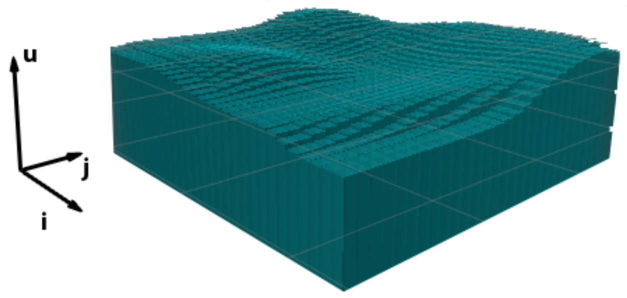


Figure 1.6: A height field consists of a number of cubes. The position is set at the top of each cube. Velocity is restricted to the y-axis and breaking waves are therefore impossible.

field is that it works well together with a texture implementation on a shader. The surface of the fluid is typically represented via a:

- continuous 2D function $u(x, y)$
- discrete 2D array u[i,j].

This kind of simulation has one major disadvantage, however: Since waves are simulated by controlling the vertical velocity of each cube, it is not possible to simulate breaking waves using a height field alone. Therefore, a height field is often coupled with a particle system which is activated where foam, sprays and splashes should form on a breaking wave.

As mentioned in Section 1.4.1, a height field has previously been used in collaboration with an Eulerian based simulation, where the height field performed the calculations of the lower part of the fluid.

1.5 Foam and Spray

For the generation of spray and foam from breaking waves, (Fournier and Reeves, 1986) defined a rule, saying that when the difference between particle speed and surface speed projected in the direction of the normal to the surface exceeds a certain threshold (depending on the curvature of the surface), spray is generated. Otherwise, foam is generated. When generated, spray is sent in the direction of the surface normal, whereas foam is sent sliding along the wave surface [17, 18].

1.6 Solid Interaction

When describing the interaction between a solid and a fluid, three general interaction types, often called *coupling*, are used [19]; one-way solid-to-fluid coupling, one-way fluid-to-solid coupling and finally two-way coupling.

Attributes of a *solid* affecting the interaction with a fluid:

- velocity and direction with which it hits the water.
- density of the solid.
- projectioned area of the solid (the part which hits the water).
- the form of the part which hits the water - convex, concave, sharp, flat etc.
- volume of the solid.
- other solids connected with the solid, e.g. a rag doll.

Attributes of the *fluid* affecting the interaction with the solid

- velocity of the part of the fluid the solid hits.
- density of the fluid.
- viscosity of the fluid.

One-way Coupling - Solid-to-Fluid interaction
Solids cause deformation in the fluid, e.g. a ball hits the fluid and creates splashes and displacement of fluid. However, the fluid has no influence on the solid, meaning that the solid will continue its motion unhindered.

One-way Coupling - Fluid-to-Solid interaction
This is the opposite situation, where the fluid affects the motion of the solid, but the solid has no influence on the fluid, e.g. when a ball hits a fluid, the ball will act as if it hit a fluid, meaning that it will float toward the surface if its density is less than that of the fluid. The fluid itself will remain unchanged, meaning that the motion of the fluid will continue unaffected by the solid.

Two-way Coupling
Solids cause deformation in the fluid which in turn affects the motion of the solids.

2 Method

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.

Sir Charles Antony Richard Hoare

This chapter will describe the formulas that will be used to create a fluid simulation based on an Eulerian grid combined with a height field. Equations in this section assume a 2D grid.

2.1 Requirements

The frame rate, measured in Frames Per Second (FPS), is a term often used to determine how smooth a game is running on a computer. More precisely, if the time to render an image is given in milliseconds, the frame rate is then simply the sum of images, rendered in a full second. For modern computer games the target frame rate is often 30-60 FPS [20]. A frame rate of 30-60 FPS means between $33\frac{1}{3}$ and $16\frac{2}{3}$ milliseconds to render each frame.

Given that the majority of the processing power is needed elsewhere to make the game running, this leaves only little processing power for the simulation of fluids. Hence, the following requirements for the simulation must be met;

- It must appear realistic.
- It must be interactive.

- It must consume little memory.
- It must be cheap to compute.
- It must be stable.

One very specific goal that must be met is that the simulation must be able to run at minimum 60FPS.

2.2 Steps of fluid simulation

This section explains in detail the algorithms which can be used to create a grid-based fluid simulation. A single simulation iteration for one time step is given by 1) force application, 2) advection, 3) diffusion and finally 4) projection. Each of which will be described in this chapter.

2.2.1 Finite Difference Form

The finite difference form for the gradient, divergence and Laplacian in two dimensions are:

$$\begin{aligned} \boldsymbol{\nabla} q = \left[\frac{\partial q}{\partial x} \; \frac{\partial q}{\partial y}\right]^T &= \frac{q_{i+1,j} - q_{i-1,j}}{2\Delta x}, \frac{q_{i,j+1} - q_{i,j-1}}{2\Delta y} \\ &= \frac{q_{i+1,j} - q_{i-1,j}}{2\Delta x}, \frac{q_{i,j+1} - q_{i,j-1}}{2\Delta x} \end{aligned} \tag{2.1}$$

$$\begin{aligned} \boldsymbol{\nabla} \cdot \mathbf{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} &= \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta y} \\ &= \frac{u_{i+1,j} - u_{i-1,j} + v_{i,j+1} - v_{i,j-1}}{2\Delta x} \end{aligned} \tag{2.2}$$

$$\begin{aligned} \boldsymbol{\nabla} \cdot \boldsymbol{\nabla} q = \frac{\partial^2 q}{\partial x^2} + \frac{\partial^2 q}{\partial y^2} &= \frac{q_{i+1,j} - 2q_{i,j} + q_{i-1,j}}{(\Delta x)^2} + \frac{q_{i,j+1} - 2q_{i,j} + q_{i,j-1}}{(\Delta y)^2} \\ &= \frac{q_{i+1,j} + q_{i-1,j} + q_{i,j+1} + q_{i,j-1} - 4q_{i,j}}{(\Delta x)^2} \end{aligned} \tag{2.3}$$

where i and j refer to the positions of individual grids, and Δx and Δy are respectively the width and breadth of individual grids along the x-axis and y-axis. Note that in fluid dynamics, it is often the case that $\Delta x = \Delta y$, in which case the finite difference forms simplify to Equations 2.1, 2.2 and 2.3. See Appendix A for further details.

2.2.2 Advection

In a particle system, the way a particle with position $\mathbf{x}$ is moved forward in a time step Δt is often simply by moving it forward using a velocity vector, or, in this case, a velocity field. This is done using the equation known as the *forward Euler*, also called *Explicit Euler* or simply *Euler*, method:

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{u}(t)\Delta t,$$

where $\mathbf{x}(t)$ is the current position, $\mathbf{u}(t)\Delta t$ is the velocity vector in a time step and $\mathbf{x}(t + \Delta t)$ is the new position after a time step. A more general formula is:

$$q(\mathbf{x} + \mathbf{u}\Delta t,\, t + \Delta t) = q(\mathbf{x}(t),\, t),$$

where q can be either a vector (e.g. velocity) or scalar quantity (e.g. density, temperature). The forward Euler is unstable, as a simulation using this method for advection will, sooner or later, blow up - especially in cases where the magnitude of $\mathbf{u}(t)\Delta t$ is larger than a grid cell size, Δx [21].

To improve stability, what is often used instead is the *semi-Lagrangian* method, which does two things: First it performs a *backward Euler*, also known as *implicit Euler*, method; where the forward Euler method moves a quantity forward in time, the backward Euler method does the opposite and traces a quantity back in time to its previous position to the quantity it had back then:

$$q(\mathbf{x}, t + \Delta t) = q(\mathbf{x}(t) - \mathbf{u}(\mathbf{x}, t)\Delta t,\, t) \tag{2.4}$$

While this ensures stability, in that a quantity will move with the velocity field but never actually change, this provides another problem for grid based simulations; on a discrete grid it is not certain that the position a quantity previously occupied is in center of a grid cell. It is in fact improbable that this should ever be the case. This problem is solved using a process known as *bilinear interpolation* (trilinear in three dimensions) [22], which is the second part of the semi-Lagrangian method, see Figure 2.1.

2.2.3 Viscosity/Diffusion

Similarly to the solution to the advection technique, wherein the semi-Lagrangian method was chosen over the simpler and more obvious forward Euler method, the solution to the diffusion has an explicit solution given by equation:

$$\mathbf{u}(\mathbf{x}, t + \Delta t) = \mathbf{u}(\mathbf{x}, t) + \nu \boldsymbol{\nabla} \cdot \boldsymbol{\nabla} \mathbf{u}(\mathbf{x}, t)$$

The above equation, however, is known to become unstable for large values

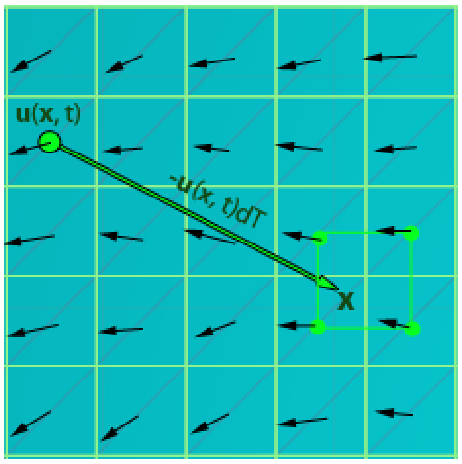


Figure 2.1: Bilinear interpolation on a discrete grid; The new quantity in the new position is calculated by tracing it back in time along the velocity vector to its old position x. Since x is outside the center of the grid, its old value is calculated from the surrounding four grid positions. Once done, the value of the old quantity is passed on to the new quantity. Note that dT is Δt.

Δt and ν. An implicit method was given by (Stam, 1999) [23]:

$$(\mathbf{I} - \nu \Delta t \boldsymbol{\nabla} \cdot \boldsymbol{\nabla})\mathbf{u}(\mathbf{x}, t + \Delta t) = \mathbf{u}(\mathbf{x}, t), \tag{2.5}$$

which is a Poisson equation for velocity. **I** is the identity matrix.

2.2.4 Pressure Projection

When pressure is applied to fluid, the fluid can either compress or expand. Mathematically, this is given by Equation 1.2, which states that an influx of fluid changes the amount of fluid at that location. The following method is based on the *Stable Fluids* technique described in (Stam, 1999) [23] as well as the *Fast Fluid Dynamics Simulation on the GPU* article by (Harris, 2004) [1].

2.2.4.1 Helmholtz-Hodge Decomposition

A sum of vector fields can be decomposed into a sum of vector fields. By defining a region D in a plane on which the fluid is defined, with a smooth boundary ∂D, with normal direction $\mathbf{n}$, the Helmholtz-Hodge Decomposition Theorem by (Chorin and Marsden) [24] states that a vector field $\mathbf{w}$ on D can be decomposed into the form:

$$\mathbf{w} = \mathbf{u} + \boldsymbol{\nabla} p, \tag{2.6}$$

where $\mathbf{u}$ is a divergence-free vector field ($\boldsymbol{\nabla} \cdot \mathbf{u} = 0$) and p is a pressure field.

For every time step the Navier-Stokes equations are solved for the velocity field for a body of fluid: Advection, diffusion and force application. The result of these computations is a velocity field with non-zero divergence. Since Equation 1.3 demands a divergence-free velocity field, further calculations have to be done. The Helmholtz Decomposition Theorem states that the divergent velocity field can be made divergence-free by subtracting the gradient of the resulting pressure field:

$$\mathbf{u} = \mathbf{w} - \boldsymbol{\nabla} p$$

To solve for a scalar field the divergence operator is applied to both sides of Equation 2.6, resulting in:

$$\begin{aligned} \boldsymbol{\nabla} \cdot \mathbf{w} &= \boldsymbol{\nabla} \cdot (\mathbf{u} + \boldsymbol{\nabla} p) \\ &= \boldsymbol{\nabla} \cdot \mathbf{u} + \boldsymbol{\nabla} \cdot \boldsymbol{\nabla} p, \end{aligned}$$

but since Equation 1.3 states that $\boldsymbol{\nabla} \cdot \mathbf{u} = 0$, it simplifies to:

$$\boldsymbol{\nabla} \cdot \boldsymbol{\nabla} p = \boldsymbol{\nabla} \cdot \mathbf{w}, \tag{2.7}$$

which is a Poisson equation for the pressure of the fluid. For further details, read (Harris, 2004) [1].

2.2.5 Jacobi Iteration

The Poisson-pressure equation, Equation 2.7, and the viscous diffusion equation, Equation 2.5, can be solved using the Poisson method [1], which is given by the equation:

$$A\mathbf{x} = \mathbf{b},$$

where A is a matrix with non-zero diagonal elements [25] given implicitly by the Laplacian operator, $\boldsymbol{\nabla} \cdot \boldsymbol{\nabla}$, $\mathbf{b}$ is a vector of constants and $\mathbf{x}$ is a vector or scalar quantity, e.g. the velocity field $\mathbf{u}$ or pressure field p. The Poisson method can be solved using the Jacobi equation that is an iterative method, which starts with an initial guess, and for every iteration the guess is improved. The Jacobi equation can be used to solve both the Poisson-pressure and the viscous diffusion equation and is given by the equation:

$$x_{i,j}^{k+1} = \frac{x_{i-1,j}^{k} + x_{i+1,j}^{k} + x_{i,j-1}^{k} + x_{i,j+1}^{k} + \alpha b_{i,j}}{\beta,} \tag{2.8}$$

where k denotes the present iteration number, α, β, x and b vary, depending on whether it is a Poisson-pressure or viscous diffusion equation that is to be solved.

Viscous diffusion equation: $\alpha = \frac{(\Delta x)^2}{\nu \Delta t}$, $\beta = 4 + \alpha$, $x = \mathbf{u}$ and $b = \mathbf{u}$.

Poisson-pressure equation: $\alpha = -(\Delta x)^2$, $\beta = 4$, $x = p$ and $b = \nabla \cdot \mathbf{w}$.

2.3 Boundary Conditions

No-slip is the most simple kind of boundary condition, relevant for viscous fluids. It states that the velocity of a quantity goes to zero at the boundaries for a stationary boundary:

$$\mathbf{u} = 0,$$

and

$$\mathbf{u} = \mathbf{u}_{solid},$$

for a moving boundary [2].

2.4 Fluid Solid Coupling

There are three major forces a fluid can induce to a solid body: buoyancy, drag and lift. Buoyancy is given by the equation:

$$\mathbf{f}^{\text{buoyancy}} = -\mathbf{g}\rho V_{\text{sub}}, \tag{2.9}$$

where $\mathbf{g}$ is the gravitational vector, ρ is the density of the fluid and V_{sub} is the submerged volume of the object.

While it is useful to be able to simulate the coupling between the fluid and primitive objects (spheres, cubes etc.), it would be more useful if the simulation could handle any type of object. One solution is to define fluid coupling for primitive objects and approximate a complex object with one or multiple of these primitive objects, e.g. approximate a boat as a number of cubes. A different solution was presented by (Yuksel, House and Keyser, 2007) [26], who performed the drag and lift calculations, given by Equation 2.11 and 2.12, on the object faces in contact with a fluid.

The calculation of the buoyancy, Equation 2.9, requires the submerged volume of an object, which can be approximated as suggested by (Chentanez and Müller, 2010) [27], by transforming the object into a number of prisms by a) calculating the projected area of each downward-facing face of the object in the xz-plane, b) multiplying the area with the distance to the surface in the y-plane for each face, and finally c) summing up the resulting volumes:

$$V_{\text{sub}} = \begin{cases} 0 & \text{if airborne} \\ \sum A_{\text{projFace}}(\eta - p_y) & \text{if } n_y < 0 \\ V_{\text{total}} & \text{if fully submerged} \end{cases} \tag{2.10}$$

where A_{projFace} is the projected area of a face, $\mathbf{p} = \begin{bmatrix} p_x & p_y & p_z \end{bmatrix}^T$ is the position of the centroid of a face, η is the height of the surface above the face, $\mathbf{n} = \begin{bmatrix} n_x & n_y & n_z \end{bmatrix}^T$ is the face normal and n_y is the face normal along the y-axis, and V_{total} is the pre-calculated volume of the object. Note that this equation will give errors with some concave objects. The equations for drag and lift are given by:

$$\mathbf{f}^{\text{drag}} = -\frac{1}{2}\rho C_D A |\mathbf{u}_{\text{rel}}| \mathbf{u}_{\text{rel}} \tag{2.11}$$

$$\mathbf{f}^{\text{lift}} = -\frac{1}{2}\rho C_L A |\mathbf{u}_{\text{rel}}| \left(\mathbf{u}_{\text{rel}} \times \frac{\mathbf{n} \times \mathbf{u}_{\text{rel}}}{|\mathbf{n} \times \mathbf{u}_{\text{rel}}|} \right), \tag{2.12}$$

where A is the effective area of the face, C_D and C_L are respectively the drag and lift coefficients, which depend on the fluid as well as the shape of the solid. $\mathbf{u}_{\text{rel}}$ is the velocity of the solid relative to the velocity of the fluid. That is: $\mathbf{u}_{\text{rel}} = \mathbf{u}_{\text{solid}} - \mathbf{u}_{\text{fluid}}$. A is given by:

$$A = \left(\frac{\mathbf{n} \cdot \mathbf{u}_{\text{rel}}}{|\mathbf{u}_{\text{rel}}|} \alpha + (1 - \alpha) \right) A_{\text{face}}, \tag{2.13}$$

where α is a control-parameter given by $0 \le \alpha \le 1$. Once calculated, the buoyancy is applied to the center of the submerged volume, and the drag and lift forces are applied to the center of each face.

2.5 Surface Rendering

This section describes the visual part of the simulation, the goal being to make the simulation seem as realistic as possible. This requires a number of things that will be described separately. All fluid calculations are performed on multiple arrays and at the end of ever iteration. The new values represent the new heights of all cubes in the height map, and these values are applied to an 8bit texture, which is then transferred to the shader which handles the fluid surface. The shader is attached to a material which in turn is attached to a plane.

2.5.1 Vertex Displacement

The shader will manipulate the individual pixels and displace them along the y-axis.

2.5.2 Reflection

When looking at the surface of e.g. water, the surface will reflect either the sky or other objects. In computer graphics this is given by the formula:

$$\mathbf{R} = \mathbf{I} - 2\mathbf{N}(\mathbf{N} \cdot \mathbf{I}),$$

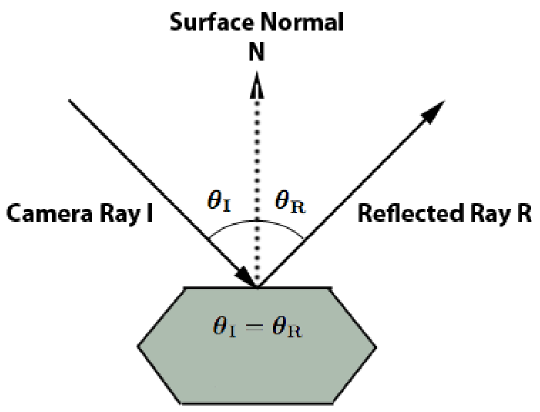
which dictates that the angle between the camera ray, **I**, and the surface normal, **N**, is equal to the angle between the surface normal and the reflected vector, **R**. This is also depicted in Figure 2.2a.

2.5.3 Refraction/Distortion

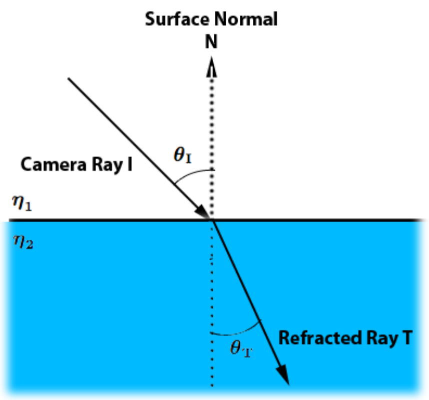
Refraction is what happens when a ray of light moves through translucent materials with different densities (e.g. water and glass). A popular explanation is that the light travels slower in materials with large densities and visa versa, and so the direction of the ray is changed. E.g. light travels fast in a vacuum, slower in air and much slower in diamond. In computer graphics this is given by a *refractive index* which varies from material to material; air = 1.0003, water = 1.3333, honey = 1.484-1.504, diamond = 2.417. The equation for refraction is given by *Snell's Law* [28] in the following equation:

$$\eta_1 \sin(\theta_I) = \eta_1 \sin(\theta_T),$$

where η_1 and η_2 are the refractive indices for two materials, θ_I is the angle between the camera ray **I** and the surface normal **N**, and θ_T is the angle between



(a)



(b)

Figure 2.2: (a) shows how reflection is calculated; the reflected ray R is given by the incoming camera ray I and its angle to the surface normal N. The reflective surface hit by the camera ray is given the color hit by the reflected ray. (b) If a ray is shot through the boundary between materials with different densities, the incoming camera ray I will be refracted.

the surface normal below the surface $-\mathbf{N}$ and the refracted vector $\mathbf{T}$, also called *transmitted*, as depicted in Figure 2.2b.

2.5.4 The Fresnel Effect

The Fresnel effect is a term used to describe the situation wherein a ray of light moves through translucent materials with different densities, and part of the ray is reflected while the remaining is refracted. The Fresnel equation dictates that the larger the angle between the surface normal, $\mathbf{N}$, and the camera ray, $\mathbf{I}$, the more will be reflected and the less will be refracted. The opposite goes as well, in that the lower the angel, the less will be reflected and the more will be refracted. Equation 2.15 by (Fernando, 2003) [29] is a Fresnel equation focusing on the visual effect rather than the mathematical precision. It depends on a reflection coefficient R_{Coeff} which is given by equation 2.14:

$$R_{\text{Coeff}} = \max(0, \min(1, bias + scale(1 + \mathbf{I} \cdot \mathbf{N})^{power})) \tag{2.14}$$

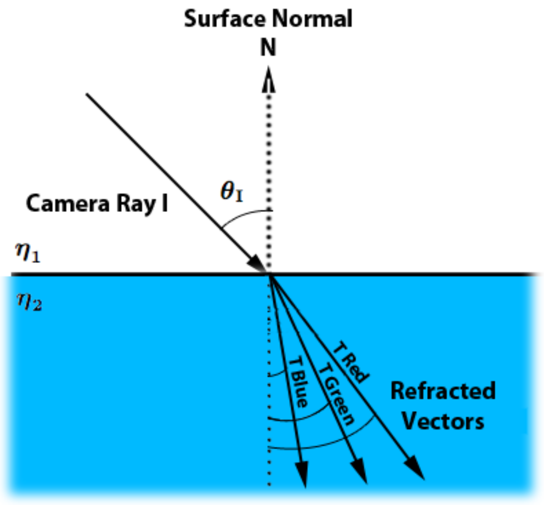
$$C_{\text{Final}} = R_{\text{Coeff}} C_{\text{Reflected}} + (1 - R_{\text{Coeff}}) C_{\text{Refracted}}, \tag{2.15}$$

where C is the color, and *bias*, *scale*, and *power* are control parameters used to adjust the final rendering.

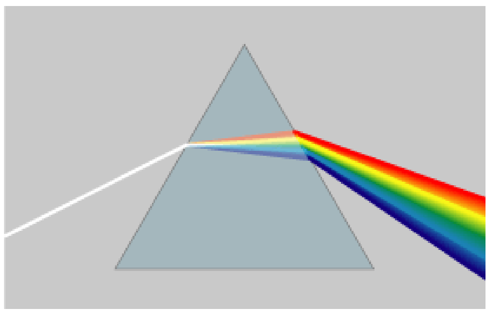
2.5.5 Chromatic Dispersion

If a single ray of light is pointed at a prism, as in Figure 2.3b, it will result in a rainbow of colors. The reason behind this is that wavelengths (colors) refract at different angles, e.g. as seen in Figure 2.3a red refracts more than blue. In computer graphics the simple solution to this problem is to refract the red, green and blue color-channel individually using their material- and color-depending refractive indices. For water, the color-depending refractive indices are:

- **Red:** 0.700 μm = refractive index of 1.3300
- **green:** 0.520 μm = refractive index of 1.3342
- **Blue:** 0.480 μm = refractive index of 1.3358



(a)



(b)

Figure 2.3: (a) Chromatic Dispersion: Colors refract differently, e.g. red refracts more than green, which refract more than blue. (b) http://en.wikipedia.org/wiki/File:Prism_rainbow_schema.png.

3 Implementation

If debugging is the process of removing bugs, then programming must be the process of putting them in.

Edsger W. Dijkstra

The basics of the fluid simulation is based on the equations from the previous chapter, as well as the work of (Stam, 2003) [30]. The total amount of code is extensive, and so only a little part will be described in this chapter.

3.1 Memory Management

The majority of the simulation is written in C# which by default passes parameters by value, meaning essentially that when passing parameters to a method, *copies* of parameters are passed. While C# is optimized to handle pass-by-value parameters, copying large arrays every iteration would put pressure on the garbage collector. To avoid this, all arrays are passed by reference using the C# keyword **ref**, ensuring that the same arrays are used throughout the simulation.

3.2 Fluid Simulation

This section describes the various parts of the implementation of the fluid simulation.

3.2.1 Setup

Unity has two “startup” methods, Awake(), which is called first, and Start(), both of which are called before the main program is started. All variables are initialized in the Awake() method in the simulation script since other scripts are dependent on the variables initialized here. Other scripts then pass information back and forth in their Start() method. Most of the calculations that only has to be performed once are done in the Awake() and Start() methods, to increase performance. This is also where all arrays are initialized along with grid, texture and height field setup. Listing 3.1 show a few of the variables initialized in the Awake() method. Note that scalar fields as well as vector fields are represented using floating point arrays; as seen in line 1-2 a vector field consist of two floating point arrays.

```
 1 private float[ , ] u;   //Fluid (vector) velocity (flow) field along the x-axis
 2 private float[ , ] v;   //Fluid (vector) velocity (flow) field along the y-axis
 3 :
 4 //Constant Jacobi values for alpha and beta
 5 private float viscAlpha, viscBeta, viscReciBeta;
 6 private float projAlpha, projBeta, projReciBeta;
 7 :
 8 private void Awake() {
 9    :
10    u = new float[xLengthOfHeightField, yWidthOfHeightField];
11    v = new float[xLengthOfHeightField, yWidthOfHeightField];
12    :
13    viscAlpha = ((deltaX*deltaX)/(deltaT*kinematicViscosity));
14    viscBeta = 4.0f + viscAlpha;
15    viscReciBeta = 1.0f/viscBeta; //Reciprocal Beta
16
17    projAlpha = (deltaX*deltaX) * (-1f);
18    projBeta = 4.0f;
19    projReciBeta = 1.0f/projBeta; //Reciprocal Beta
20    :
```

Listing 3.1: Setup of the simulation program. Here all variables are initialized and precomputed if possible.

3.2.2 Main Loop

The main loop of the program is given by Listing 3.2. FixedUpdate() is a Unity3D-specific method, which run every 0.02 seconds. Besides performing the Navier-Stokes step, this is also where data from solids, user input etc. is added. Note that the arrays xVar and yVar are temporary arrays used to hold old values during computations and are therefore swapped with the velocity field twice every iteration. Once all calculations are done, the results are stored in a texture and passed on to the shader.

```
 1 private void FixedUpdate() {
 2    :
 3    //FULL NAVIER-STOKES STEP
 4    //Add Force
 5    AddForce ( N,  ref  u,  ref  xVar);
 6    AddForce ( N,  ref  v,  ref  yVar);
 7    Utils.Swap<float>(ref xVar, ref u);
 8    Utils.Swap<float>(ref yVar, ref v);
 9    //Viscous Diffusion
10    Jacobi( N, 1,  ref  u, ref  u,  viscAlpha,  viscReciBeta,  viscIterations);
11    Jacobi( N, 2,  ref  v, ref  v,  viscAlpha,  viscReciBeta,  viscIterations);
12    Utils.Swap<float>(ref xVar, ref u );
13    Utils.Swap<float>(ref yVar, ref v);
14    //Advect
15    Advect( N, 1, ref  u, ref  xVar, ref  xVar, ref  yVar, ref  gridPos,  dX,  dT);
16    Advect( N, 2, ref  v, ref  yVar, ref  xVar, ref  yVar, ref  gridPos,  dX,  dT);
17    //Project
18    Project( N,  ref  u,  ref  v,  ref  xVar,  ref  yVar,
19    .          projAlpha,  projReciBeta,  projIterations);
20    :
21 }
```

Listing 3.2: This is the main loop running every 0.02 seconds. The main part happening here is the computation of the Navier-Stokes equations.

3.2.3 Force

Force is applied in a separate script attached to the active camera object. The user can add force by click-and-drag on a separate plane standing beside the fluid in the scene. The plane has a texture attached and when the user clicks the mouse, force is added to the grid cell which correspond to the pixel in the texture. If the user click-and-drags the mouse over the plane, force is added in the direction of the vector created with this drag. Force is saved in the temporary arrays xVar and yVar in the simulation program and added to the program in line 5-6 in Listing 3.3.

```
1 private void AddForce (int _N, ref float[ , ] _x, ref float[ , ] _f) {
2    for (int i = 1 ; i <= _N ; i++ ) {
3       for (int j = 1 ; j <= _N ; j++ ) {
4          _x[i,j] += _f[i,j];
5       }
6    }
7 }
```

Listing 3.3: This is where the user input is added to the program.

3.2.4 Advection

The backward Euler, which is the part of advection that traces a fluid quantity back in time, is given by Equation 2.4, repeated here for convenience:

$$q(\mathbf{x}, t + \Delta t) = q(\mathbf{x}(t) - \mathbf{u}(\mathbf{x}, t)\Delta t,\ t)$$

The implementation of this equation is seen in line 20 in Listing 3.4. The previous quantity is then found using bilinear interpolation in line 21-23.

```
1  private Vector2 velTmp, posImplicit;
2
3  private static void Advect(int _N,
4                             int _boundary,
5                             ref float[ , ] _d,
6                             ref float[ , ] _d0,
7                             ref float[ , ] _u,
8                             ref float[ , ] _v,
9                             ref float[ , ] _gridPositions,
10                            float _dX,
11                            float _dT)
12 {
13    //Used for changing between positions and indexes of grid points
14    float posToGrid = 1.0f/_dX;
15
16    for(int i = 1; i <= _N; i++) {
17       for(int j = 1; j <= _N; j++) {
18          velTmp.x = _u[i,j];
19          velTmp.y = _v[i,j];
20
21          //Follow the velocity field "back in time"
22          posImplicit = _gridPositions[i,j] - (_dT * _dX * velTmp);
23
24          //Interpolate the values according to the nearest four cubes,
25          // and give the interpolated value to the first first argument _q[x,y]
26          Utils.BilinearInterpolation(ref _d[i,j],
27                                      new Vector2(posImplicit.x*posToGrid,
28                                                  posImplicit.y*posToGrid),
29                                      ref _d0);
30       }
31    }
32 }
```

Listing 3.4: Advection.

3.2.5 Jacobi

The Jacobi iteration given by Equation 2.8, repeated here for conveniency:

$$x_{i,j}^{k+1} = \frac{x_{i-1,j}^{k} + x_{i+1,j}^{k} + x_{i,j-1}^{k} + x_{i,j+1}^{k} + \alpha b_{i,j}}{\beta,}$$

The implementation of this equation is seen in line 14-15 in Listing 3.5.

3.2.6 Viscous Diffusion

The viscous diffusion a Jacobi iteration implemented in Listing 3.5 with constant α and β values given by Listing 3.1, and both x and b is the velocity field. Notice that the reciprocal of β is used and precomputed, since this should give a minor increase in performance as opposed to dividing with β.

```
private static void Jacobi(int _N,
                           int _boundary,
                           ref float[,] _x,
                           ref float[,] _b,
                           float _alpha,
                           float _reciprocalBeta,
                           int _jacobiIterations)
{
   for (int k = 0; k < _jacobiIterations; k++) {
      for(int i = 1; i <= _N; i++) {
         for(int j = 1; j <= _N; j++) {
            //Multiply with reciprocal Beta
            _x[i,j] = (_x[i-1,j] + _x[i+1,j] + _x[i,j-1] + _x[i,j+1] +
                           _alpha*_b[i,j]) * _reciprocalBeta;
         }
      }
   }
}
```

Listing 3.5: Jacobi Method.

3.2.7 Project

As described in Section 2.2.4, the projection step is what ensures mass conservation in the program. It is implemented in Listing 3.6; First the divergence is calculated using Equation 2.2 and implemented in line 16. Following this the Poisson-pressure equation is solved using the Jacobi() method again, this time with the values for α and β given in Listing 3.1, x is the pressure, initially set to zero in line 17, and b is the divergence from before. Finally in line 28-29 the gradient is calculated using Equation 2.1 and subtracted from the pressure field.

3.3 Fluid Coupling

This section describes the implementation of the solid-to-fluid and fluid-to-solid coupling. Due to time limitations only interaction with cubes and spheres are possible at the moment.

The buoyancy for a sphere is given by Equation 2.9 and calculated in Listing 3.7.

$$\mathbf{f}^{\text{buoyancy}} = -\mathbf{g}\rho V_{\text{sub}}$$

The drag of the object is given by Equation 2.11 and calculated in Listing 3.8.

$$\mathbf{f}^{\text{drag}} = -\frac{1}{2}\rho C_D A|\mathbf{u}_{\text{rel}}|\mathbf{u}_{\text{rel}}$$

Once calculated buoyancy is applied to the center of the submerged volume and the drag is added to the center of the object.

```
private static void Project(int _N,
                            ref float[ , ] _u,
                            ref float[ , ] _v,
                            ref float[ , ] _p,
                            ref float[ , ] _div,
                            int _jacobiIterations,
                            float _projAlpha,
                            float _projReciBeta,
                            float _dX,
                            float _dT)
{
   //Instead of dividing the gradient subtraction part with 2*deltaX,
   //  we instead multiply with (1/2)*delta x
   float halfDeltaX = 0.5f*_dX;
   float h = 1.0f/_N;

   //Calc divergence
   for (int i = 1; i <= _N; i++) {
      for (int j = 1; j <= _N; j++)  {
        _div[i,j] = -0.5f * h * (_u[i+1,j] - _u[i-1,j] + _v[i,j+1] - _v[i,j-1]);
        _p[i,j] = 0.0f;
      }
   }

   //Poisson-pressure
   Jacobi(_N, 0,  ref _p, ref _div, _projAlpha, _projReciBeta, _dT,
            _jacobiIterations);

   //Helmholtz-Hodge Decomposition - Gradient Subtraction
   for (int i = 1; i <= _N; i++) {
      for (int j = 1; j <= _N; j++) {
         _u[i,j] -= (_p[i+1,j]-_p[i-1,j])*halfDeltaX;
         _v[i,j] -= (_p[i,j+1]-_p[i,j-1])*halfDeltaX;
      }
   }
}
```

Listing 3.6: Project.

```
//Finds the volume V_submerged of the submerged part of the sphere
V_submerged = findV_Submerged(transform.position.y, r, surfaceHeight);

//Calc buoyancy: Gravity*density*submergedVolume
F_buoyancy = gravity * density_fluid * V_submerged * new Vector3(0,1,0);
```

Listing 3.7: Buoyancy Listing.

3.4 Rendering

This section describes the part of the visualization which handles vertices displacement of the fluid surface. This is implemented in a shader, written in CG.

3.4.1 Setup

Listing 3.9 contains the initialization of the shader, and of control parameters. It is a surface shader and the parts described in the following parts of this section are implemented where the dotted line is in Listing 3.9.

```
private void FixedUpdate() {
   F_inertiaDrag = calculateDrag(
      0.47f,    //drag coefficient for a sphere
      Mathf.PI*r*r,   //
      new Vector3(fluid.u[cubeI, cubeJ],
                  0,
                  fluid.v[cubeI, cubeJ]),
      rigidbody.velocity);
}

private static Vector3 calculateDrag(float _dragCoefficient,
                                     float _area,
                                     Vector3 _velocityOfFluid,
                                     Vector3 _velocityOfSolid)
{
   Vector3 relativeVelocity = _velocityOfSolid - _velocityOfFluid;
   return (0.5f)*_dragCoefficient*_area*
          Vector3.Magnitude(relativeVelocity)*relativeVelocity;
}
```

Listing 3.8: Inertia Drag Listing.

```
1  Shader "Custom/HeightFieldFluid" {
2     Properties {
3        _MainTex ("Base_(RGB)", 2D) = "white" {}
4        _FluidTex ("Fluid_texture", 2D) = "white" {}
5        _CubeMap ("Cube_Map", CUBE) = "" {}
6        //1.0003 for air, 1.3333 = water, 1.5 = glass/plastic, 2.417 = diamond
7        _RefractiveIndex("Refractive_Index", Float) = 1.3333
8        _AlphaOfFluid("Alpha_of_Fluid", Range(0,1)) = 0.5
9     }
10    SubShader {
11       Tags { "Queue"="Transparent" "RenderType"="Transparent" }
12
13       CGPROGRAM
14       #pragma glsl
15       #pragma target 3.0
16       #pragma surface surf Lambert vertex:vert alpha
17       .
18       .
19       .
20       ENDCG
21    }
22 }
```

Listing 3.9: Vertice Displacement.

3.4.2 Vertice Displacement

The result of the fluid simulation is a height field which is transferred to an 8bit texture, which is passed to this shader every time step. The vertices displacement is performed in line 3-4 in Listing 3.10. Note that the tex2Dlod method is a GLSL method, and the code is therefore translated to GLSL in line 14 in Listing 3.9.

```
void vert (inout appdata_full v, out Input o) {
    //For pixel displacement
    float fluidTex = tex2Dlod(_FluidTex, float4(v.texcoord.xy, 0.0, 0.0)).r;
    v.vertex.y = fluidTex;
}
```

Listing 3.10: Vertice Displacement.

4 Results and Discussion

Testing is an infinite process of comparing the invisible to the ambiguous in order to avoid the unthinkable happening to the anonymous.

James Bach

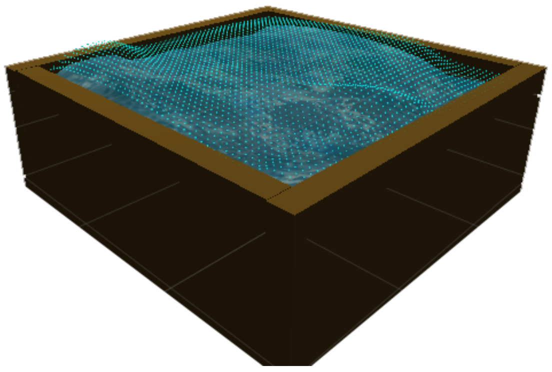
The goal of this thesis was to create a fluid simulation for the Unity3D game engine version 4.1.3, capable of simulating viscosity and solid coupling at minimum 60 FPS non-compiled. To this end the Navier-Stokes equations were implemented on a 2D grid which was used to control a height field.

4.1 Results

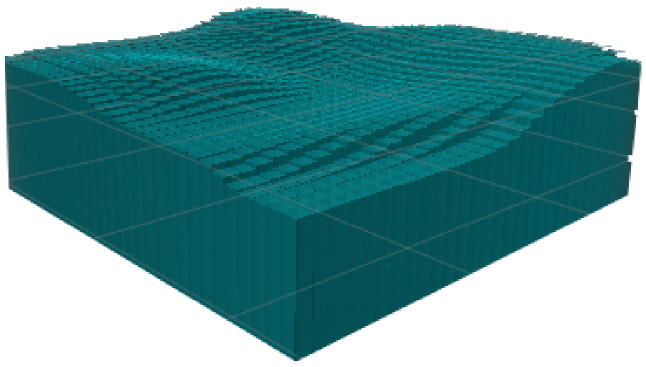
The tests were performed on a Macbook Pro, 2.66 GHz Intel Core 2 Duo, OS X 10.8.3. The performance of the simulation was tested against various parameters:

- Grid sizes
- Number of Jacobi steps for viscous diffusion
- Number of Jacobi steps for poisson-pressure
- Number of objects

The tests in this section will focus on performance, measured in FPS and lack of artifacts. For every time step the simulation will run through a number of Jacobi iterations for respectively viscosity and pressure project, so the first test was about testing the FPS for various grid sizes against the number of Jacobi iterations to find a reasonable trade off for the later test. The results in Table 4.1 clearly show that the simulation works at interactive rates for smaller grid sizes.



(a)



(b)

Figure 4.1: (a) The resulting simulation with the velocity field draw on top with lines. (b) The height behind the simulation.

The subsequent test used 10 pressure iterations and 5 viscosity iterations, since this combination produced reasonable results with no artifacts, while keeping below the minimum requirement of 60 FPS for grids of 48x48 and below.

The results in Table 4.2 show FPS at various grid sizes and various numbers of simple objects in the fluid. The results show that the simulation is capable of running with up to 40 primitive objects (spheres and cubes) at any of the three grid sizes while keeping within the minimum of 60 FPS. For smaller grid sizes, the simulation can handle 100+ primitive objects.

The results prove that the product can be used to simulate small bodies of fluid with various viscosity, user-applied force and two-way coupling with solids of adjustable densities at interactive rates (> 60 FPS). This makes it possible to use the simulation as part of a computer game.

To make the simulation more generally useable the size of the grid should be upped while still working at interactive rates. Previous research has proven

	Pressure and Viscosity Iterations					
Grid Sizes	40 & 20	30 & 15	20 & 10	10 & 5	4 & 2	2 & 1
64x64	< 3	< 3	5	19	67	75*
48x48	9	20	44	84	110	114*
32x32	65	79	105	130	141	150*
16x16	140	149	162	178	180	184*

Table 4.1: The above table shows the FPS values for various Viscosity and Pressure iterations for each time step. All results with the '*' symbol produced distinct artifacts.

	Objects						
Grid Sizes	1	10	20	40	60	80	100
48x48	80	78	75	60	54	50	45
32x32	129	128	122	102	90	71	61
16x16	178	170	148	155	170	178	190

Table 4.2: Testing the performance with various numbers of objects.

that fluid dynamics do well when implemented on the GPU; GPUs perform computations slowly on every fragment simultaneously as opposed to CPUs which perform computations fast, one at a time. Many algorithms for CFD based on the Eulerian method are parallel in nature since the same algorithms are often performed on each grid cell in a grid, and, as a consequence, are well suited for GPU implementation, which is also true for the algorithms used in this thesis. Therefore, the performance should increase with a GPU implementation, and it should then be possible to increase the grid size. A GPU implementation could therefore be the next step to increase performance and grid size.

5 Conclusion

Science is built up with facts, as a house is with stones. But a collection of facts is no more a science than a heap of stones is a house.

Henri Poincaré

The results show that the simulation can handle smaller bodies of fluids with various viscosity, user-applied force and two-way coupling with solids of adjustable densities at interactive rates (> 60 FPS).

Typically, the research by others focused on realistic appearance (either at interactive rates or offline rendering), solid coupling, viscosity or waves. Therefore, simulations exist that are capable of producing fluids that either appear more realistic, can handle larger grids, can perform more realistic solid coupling, solid coupling of more complex objects, or can produce more realistic waves than is possible with the product of this thesis. Many of these have been produced using CUDA or other platforms capable of handling General-Purpose computing on Graphics Processing Units (GPGPU).

What has been achieved in this thesis is the creation of a prototype, capable of handling simple versions of the above mentioned aspects at interactive rates. The prototype works within the Unity3D game engine on Unity3D-supported platforms, which make the simulation capable of running as part of a computer game.

6 Future Work

Imagine if every Thursday your shoes exploded if you tied them the usual way. This happens to us all the time with computers, and nobody thinks of complaining.

Jef Raskin

The present simulation can be extended in various ways. Here follows a description of a few that will either produce speed, precision and/or realism to the simulation.

6.1 GPGPU Implementation

The Unity3D game engine has supported compute shaders since version 4.0, which allows for massively parallel GPGPU algorithms. Modern CFD models often use massively parallel algorithms and should therefore benefit from being implemented into a compute shader. As of Unity3D version 4.1.3, GPGPU is built on top of DirectX 11 and therefore only works on Windows if the GPU supports Shader Model 5.0.

6.2 Shader Implementation

If the goal is to simulate fluids on multiple platforms, an alternative implementation into a "normal" shader is also possible, the greatest difference being that instead of arrays, textures are used to store vector and scalar fields. Textures typically have four color channels, each of which can store an array of scalars. Another difference appears when algorithms are performed on texture. Normally an algorithm would be placed within a nested loop and run on some or all of the elements of a grid. However, when an algorithm is performed on a texture in a shader, it runs on every pixel/fragment of the texture, and the algorithm must take this into account.

6.3 Staggered Grid

Implementing scalar fields in the center of each cell and vector fields in the boundaries of each cell should give a more stable and precise simulation. This was never fully implemented. See Section 1.4.1.1 for further details.

6.4 Fluid Coupling for Non-Simple Solids

Presently, the simulation works with primitive objects, such as spheres and cubes. However, the equations for complex objects described in Section 2.4 were never fully implemented. Doing so would make it possible to simulate a two-way fluid coupling between fluids and non-primitive solids.

6.5 Additional Substances

To make the simulation more useful, it should be made possible to add substances to the fluid, e.g. dye.

6.6 Vorticity

The occurrence of rotational flow is known as *vorticity* and is a well-studied phenomenon, which could add more realism to the simulation.

6.7 Breaking Waves

The present simulation is a height field and therefore cannot handle breaking waves. This can be implemented in various ways:

- Detect where waves should break and spawn particles for foam, spray and splashes that follow the underlying height field. Section 1.5 provides a few more details on the subject.

- Implement a Lagrangian or 3D Eulerian simulation on top of the height field, similar to the tall cell grid, described in Section 1.4.1

6.8 Interactivity and Custom Tool Development

Should the user wish to control the viscosity of the fluid, the density of objects etc., it is only possible by changing numbers in the editor. A more user-friendly tool should be implemented to make the simulation useable for non-programmers.

7 Epilogue

I think computer viruses should count as life ... I think it says something about human nature that the only form of life we have created so far is purely destructive. We've created life in our own image.

Stephen Hawking

The goal of this project was to create an interactive fluid simulation for the Unity3D game engine, capable of handling different kinds of fluids and coupling of solids. It was a requirement that the simulation should be stable, appear realistic and able to run at minimum 60 FPS on consumer hardware in order to be useable for computer games.

The first two months of this project was spent analyzing the existing work within the field of CFD for graphics to get an overview of previously used methods with their pros and cons regarding implementational difficulties, performance, solid coupling and realism. The purpose of this analysis was to find the optimal solution which could handle the above mentioned criteria, while also being easy (and therefore fast) to implement.

A Lagrangian approach was decided against for a number of reasons; a Lagrangian approach will waste a lot of processing power on particles far below the surface; it is far from clear how to render a smooth surface from a massive amount of particles. Also, the Unity3D game engine uses the PhysX physics engine developed by Nvidia, which already contains a fluid simulation based on SPH. As of Unity3D version 4.1.3, the fluid simulation is not included in Unity3D, but it is possible that it will be included in a future version.

In the end, a CPU-based simulation combining a height field with the Eulerian approach was chosen for a number of reasons:

- It is relatively easy to implement.
- Coupling with solids can be achieved with a great deal of precision.
- Performance wise the use of a height field gives the appearance of a 3D simulation, while in effect done on a 2D grid.
- The transfer from a 2D grid to a texture is trivial when the amount of grid cells equal the amount of pixels in the texture.
- The algorithms used with the Eulerian approach can often be used in a later implementation on a GPU.

The consequence of this choice is that the detail of the product is in the low end; the product cannot handle large grids (larger than 48x48), meaning that it cannot in its present state simulate larger basins or water, such as lakes, rivers etc., without producing visual artifacts.

References

[1] M. Harris, "Fast fluid dynamics simulation on the gpu," *GPU gems*, vol. 1, pp. 637–665, 2004. 1, 18, 19

[2] R. Bridson and M. Müller-Fischer, "Fluid simulation: Siggraph 2007 course notes," in *ACM SIGGRAPH 2007 courses*, pp. 1–81, ACM, 2007. 1, 8, 11, 20, 47

[3] Bioware, *Mass Effect 2 (Version 1.02) [Software]*. Redwood City, USA: Electronic Arts Inc., January 2010. 2

[4] Toys For Bob, *Skylander: Spyro's Adventure [Software]*. Santa Monica, USA: Activision, October 2011. 2

[5] Cyan Worlds, *Uru: Ages Beyond Myst [Software]*. Washington, USA: Ubisoft Montpellier, November 2003. 2, 11

[6] Valve Corporation, *Portal 2 [Software]*. Kirkland, USA: Valve Corporation, April 2011. 2

[7] Ubisoft Montpellier, *From Dust [Software]*. Montreuil, France: Ubisoft Entertainment S.A., July 2011. 2

[8] Gearbox Software, *Borderlands 2 [Software]*. Novato, USA: 2K Games, September 2012. 2

[9] M. J. Gourlay, "Fluid simulation for video games (part 1-15)," 2009. 7

[10] G. Irving, E. Guendelman, F. Losasso, and R. Fedkiw, "Efficient simulation of large bodies of water by coupling two and three dimensional techniques," *ACM Transactions on Graphics (TOG)*, vol. 25, no. 3, pp. 805–811, 2006. 8

[11] N. Chentanez and M. Müller, "Real-time eulerian water simulation using a restricted tall cell grid," in *ACM Transactions on Graphics (TOG)*, vol. 30, p. 82, ACM, 2011. 8

[12] F. H. Harlow and J. E. Welch, "Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface," *Physics of fluids*, vol. 8, p. 2182, 1965. 8

[13] L. B. Lucy, "A numerical approach to the testing of the fission hypothesis," *The astronomical journal*, vol. 82, pp. 1013–1024, 1977. 11

[14] R. A. Gingold and J. J. Monaghan, “Smoothed particle hydrodynamics-theory and application to non-spherical stars,” *Monthly notices of the royal astronomical society*, vol. 181, pp. 375–389, 1977. 11

[15] Wikipedia, “Smoothed-particle hydrodynamics — wikipedia, the free encyclopedia.” `http://en.wikipedia.org/w/index.php?title=Smoothed-particle_hydrodynamics&oldid=540773130`, 2013. [Online; accessed 8-March-2013]. 11

[16] M. Finch, “Effective water simulation from physical models,” *GPU Gems*, vol. 1, pp. 5–29, 2004. 11

[17] A. Iglesias, “Computer graphics for water modeling and rendering: a survey,” *Future generation computer systems*, vol. 20, no. 8, pp. 1355–1374, 2004. 12

[18] A. Fournier and W. T. Reeves, “A simple model of ocean waves,” *ACM Siggraph Computer Graphics*, vol. 20, no. 4, pp. 75–84, 1986. 12

[19] M. Carlson, P. J. Mucha, and G. Turk, “Rigid fluid: animating the interplay between rigid bodies and fluid,” in *ACM Transactions on Graphics (TOG)*, vol. 23, pp. 377–384, ACM, 2004. 13

[20] Wikipedia, “Frame rate — wikipedia, the free encyclopedia.” `http://en.wikipedia.org/w/index.php?title=Frame_rate&oldid=547987301`, 2013. [Online; accessed 9-May-2013]. 15

[21] R. Fernando, E. Haines, and T. Sweeney, *GPU Gems: Programming Techniques, Tips & Tricks for Real-Time Graphics*. Addison-Wesley Professional, 2004. 17

[22] Wikipedia, “Bilinear interpolation — wikipedia, the free encyclopedia.” `http://en.wikipedia.org/w/index.php?title=Bilinear_interpolation&oldid=540473612`, 2013. [Online; accessed 11-May-2013]. 17

[23] J. Stam, “Stable fluids,” in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pp. 121–128, ACM Press/Addison-Wesley Publishing Co., 1999. 18

[24] A. J. Chorin and J. E. Marsden, *A mathematical introduction to fluid mechanics*. Springer New York, 3 ed., 1993. 18

[25] G. H. Golub and C. F. Van Loan, *Matrix computations. 1996*, pp. 509–520. Johns Hopkins University Press, 1996. 19

[26] C. Yuksel, D. H. House, and J. Keyser, “Wave particles,” *ACM Transactions on Graphics (TOG)*, vol. 26, no. 3, p. 99, 2007. 21

[27] N. Chentanez and M. Müller, “Real-time simulation of large bodies of water with small scale details,” in *Proceedings of the 2010 ACM SIGGRAPH/Eurographics symposium on computer animation*, pp. 197–206, Eurographics Association, 2010. 21

[28] Wikipedia, “Snell’s law — wikipedia, the free encyclopedia.” `http://en.wikipedia.org/w/index.php?title=Snell%27s_law&oldid=550947773`, 2013. [Online; accessed 20-May-2013]. 22

[29] R. Fernando and M. J. Kilgard, *The Cg Tutorial: The definitive guide to programmable real-time graphics*, pp. 119–137. Addison-Wesley Longman Publishing Co., Inc., 2003. 23

[30] J. Stam, “Real-time fluid dynamics for games,” in *Proceedings of the Game Developer Conference*, vol. 18, 2003. 25

A Appendix Math

A.1 Vector Calculus

Throughout this thesis, the following three operators have been used; The gradient $\boldsymbol{\nabla}$, the divergence $\boldsymbol{\nabla}\cdot$ and the Laplacian $\boldsymbol{\nabla}\cdot\boldsymbol{\nabla}$. The examples in this appendix are based on (Bridson and Müller-Fischer, 2007) [2] and assumes that the operators are used in three dimensions, unless otherwise stated.

A.1.1 Gradient

The gradient, denoted with the *nabla* operator, $\boldsymbol{\nabla}$, or sometimes *grad*, takes a function and returns a vector of the spatial partial derivatives of it.

$$\boldsymbol{\nabla} f(x,y,z) = \left[\frac{\partial f}{\partial x}\ \frac{\partial f}{\partial y}\ \frac{\partial f}{\partial z}\right]^T$$

However, the gradient is not always used together with a function, so a different, and perhaps more useful, notation is:

$$\boldsymbol{\nabla} = \left[\frac{\partial}{\partial x}\ \frac{\partial}{\partial y}\ \frac{\partial}{\partial z}\right]^T$$

In this report, the gradient is often used together with a scalar. This results in a vector;

$$\boldsymbol{\nabla} p = \left[\frac{\partial p}{\partial x}\ \frac{\partial p}{\partial y}\ \frac{\partial p}{\partial z}\right]^T$$

If the gradient is used together with a vector, it results in a matrix;

$$\boldsymbol{\nabla}\mathbf{u} = \boldsymbol{\nabla}\left[u\ v\ w\right]^T = \begin{pmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & \frac{\partial u}{\partial z} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} & \frac{\partial v}{\partial z} \\ \frac{\partial w}{\partial x} & \frac{\partial w}{\partial y} & \frac{\partial w}{\partial z} \end{pmatrix}$$

The finite difference form of the gradient in two dimensions is given by:

$$\begin{aligned}\boldsymbol{\nabla} p &= \left[\frac{\partial p}{\partial x}\ \frac{\partial p}{\partial y}\right]^T \\ &= \frac{p_{i+1,j} - p_{i-1,j}}{2\Delta x}, \frac{p_{i,j+1} - p_{i,j-1}}{2\Delta y},\end{aligned}$$

where i and j refer to the index of individual grid cells, and Δx and Δy is the distance between neighboring grid cells along the x-axis and y-axis, respectively. In fluid dynamics, it is often the case that $\Delta x = \Delta y$, in which case the previous equation is simplified to:

$$\boldsymbol{\nabla} p = \frac{p_{i+1,j} - p_{i-1,j}}{2\Delta x}, \frac{p_{i,j+1} - p_{i,j-1}}{2\Delta x}$$

In three dimensions:

$$\begin{aligned}\boldsymbol{\nabla} p &= \left[\frac{\partial p}{\partial x}\ \frac{\partial p}{\partial y}\ \frac{\partial p}{\partial z}\right]^T \\ &= \frac{p_{i+1,j,k} - p_{i-1,j,k}}{2\Delta x}, \frac{p_{i,j+1,k} - p_{i,j-1,k}}{2\Delta y}, \frac{p_{i,j,k+1} - p_{i,j,k-1}}{2\Delta z},\end{aligned}$$

where i, j and k refer to the index of individual grid cells, and Δx, Δy and Δz is the distance between neighboring grid cells along the x-axis, y-axis and z-axis, respectively. In fluid dynamics, it is often the case the $\Delta x = \Delta y = \Delta z$, in which case the previous equation is simplified to:

$$\boldsymbol{\nabla} p = \frac{p_{i+1,j,k} - p_{i-1,j,k}}{2\Delta x}, \frac{p_{i,j+1,k} - p_{i,j-1,k}}{2\Delta x}, \frac{p_{i,j,k+1} - p_{i,j,k-1}}{2\Delta x}$$

A.1.2 Divergence

The divergence, denoted with the $\boldsymbol{\nabla}\cdot$ operator, or sometimes *div*, can only be applied to vector fields; the input is a vector field and the output is a scalar scalar. It measures how much the vectors are converging or diverging at any point. In relation to fluid dynamics, it measures how much of a quantity (which can be velocity, pressure, density etc.) is exiting and entering a given point:

$$\boldsymbol{\nabla} \cdot \mathbf{u} = \boldsymbol{\nabla} \cdot \left[u\ v\ w\right]^T = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}$$

The divergence operator looks like the dot-product between a gradient and the vector field that comes after, which is exactly what it does:

$$\begin{aligned}\boldsymbol{\nabla} \cdot \mathbf{u} &= \left(\frac{\partial}{\partial x} + \frac{\partial}{\partial y} + \frac{\partial}{\partial z}\right) \cdot \left[u\ v\ w\right]^T \\ &= \frac{\partial}{\partial x} u + \frac{\partial}{\partial y} v + \frac{\partial}{\partial z} w\end{aligned}$$

The finite difference form of the divergence in two dimensions is given by:

$$\begin{aligned}\boldsymbol{\nabla}\cdot\mathbf{u} &= \frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}\\ &= \frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}+\frac{v_{i,j+1}-v_{i,j-1}}{2\Delta y},\end{aligned}$$

where i and j refer to the index of individual grid cells, and Δx and Δy is the distance between neighboring grid cells along the x-axis and y-axis, respectively. In fluid dynamics, it is often the case that $\Delta x = \Delta y$, in which case the previous equation is simplified to:

$$\boldsymbol{\nabla}\cdot\mathbf{u} = \frac{u_{i+1,j}-u_{i-1,j}+v_{i,j+1}-v_{i,j-1}}{2\Delta x}$$

In three dimensions:

$$\begin{aligned}\boldsymbol{\nabla}\cdot\mathbf{u} &= \frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}+\frac{\partial w}{\partial z}\\ &= \frac{u_{i+1,j,k}-u_{i-1,j,k}}{2\Delta x}+\frac{v_{i,j+1,k}-v_{i,j-1,k}}{2\Delta y}+\frac{w_{i,j,k+1}-w_{i,j,k-1}}{2\Delta z},\end{aligned}$$

where i, j and k refer to the index of individual grid cells, and Δx, Δy and Δz is the distance between neighboring grid cells along the x-axis, y-axis and z-axis, respectively. In fluid dynamics, it is often the case the $\Delta x = \Delta y = \Delta z$, in which case the previous equation is simplified to:

$$\boldsymbol{\nabla}\cdot\mathbf{u} = \frac{u_{i+1,j,k}-u_{i-1,j,k}+v_{i,j+1,k}-v_{i,j-1,k}+w_{i,j,k+1}-w_{i,j,k-1}}{2\Delta x}$$

A.1.3 Laplacian

The Laplacian is denoted with the $\boldsymbol{\nabla}\cdot\boldsymbol{\nabla}$ operator, $\boldsymbol{\nabla}^2$ or Δ. As the operator implies, the laplacian is the dot-product of two gradients, or, more correctly, the divergence of the gradient:

$$\boldsymbol{\nabla}\cdot\boldsymbol{\nabla}f = \frac{\partial^2 f}{\partial x^2}+\frac{\partial^2 f}{\partial y^2}+\frac{\partial^2 f}{\partial z^2}$$

A few notes of interest; the partial differential equation $\boldsymbol{\nabla}\cdot\boldsymbol{\nabla}f = 0$ is called Laplace's equation. And if the right-hand side is replaced by something non-zero, $\boldsymbol{\nabla}\cdot\boldsymbol{\nabla}f = q$, it is called the Poisson equation.

The finite difference form of the Laplacian in two dimensions is given by:

$$\begin{aligned}\boldsymbol{\nabla}\cdot\boldsymbol{\nabla}p &= \frac{\partial^2 p}{\partial x^2}+\frac{\partial^2 p}{\partial y^2}\\ &= \frac{p_{i+1,j}-2p_{i,j}+p_{i-1,j}}{(\Delta x)^2}+\frac{p_{i,j+1}-2p_{i,j}+p_{i,j-1}}{(\Delta y)^2},\end{aligned}$$

where i and j refer to the index of individual grid cells, and Δx and Δy is the distance between neighboring grid cells along the x-axis and y-axis, respectively. In fluid dynamics, it is often the case that $\Delta x = \Delta y$, in which case the previous equation is simplified to:

$$\boldsymbol{\nabla}\cdot\boldsymbol{\nabla}p = \frac{p_{i+1,j}+p_{i-1,j}+p_{i,j+1}+p_{i,j-1}-4p_{i,j}}{(\Delta x)^2}$$

In three dimensions:

$$\begin{aligned}\boldsymbol{\nabla}\cdot\boldsymbol{\nabla}p &= \frac{\partial^2 p}{\partial x^2}+\frac{\partial^2 p}{\partial y^2}+\frac{\partial^2 p}{\partial z^2}\\ &= \frac{p_{i+1,j,k}-2p_{i,j,k}+p_{i-1,j,k}}{(\Delta x)^2}+\frac{p_{i,j+1,k}-2p_{i,j,k}+p_{i,j-1,k}}{(\Delta y)^2}+\frac{p_{i,j,k+1}-2p_{i,j,k}+p_{i,j,k-1}}{(\Delta z)^2},\end{aligned}$$

where i, j and k refer to the index of individual grid cells, and Δx, Δy and Δz is the distance between neighboring grid cells along the x-axis, y-axis and z-axis, respectively. In fluid dynamics, it is often the case the $\Delta x = \Delta y = \Delta z$, in which case the previous equation is simplified to:

$$\boldsymbol{\nabla}\cdot\boldsymbol{\nabla}p = \frac{p_{i+1,j,k}+p_{i-1,j,k}+p_{i,j+1,k}+p_{i,j-1,k}+p_{i,j,k+1}+p_{i,j,k-1}-6p_{i,j,k}}{(\Delta x)^2}$$

A.1.4 Calculation Rules

$$(\mathbf{u}\cdot\mathbf{v})f = (\mathbf{v}\cdot\mathbf{u})f \tag{A.1}$$

$$(\boldsymbol{\nabla}\cdot\mathbf{u})f \neq (\mathbf{u}\cdot\boldsymbol{\nabla})f \tag{A.2}$$

$$(\boldsymbol{\nabla}\cdot\mathbf{u})f = \left(\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}+\frac{\partial w}{\partial z}\right)f = \frac{\partial u}{\partial x}f+\frac{\partial v}{\partial y}f+\frac{\partial w}{\partial z}f \tag{A.3}$$

$$(\mathbf{u}\cdot\boldsymbol{\nabla})f = \left(u\frac{\partial}{\partial x}+v\frac{\partial}{\partial y}+w\frac{\partial}{\partial z}\right)f = u\frac{\partial f}{\partial x}+v\frac{\partial f}{\partial y}+w\frac{\partial f}{\partial z} \tag{A.4}$$

$$(\mathbf{u}\cdot\boldsymbol{\nabla})\mathbf{f} = \begin{pmatrix}\left(u\frac{\partial}{\partial x}+v\frac{\partial}{\partial y}+w\frac{\partial}{\partial z}\right)f\\ \left(u\frac{\partial}{\partial x}+v\frac{\partial}{\partial y}+w\frac{\partial}{\partial z}\right)g\\ \left(u\frac{\partial}{\partial x}+v\frac{\partial}{\partial y}+w\frac{\partial}{\partial z}\right)h\end{pmatrix} \tag{A.5}$$

$$= \begin{pmatrix}u\frac{\partial f}{\partial x}+v\frac{\partial f}{\partial y}+w\frac{\partial f}{\partial z}\\ u\frac{\partial g}{\partial x}+v\frac{\partial g}{\partial y}+w\frac{\partial g}{\partial z}\\ u\frac{\partial h}{\partial x}+v\frac{\partial h}{\partial y}+w\frac{\partial h}{\partial z}\end{pmatrix} \tag{A.6}$$