NTNU
Innovation and Creativity

Visualization of water surface using GPU

Jostein Gustavsen
Dan Lewi Harkestad

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Torbjørn Hallgren, IDI
Co-supervisor: Odd Erik Gundersen, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

This is a continuation of the work done in the in-depth study of water visualization during fall 2005. The thesis will focus on improving this work on physical water surface models (using the shallow water equations based on Navier-Stokes with the conjugate gradient method as the solver), as well as research and implement alternative methods. This could be volume based models, and using other solvers like Jacobi and other variants of the conjugate gradient solver.

The results will then be compared and evaluated to see what might be the best approach for visualizing a water surface and water waves based on physical models. Based on these data, one method is selected and implemented in the Sverresborg simulation, if a usable method is found.

Assignment given: 20. January 2006
Supervisor: Torbjørn Hallgren, IDI

Table of Contents

Chapter 1

Introduction

This chapter gives an introduction to our work on simulating water on the GPU. It presents our motivation for improving existing methods of water simulation as well as our objectives and requirements. The approach to achieve our goals is also stipulated. The chapter is round up with a description of the structure of the paper.

1.1 Motivation

The motivations for this project were to enhance the water in the well in the Sverresborg simulation, and explore the new possibilities for enhancing real-time rendering using the graphics processing unit (GPU).

The Sverresborg simulation was a masters project done by the former NTNU students Espen Almdahl and Bård Terje Fallan during spring 2005. It was a continuation of former student projects and shows a three-dimensional presentation of the every day life in the Sverresborg fortress in the 12^{th} and 13^{th} century. The project was extensive and they had to cut a couple of corners. One of them were the water surface in the well in the fortress, where the water is a textured, non-moving flat surface. Our supervisor presented the opportunity to create a better water surface simulation for Sverresborg. This proved interesting, especially since the animation- and special effects industry has shown that a very beautiful body of water can be rendered offline, and the gaming industry has shown that realistic water surfaces can be rendered in real-time.

Another motivation was that most of the new graphics cards have programmable processing units. This new possibility seemed very interesting since it is not only applicable in pure graphics applications like games and simulations, but also in general purpose programming. The possibility to program the GPU is used not only by the graphics community, but also researchers and engineers. It is therefore interesting to acquire knowledge about programmable GPUs.

1.2 Objectives and Requirements

Our objectives for this project are:

- Comprehend current research on water simulation using physical models. This is an interesting, ever-changing field. In the last fifteen years much research has been done on simulating real-time water. To find a method that best suits our objectives and requirements, we need to look at articles all the way back to the beginning of real-time water rendering. This will make us able to follow and understand the differences between each paper, and give us a better understanding when researching newer articles.
- Find a suitable method to implement. This goal is closely connected to the previous goal, and without a good method to implement we will not get the results we require. The method or methods we base our implementation on may or may not have been implemented on the GPU before. Our implementation may therefore be based on a single method or a mix of several existing methods.
- The water surface should be quadratic with a surrounding well.
- Implement the chosen method for the GPU using OpenGL and GLSL, and compare with a similar implementation for the CPU.
- Implement a lighting model with reflections, refractions, and a surrounding environment.

Our requirements will be met if we are able to render a physical based real-time water surface on the GPU, and collect result to compare with a CPU implementation.

1.3 Approach

The following approach will be taken to meet our goals:

- Research previous work on simulation of water surfaces. This includes the research of offline simulations to get a grasp of most techniques that are, and have been used for water simulation. Former offline simulations may be interesting since they may be able to run in real-time on todays graphics hardware. From there on we will work our way up to todays cutting-edge simulations and choose one or several methods to integrate into our own model.
- Combine work into a method suitable for implementation on the GPU. After finding a suitable water model, and a suitable algorithm to be implemented on the GPU, we need to merge these into a method to implement.
- Implement the method for CPU. After a suitable method has been found, we need to implement it for the CPU to eliminate errors that are related to the algorithm, and not the GPU implementation. If the method fails to meet our requirements for simulation on the CPU, it will likely not meet our requirements for simulations on the GPU.
- Implement the final method on the GPU. The method needs to be programmed for efficient use of the GPU. It is difficult to predict the speed of the GPU implementation compared to the implementation for the CPU.

- Compare the result from the CPU implementation to the GPU implementation. If we implement different solvers these should also be compared.

1.4 Structure

The thesis is divided into seven chapters:

Introduction is a short introduction of the report. It shows our motivation, our objectives, and the approach we are planning to take to solve the simulation of the water.

Background describes the theory behind the motion of water, water optics, and gives a short introduction to programmable GPUs and some of the available languages.

Related research describes important and popular approaches to water simulation. We mention both real-time and non real-time approaches, and some pros and cons for the two main approaches we have investigated; grid-based waves and physically based waves.

Conceptual design describes one model we abandoned after implementing it for the CPU; Fourier based waves, and the final model we decided to implement; physically based waved.

Implementation contains the description of the implementation of the final model we decided on.

Experiments & results describe the experiments we did, and the results from these. A general discussion is also included.

Discussion contains an evaluation of our results, the conclusion and a description of possible future work.

1.5 Summary

The motivation for this project was to enhance a former student project, and to learn about new and exiting possibility for efficient parallel computations on ordinary desktop computers. We start by researching different approaches to water rendering, and look into general programming on the GPU. Thereafter we try to compose a method that is suitable for implementation on the GPU. This method will first be implemented for the CPU, and if that is successful we will try to implement it on the GPU.

Chapter 2

Background

Motion of water can be described by the means of fluid dynamics. Fluid dynamics has been studied by engineers for a long time. The flow of air is important for car and aircraft manufacturers, and the flow of liquids are important for e.g. mechanical engineers.

The appearance of water is dependent on several factors, like caustics (patterns of light on the pool floor and shafts of light), murk (how light filters out over distance), reflection and refraction (the amount of light from above and below water). It is also dependent on artefacts like algae or fine sand as well as the colour of the environment. E.g. if the sky is cloudy the water surface becomes grey, but with a clear blue sky the water gets a bluish look.

Graphics processing units (GPUs) in modern commercial graphics cards are now being considered very cost effective. Recent GPUs have full programmability and floating pointer arithmetic which also allow general purpose computations on the GPU (GPGPU). Engineers and scientists are devoting a significant amount of time on new algorithms for GPGPU to increase the speed of the calculations.

2.1 Motion of Water

Fluid dynamics involve many properties of the fluid, such as velocity, pressure, density, and temperature. The fundaments of fluid dynamics is the conservation laws; conservation of mass, conservation of momentum (Newton's first law), and conservation of energy. Fluid particles behave according to the laws of physics. In general, particles behave according to Newton's second law of motion, $F = ma$.

In the nineteenth century the French engineer and physicist Claude-Louis Navier and the Irish mathematician and physicist George Gabriel Stokes applied Newton's second law to fluid dynamics. They derived the Navies-Stokes equations which establish that changes in the momentum (acceleration) of the particles of a fluid are the product of changes in pressure and friction acting inside the fluid. The equations can take many forms depending on the assumptions made. Water is usually simulated using the

Navier-Stokes partial differential equations for incompressible fluids since water is nearly incompressible [1]. Water will hereafter be referred to as incompressible.

From [2]: The Navier-Stokes equations describe the flow of incompressible fluids. There are three forces acting on fluid: The viscous[1] force, the pressure force and the body force.

$$\frac{\mathbf{F}_{viscous}}{V} = \eta \nabla^2 \mathbf{u} \tag{2.1}$$

$$\frac{\mathbf{F}_{pressure}}{V} = -\nabla P \tag{2.2}$$

$$F = \frac{\mathbf{F}_{body}}{V} \tag{2.3}$$

Adding the forces, equating them to Newton's law of fluid and dividing by the density ρ yields:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\frac{1}{\rho} \nabla P + \nu \nabla^2 \mathbf{u} + \frac{F}{\rho} \tag{2.4}$$

where the kinematic viscosity is defined by $\nu \equiv \frac{\eta}{\rho}$.

When we look at (2.4) in Cartesian coordinate components of the velocity vector given by $\mathbf{u} = (u, v, w)$, and set the body force to be gravity $\mathbf{g}$, the Navier-Stokes equations are given by

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = g_x - \frac{1}{\rho}\frac{\partial P}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) \tag{2.5}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} = g_y - \frac{1}{\rho}\frac{\partial P}{\partial y} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) \tag{2.6}$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} = g_z - \frac{1}{\rho}\frac{\partial P}{\partial z} + \nu\left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right) \tag{2.7}$$

The equations are usually applied with the continuity equation, which states the law of conservation of mass:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{2.8}$$

Since the equations are in three dimensions, the execution time for algorithms based on the full equations is $O(N^3)$. It is therefore difficult to solve these equations fast. But by assuming zero viscosity and only considering two-dimensional motions we get a much simpler set of equations, which describe flows of thin layers (Haltiner and Williams

[1] The viscosity of a fluid can be defined as the measure of how resistive the fluid is to flow.

1980):

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + g\frac{\partial h}{\partial x} = 0 \tag{2.9}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + g\frac{\partial h}{\partial y} = 0 \tag{2.10}$$

$$\frac{\partial h}{\partial t} + \frac{\partial}{\partial x}\left[u\left(h-b\right)\right] + \frac{\partial}{\partial y}\left[v\left(h-b\right)\right] = 0 \tag{2.11}$$

where h is the height field, b is the bottom height, u and v are the velocity in the x-, and y-direction respectably. Equations (2.9) and (2.10) are derived from Newton's second law of motion, $F = ma$. Equation (2.11) is an equation of conservation of mass.
Other 2D equations may be derived from other assumptions.

2.2 Optics of water

To understand how water optics works it is important to look at how light behaves when we look at water. Water is transparent, but also a near perfect specular reflector. In a perfect specular reflection, the reflected ray of light has the same angle as an incoming ray of light with respect to the surface normal. Water has a different refractive index than air which causes the light to either be reflected or transmitted inside the other medium (see figure 2.1).

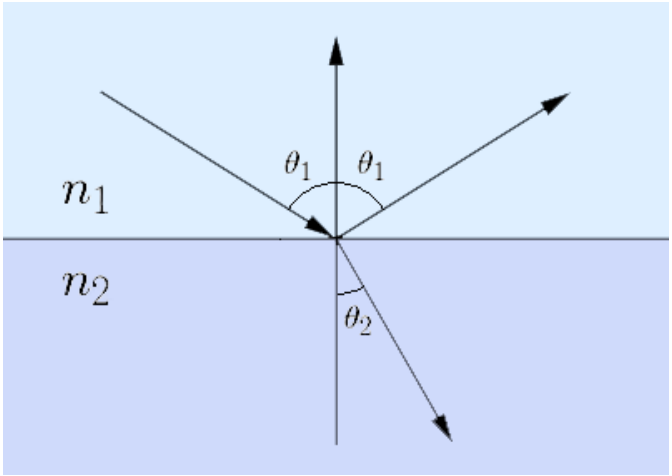

Figure 2.1: Reflection and refraction on air/water boundary

If we assume perfect specular reflection, the angle of refraction can be obtained from Snell's law:

$$n_1 \sin\left(\theta_1\right) = n_2 \sin\left(\theta_2\right) \tag{2.12}$$

where n_1 and n_2 are the refractive index for the two media. The approximated reflective index is 1.00 for air, and 1.33 for water.

If we look straight into the water, it looks different that when we look at the horizon. When looking straight down into the water we can clearly see what lies beneath the surface as long as the water is clear. When looking at the horizon it will look more like a mirror which reflects the sky. Augustin-Jean Fresnel deduced the Fresnel equations which describe the behaviour of light when moving from one medium to another. R is the reflection coefficient, and T is the refraction coefficient. No light is lost, so $R+T=1$. Since most light is non-polar the reflection coefficient is calculated by:

$$R = \frac{1}{2}\left(\left[\frac{n_1\cos(\theta_1) - n_2\cos(\theta_2)}{n_1\cos(\theta_1) + n_2\cos(\theta_2)}\right]^2 + \left[\frac{n_1\cos(\theta_2) - n_2\cos(\theta_1)}{n_1\cos(\theta_2) + n_2\cos(\theta_1)}\right]^2\right) \tag{2.13}$$

In 1994 Schlick [3] introduced a rational approximation of Fresnels equations that give adequate results for simulations. OpenGL Shading Language (Orange Book) [4] uses Schlick's approximation in an example.

$$f = \frac{(1.0 - \frac{n1}{n2})^2}{(1.0 + \frac{n1}{n2})^2} \tag{2.14}$$

$$R = f + (1-f)(1-\theta_1)^5 \tag{2.15}$$

2.3 Programmable GPU

Vector arithmetic and very many pipelines make the GPU a powerful SIMD (Single Instruction Multiple Data) platform. GPUs were designed to work on vertices and fragments, and they are therefore effective on vector and matrix operations. This can greatly improve the speed of graphics programs like simulations and games.

Programs on the GPU are called shader programs. They can be divided into vertex shaders and fragment shaders and are used according to figure 2.2.

2.3.1 Vertex shaders

Vertex shaders perform calculations during the Vertex Shading stage (also called Transform and Lighting) in 2.2. They perform mathematical operations on vertex data, but can not create or destroy vertices. The vertices are defined by its location in the 3D environment, but can also have colour, textures and lighting properties. The vertex shader can manipulate these data to give objects a different position, different colour, or different texture. The results from the vertex shader computations are used to provide the graphics pipeline with interpolated fragments attributes. Vertex shaders are often used to deform objects, and computing linearisable attributes that are to be used in the fragment shaders.

Input to vertex shaders are uniform variables, and the vertex attributes already mentioned. Uniform variables are usually constant basic data types that can only be changed between each invocation. Varying attributes can be defined and passed onto the fragment shaders. The newest shader models are able to do texture lookup also in the vertex shader. This has usually been confined to the fragments shaders.

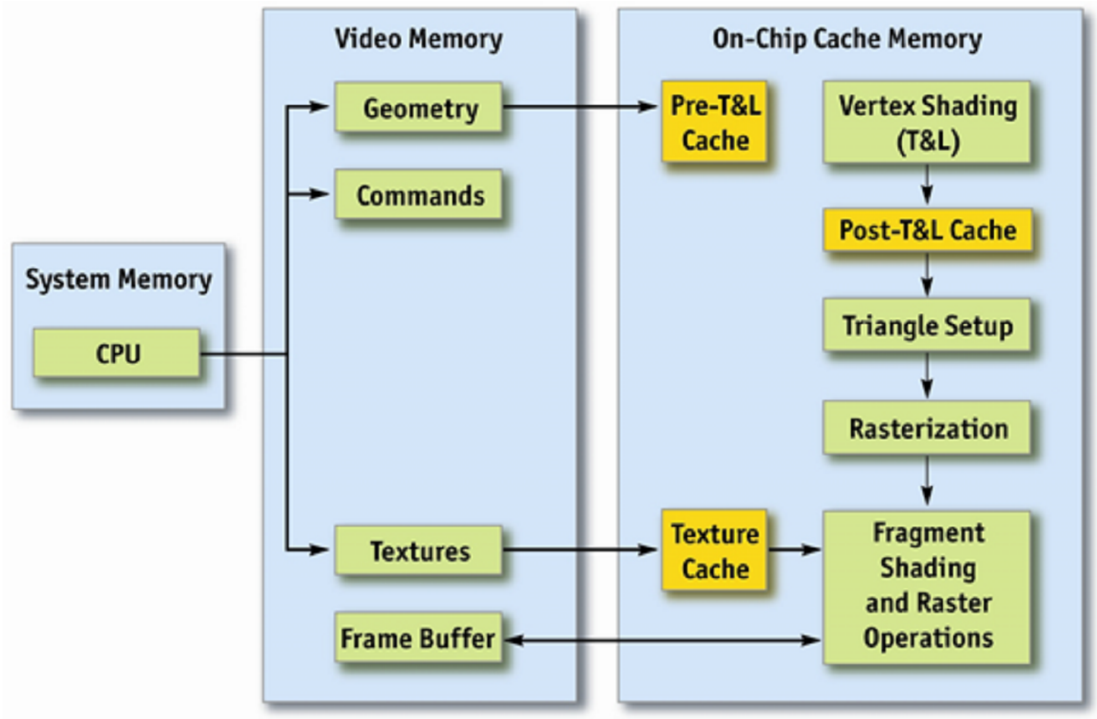

Figure 2.2: GPU Pipeline from NVIDIA GPU Programming Guide [5]

2.3.2 Fragment shaders

The fragment processor usually has much more processing power than the vertex processor. It performs calculations on every fragment that is produced by the rasterization stage. With high resolution this creates a huge computational load. The programmable fragment shaders (also called pixel shaders) give the opportunity to add effects that requires better control than computations on a triangular level can give. They are typically used for lighting models, texture access, fog and other environmental properties. They are also used in vector and matrix computations. Pixels can be discarded by the fragment shader. Input to fragments shaders are almost the same as for vertex shaders, but the fragments shaders can not change varying attributes as they are not passed on.

2.3.3 Shading languages

There are mainly three shading languages that are used for real-time rendering today; Cg programming language, DirectX High-Level Shader Language, and OpenGL shading language. The languages have a C-like syntax, and the developer writes explicit vertex and fragment programs. The code is compiled to a pseudo-assembly which are used differently for each language. Like in C each shader program has one main function which starts it.

Cg programming language is a language developed by nVidia. It is platform and architecture independent, and were one of the first GPGPU languages used widely.

It is still used widely today.

DirectX high-level shader language (HLSL) is a language developed by Microsoft. Its support from Microsoft and tight connection to DirectX makes it a natural choice for many graphics application developers.

OpenGL shading language (GLSL or glslang) is the newest shader language. It was originally introduced as an extension to OpenGL 1.5, but formally introduced in OpenGL 2.0 as a collaboration by 3DLabs and the OpenGL Architecture Review Board (ARB). It requires OpenGL 2.0 and has an extensive feature set. GLSL relies on the graphics driver to compile the program into GPU assembly code, which allows the vendors to create optimized code for their graphics cards. Most well-known vendors are providing the possibility to use GLSL, but some commands or constants may be vendor specific. We chose to use GLSL since it is a new and promising shader language with a well-known and easy-to-understand syntax.

2.4 Summary

A basic understanding of the motion of water and it's optic qualities is necessary to comprehend articles about water simulation.

Navier-Stokes equations are the basis of physical based research on liquids. Because of the complexity of the complete equations a number of assumptions can be made to simplify the equations to a spesific application.

The appearance of water consist of several effects like reflection, refraction, caustics and particles of e.g. sand. We have concentrated on reflection and refraction and found a simplification to the Fresnel equations by Schlick [3].

Programmable GPUs have a high number of pipelines and are therefore ideal for parallel execution. By utilizing the GPU the CPU can be used by other waiting processes. A shader language is used when programming the GPU.

Chapter 3

Related Research

Research on water motion can be divided into two main directions: grid based and particle based as seen in figure 3.1.

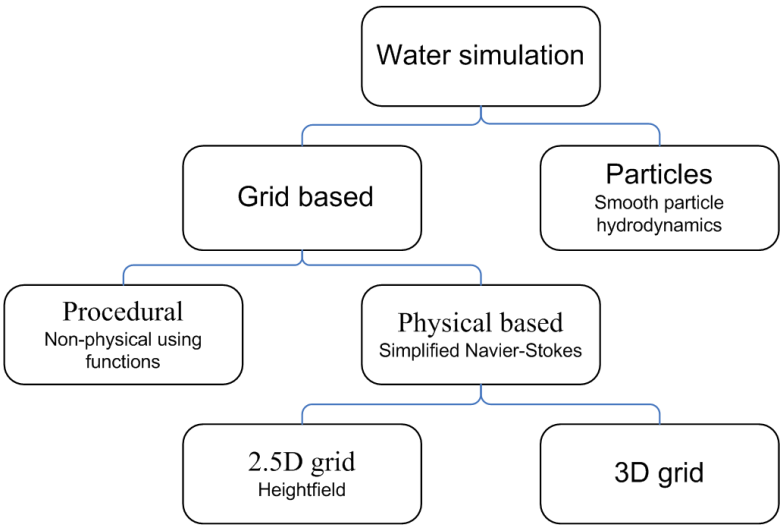

Figure 3.1: Water modelling techniques

3.1 Introduction

Grid based methods divide the simulation space into small cells, and computations are done on this finite number of cells. Procedural techniques apply are non-physical functions to grids. E.g. Sine waves and Gerstner waves applied to heightfields. Procedural techniques may give adequate effects where physical correctness is not important.

Physical based techniques are based on Computational fluid dynamics (CFD). The CFD community has been researching the Navier-Stokes equations for uses in engineering applications for several decades. Aerospace and naval research and development are among the areas that has benefited from this research. This grid based approach to

simulating water has also been used by the special effects industry for a long time. It gives realistic and often impressive effects, but is far to complex for real-time simulations.

Particle based methods model the volume of the liquid using a set of particles. This approach is called the Lagrangian approach. The particles conserve volume, and only pressure and viscosity has to be computed. The method is used to simulate liquids and melting objects at interactive rates.

3.2 Grid Based Techniques

Grid based methods can be divided into procedural techniques and physical based techniques, and they may be computed on both two-dimensional and three-dimensional grids. Two dimensional grids are still too complex for real-time rendering, but a cross between the two dimensional grid and the three dimensional grid can give realistic results for a water surface. These representations are called height fields, or 2.5D grid.

3.2.1 Procedural Techniques

One of the first applications of procedural waves in computer graphics was by Fournier and Reeves [6] who implemented Gerstner waves in 1986. In 2004 Finch implemented Gerstner waves on the GPU in GPU Gems [7]. An improvement of these simple methods uses Fast Fourier transformations (FFT). Tessendorf [8] showed that this method can run in real-time for large bodies of water, and give a result that looks more realistic than Gerstner waves (figure 3.2). Mitchell [9] implemented a FFT-based water simulation on GPU in 2005.
We are interested in physical based waves and will therefore not investigate procedural techniques further.

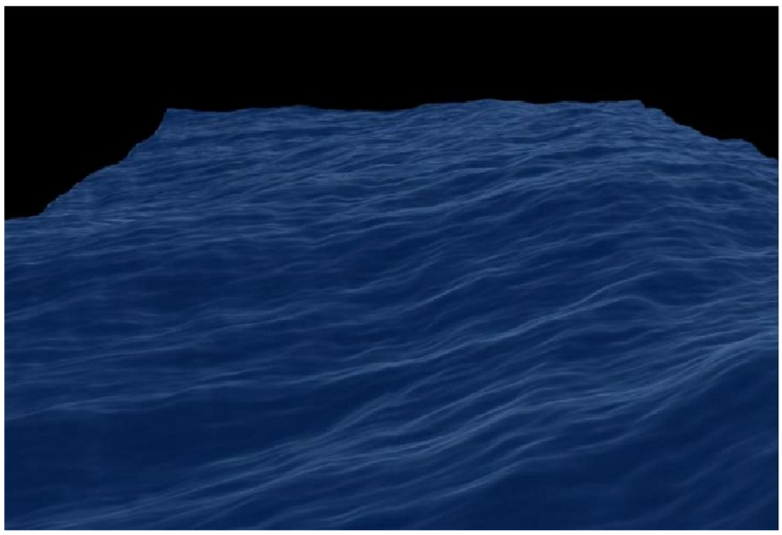

Figure 3.2: From [8]: Fourier based waves

3.2.2 Physical Techniques

CFD methods are both very time-consuming and hard to grasp without a good understanding of the equations and principles. It is therefore often necessary to restrict the simulations to get more effective computations. This especially applies to real-time water simulations. The normal approach is to discretize a continuous fluid onto an Eulerian two-dimensional grid or three-dimensional voxel framework for two-dimensional and three-dimensional graphics systems, respectably. Then a suitable algorithm is applied to solve the equations of motion.

In 1990 Kass and Miller [10] used a linearized form of the shallow water equations which are derived from the Navier-Stokes equations, without the Coriolis term. They used the equations in two dimensions with a heightfield representing the fluid surface and assume that water speed vary slowly in space. The equations are solved using the ADI (alternating-direction implicit) method, which make the solution stable. This approach restricted the possibility to simulate the true three-dimensional properties like wave breaking, but it enabled the use of reflection and refraction. They solved the following equations in real-time:

$$\frac{\partial u}{\partial t} + g\frac{\partial h}{\partial x} = 0 \tag{3.1}$$

$$\frac{\partial v}{\partial t} + g\frac{\partial h}{\partial y} = 0 \tag{3.2}$$

$$\frac{\partial h}{\partial t} + d\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) = 0 \tag{3.3}$$

Noe and Trier [11] implemented Kass on the GPU in 2004.

In 1996 Foster and Metaxas [12] solved the Navier-Stokes equations in three dimensions on a low resolution grid of voxels as in figure 3.3. They then calculated a detailed heightfield representing the fluid surface. The method gives highly realistic results, with effects like swirling motion and flows past objects. But since it's using the Navier-Stokes equations in three dimensions it takes $O\left(N^3\right)$ to run. The model is unstable for large time steps since it solves the equations explicitly.

In 1999 Stam [13] replaced Foster and Metaxas finite difference with an implicit semi-Lagrangian method to efficiently solve the Navier-Stokes equations on grid. He proposed methods for both three dimensional grids and heightfields. The implicit semi-Lagrangian approach is unconditionally stable regardless of the size of the time-steps. The method is not accurate enough for engineering applications, but was an important contribution to visualizing fluids. Stam did however not address the problem of simulating with free boundaries (such as water) and with obstacles.
The method was implemented on GPU in GPU Gems [14].

In 2001 Foster and Fedkiw [15] introduced a hybrid liquid volume model by solving the three-dimensional Navier-Stokes equations using an adaptation of the semi-Lagrangian method introduced by Stam [13]. The method can animate viscous liquids from water to thick mud within a fixed three-dimensional grid.

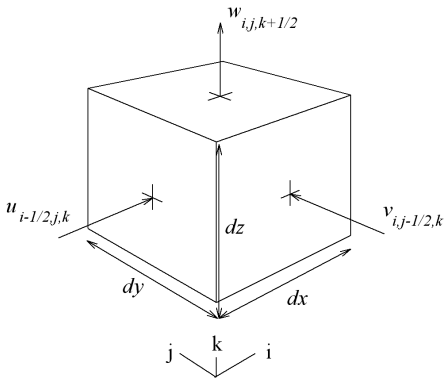

Figure 3.3: From [12]: Location of velocity components in a typical cell

In 2002 Enright, Marschner and Fedkiw [16] extended the work of Foster and Fedkiw [15] by focusing on the surface as opposed to the liquid volume. The surface modelling creates a more realistic surface. They also solve their system using a semi-Lagrangian method on a fixed three-dimensional grid.

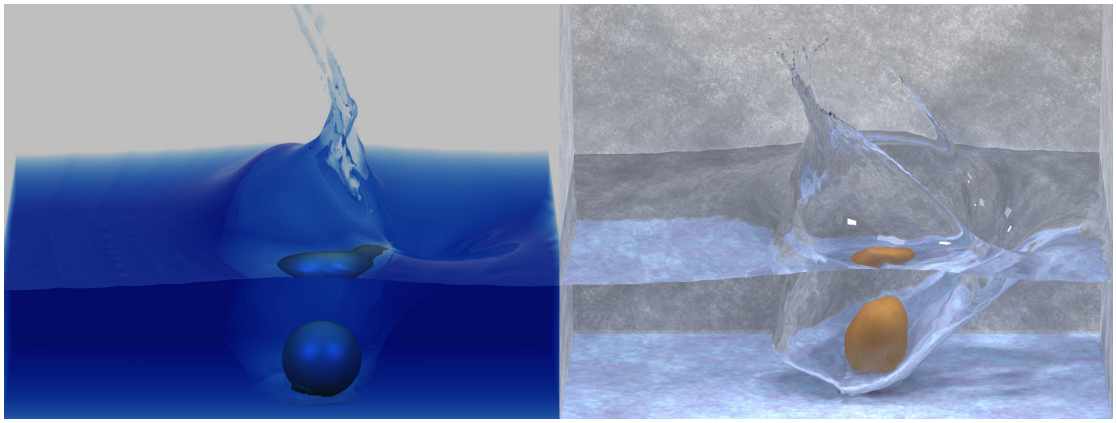

Figure 3.4: Left: Foster and Fedkiw [15]. Right: Enright et al. [16]

In 2002 Layton & van de Panne [17] also used an implicit semi-Lagrangian method to solve the Navies-Stokes equations, but in contrast to Stam, they addressed the problem of interaction with boundaries. This improvement gives the simulation a much more realistic appearance. This approach will be described in more detail in chapter 4.

Foster and Fedkiw [15] and Enright et al. [16] have one-way solid to fluid coupling where e.g. the motion of the ball in figure 3.4 is predetermined and the fluid motion is a secondary effect in response to the ball. The fluid has no effect of the motion of the ball. Therefore Carlson et al. [18] in 2004 presented the Rigid Fluid method which is a technique to animate the interaction of rigid bodies and viscous incompressible fluid with free surfaces. They solved this with a Lagrangian method by treating rigid objects

as if they were made of liquids. The velocity inside the rigid body is constrained to the motion of the body. Solid objects with different densities can be put into the same scene, and very realistic result can be rendered (see figure 3.5).

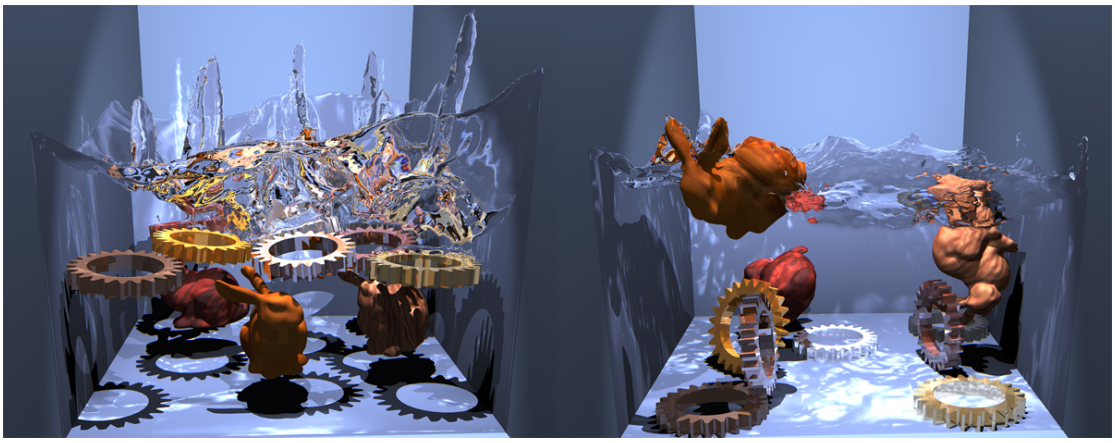

Figure 3.5: From [18]: Metal gears dropped into a body of water with wooden bunnies

3.3 Particle systems

Particle systems are an important technique in computer graphics, and was formally introduced by Reeves [19] in 1983. In some particle system the particles move according to Newton's laws, but independent of each other. These are called uncoupled particle systems and do not simulate fluids realistically because internal forces are ignored.

In a coupled system the particles can interact with each other. This makes it possible to simulate the particle system more realistically. A common method for simulating the particles is to repel each other if they are very close, attract each other at medium distance, and let the attracting forces approach zero as the distance increases.

Smoothed particle hydrodynamics (SPH) was introduced in 1977 by Lucy [20]. It is a Langrangian method which moves the particles according to hydrodynamic and gravitational forces. SPH was originally developed for simulating compressible fluids, but it has been modified to simulate incompressible fluids like water. Large time-steps which is needed for many particles makes SPH unstable.

In 2003 Müller et al. [21] showed that SPH and Navier-Stokes may be used in interactive applications to simulate water with free surfaces for up to 5000 particles.

In 2003 Premoze et al. [22] introduced the Moving-Particle Semi Implicit (MPS) method. The method is closely related to SPH but it allows the simulations of incompressible fluids, as opposed to standard SPH.

Müller et al. [21] and Premoze [22] assume the water to interact with solid objects as drinking glasses or walls, and therefore do not treat the boundary conditions explicitly. In 2004 Müller et al. [23] continued the work of Müller et al. [21] and proposed a method for fluid-solid interaction. The solids were represented by polygonal meshes with virtual boundary particles placed on the surface.

In 2005 Müller et al. [24] based their work on Müller et al. [21] and showed that SPH may be used to model fluids of different buoyancy as in figure 3.6. The result was realistic looking, but the number of particles is restricted because of the heavy computations.

While particle models are useful when rendering free flowing models they are not very practical for pure water surfaces. A surface needs to be created around the boundary of the particle system. This surface often look unnatural dues to the methods that need to be used for real-time rendering. A popular method is the Marching cubes algorithm.

Figure 3.6: From Müller et al. [24]: Particles pouring into a glass. Air particles are inserted and have different buoyancy than the water, and therefore rice to the top

3.4 Discussion

One of our goals is to render a physical based water surface in real-time. It is therefore natural to choose a two dimensional simplification, but we also investigated three dimensional techniques to explore the possibilities of implementing older techniques on todays hardware. We also looked at particle models.

After reading the articles which describe three dimensional methods we quickly realised that these methods are still too computational heavy. And since we were looking for a physical model of a single water surface, a particle model would probably not give a good result. Kass and Miller [10] and Stam [13] have been implemented on GPU, but we could not find a GPU implementation of Layton and van de Panne [17]. They claim their method is better than Stams method, and it therefore seemed like an interesting method to implement.

[1]Stam proposed both a 2.5D and 3D solution. The 2.5D method has been implemented in real-time and on a GPU.

	Procedural	Physical 2.5D	Physical 3D	Particles	Real-time	GPU
Fournier [6]	✓				✓	✓
Tessendorf [8]	✓				✓	✓
Kass and Miller [10]		✓			✓	✓
Foster and Metaxas [12]			✓			
Stam [13]		✓	✓[1]		✓	✓
Foster and Fedkwi [15]			✓			
Enright et al. [16]			✓			
Layton and van de Panne [17]		✓			✓	
Carlson [18]			✓			
Müller et al. 2003 [21]				✓	✓	
Premoze [22]				✓		
Müller et al. 2004 [23]				✓	✓	
Müller et al. 2005 [24]				✓	✓	

Table 3.1: Comparison of the different articles we have studied

3.5 Summary

There has been much research on water simulation, and we have tried to mention the most important and innovative papers. The different approaches to water simulation arise from different requirements. For water surfaces with limited computing power a procedural approach will be a natural choice. Particle models are based on the need for modelling free flowing water. The number of particles are still limited by computing power. Grid based physical techniques can be divided into two different approaches. The three dimensional approach is heavy to compute since it solves Navier-Stokes equations in three dimensions, while two dimensional simplifications are better for real-time applications.

Chapter 4

Conceptual Design

We have chosen to adapt an article written by Layton & van de Panne [17] to a GPU implementation. To achieve this we need to adjust and simplify several stages. We also need to create realistic reflections and refractions.

4.1 Selected Method

Layton & van de Panne [17] use an implicit semi-Lagrangian method to integrate the equations in time. The semi-Lagrangian method tries to combine the regularity of the Eulerian method, and the stability from the Lagrangian method. In the Eulerian method the observer stays fixed at a point and observes the world around him, like a fixed weather station, and require small time steps to maintain stability. In the Lagrangian method the observer travel with the world, like a weather balloon, and allow larger time steps.

They base their semi-Lagrangian method on the shallow water equations (2.9), (2.10) and (2.11) derived in 1980 by Haltiner and Williams. If we look at the Lagrangian method we can imagine particles travelling along a trajectory and arriving at position x_i at time t_{n+1}. If we then use the semi-Lagrangian method, the derivatives are computed from the position at t_{n+1}, which is x_n, and at t_n, which is the departure point, α_i^n. Figure 4.1 illustrate the semi-Lagrangian integration method described.

Layton & van de Panne's implicit semi-Lagrangian wave simulation algorithm where h^t, u^t and v^t is the height and velocity in x and z-direction at time t.

1. Initialize h^0, u^0 and v^0.
2. For each time step t_{n+1} do:
   - i Compute departure points using equation (4.1).
   - ii Compute h, u and v at departure points using equations (4.2).
   - iii Solve equation (4.3) for h^{n+1} using Conjugate Gradient method.
   - iv Update u^{n+1} and v^{n+1} with equations (4.4) and (4.5).

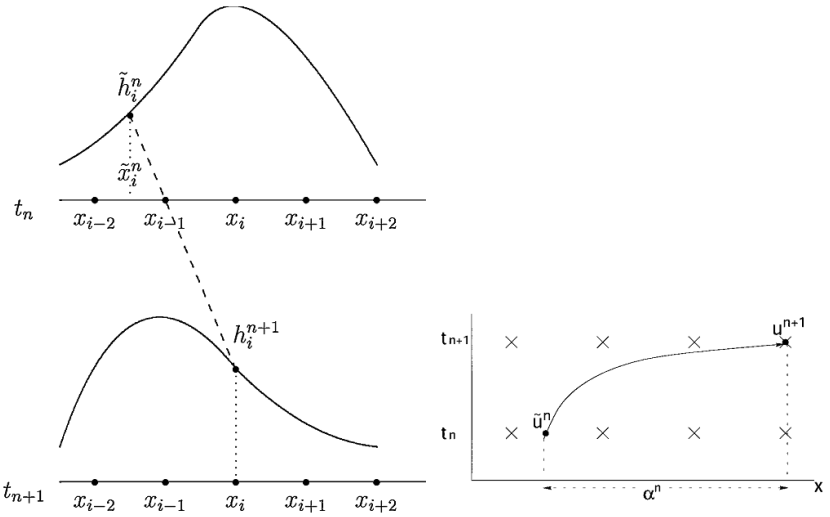

Figure 4.1: From [17]: Illustration of the semi-Lagrangian time integration method

$$\alpha_i^n = \Delta t u^n (x_i) \tag{4.1}$$

$$\tilde{u}^n (x_{i,j}) \equiv u^n (x_i - \alpha_i^n) \tag{4.2}$$

Equations (4.1) and (4.2) are formulas for one dimension. The formulas for two dimensions are similar and it is therefore unnecessary to write out these. A similar equation to (4.2) can be applied to the height field to obtain its values at the departure points.

$$\begin{aligned}
&h_{i,j}^{n+1} + \Delta t^2 g \left(\frac{b_{i+1,j} - b_{i-1,j}}{2\Delta x} \frac{h_{i+1,j}^{n+1} - h_{i-1,j}^{n+1}}{2\Delta x} + \frac{b_{i,j+1} - b_{i,j-1}}{2\Delta z} \frac{h_{i,j+1}^{n+1} - h_{i,j-1}^{n+1}}{2\Delta z} \right) \\
&\quad -\Delta t^2 g d_{i,j}^n \left(\frac{h_{i-1,j}^{n+1} - 2h_{i,j}^{n+1} + h_{i+1,j}^{n+1}}{\Delta x^2} + \frac{h_{i,j-1}^{n+1} - 2h_{i,j}^{n+1} + h_{i,j+1}^{n+1}}{\Delta z^2} \right) \\
&= \tilde{h}_{i,j}^n + \Delta t \left(\tilde{u}_{i,j}^n \frac{b_{i+1,j} - b_{i-1,j}}{2\Delta x} + \tilde{v}_{i,j}^n \frac{b_{i,j+1} - b_{i,j-1}}{2\Delta z} \right) \\
&\quad -\Delta t d_{i,j}^n \left(\frac{\tilde{u}_{i+1,j}^n - \tilde{u}_{i-1,j}^n}{2\Delta x} + \frac{\tilde{v}_{i,j+1}^n - \tilde{v}_{i,j-1}^n}{2\Delta z} \right)
\end{aligned} \tag{4.3}$$

where h is the heightfield, b is the bottom height, u and v are the velocity of the particles in the x-, and z-direction respectably, and $d_{i,j}$ is the depth at i, j.

$$\frac{u^{n+1} - \tilde{u}^n}{\Delta t} + g \frac{\partial h^{n+1}}{\partial x} = 0 \tag{4.4}$$

$$\frac{v^{n+1} - \tilde{v}^n}{\Delta t} + g \frac{\partial h^{n+1}}{\partial z} = 0 \tag{4.5}$$

4.2 Preconditions and simplifications

One of our preconditions was to model a watersurface for the Sverresborg simulation. This means that the available GPU time is limited, and we need to simplify the calculations. By assuming a flat bottom and that $d_{i,j} = h_{i,j}$ we get the following simplifications to (4.3):

$$\begin{aligned} & h_{i,j}^{n+1} - \Delta t^2 g d_{i,j}^n \left(\frac{h_{i-1,j}^{n+1} - 2h_{i,j}^{n+1} + h_{i+1,j}^{n+1}}{\Delta x^2} + \frac{h_{i,j-1}^{n+1} - 2h_{i,j}^{n+1} + h_{i,j+1}^{n+1}}{\Delta z^2} \right) \\ & = \tilde{h}_{i,j}^n - \Delta t d_{i,j}^n \left(\frac{\tilde{u}_{i+1,j}^n - \tilde{u}_{i-1,j}^n}{2\Delta x} + \frac{\tilde{v}_{i,j+1}^n - \tilde{v}_{i,j-1}^n}{2\Delta z} \right) \end{aligned} \tag{4.6}$$

The well in the Sverresborg simulation is quadratic, and we can therefore also assume that $\Delta x = \Delta z$. We then get the following simplification to (4.6):

$$\begin{aligned} & \left(\frac{\Delta x^2}{\Delta t^2 g h_{i,j}^n} + 4 \right) h_{i,j}^{n+1} - h_{i-1,j}^{n+1} - h_{i+1,j}^{n+1} - h_{i,j-1}^{n+1} - h_{i,j+1}^{n+1} \\ & = \frac{\Delta x^2}{\Delta t^2 g h_{i,j}^n} \tilde{h}_{i,j}^n - \frac{\Delta x}{2\Delta t g} \left(\tilde{u}_{i+1,j}^n - \tilde{u}_{i-1,j}^n + \tilde{v}_{i,j+1}^n - \tilde{v}_{i,j-1}^n \right) \end{aligned} \tag{4.7}$$

To solve (4.7) we need to transfer it to a matrix system, which gets the following structure:

$$\begin{bmatrix} a_1 & -1 & & -1 & & & & & \\ -1 & a_2 & -1 & & \ddots & & & & \\ & -1 & \ddots & \ddots & & \ddots & & & \\ -1 & & \ddots & \ddots & \ddots & & \ddots & & \\ & \ddots & & \ddots & \ddots & \ddots & & \ddots & \\ & & \ddots & & \ddots & \ddots & \ddots & & -1 \\ & & & \ddots & & \ddots & \ddots & \ddots & \\ & & & & \ddots & & \ddots & a_{n^2-1} & -1 \\ & & & & & -1 & & -1 & a_{n^2} \end{bmatrix} \begin{bmatrix} h_{1,1} \\ h_{1,2} \\ \\ \vdots \\ \\ \\ \vdots \\ \\ h_{n,n-1} \\ h_{n,n} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \\ \vdots \\ \\ \\ \vdots \\ \\ b_{n^2-1} \\ b_{n^2} \end{bmatrix} \tag{4.8}$$

From (4.8) we see that all non-diagonal elements in the matrix are either empty or -1. We also know the positions of the -1-elements. This means that we can describe the matrix with a single vector along the diagonal, and have less data to send to the GPU.

Layton and van de Panne's original method use the Conjugate Gradient method. A simpler method to implement on the GPU is the Jacobi solver. By using Jacobi we can also exploit our simplifications better.

4.3 Boundary conditions

The boundary conditions are important for this method. The authors of Physics-based animation [25] experienced that setting the derivative of h across the boundary to be zero, and setting u and v to be zero at the boundary, giva useful results. The boundary conditions on h is of the type von Neumann, while the boundary conditions on u and v is of the type Dirichlet.

We adapted the Von Neumann boundary conditions algorithm from Physics-based animation [25] to our solution. $h(i, j)$ denote the grid node at positions (i, j). The $N \times N$ computational grid is represented on a $(N + 2) \times (N + 2)$ grid and we end up with the following algorithm:

Algorithm 1 Applying von Neumanns border conditions

```
1:  procedure APPLY-NEUMANN(h, N)
2:      for i ← 1, N do
3:          h(0, i) = 2 * h(1, i) − h(2, i)
4:          h(N + 1, i) = 2 * h(N, i) − h(N − 1, i)
5:          h(i, 0) = 2 * h(i, 1) − h(i, 2)
6:          h(i, N + 1) = 2 * h(i, N) − h(i, N − 1)
7:      end for
8:      h(0, 0) = 2 * h(1, 1) − h(2, 2)
9:      h(0, N + 1) = 2 * h(1, N) − h(2, N − 1)
10:     h(N + 1, 0) = 2 * h(N, 1) − h(N − 1, 2)
11:     h(N + 1, N + 1) = 2 * h(N, N) − h(N − 1, N − 1)
12: end procedure
```

Dirichlet boundary conditions simply means setting values in the computational grid, before using it to perform any computations.

4.4 Optics

To make the water surface look realistic we need to add reflections and refractions. Static environment maps is the classic method of choice when one need to quickly render reflections. By surrounding the scene with a sphere map, or cube map, reflections may come from any direction. It requires only a sinle pass and can handle steep waves. A big drawback with static environment maps is that they can't reflect local objects without severe visual effects. The well surrounding the water will be the most noticable object missing in the reflections and refractions. We considered two different solutions to the problem with the local reflections and refractions; projective texturing and dynamic environment maps.

Projective texturing is a good method to use when the water surface is nearly flat. It is not physically accurate, but has a convincing look without the severe visual effects that occur with local reflections when using static environment maps. By projecting

the scene onto the surface the result is good enough for most applications. Vlachos et al. [26] proposed a GPU implementation of projective texturing. We did not implement projective texturing because our water surface has the possibility to generate steep waves. It would then reflect objects that's not necassarily available with projective texturing.

We chose to implement a dynamic environment map. This works by rendering a new environment map from the center of the reflective object, without the rendering the object. It is then possible to reflect local objects like our well around the water. A problem with this approach is that enviroment maps are assumed to be infinitely far away. To fix this problem we adjusted a solution by Brennan [27] which adjust for object distance. We find the intersection between the reflecting and refrating ray, and then use the vector, from the center of the environment map to the intersecting point, to index the environment map.

The solution is specific to our water surface and only fixes the reflections of the well surrounding the water surface. Distortions in the reflection from other objects will still occur.

4.5 Summary

We have achieved to simplify Layton & van de Panne's article and made it more adaptable for GPU implementation. The simplifications are simple and effective, but also restrictive. The restictions suit our water surface since it is a well with a flat bottom. The reflections and refractions are also specific to our simulation since they are dependent on the well to have straight sides and to be quadratic.

Chapter 5

Implementation

5.1 Overview

The Well is the name of the implementation of the modified Layton & van de Panne water surface method. The figure (5.1) outlines the main steps of the implementation of *The Well*. The implementation starts by initializing the engine, specifying window size, and which drawable objects will be part of the scene, including the water surface. When the initialization phase is done, the main loop is started. The main loop first updates all drawable objects in the scene. This could include rotating and moving an object, but the current implementation mainly calculates the water surface in the update phase.

If there are any updates (apart from the water surface) that change the scene, the reflection cube map is updated. The rendering of the first frame is considered such a change, thus the cube map is created during the first frame before it is needed. Also moving and rotating an object is considered a change that requires an update of the cube map. The cube map is updated by rendering all drawable objects except the water surface. The water surface is the reflective surface and cannot give a reflection upon itself using the reflection cube map method. When the drawable objects have been rendered, the scene is saved into the reflection cube map texture set.

Finally all the drawables, including the water surface, is rendered and displayed on the screen. After this, events are signalled to the application. This can be a resize of the window or the user pressing a key on the keyboard. Any events are handled and the cycle starts again with updating all the drawables.

5.2 Program framework

The Well consist of a few main modules; the engine, the water surface and all the other drawables. The engine is the module responsible for controlling what happens in the program. The viewport, camera positioning and transformation matrices are handled here, events are gathered and dispatched. The engine also keeps a list of all the drawables in the scene, and initiates updates and renderings in all the drawables.

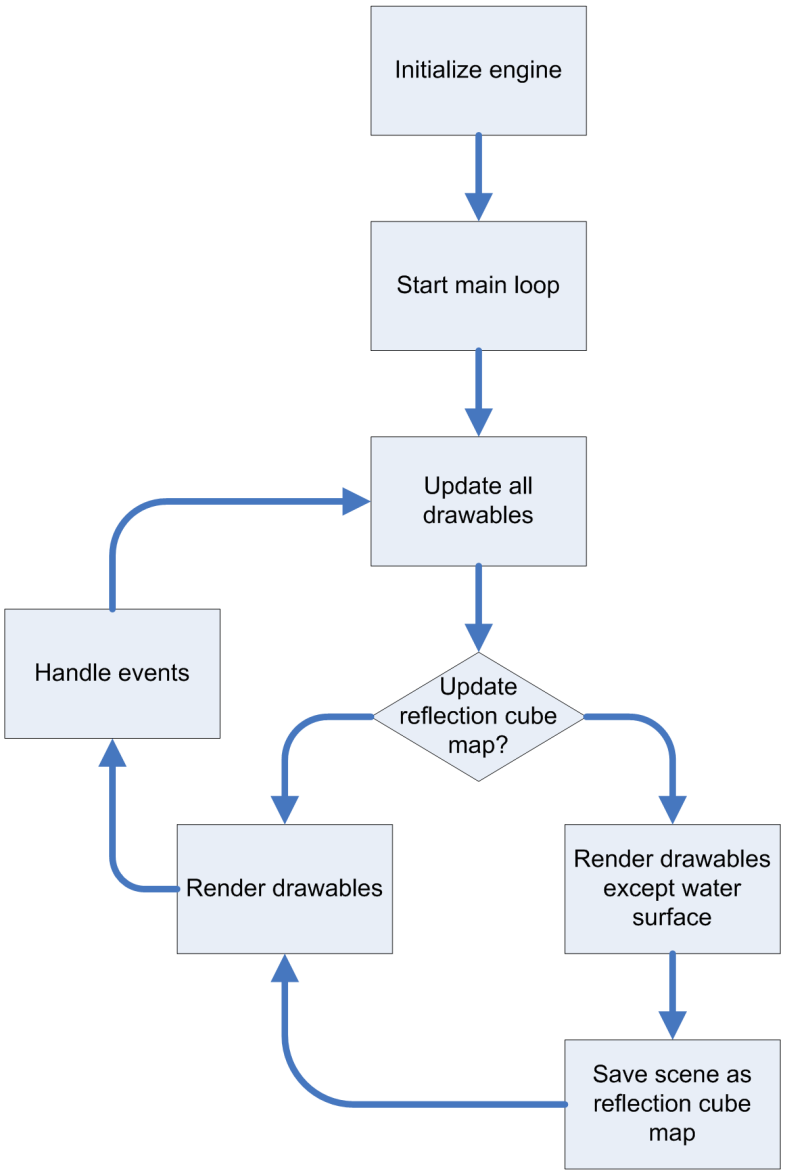

Figure 5.1: Overview of the main program loop

The water surface module receives update and render notifications from the engine. The update method of the water surface computes the new water surface and stores it. There are several solvers implemented for computing the surface, but only one is used in the lifetime of the program. The various solvers are used to evaluate the performance of computing such a surface on the GPU vs. the CPU, and also comparing the speed and surface quality of the various solvers.

The render method of the water surface draws the surface. However a shader is loaded before the actual drawing to handle reflection mapping. The reflection mapping is done per vertex and the resulting colour values are interpolated across the fragments.

All the other drawables also receives update and render notifications. To keep things simple, nothing happens in the update methods of the objects, they're static. The render method draws the object.

5.3 Class structure

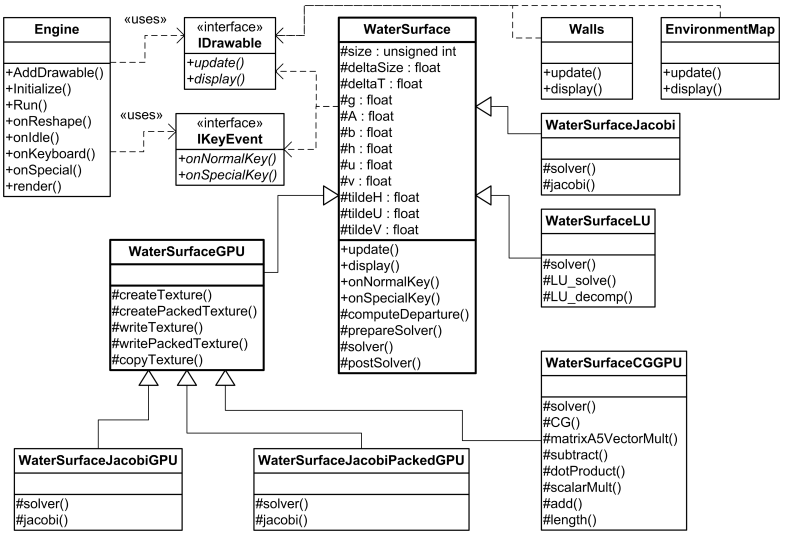

Figure 5.2: UML Class diagram

The UML class structure figure (5.2) describes how the class structure is built.

5.3.1 Engine class

The engine class is responsible for setting up the graphical environment, controls the objects in the scene, the flow of the program and gathers events. The engine is started like this:

1. **Initialize(...)** - Must be run first to setup the graphical environment, before any drawable objects are created. The drawable objects might call graphical functions which rely on the graphical environment to have been initialized.

2. **Add(...)** - Drawables can be added after initialization. The drawables are stored in the drawlist of the engine. They are also checked to see if they implement the IKeyEvent and IStatistics interfaces, and stored in the keylist and statlist respectively if they are.

3. **Run()** - Starts the main loop, and the main loop calls the various on[...]-methods. These methods are the event gatherers, and controls the flow of the program.

The event gatherers are:

- **onIdle()** - Called whenever there are no events in the event queue. onIdle triggers the next update/render cycle by posting a redisplay event.
- **render()** - The display event is triggered when a redisplay event is posted or the window needs to refresh itself. The render method initiates updates and renders the scene. It also decides if the reflection cube map needs to be updated and calls the updateCubeMap()-method if necessary.
- **onReshape(...)** - The reshape event is triggered when the user resizes the program window. The event handler changes the viewport to provide the same aspect ratio and line-of-sight as before the reshape.
- **onKeyboard(...)** - The keyboard event is triggered when the user press a key. The key press is read and possibly handled in the engine (for instance going into frame-by-frame mode or pausing the program). If the key pressed is not recognized by the engine, all the drawables which implements IKeyEvent receives the key event and has a chance to process it (for instance changing from flat to wireframe in the water surface).
- **onSpecial(...)** - The special event is triggered when the user press a special key like the arrow keys or function keys. It's the same mechanism as with the key event.

5.3.2 IDrawable interface

The IDrawable interface specifies the methods a drawable object must implement. The methods are:

- **update(int ticks)** - The update method receives the number of ticks elapsed since the previous update **started**. Using this value, the drawable objects are able to update their state based on elapsed time rather than the number of frames processed.
- **display()** - The display method instructs the drawable to render itself.
- **isReflecter()** - The method should return true if the object can be reflected in the water surface. The water surface must return false.

5.3.3 IKeyEvent interface

The IKeyEvent interface specifies the two methods an object which receives key presses must implement. The methods are onNormalKey(...) and onSpecialKey(...). They are described above in section 5.3.1. Only the keys not trapped by the engine are passed down.

5.3.4 Drawable classes

The drawable classes apart from the water surface are the Walls class and the EnvironmentMap class. The Walls class draws the walls of the well, complete with a brick

texture. The EnvironmentMap class draws the textured environment cube map which gives a nice reflection in the water surface.

5.3.5 WaterSurface classes

The water surface classes are also drawable objects, but only one should be active at any given time.

WaterSurface class

The WaterSurface class is the base class of all the water surface classes. This class implements the update and render methods. The update method has the following pseudocode:

Algorithm 2 The WaterSurface update method

```
1: procedure UPDATE-WATERSURFACE(intticks)
2:     dT ← ticks/1000.0
3:     computeDeparture()
4:     prepareSolver()
5:     solver()
6:     postSolver()
7: end procedure
```

dT means delta time and is the number of seconds since the last update. The methods called are all virtual methods, and the solver method is a virtual abstract method. This means the solver method has to be defined in a subclass. The other methods thus have a CPU implementation which can be overridden in subclasses.

The render method draws the water surface. Also a shader, ReflectionRefractionShader, is loaded while drawing the water surface. The shader computes the reflection and refraction on the water surface per vertex and determines a colour. This colour is then interpolated across the fragments.

WaterSurfaceJacobi class

The WaterSurfaceJacobi class defines the solver method and is the CPU implementation of the Jacobi method. Our implementation of the Jacobi method can be described with the following pseudocode:

The Jacobi method is actually a bit more complicated, but because of the simplifications we've made (4.2) we have four columns in the A matrix with the value -1 and only the diagonal contains varying data.

WaterSurfaceLU class

The WaterSurfaceLU class implements the LU decomposition method. It is very slow, but it was used to verify the correctness of the various methods that were implemented. The class was not used for performance evaluations.

Algorithm 3 The Jacobi CPU solver

```
1: procedure JACOBI-CPU(imax, tolerance)
2:     repeat
3:         for i ← 1, n do
4:             for j ← 1, n do
5:                 newH(i, j) ← (b(i, j) + 4.0 ∗ H(i, j))/A(i, j)
6:             end for
7:         end for
8:         difference ← distance(newH, H)
9:         H ← newH
10:        iterNum ← iterNum + 1
11:    until iterNum > imax or difference < tolerance
12: end procedure
```

WaterSurfaceGPU

The WaterSurfaceGPU class does not implement the solver method and is therefore also an abstract class like WaterSurface. It does however implement a few functions for handling textures used in the shaders defined by its subclasses:

- **createTexture(...)** - Creates a new texture of the same height and width as the water surface
- **createPackedTexture(...)** - Creates a new texture with half the width and half the height of the water surface. Textures made with this method are meant to receive data in all colour channels (RGBA). Also note that all textures in a single framebuffer object have to be the same size. Therefore textures created with createTexture and createPackedTextures cannot be attached to the same framebuffer object.
- **writeTexture(...)** - Writes data from up to four arrays to the texture. The arrays are written simultaneously, spread over all the colour channels.
- **writePackedTexture(...)** - Writes a single array to a texture, filling values into all four colour channels of the texture.
- **copyTexture(...)** - Copies data from one texture to another, using framebuffer objects.

The textures are generally not read directly. Instead, they are attached to a framebuffer object, and the framebuffer is read instead. This is a much faster method on most modern graphics cards.

WaterSurfaceJacobiGPU class

The WaterSurfaceJacobiGPU class implements the Jacobi method on the GPU. The computations are almost the same as the CPU version, but they're made on the GPU

instead. The following pseudocode outlines the necessary steps for performing the computations on the GPU:

Algorithm 4 The Jacobi GPU solver

```
1: procedure JACOBI-GPU(imax)
2:     jacobiShader.initialize()
3:     fbo.bind()
4:     writeTexture(texture1, A, h, b, NULL)
5:     fbo.attach(texture1)
6:     fbo.attach(texture2)
7:     jacobiShader.load()
8:     for i ← 0, imax do
9:         setup texture1 as read and texture2 as write
10:        execute shader code (draw a quad as big as the data texture)
11:        swap read and write textures
12:    end for
13:    jacobiShader.unload()
14:    fbo.read(texture, h)
15:    fbo.unbind()
16:    jacobiShader.reset()
17: end procedure
```

A major motivation when doing computations on the GPU is to avoid reading and writing textures more than absolutely necessary. Therefore two textures are created to implement the ping-pong technique. The idea is that in an iteration, one texture is used for reading and the other for writing. When the first iteration is complete, the textures are swapped, and the texture written to is the texture being read in the next iteration and vice versa. The swapping continues until no more iterations are needed. At the end, the texture last written to is read as the result of the computation.

WaterSurfaceJacobiPackedGPU class

The WaterSurfaceJacobiPackedGPU class implements a slightly different version of the solver from the one WaterSurfaceJacobiGPU uses. The motivation behind this class is an attempt to utilize the pipelines of the GPU better and making use of the GPUs ability to do four multiplications at the same time by multiplying on all four colour channels. Also, less data needs to be transferred to and from the GPU as there are no empty colour channels, and the textures are a quarter of the size used in WaterSurfaceJacobiGPU.

WaterSurfaceCGGPU class

The WaterSurfaceCGGPU class implements the Conjugate Gradient (CG) method on the GPU. This is the method recommended by Layton & van de Panne [17]. The method makes use of eight textures of the same size as the water surface, and requires a lot of video memory to run if the water surface is relatively large. In contrast, the Jacobi on the GPU method described above makes use of only two textures. The pseudocode is

defined in algorithm 5. The comments to the right in algorithm 5 are the mathematical

Algorithm 5 The Conjugate Gradient GPU solver

```
1:  procedure CG-GPU(imax)
2:      shader.initialize()
3:      fbo.bind()
4:      writeTexture(texA, A)
5:      writeTexture(texH, h)
6:      writeTexture(texB, b)
7:      matrixA5VectorMult(texH, texR0)                    ▷ r_i = b − (A ∗ h)
8:      subtract(texB, texR0, texRi)
9:      copyTexture(texRi, texPi)                          ▷ p_i = r_i
10:     for i ← 0, imax do
11:         matrixA5VectorMult(texPi, texAPi)              ▷ α_i = (r_i·p_i) / (p_i·(A∗p_i))
12:         dAPi ← dotProduct(texPi, texAPi)
13:         α_i ← dotProduct(texRi, texPi) / dAPi
14:         scalarMult(α_i, texPi, texR0)                  ▷ h = h + α_i p_i
15:         add(texH, texR0, texH)
16:         scalarMult(α_i, texAPi, texR0)                 ▷ r_i = r_i − α_i(A ∗ p_i)
17:         subtract(texRi, texR0, texRi)
18:         matrixA5VectorMult(texRi, texR0)               ▷ β_i = −(r_i·(A∗r_i)) / (p_i·(A∗p_i))
19:         β_i ← − dotProduct(texRi, texR0) / dAPi
20:         scalarMult(β_i, texPi, texR0)                  ▷ p_i = r_i + β_i p_i
21:         add(texRi, texR0, texPi);
22:     end for
23:     fbo.read(texH, h)
24:     fbo.unbind()
25:     Shader.reset()
26: end procedure
```

equivalents of the pseudocode to the left. Behind each of the function calls in the pseudocode, a shader performs the computation on the GPU.

5.4 Summary

We have implemented our proposed method on the CPU and GPU with several solvers. Using different solvers gives us the opportunity to compare and find a best possible solution. The implementation consist of an engine and a few drawable objects. The water surface is one of the drawable objects, and contains the logic for computing and displaying the water surface. The engine controls the drawables and handles events.

Chapter 6

Experiments & Discussion

6.1 Overview

Experiments were run to evaluate the performance of the implementation and the different solver methods. Time spent in the various steps of the water surface algorithm were recorded. For the solvers implemented on the GPU, time spent on writing to and reading from textures were recorded, as well as time spent waiting for the GPU shaders to finish.

The experiments were run on an Intel Pentium 4 HT 3.0GHz computer with 768MB RAM. The GPU used was a nVidia 7800GT with 256MB of video RAM on a PCI Express 16x bus. The program was built using Visual Studio 2005 Professional Edition. The speed of the implementation and the solvers were measured using two different water surface sizes, 128 x 128 and 256 x 256.

6.2 Visual quality

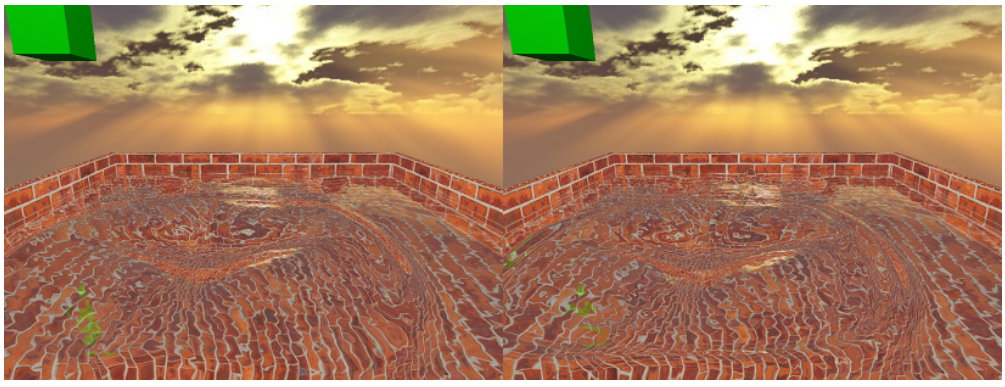

Figure 6.1: CG and Jacobi after 100 frames

The visual quality using the different solvers were the same for the Jacobi solvers. However the conjugate gradient (CG) method gave a slightly better representation,

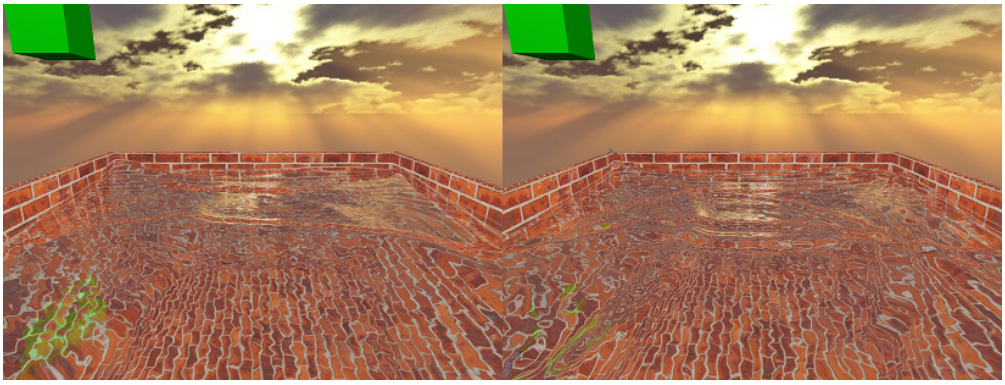

Figure 6.2: CG and Jacobi after 400 frames

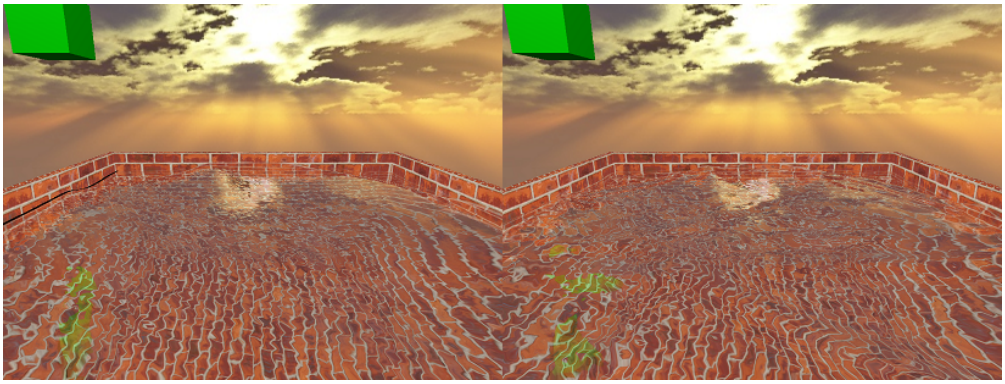

Figure 6.3: CG and Jacobi after 2000 frames

having less choppier waves. Figure 6.1 shows the rendered scene after 100 frames of the CG and the Jacobi solver, the CG version to the left. The Jacobi waves don't have as high amplitudes and there are slightly more of them than with the CG waves. Figure 6.2 is the rendered scene after 400 frames. It is now more prominent that the CG version has fewer and larger waves. In figure 6.3 the CG waves are now much broader and calmer than the Jacobi waves which are a bit small, choppy and numerous.

Both versions are stable and can run for a very long time without unwanted artifacts appearing. We believe the CG version to be the more physically correct version. However, the Jacobi version also gives a nice and acceptable result, albeit a bit choppier. Based on visual appearance alone, the CG version is the better, though it cannot compare to the Jacobi method on speed, which will be shown later.

6.3 Measurements

6.3.1 Framerate

Figure 6.4 shows the average number of frames per second per method. Frames per second isn't a very accurate measurment, but comparing the values gives a good overview

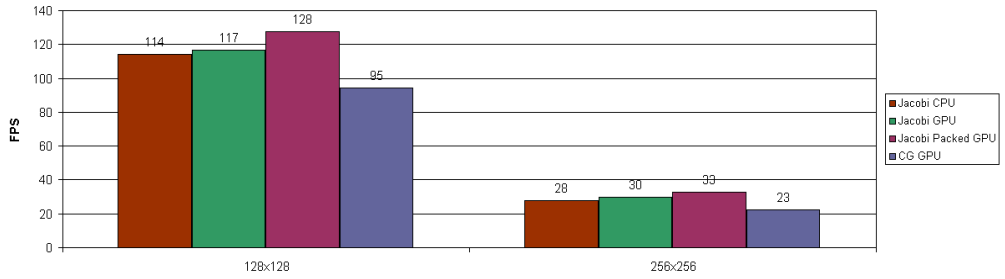

Figure 6.4: Average frames per second

of the general performance of the methods. As we see, the CPU version of the Jacobi method performs quite well, while the packed version of the Jacobi method on the GPU is the fastest one. The CG method is the slowest, although in 128x128 it is able to keep up a framerate good enough for a real-time representation. In 256x256 only the Jacobi methods on the GPU has high enough framerate (above 30 FPS) to really be real-time. One point worth mentioning is that the reflection and refraction shader is active in all the measurements, which will be apparent further on. Without this shader, the framerate would be higher and all the methods could be considered real-time.

6.4 Time measurements

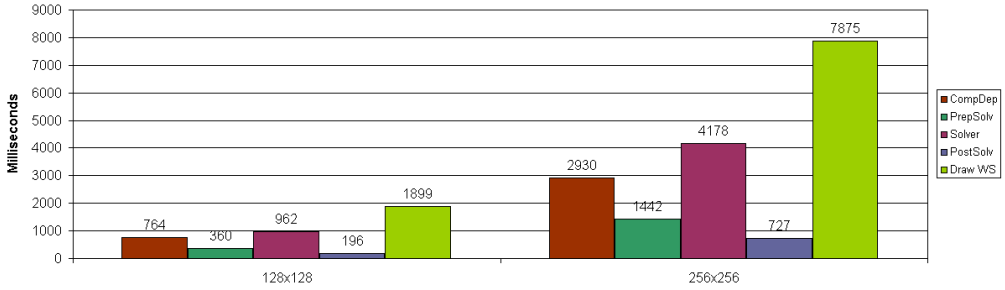

Figure 6.5: Overview of time elapsed

Figure 6.5 shows the time taken computing 500 frames in the various steps of the update algorithm (algorithm 2), as well as the display method which includes the reflection and refraction shader. The measurements are taken from the Jacobi CPU version, and apart from the time taken by the solver, the timings are very close to equal for all the versions. Therefore only the Jacobi on the CPU version is shown. Comparing the values, we see that the display method takes the most time, mostly due to the computation of the reflections and refractions, which are rather computationally intensive.

6.4.1 Solvers

Figure 6.6 shows time elapsed in the solver for the various versions. The CG version spends a lot more time computing the water surface than the other methods, more

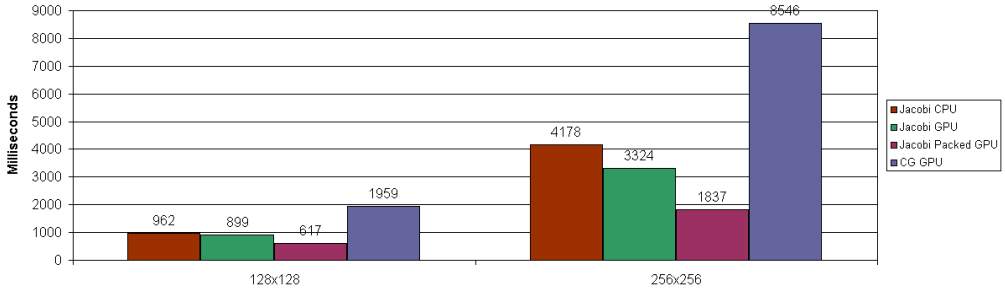

Figure 6.6: Time elapsed in solver

than twice as long as the jacobi method on the CPU. Comparing the jacobi versions, the GPU versions are slightly faster than the CPU version in 128x128 and the packed version more so, but when the size is increased to 256x256, the packed jacobi version performs a lot better than the other jacobi versions, almost twice as fast as the regular jacobi version on the GPU.

6.5 Conjugate Gradient vs. Jacobi

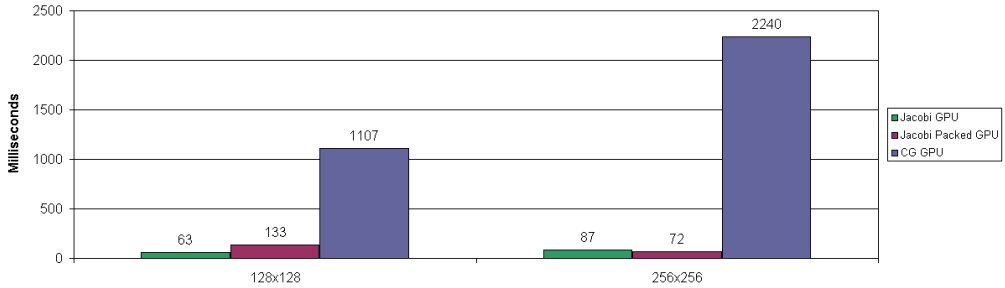

Figure 6.7: Time elapsed in shader

The conjugate gradient solver was the solver recommended by Layton & van de Panne ([17]), but it is outperformed by the Jacobi solver. However, we've made quite a few simplifications (4.2) of the method proposed by Layton & van de Panne due to our preconditions. Thus we've moved from a rather complex system of linear equations to a system where only the diagonal of the A matrix has variable values and the other four are always -1. This condition were exploited in our Jacobi solver and each iterative step now contains only one addition, multiplication and division for a single point in the height field. It was possible to implement similar simplifications in the CG solver, but with much less impact because of the solver's more complex nature.

Usually the CG solver is quite effective since it is able to reach an acceptable solution with very few iterations. In our measurements, the CG solver only has a single iteration (and skips computing the values needed for the next iteration, since there's only one iteration), and even this gives a more correct solution than the jacobi method. The jacobi solver iterates five times, more iterations doesn't seem to improve the visual

quality. However, even iterating five times, the jacobi solver is a lot faster because it has a lot less to do each iteration. This is clearly seen in figure 6.7, where the CG version spends a lot more time running shader code compared to the Jacobi version.

6.6 Jacobi on the CPU and the GPU

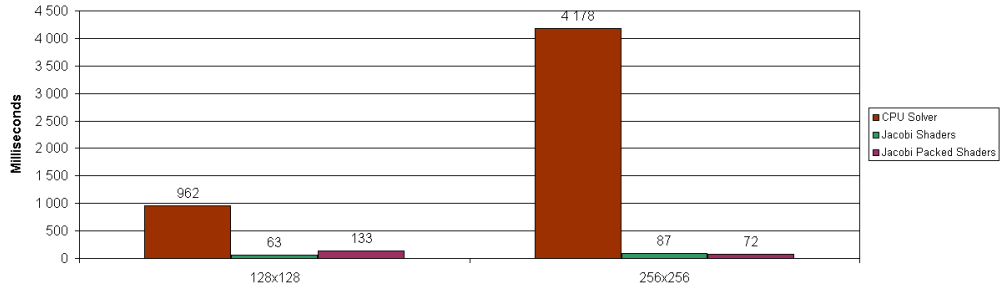

Figure 6.8: Time elapsed computing

The GPU versions of the Jacobi solver are only slightly faster than the CPU version. On the other hand, if we consider only the time spent actually doing the computation, we see that the GPU version is extremely fast in comparison (figure 6.8).

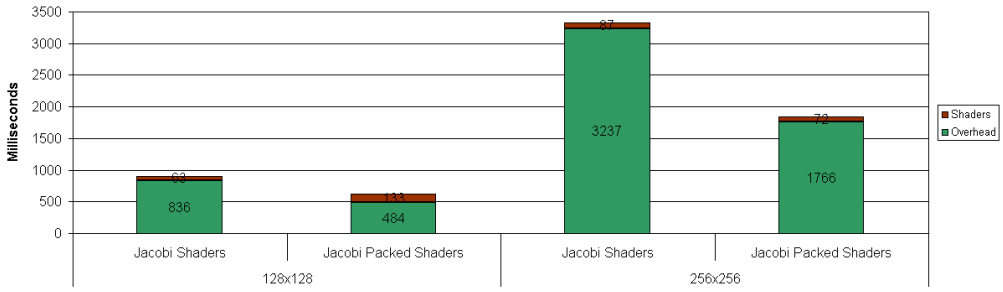

Figure 6.9: Overhead and shader comparison

So why isn't the GPU version that much faster? The answer lies in the overhead of writing and reading textures stored on the GPU's video memory. Transferring data between the CPU and the GPU is very slow and should be avoided as much as possible. This is also the reason why the packed version has a lot less overhead than the regular GPU version because it operates with textures a quarter of the size which the regular GPU version needs. See figure 6.9.

This also means that if we were to implement a more complex algorithm which took more clock cycles on the CPU and was transferrable to the GPU as shader code, the impact of the overhead would become less. We would see a larger time difference between the CPU and the GPU versions, making the GPU version more efficient.

6.7 Conclusion

Water rendering has become more interesting during the last years due to new generations of programmable graphics hardware. The increased computational power has made it possible to render realistic water motion in real-time, instead of just a flat surface.

The method described in this paper has been derived from future work by Layton and van de Panne, which is stable and physical correct. We successfully implemented the method on the GPU and gained a speedup. The CPU time released may be used for other computational purposes. GPU implementations of physical based water is therefore more efficient than CPU implementations. It is also possible to increase the speedup by transferring more of the computations to the GPU.

6.8 Future Work

- Our implementation doesn't run entirely on the GPU. A natural extension of our project is therefore to make this work. By having a shader for each step of the computations, the need to transfer data between the CPU and the GPU is minimized.
- Layton and van de Panne's method incorporates the possibility to add obstacles in the well by i.a. setting the velocities at the location of the obstacles. This is easy to accomplish in a CPU implementation, but a little more tricky on the GPU. One possibility to solve the problem is to add a stencil map which defines which areas that includes obstacles.
- Improve the water surface to support rectangular and even ellipsoidal surfaces.
- One of the restrictions of the implementation is a flat bottom. The water surface model could be improved to include support for a sloped bottom, for instance. This will increase the load on the GPU, as well as the amount of data that needs to be transferred, but may not be an issue in the near future with improved hardware technology.

Bibliography

[1] Professor M.S. Cramer of Virginia Tech. Foundation of fluid mechanics. Available from: `http://www.navier-stokes.net/`.

[2] Eric W. Weisstein. Navier-stokes equations, a wolfram web resource. Available from: `http://scienceworld.wolfram.com/physics/Navier-StokesEquations.html`.

[3] Christophe Schlick. An inexpensive BRDF model for physically-based rendering. *Computer Graphics Forum*, 13(3):233–246, 1994. Available from: `citeseer.ist.psu.edu/schlick94inexpensive.html`.

[4] Randi J. Rost. *OpenGL Shading Language*, chapter 14, page 359. Addison-Wesley Professional, second edition, 2006.

[5] NVIDIA Corporation. Nvidia gpu programming guide, 2005. Available from: `http://developer.nvidia.com/object/gpu_programming_guide.html`.

[6] Alain Fournier and William T. Reeves. A simple model of ocean waves. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 75–84, New York, NY, USA, 1986. ACM Press. Available from: `http://doi.acm.org/10.1145/15922.15894`.

[7] Mark Finch. Effective water simulation from physical models. In *GPU Gems*, chapter 1, pages 5–29. Addison Wesley, March 2004.

[8] Jerry Tessendorf. Simulating ocean water. *Siggraph 2004*, 1999–2004. Available from: `http://www.finelightvisualtechnology.com/pages/coursematerials.php`.

[9] Jason L. Mitchell. Real-time synthesis and rendering of ocean water. Technical report, ATI Research, April 2005. Available from: `http://www.ati.com/developer/techreports.html`.

[10] Michael Kass and Gavin Miller. Rapid, stable fluid dynamics for computer graphics. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 49–57, New York, NY, USA, 1990. ACM Press. Available from: `http://doi.acm.org/10.1145/97879.97884`.

[11] Karsten Ø. Noe and Peter Trier. Implementing rapid, stable fluid dynamics on the gpu. Available from: `http://projects.n-o-e.dk/GPU_water_simulation/gpu-water.pdf`.

[12] Nick Foster and Dimitri Metaxas. Realistic animation of liquids. *Graph. Models Image Process.*, 58(5):471–483, 1996. Available from: `http://dx.doi.org/10.1006/gmip.1996.0039`.

[13] Jos Stam. Stable fluids. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co. Available from: `http://doi.acm.org/10.1145/311535.311548`.

[14] Mark J. Harris. Fast fluid dynamics simulation on the gpu. In *GPU Gems*, chapter 38, pages 637–665. Addison Wesley, March 2004.

[15] Nick Foster and Ronald Fedkiw. Practical animation of liquids. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 23–30, New York, NY, USA, 2001. ACM Press. Available from: `http://doi.acm.org/10.1145/383259.383261`.

[16] Douglas Enright, Stephen Marschner, and Ronald Fedkiw. Animation and rendering of complex water surfaces. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 736–744, New York, NY, USA, 2002. ACM Press. Available from: `http://doi.acm.org/10.1145/566570.566645`.

[17] Anita T. Layton and Michiel van de Panne. A numerically efficient and stable algorithm for animating water waves. *The Visual Computer*, 18:41–53, 2002. Available from: `http://www.amath.unc.edu/Faculty/layton/research/water/`.

[18] Mark Carlson, Peter J. Mucha, and Greg Turk. Rigid fluid: animating the interplay between rigid bodies and fluid. *ACM Trans. Graph.*, 23(3):377–384, 2004. Available from: `http://doi.acm.org/10.1145/1015706.1015733`.

[19] W. T. Reeves. Particle systems – a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, 1983. Available from: `http://doi.acm.org/10.1145/357318.357320`.

[20] Lucy. A numerical approach to the testing of the fission hypothesis. *Astronomical Journal*, 82:1013–1024, December 1977. Available from: `http://adsabs.harvard.edu/cgi-bin/nph-bib_query?bibcode=1977AJ.....82.1013L&db_key=AST`.

[21] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[22] Simon Premoze, Tolga Tasdizen, James Bigler, Aaron Lefohn, and Ross T. Whitaker. Particle-based simulation of fluids. In *Proceedings of Eurographics 2003*, pages 401–410, 2003. Available from: `http://www.cs.utah.edu/vissim/papers/ParticleFluid/`.

[23] Matthias Müeller, Simon Schirm, Matthias Teschner, Bruno Heidelberger, and Markus Gross. Interaction of fluids with deformable solids, 2004. Available from: `citeseer.ist.psu.edu/mueller04interaction.html`.

[24] Matthias Müller, Barbara Solenthaler, Richard Keiser, and Markus Gross. Particle-based fluid-fluid interaction. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 237–244, New York, NY, USA, 2005. ACM Press. Available from: `http://doi.acm.org/10.1145/1073368.1073402`.

[25] Kenny Erleben et al. *Physics-based animation*, pages 329–391. Charles River Media, first edition, 2005.

[26] Alex Vlachos et al. *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, chapter Rippling Reflective and Refractive Water. Wordware Publishing, 2002. Available from: `http://www.ati.com/developer/shaderx/`.

[27] Chris Brennan. *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, chapter Accurate Environment Mapped Reflections and Refractions by Adjusting for Object Distance. Wordware Publishing, 2002. Available from: `http://www.ati.com/developer/shaderx/`.