# Simulation of Liquid Surfaces for Games

Bachelor's Thesis
by
**Daniel Gritzner**
from
Worms

submitted to
Lehrstuhl für Praktische Informatik IV
Prof. Dr.-Ing. W. Effelsberg
Fakultät für Mathematik und Informatik
University of Mannheim

August 2010

Supervisor: PD Dr. rer. nat. habil. Thomas Haenselmann

## Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mannheim, den 10.08.2010 Daniel Gritzner

# Contents

# List of Figures

# List of Tables

# Abbrevations

**ACM** Association for Computing Machinery

**CPU** Central Processing Unit

**DDR** Double Data Rate

**FFT** Fast Fourier Transform

**GCC** GNU Compiler Collection

**GDC** Game Developers Conference

**GPU** Graphics Processing Unit

**HDD** Hard Disk Drive

**LOD** Level of Detail

**OpenGL** Open Graphics Library

**PC** Personal Computer

**RAM** Random Access Memory

**rpm** rotations per minute

**SATA** Serial Advanced Technology Attachment

**SDL** Simple DirectMedia Layer

**SDRAM** Synchronous Dynamic Random Access Memory

**SIGGRAPH** Special Interest Group on Graphics and Interactive Techniques

# 1. Introduction

## 1.1. Motivation

In the last few years it has become quite common for games to include physics-based elements. The idea is that this creates a more immersive ("believable") game world for the player to experience. Also these elements themselves might be interesting enough to entertain players. In the end the idea is to make the game more enjoyable and thus sell more copies by getting good reviews from the media or by creating a franchise which players associate with a fun experience.

One of these elements are so called ragdoll physics which simulate the way a humanoid body moves through space when external forces are applied. E.g. first person shooter games use this approach to increase the immersion by causing each dead body to fall to the ground differently. Before ragdoll physics were used there usually was a finite set of animations which showed a body falling to the ground and how it lays there. When playing a game for longer periods this might remind the player that the world he sees is just virtual and thus break the immersion.

Though ragdoll physics are an interesting concept this bachelor's thesis will focus on another physical phenomenon namely liquid surfaces. With large parts of the earth covered with water and water being such a central element of our lives it is easy to see how including a believable representation of liquids may increase the immersion of the player into the game world. This thesis will show a way to simulate the surface of liquids in a way so that interactions with other objects can easily be implemented. Even today many games only provide fake interactions at most which makes the liquids (usually water) look less believable as soon as some object comes into contact with it.

While there are already visually convincing solutions from the field of fluid dynamics, which liquid surfaces are a subset of, these solutions usually are computationally expensive making them unsuitable for games. Because of this downside this thesis' focus is on just the surface of liquids to reduce the complexity of the problem and with it the complexity of the solution. After all games consist of many more things than just graphics. E.g. the behavior of the non-player characters has to be determined or the player's input has to be processed. This means that the simulation of the liquid surfaces may only use a small fraction of the already quite limited time which is available for each rendered frame. With 30 to 60 frames per second being the de facto standard the simulation has to be done in a fraction of 16 to 33ms without the use of expensive supercomputers.

## 1.2. Related Work

The work related to liquid surfaces can be divided into three categories:

- **Games:** The state of the art in modern games
- **Fluid dynamics:** Solutions using full fluid dynamics
- **Liquid surfaces:** Solutions with the same scope as this thesis

Each category is relevant in a slightly different way. The state of the art in games is interesting to get a clearer idea of the problem of liquid surfaces and their use in games. As already mentioned liquid surfaces are a subset of fluid dynamics. Thus a short overview of full fluid dynamics and why they are or might be unsuitable for games is appropriate. The relevance of the last category is obvious. It simply contains works that cover the same problem.

Due to the vast amount of work done in the field, especially concerning the simulation of water, each category contains only a representative selection of the existing work.

### 1.2.1. Games

Three points are important for this category. How do liquids look in current games, how are they implemented and in what way, in terms of gameplay, are they used.

At first the look of water in games. Other liquids only play very minor roles and usually the only liquid found in games is water. Most modern games only aim to achieve real looking water without really simulating it. This can be seen quite easily when looking at how it interacts with other objects in the scene like characters or bullets shot at the surface. If left alone the water in games like Crysis or Unreal Engine 3-based games like Gears of War 2 and BioShock looks quite convincing. As soon as the water has to interact with other objects aside from static geometry the immersion often breaks. This is due to the water not being simulated properly. It's look is only faked. Instead of real interactions an effect showing a concentric ring is rendered on top of the surface or particles trying to emulate a special effect like a splash are added to the scene. While some implementations, e.g. when the water is represented by a real mesh instead of a "simple" texture and normal map and the mentioned ring effect affects said mesh, might look quite convincing for a brief moment this impression does not last. These added effects only have a limited lifetime and only effect a limited part of the water surface. While this might look convincing enough at first, the way the water surfaces returns to it's previous behavior as if nothing has happened usually reveals that no real simulation is taking place. Also the amount of additional effects added is limited due to each effect costing additional CPU time and memory. This means that limits like the number of such effects and their lifetime might have a negative impact on how real the water looks if they are too low.

The second point that needs discussion is how the water is represented in the first place if it is not properly simulated. Due to the proprietary nature of most games only educated guesses can be made here. Looking at available solutions and the way water

behaves in games, as outlined in the previous paragraph, it seems like the water in most games is generated procedurally. This means for each point $(x, y)$ of the water surface and each point in time $t$ there is a function $h(x, y, t)$ which returns the height of the water at the given point in space and time. The function $h$ might actually be a combination of multiple functions, e.g. a sum of sine functions with different parameters. This model fits as it explains the behavior of games with a function $h_0$ defining the behavior of the water surface without any interactions and several functions $h_n$ added to it temporarily when interactions occur (and no particles are used). With the selection of proper functions and parameters this may already look quite convincing which is the reason why water in modern games usually looks real enough for their purposes. But it is apparent that this kind of implementation has strict limitations in terms of the number of effects because each additional effect adds processing time and occupies memory as already mentioned.

The last point that has to be considered is the way water is used in games. With a few exceptions water is used mostly for visual purposes to create more lifelike scenes. Sometimes this water is even used as a kind of "natural" border which limits the regions the player can visit in the virtual world. In these games swimming simply is not possible. Even in games where swimming or diving is possible most of the gameplay usually takes place on land. This minor role for water and other issues like development time constraints might be a reason why so many if not all games use procedurally generated water (solutions for this are easily available) instead of proper simulations. But as said before there are exceptions. Some games use water as a central gameplay element. Notable games in this regard are Wetrix, Wave Race: Blue Storm and BioShock. Wetrix is a Tetris-like puzzle game in which the player has to catch rain on an initially flat platform. Wave Race: Blue Storm is, as the name indicates, a racing game. The races are taking place on water using stand-up personal watercrafts (colloquially called Jet Skis). BioShock is a first person shooter set in an underwater city. At the time the player arrives in said city it has fallen into disrepair. The game tries to create a threatening atmosphere by effects like water leaking from the outer walls into the city. The game mechanics also allow larger pools of water to be electrically charged and by that electrocuting characters in such pools.

### 1.2.2. Fluid dynamics

Fluid dynamics is the term used in physics as the study of fluids which are in motion. Fluids can either be liquids or gases. As a result of the total motion of a fluid the motion or behavior of it's surface is implicitly defined, making the behavior of liquid surfaces a subset of fluid dynamics. There exists numerous implementations of fluid dynamics. Examples are Autodesk Maya ([13]) and the work of Robert Bridson ([17], [2]). Maya is an application used for creating 3D models, animations and simulations. It is used by people working in different industries that rely on visualization, e.g. the film industry or the video game industry. One of it's features is fluid simulation which allows the user to simulate the behavior of smoke, steam, different bodies of water (the ocean, ponds, etc.) among other things. Maya tries to achieve the highest possible level of realism. Thus it is used in environments where processing time is no critical issue. E.g. it does not

matter for a movie if it takes several minutes or even hours to render one frame. The audience will only see the precomputed result and thus the level of realism achieved is more important than the time it took to compute it. Unfortunately Maya is a proprietary product and so one has to rely on educated guesses again to asses how Maya works. The feature list on Autodesk's website ([13]) mentions particles among other things. So it might be possible that Maya uses a large amount of particles to simulate fluids. Each particle has to be simulated according to Newton's laws of motion. To do that the forces applied to each particle have to be evaluated. Other physical models like the Navier-Stokes equation ([5]) are necessary to do that. Depending on the actual model used it is easy to see why each simulation step might take long to compute. If the interaction between particles is taken into account (e.g. in the form of pressure) the calculations needed to find the forces applied to each particle rises quickly and also depends on the total number of particles or at least the number of particles in the close vicinity. Also to achieve a high level of realism a large number of particles each representing only a tiny amount of space is needed. The combination of at least quadratic complexity and a large problem instance (many particles) leads to the long computation time. This does not take into account that after each particle has been moved the implicitly defined surface has to be determined and rendered which also takes time.

As already mentioned this is just a guess of how Maya might work when simulating fluids. It is also entirely possible that it works similar to how Robert Bridson, associate professor at the University of British Columbia and co-founder of a company that produces software for physics simulations for use by the film industry, achieves his results. A paper by written by Jos Stam ([7]) indicates that at least older versions of Maya used techniques similar to the one described in this paragraph. Robert Bridson uses the Navier-Stokes equation as model for fluids. But instead of solely relying on particles he uses a hybrid approach using both grids and particles. He bases his algorithms on having a large grid which stores values like pressure or fluid density. Only where convenient Robert Bridson assumes that particles are at the locations defined by the grid. His algorithms consist of several steps to simulate the fluid motion. What makes his implementation too slow for games is that he has to solve large linear equation systems. Basically each grid cell contributes at least one equation and unknown variable (not accounting for the boundaries) and with sufficiently large grids like 256x256x256 this results in long computation times.

Even though the results are impressive as can be seen in the demo videos on Autodesk's website or in the images posted on Robert Bridson's website ([17]) the required computations are simply too expensive for video games which have only a few milliseconds per frame and have to run on cheap hardware. In this context one should mention the chapter in fluid dynamics in GPU Gems 3 ([10]) and look at Jos Stam's paper ([7]) again. Both claim being able to do fluid dynamics simulations in real-time. Unfortunately neither work includes proper performance evaluations. There are no sections giving details on the hardware used, the parameters used (e.g. grid sizes) and the performance achieved in frames per second or processing time per simulation step. Only rough indications are given pointing to ordinary consumer PC hardware being used. Jos Stam's paper also mentions in a section on extensions to his simulation that liquid simulations (specifically

water simulations) might not yet run in real-time at the time the paper was published in 2003. The game mentioned in GPU Gems 3 only features gases as well. But without more details it is difficult to asses if fluid dynamics simulations can really be done in real-time for games and what limitations still might exist like being able to simulate gases but not liquids. For future games these works look promising though.

### 1.2.3. Liquid surfaces

Most of the work done in this area are procedurally generated water surfaces. These have already been discussed in the subsection about games. The complexity of the functions varies quite a bit. While some functions can be evaluated directly like those based on sine functions as used by Intel ([15], [16]), other functions like one used by Vladimir Belyaev ([1]) are based on more complex physical models and require solutions like fast Fourier transform (FFT). A FFT has a complexity of $O(n \cdot \log n)$ which seems quite reasonable and in fact can be used for real-time applications as shown in the mentioned paper. But in the case of water simulations $n$ is actually dependent on a two dimensional plane and thus grows quadratically. So an even lower complexity like being able to evaluate the height at each point in constant time which results in a total complexity of $O(n)$ (again with a quadratically increasing $n$) would be preferable. Matthias Müller-Fischer achieves this with his height field simulations ([6]). To store the current state of the surface a grid is used. At each grid cell the surface height at that point on the plane is stored. How each height value changes over time is derived from looking at the forces applied to a given point of a membrane which serves as model for the liquid surface. It turns out that the change can be calculated with just looking at each neighboring cell resulting in the mentioned linear complexity. This also makes the simulation physically correct for all bodies of water which can be modeled in this way which means all bodies of water which only have waves with low amplitudes. Another advantage of this approach over procedurally generated water surfaces is that interactions with objects can easily be included without a cost penalty on a per object basis for the surface simulation.

Many papers on water surface simulation focus heavily on the rendering aspects. While this bachelor's thesis is more concerned with the simulation of the surface, rendering is still a related topic. The works of Vladimir Belyaev ([1]) and Rene Truelsen ([19]) are examples works showing how a water surface can be rendered in a convincing way. Rendering is non-trivial because of the various effects affecting how water looks. In terms of optics the water surface represents the border of two mediums with different optical properties which means that effects like refraction and reflection occur. Combined with the fact that water itself is colorless and the perceived color comes from particles dissolved in it and the color of the light shining on it, the water's color depends heavily on the environment it is in. Due to the refractions and reflections the water might also have an impact on it's environment in the form of caustics.

# 2. Simulating liquid surfaces

As seen in the previous chapter there is a vast amount of existing work related to liquid surfaces or in the most cases water surfaces. Many different models and solutions to the problem have been proposed. Thus at first the goals of this bachelor's thesis will be discussed in more detail in this chapter. From these goals a model and it's physical background will be derived. After the basics have been established there will be a discussion of additional phenomena like damping, reflections and interactions between liquids and objects. To give a complete picture of the simulation implementation details will be pointed out before concluding this chapter with hints on rendering, simulating arbitrary liquids and limitations.

## 2.1. Goals

The introduction already loosely stated some of the goals like creating more immersive virtual worlds but in a way that is fast to compute. However it is not exactly clear what that means. Thus this section provides a list of key points and a short discussion of those.

1. High performance
2. Realism
3. Simplicity
4. Low memory requirements

As initially mentioned games have tight requirements concerning the time that may be spent per frame to be rendered. In order to be able to render enough frames per second, to achieve a smooth looking simulation, only 16 to 33ms may be spent on each frame. Within this timeframe several different problems have to solved including among other things processing the players input, updating the state of the game world (e.g. by moving objects), detecting collisions and determining the behavior of non-player characters. Thus only a fraction of those 16ms to 33ms may be used for the simulation of liquids.

The second point should probably be called immersion instead of realism. It is more important to create virtual worlds that are believable and allow the player to immerse into than it is to represent the laws of physics of the real world as accurate as possible. Doing so would even exclude entire settings from being used like fantasy settings with monsters and magic or science fiction settings depicting technology which does not exist

yet. Basing the simulation on actual physics is helpful to create a believable world since it reflects what people can actually experience outside of games but accuracy may be sacrificed to some extent to improve on other points like performance.

The last two points are only minor issues. Both points are similar to the first one in that they are a result of games being usually about more than just water simulations. Depending on the available budget it might not be economically viable to spend too much time on implementing and debugging a complex simulation or buy and integrate middleware when developing a game. Depending on the role of water in the game it might simply be more effective to spend time and money elsewhere. This speaks in favor of a simple solution. But since one might also argue that the games industry has become so big that many games are produced with a large budget simplicity is only a minor issue. Low memory requirements are also just a minor issue. The combined memory available to the CPU and GPU of modern systems is in the range of several gigabytes already. This point is mainly mentioned with regard to video game consoles like the Xbox or the PlayStation which still only have available memory in the magnitude of megabytes.

Matthias Müller-Fischer's work outlined in the first chapter fits these goals quite well. It has a low computational complexity, it is based on actual physics and allows the inclusion of interactions between the liquid and objects relatively easy. It even fits the minor goal of simplicity. For these reasons the further work done in this bachelor's thesis is based on the work on height field fluids by Müller-Fischer. It is also worth noting that the version from SIGGRAPH 2007 ([6]) was used even though Müller-Fischer showed an updated version at the Game Developers Conference 2008 ([12]). But no video recording of the lecture at the GDC was available and the presentation slides alone are hardly comprehensible. A video of the SIGGRAPH 2007 version is available at the ACM portal ([4]).

## 2.2. Model

The model of the liquid surface has been chosen to be a continuous surface such that a function $f_s(x, y)$ exists which represents the surface by assigning a height value to each point on a plane. Said function represents the state of the liquid surface for one specific point in time. An example is shown in figure 2.1. To be able to represent arbitrary functions in memory a grid is used. This grid stores height values representing the distance from the bottom of the body of liquid being simulated to it's surface. It does not matter whether the intersections of the lines of the grid or any point inside each grid cell is used to discretize and store the height values. However we will see that it is beneficial to discretize the surface in such a way that the points are evenly spaced. This simplifies the implementation.

This model does not cover the behavior of the liquid surface yet. Thus another function $f_t$ is needed which assigns a given state $s_0 = f_s(x, y)$ and time difference $t$ the state $s_t = f_t(s_0, t)$. The state $s_t$ represents how the surface looks like after time $t$ has passed.

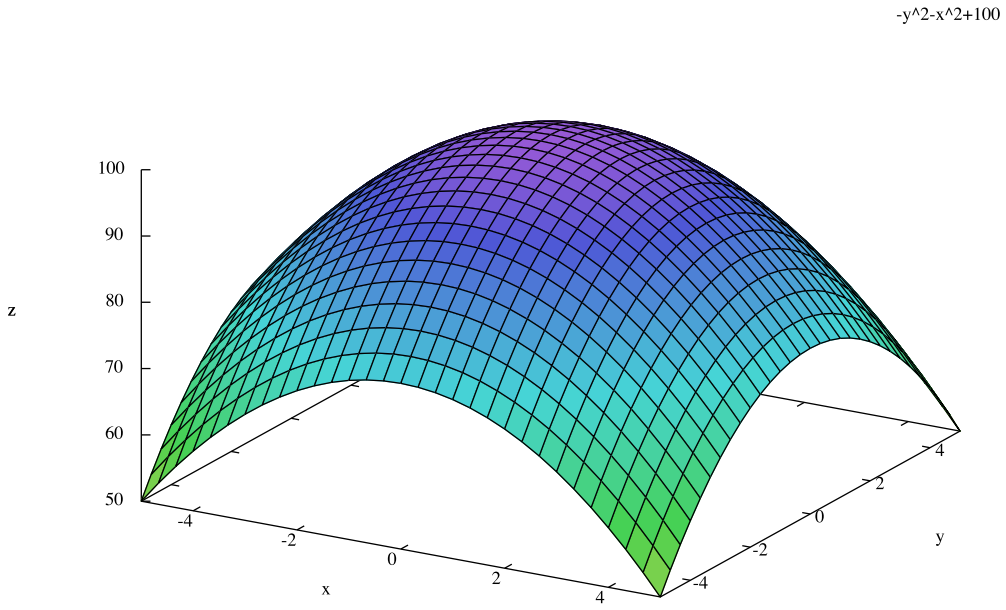



Figure 2.1.: An example for a function $f_s(x, y)$ describing a surface. The function is $f_s(x, y) = z(x, y) = 100 - x^2 - y^2$.

## 2.3. Physical background

As a physical background Matthias Müller-Fischer proposes the model of an elastic membrane with low stiffness for the liquid surfaces. The displacement of the membrane at different points translates to the heights of the liquid surface it shall model. Müller-Fischer uses the force acting on a infinitesimal point on the membrane surface which are caused by stress which in turn is a result of the displacement. This force in combination with Newton's laws of motion is then used to determine the height changes of the liquid surface.

Figure 2.2 shows the one-dimensional case of such a membrane. It shows a membrane in vibration at a fixed point in time. Due to the vibration most points of the membrane are displaced by a certain amount $u(x)$, or $u(x, t)$ to account for the fact that this displacement is dependent on both space and time. The figure also shows an enlarged cross section of the membrane. This cross section shows the forces acting on an infinitesimal point due to stress. This force acts in two directions. It acts from any given infinitesimal point onto it's surrounding and it also acts from the (infinitesimal) surrounding of each point onto the point itself. This is in accordance to Newton's laws of motion which state that for each force there is a reactive force with the same magnitude but opposing

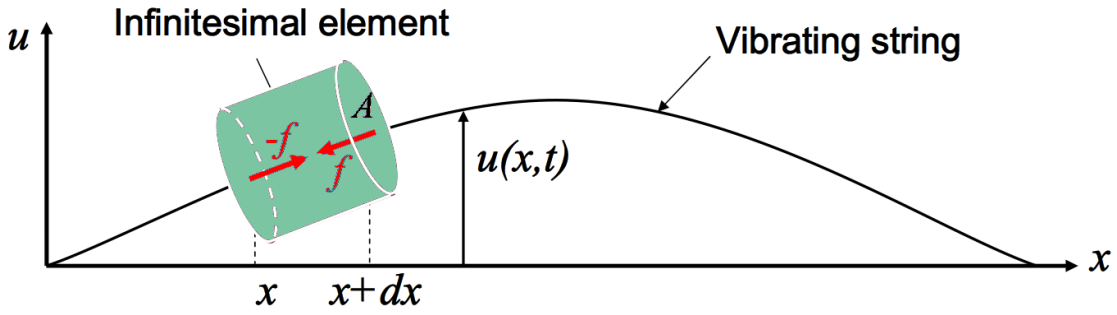



Figure 2.2.: The one-dimensional case of the membrane model. In this case the membrane becomes a string. This picture is taken from the slides of Matthias Müller-Fischer's lecture at SIGGRAPH 2007. The colors have been adjusted to fit this thesis.

direction.

With the assumption of constant stress $\sigma$ the force on an infinitesimal point is

$$f = \sigma \cdot A \tag{2.1}$$

with $A$ being the area of the membrane's cross section at this point. This follows directly from the physical definition of stress. This force is directed along the tangent of the membrane at that point. The part of this force that is actually interesting for determining the change of the displacement is the part in the direction of $u(x,t)$. As an simple approximation to get this magnitude one can simply multiply the right-hand side of the equation with the derivative of $u(x,t)$ with respect to $x$. To see that this works one has to look at the magnitude of $\frac{d}{dx}u(x,t)$. If $u(x,t)$ is constant the derivative and also the force in the direction of $u(x,t)$ becomes zero. The steeper the function gets the higher the magnitudes of both the derivative and the force get. This approximation works well for low displacements. Thus the magnitude of the force $f$ in the direction of $u(x,t)$ becomes

$$f_u = \frac{d}{dx}u(x,t) \cdot \sigma \cdot A. \tag{2.2}$$

This equation is not complete yet. As already mentioned Newton's laws also state that every force has a reactive force. Accounting for this reactive force the actual magnitude of $f_u$ is

$$f_u = \frac{d}{dx}u(x+\Delta x,t) \cdot \sigma \cdot A - \frac{d}{dx}u(x,t) \cdot \sigma \cdot A \tag{2.3}$$

$$= \left(\frac{d}{dx}u(x+\Delta x,t) - \frac{d}{dx}u(x,t)\right) \cdot \sigma \cdot A. \tag{2.4}$$

The first part with the derivative at $x + \Delta x$ is the force acting on the infinitesimal point from the surrounding points and the second part with the derivative at $x$ is the

counteracting force of the point on it's surrounding points. Note: this derivation will use $\Delta x$ when referring to the $dx$ shown in figure 2.3. This is done to avoid confusion with the derivative with respect to $x$.

Now one knows the force required for applying Newton's second law of motion which states $f = m \cdot a$. The acceleration $a$ in this case is actually $\frac{d^2}{dt^2}u(x,t)$ and thus the parameter which can be used to determine the change of the displacement which equals the change in height of the modeled liquid surface. The mass $m$ of the membrane cross section can easily be computed by multiplying the density $\rho$ with the volume $V = \Delta x \cdot A$. So the right-hand side of Newton's second law of motion becomes

$$(\rho \cdot \Delta x \cdot A) \cdot \frac{d^2}{dt^2}u(x,t). \tag{2.5}$$

The left-hand side of Newton's law is the force $f_u$ discussed in the previous paragraph. So the equation becomes

$$\left(\frac{d}{dx}u(x+\Delta x,t) - \frac{d}{dx}u(x,t)\right) \cdot \sigma \cdot A \quad = \quad (\rho \cdot \Delta x \cdot A) \cdot \frac{d^2}{dt^2}u(x,t). \tag{2.6}$$

To get the acceleration one has to divide by the mass $m = \rho \cdot \Delta x \cdot A$ which turns equation 2.6 into

$$\frac{d^2}{dt^2}u(x,t) = \frac{\frac{d}{dx}u(x+\Delta x,t) - \frac{d}{dx}u(x,t)}{\Delta x} \cdot \frac{\sigma}{\rho}. \tag{2.7}$$

To get the acceleration on an infinitesimal point the surroundings have to be chosen infinitesimally small as well which means that $\Delta x$ goes to 0. This means that the first part of the right-hand side of the last equation becomes another derivate simplifying the equation to

$$\frac{d^2}{dt^2}u(x,t) = \frac{\sigma}{\rho} \cdot \frac{d^2}{dx^2}u(x,t) \tag{2.8}$$

which can be further simplified to

$$\frac{d^2}{dt^2}u(x,t) = c^2 \cdot \frac{d^2}{dx^2}u(x,t) \tag{2.9}$$

with $c^2 = \frac{\sigma}{\rho}$. This differential equation can be solved with any function $f$ and defining $u(x,t) := a \cdot f(x + c \cdot t) + b \cdot f(x - c \cdot t)$. When calculating the derivatives of this $u(x,t)$ one sees why $\frac{\sigma}{\rho}$ was defined as $c^2$ instead of just $c$. In this definition of $u(x,t)$ one can also see the meaning of $c$. It is the speed with which waves travel.

For the proposed implementation using a grid storing heights at discrete positions $(x,y)$ equation 2.9 has to be discretized. To get the second order derivative in time one can simply use two first order derivatives. This means another parameter $v$ representing the velocity is introduced. The change in displacement at $x$ when time $\Delta t$ has passed in the discrete case is simply

$$u(x,\Delta t) = v(x,t) \cdot \Delta t \tag{2.10}$$

with the multiplication being the discrete version of integration necessary to get from the velocity to the displacement. The change in the velocity can be calculated similarly as

$$v(u, \Delta t) = c^2 \cdot \frac{d^2}{dx^2} u(x,t) \cdot \Delta t. \tag{2.11}$$

Since these equations only determine the magnitude of change over time the old values for $u(x,t)$ and $v(x,t)$ have to be stored and the change simply added to each position. This means the actual heights of the liquid surface are determined by

$$u(x, t + \Delta t) = u(x,t) + u(x, \Delta t) \tag{2.12}$$

and

$$v(x, t + \Delta t) = v(x,t) + v(x, \Delta t). \tag{2.13}$$

The second order spatial derivatives can be calculated with simple differences in the discrete case. To get the first order derivatives one simply calculates the differences to an immediate neighbor. There are two possibilities because there are two neighbors to choose from. The possibilities are

$$\frac{d}{dx} u \left[ x - \frac{1}{2} \right] = \frac{u[x] - u[x-1]}{h} \tag{2.14}$$

and

$$\frac{d}{dx} u \left[ x + \frac{1}{2} \right] = \frac{u[x+1] - u[x]}{h}. \tag{2.15}$$

Both are needed to calculate the second order derivative. It is the difference of those two possibilities. The second order derivative thus is

$$\frac{d^2}{dx^2} u[x] = \frac{\frac{d}{dx} u[x + \frac{1}{2}] - \frac{d}{dx} u[x - \frac{1}{2}]}{h} \tag{2.16}$$

$$= \frac{u[x+1] + u[x-1] - 2 \cdot u[x]}{h^2}. \tag{2.17}$$

This derivative can then be used to calculate the change in velocity and consequently the change in displacement or height. Note: square brackets have been used to indicate that values for discrete spatial points are being discussed. The parameter $t$ has been omitted because these values are the heights stored in the gird discussed early. This grid only represent the state of the liquid surface for a specific point in time.

So far only the one-dimensional case has been discussed. When doing the same for the two-dimensional case equation 2.9 becomes

$$\frac{d^2}{dt^2} u(x,y,t) = c^2 \cdot \left( \frac{d^2}{dx^2} u(x,y,t) + \frac{d^2}{dy^2} u(x,y,t) \right) \tag{2.18}$$

and the discrete second order spatial derivate becomes

$$\frac{u[x+1,y]+u[x-1,y]+u[x,y+1]+u[x,y-1]-4\cdot u[x,y]}{h^2}. \tag{2.19}$$

The other equations like the velocity change or height change (equations 2.10 to 2.13) stay the same. With these equations one arrives at the following pseudo-code implementation of the liquid surface simulation:

```
1 forall i,j:
2         du = u[i+1,j] + u[i-1,j] + u[i,j+1] + u[i,j-1] - 4*u[i,j]
3          f = c**2 * du / h**2
4     v[i,j] = v[i,j] + f * t
5    u2[i,j] = u[i,j] + v[i,j] * t
6 endfor
7 forall i,j:
8    u[i,j] = u2[i,j]
9 endfor
```

This pseudo-code implementation uses the following symbols:

- $u$: an array storing the heights at each point $(i, j)$
- $u2$: an array storing the heights after time $t$ has passed
- $v$: another array storing the velocity with which the heights in $u$ change (initialized to 0 at all points before the first simulation step)
- $f$: force derived from the physical model which causes the change in height
- $t$: time which has passed between the states in $u$ and $u2$
- $c$: the speed with which waves travel over the surface
- $h$: the grid spacing (assumed to be constant)

Note: the notation a**b is used in the pseudo-code for $a^b$.

This pseudo-code already implements the basic surface simulation allowing waves to spread when $u$ gets changed by external forces. Once such an external force has been completely applied it is not needed anymore to simulate how the waves will spread. The force is only needed for the initial change in height. The pseudo-code is the function $f_t$ mentioned at the end of the section discussing the model.

## 2.4. Damping

In his lecture at the GDC 2008 Matthias Müller-Fischer also discusses the physical phenomenon of damping. The above pseudo-code causes the waves to spread endlessly. With a simple experiment it can be shown that this behavior is not realistic: if you fill

a bowl with water and causes some waves maybe by throwing a small object into it, the amplitude of those waves will decrease over time. This is called damping. The approach taken when deriving the force from the membrane model simply does not take such a damping force into account. Müller-Fischer proposes three ways to fix this:

1. including a damping force
2. scaling the values in the array $v$ with some factor $s \in (0, 1)$
3. limiting the magnitude of the change made to the array $u$ at each point

While the first idea is the only one with a physical background it is also difficult to derive how this force should be included and which parameters should be used for it to achieve the damping effect. It simply does not follow from the membrane model. With this problem in mind the second idea was adopted for this bachelor's thesis. Both the the second and third ideas offer easy to understand and direct control of what happens. The third idea has the disadvantage of requiring more changes to the code including conditional statements and having no effect on waves with small amplitudes. The negative impact of branching (conditional statements) is highly dependent on the hardware architecture used to run the code on. While most modern CPUs have features like branch prediction to mitigate the negative impact of conditional statements to some extend other architectures might be simpler in order to dedicate more hardware to actual computations. E.g. on GPUs branching is considered to be expensive and should be avoided. GPU Gems 2 ([9]) features a short article on this issue.

## 2.5. Reflections

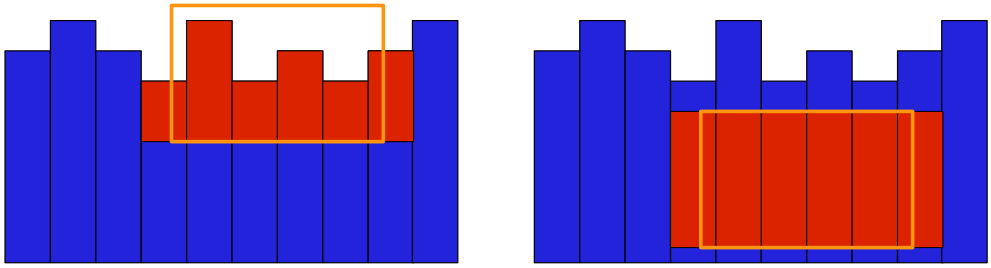
So far there has been no discussion on how to handle the boundaries or for the same matter large objects floating on the surface. Since the size of the simulated liquid surface is fixed (for more details see the section on limitations later int his chapter) one can assume that some kind of wall is at the borders of the simulated surface. What happens when a wave collides with a solid wall is that it is reflected. Matthias Müller-Fischer also gives a solution on how to implement those reflection. The solution is to simply clamp the height values at all boundaries. This means that if a value beyond the grid, e.g. $u[-1, j]$, or value for a grid cell which is occupied by large floating object at surface is addressed, one simply uses the height value of the current cell instead. E.g. when updating $u[0, 0]$ the values for $u[-1, 0]$ and $u[0, -1]$ are needed. But instead of trying to guess some values for these two non-existent cells the value for $u[0, 0]$ is used instead in both cases. Unfortunately Müller-Fischer does not clarify why this causes reflections. When giving is lecture at SIGGRAPH 2007 he indicated that the answer might be given in Robert Bridson's part of the same lecture ([6], [4]).

As already stated large objects, e.g. a ship, have the same effect as solid walls and will reflect waves. This means that one has to know which grid cells are occupied by such objects when updating the height values. To avoid having to do collision queries each time a height value is read from $u$, a boolean array $b$ can be used to store if any

given cell is occupied and clamping should be used. When updating the height values occupied cells might be skipped. But in order to avoid more or less obvious artifacts when visualizing the surface it is still advisable simply update the values for those cells as well. The rendered surface still has to extent to the real boundaries of the object, which might not be aligned with grid. Depending in on how the visualization is done a constant (non-updated) height value for such a cells might be visible. E.g. when a triangle mesh is used to represent the surface some vertices would remain constant which might be visible in the way the surrounding triangles change their "shape" over time. When not skipping over blocked cells one should use extra damping for those cells to avoid numerical explosion. Due blocked cells likely being surrounded by more blocked cells the force acting on the heights at those cells is low meaning the velocity at those cells will hardly change. Thus over time huge heights might develop without extra damping.

## 2.6. Object to liquid interactions

What happens when an object comes into contact with a liquid surface is that it pushes liquid aside. To keep track of how much liquid is replaced by an object Matthias Müller-Fischer proposes to add another array called $r$. For each grid position $r$ holds the amount of liquid currently replaced. Before each simulation step $r$ is updated and the difference at each grid point is calculated. This difference is then either added equally to the neighboring cells if it is positive or subtracted if it is negative.



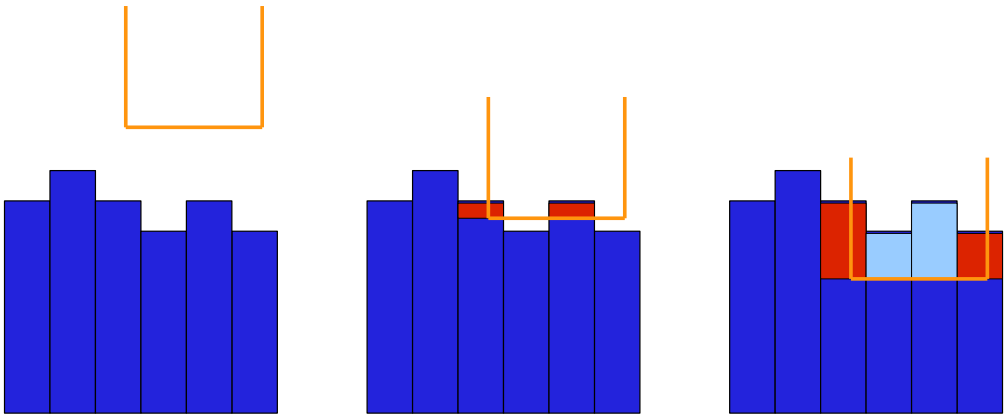
(a) Object at surface (b) Object fully submerged

Figure 2.3.: Water being replaced by an object. The water is shown in blue, the replaced volume is marked red and the object's boundaries are indicated by a brown box.

To calculate $r$ one may do collision queries at each grid position. The problem to be solved is finding out if the point defined by the grid position and the corresponding height value is inside any object or if any objects are inside the interval defined by the grid position starting at the bottom of the simulated body of liquid and extending to it's surface. If the query concerning the point inside an object evaluates to "true" this

means there is an object at the surface which means the array $b$ has to be updated. If the interval query evaluates to "true" the amount of overlap has to be determined and this will be the new value of $r$. Going into detail here is beyond the scope of this thesis. Any paper or book that covers collision detection might be consulted on this issue, e.g. the book on collision detection by Christer Ericson ([3]). To reduce the number of collision queries one could define a bounding box around the body of liquid and use this box to determine a set of objects which might collide with the liquid. The projection of the bounding boxes of the objects in this set may then be used to determine for which $r[i, j]$ collision queries have to be made. For all other position logically follows that $r$ is 0 and $b$ is "false".

Another issue arrises when the point query is "true". Then water cannot simply be added or subtracted from the height value at the current position or it's neighbors. The vicinity of the object at the surface has to be found. This problem is similar to finding regions in a digital image. The region of the object on the surface is defined by a connected set of neighboring grid positions for which $b$ is set to "true". The direct neighbors of this region is the vicinity that has to be modified. Again too much detail would be beyond the thesis' scope. One solution to region finding is so called flood filling but other techniques might be used as well.



(a) Bucket above water (b) Bucket colliding with water (c) Bucket partially submerged

Figure 2.4.: This figure depicts the problem with certain objects and the water they replace. A bucket falling into water is shown. In (c) there should be no water in the bucket but the simulation will still show some water inside it (shown in light blue).

This solution still has an unsolved problem namely that some objects are not handled properly. E.g. a bucket which falls down straight with it's bottom facing the surface will seem to have non-solid walls with water magically getting inside it once the bottom is submerged below the surface but the rest is not.

## 2.7. Liquid to object interactions

On any object in a liquid a force called buoyancy is applied. This force pushes the object upwards toward the surface or maybe even on top of it. The magnitude of this force on floating objects can be determined by Archimedes' principle. It states that the force pushing the object up is the reaction to force pushing the water away. This means it is a reaction to the gravity affecting the object. Thus follows that the direction of this force is the opposite of the direction of gravity and that the magnitude is the mass of the replaced liquid times the gravitational acceleration ($g \approx 9.81$ m/s$^2$). The mass of the replaced liquid is the volume times the density $\rho$ and the volume at each grid position is $h^2$ (the size of a grid cell) times $r$ (note: if multiple objects contribute to $r$ at a given position only the contribution from each object itself must be used to the determine the buoyancy applied to it). Written in one formula the magnitude of this force at each grid cell is is

$$f = r \cdot h^2 \cdot \rho \cdot g. \tag{2.20}$$

The total force on the object is naturally the sum of the forces at all grid cells.

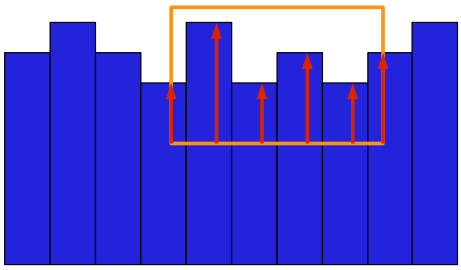


Figure 2.5.: Buoyancy. The force caused by each water column on an object is shown as a red arrow. The total buoyancy is the sum of all these individual forces.

## 2.8. Extended implementation

With various extensions having been discussed so far it is time to look at an updated implementation of the simulation:

```
1 forall i,j:
2   dr[i,j] = -r[i,j]
3    r[i,j] = 0
4    b[i,j] = false
5 endfor
6 forall objects o colliding with liquid's bounding box:
7   f = 0
```

```
 8    forall i,j in projection of o's bounding box on liquid surface:
 9       b[i,j]  = doPointQuery(i,j,u[i,j],o)
10            x  = doIntervalQuery(i,j,u[i,j],bLvl,o)
11       r[i,j] += x
12      dr[i,j] += x
13            f += h**2 * x * p * g
14    endfor
15    applyBuoyancy(o,f)
16  endfor
17  distributeReplacedWater(u,dr,b)
18
19  forall i,j:
20         du = -4 * u[i,j]
21         if i == N-1 or b[i+1,j]:
22           du += u[i,j]
23         else:
24           du += u[i+1,j]
25         if i == 0 or b[i-1,j]:
26           du += u[i,j]
27         else:
28           du += u[i-1,j]
29         if j == M-1 or b[i,j+1]:
30           du += u[i,j]
31         else:
32           du += u[i,j+1]
33         if j == 0 or b[i,j-1]:
34           du += u[i,j]
35         else:
36           du += u[i,j-1]
37          f = c**2 * du / h**2
38      v[i,j] = s * (v[i,j] + f * t)
39         if b[i,j]:
40           v[i,j] = v[i,j] * sb
41    u2[i,j] = u[i,j] + v[i,j] * t
42  endfor
43  forall i,j:
44    u[i,j] = u2[i,j]
45  endfor
```

New symbols added since the previous pseudo-code listing are:

- *dr*: array containing the change in displaced liquid compared to the previous simulation step
- *r*: array containing the amount of displaced liquid (has to be initialized to 0 at all

positions before the first simulation step)

- $b$: array indicating at which positions reflections should occur (indicates the presence of floating objects or objects which are entering the liquid)
- $f$ (lines 7-15): buoyancy
- $bLvl$: the height of the bottom of the liquid body in world space
- $x$: liquid displaced by a given object
- $p$: the density of the liquid
- $g$: gravitational acceleration
- $N$: size of the grid in the dimension of $i$ ("horizontal size")
- $M$: size of the grid in the dimension of $j$ ("vertical size")
- $s$: scaling to achieve a (fake) damping effect
- $sb$: extra scaling/damping for blocked cells

With all these extension the simulation has become a lot more complicated already. Lines 1-17 have been added to do liquid-object interactions. These lines left some details undecided like how to do collision detection. This was done on purpose to stay within this thesis' scope. Lines 20-36 are an updated version of the previous implementation which calculated $du$. The change was done to include reflections. There is also a change in line 38 and lines 39-40 have been added to include damping.

While it is difficult to argue about lines 1-17 without filling in more details one might take a closer look at lines 19-45 though. These lines do the updating of the actual heights. One property of the for-loops in those lines is that the result at each $u[i, j]$ is independent of the result at all other positions $(i', j')$. From that follows that lines 19-43 might be parallelized. Each thread calculates the results for a different subset of positions $(i, j)$. One might even try to calculate the simulation on the GPU. Problems when doing so are that branching should be avoided (meaning the five if statements might cause performance problems) and the values in $u$ need to be read back to the main memory if the code in lines 1-17 cannot be executed on the GPU as well.

## 2.9. Rendering

The section on related work in the first chapter already stated that the focus of this bachelor's thesis is more on the simulation itself than on rendering it's results. But since the simulation is basically useless without a visualization and that there is potential for optimizations a short discussion is still appropriate.

The first idea one might have is to simply use bars to represent the surface. For each value stored in the grid a bar is used with a height depending on the value itself.

The base area of the bars is chosen such that all bars have the same base area and the entire surface covered by the grid is covered with bars. The bars then extend from the ground to the surface of the liquid. Unfortunately one would need large grids to achieve a satisfying results and the resulting surface normals, which are required for calculating optical effects like reflection and refraction, would still be basically useless.

A better approach is to derive a triangle mesh representing the surface directly. Fortunately this is quite easy. As basis one simply defines a vertex for each value in the grid. The vertex' $x$ and $z$ values are determined by the position in the grid and $y$ value is the height stored in the grid. Each of these vertices (aside from the ones on the border) has to be connect with all his direct neighbors including the diagonal neighbors. As an optimization the heights of the grid could be stored and read directly in the triangle mesh. This means that no dedicated array $u$ is needed. This triangle mesh can then be easily used for all further processing required for rendering. In modern games this usually means deriving normals from the triangle faces, transforming the vertices from object- to world- to screen-coordinates using matrix multiplications, rasterize the triangles and execute fragment shader programs to determine the color of each pixel covered by the triangles. These shader programs then can produce effects like refraction and reflection. But this entire process is beyond the scope of this thesis as it is a topic of it's own. Works which give more details on this are already mentioned in the section on related work.

## 2.10. Liquids other than water

So far the word liquid could have been used interchangeably with the word water. Given the importance of water in our lives it is not surprising that water is the most interesting liquid to talk about. But games may also include scenarios in which other liquids like slime or lava play an important role. Thus it is interesting to discuss what liquid-specific parameters there are in the simulation, how they affect the results and consequently how different parameters might be used to simulate different liquids.

The liquid-specific parameters are:

- $p$: the density of the liquid which in turn affects the buoyancy
- $c$: the speed with which waves travel
- $s$: the damping

These parameters may be used to adjust the perceived "viscosity" of the liquid. If $p$ is sufficiently large objects will sink slower and less far into the liquid which in turn causes smaller waves. The spreading of the waves may be slowed down by using low values for $c$ and $s$. By making a liquid seem more or less viscose and using different fragment shaders to achieve different colors one may simulate liquids that do not necessarily feel and look like water.

## 2.11. Limitations

There are basically two kinds of limitations. Some limitations follow from the model itself others from the proposed implementation. These will be discussed here in addition to some extensions of the simulation Matthias Müller-Fischer proposes but which have not been included here.

### 2.11.1. Model

In the beginning of this chapter the model which was proposed included the idea that the water surface can be represented by a continuous height function at any given point in time. While the model might seem to suffice for all water surfaces at the first glance, there are two major phenomena which cannot be represented in this model. The first one is a breaking wave. Under certain circumstances the top of a wave might tip over such that there is an overlap of parts of the surface when viewed from above. A function which assigns height values to points on a plane simply is not expressive enough to model this phenomenon. In real life this occurs usually when ocean waves arrive at a beach and is known by most people in contexts such as surfing or tsunamis. The other phenomenon are splashes of liquid which become disconnected from the rest. E.g. the water in a swimming pool might splash over the edge of said pool and thus a portion of water becomes disconnected from the rest of the water. Due to the assumed continuity of the surface in the model this cannot be simulated by this model either.

### 2.11.2. Implementation

The proposed implementation basically uses arrays to represent values on a fixed, evenly spaced grid. The even spacing is used to optimize the simulation by simplifying the calculation of the force $f$. The spatial derivatives could not be calculated so easily if the points in the grid were not evenly spaced. As a result the size of the simulated liquid surface is also fixed. This becomes a problem when trying to simulate liquids arriving at sloped surfaces, e.g. water waves arriving at a beach. What happens is that some water splashes onto the beach and then flows back leaving (wet) sand behind. This means that the border of water surface changes over time. From that observation follows that this implementation is suited for lake- or pool-like bodies of water where the water is confined inside a specific space. Another implementation issue is that information may propagate only one cell at a time. This is a result of only using the direct neighbors to determine the force a each point of the grid. In practice this means that the equation $c < \frac{h}{t}$ must be true at all times. As a reminder $c$ is the speed with which waves travel, $h$ is the grid spacing and $t$ is the time that has passed between the old state of the surface and the new one to be calculated. While this condition might be relaxed by taking more neighbors into account when updating the heights on the grid one has to consider that for each additional cell that information may travel, and therefore increasing the maximum allowed value for $c$, four additional neighbors have to be taken into account. This means that the cost (additional neighbors considered and the resulting complexity

of the expression) grows faster than the gain (number of cells information may travel).

### 2.11.3. Other extensions

For the sake of completeness other extensions proposed by Müller-Fischer are mentioned here even though they could not be properly included into this thesis due to the lack of a video of his lecture at the Game Developers Conference 2008.

He recognized the problem that a fixed surface sizes poses in terms of scenarios the simulation is applicable to, too. His idea for a solution was to use a hybrid solution which uses not only height fields but also other techniques. E.g. a part of the water could be simulated with height fields and surrounding this simulation procedurally generated water is placed. A viable solution though one has to remember that this requires two different implementations. Both of which might be quite complex or, to reduce the complexity of one of the implementations, which might result in parts of liquid being simulated with fewer features like no object-liquid interactions. He even showed a demo of a pool of water from which one side gets removed and consequently the water flows out of the pool. Even though there is a slide which indicates his approach (a ghost column at the border with a memory of the height of the liquid) it is basically impossible to deduce his entire approach from one slide alone without knowing what he said when showing the slide.

The model and implementation as discussed in this thesis might be able to handle horizontal movement of objects in the liquid but this is an extension which was not directly considered when deriving the equations used for the simulation from the membrane model. Thus Müller-Fischer proposes to use a more realistic physical model like the shallow water equations. But the slides indicate that even at the lecture no details were discussed.

Another proposed idea to improve the realism is by adding specific phenomena using specialized implementations. Criteria to identify the occurrence of such special case have to be added and if appropriate specialized simulations have to be triggered. This is similar to the approach used for liquid-object interactions when using procedurally generated liquid surfaces. This idea is given as an example in the form of breaking waves. The specific idea in this case is to identify steep wave fronts and add a special polygon mesh just for simulating the breaking wave. Adding particles is mentioned as well. By using special implementations for special cases the possibilities are virtually endless but so is the required implementation effort and the potential hit on performance.

# 3. Evaluation

This chapter will cover the results of implementing the algorithms discussed in the previous chapter. At first the setup used for testing will be described in detail. This is then followed by analyses of the performance and of the realism of the simulation.

## 3.1. Test Setup

An application implementing the algorithms from chapter 2 was created. Aside from simulating the liquid surface it was also able to visualize the results. The application was build using C++, boost ([8]), SDL ([18]), SDL_ttf ([11]) and OpenGL ([14]). Table 3.1 shows the test system used both for developing and benchmarking. The application allowed to vary various parameters like the grid sizes, the wave speed or the liquid density. Additionally crates could be thrown into the liquid to test liquid-object interactions. The liquid pool and the crates were always axis-aligned simplifying the implementation of the interactions by making collision detection, projection of the crates onto the liquid surface and finding the vicinity of the crates easy. This also means that in a real game interactions could be more costly than in this application. The simulation of the liquid surface (lines 19-45 of the extended pseudo-code in chapter 2) could be computed using two threads each thread computing half of the new heights. The visualization was quite simple using a transparent, textured triangle mesh derived from the heights without any of the more advanced effects mentioned in the rendering section.

| **Component** | **Component name and model** |
|---|---|
| CPU | 2.4GHz Intel Core 2 Duo |
| RAM | 2x 2 GB 1066MHz DDR3 SDRAM |
| GPU | NVIDIA GeForce 9600M GT (256 MB shared memory) |
| HDD | 250 GB SATA (5400 rpm) |
| Operating system | Mac OS X 10.6.4 |
| C++ Compiler | GCC v4.2.1 |

Table 3.1.: Test system used for development and benchmarking

## 3.2. Performance and memory consumption

The performance was measured by recording the times required for computing liquid-object interactions and for computing the liquid surface simulation for 100 consecutive

frames. These times were then averaged. The appendix contains detailed tables with the results. An excerpt is shown in table 3.2. It shows the times needed for computing liquid-object interactions, the time needed for liquid surface simulation and the sum of both. Time needed for other things like rendering is not included. The dependence of the computation times on the number of threads and the grid size is shown.

| **no. of threads** | **grid size** | **time for interactions** | **time for liquid simulation** | **total time** |
|---|---|---|---|---|
| 1 | 128x128 | 0.10 ms | 0.91 ms | 1.01 ms |
| | 256x256 | 0.31 ms | 2.59 ms | 2.90 ms |
| | 512x512 | 1.22 ms | 10.06 ms | 11.28 ms |
| | 1024x1024 | 5.58 ms | 40.70 ms | 46.28 ms |
| 2 | 128x128 | 0.16 ms | 0.32 ms | 0.48 ms |
| | 256x256 | 0.32 ms | 1.51 ms | 1.83 ms |
| | 512x512 | 1.23 ms | 5.87 ms | 7.10 ms |
| | 1024x1024 | 5.58 ms | 22.98 ms | 28.56 ms |

Table 3.2.: Simulation performance when 30 objects interact with a liquid

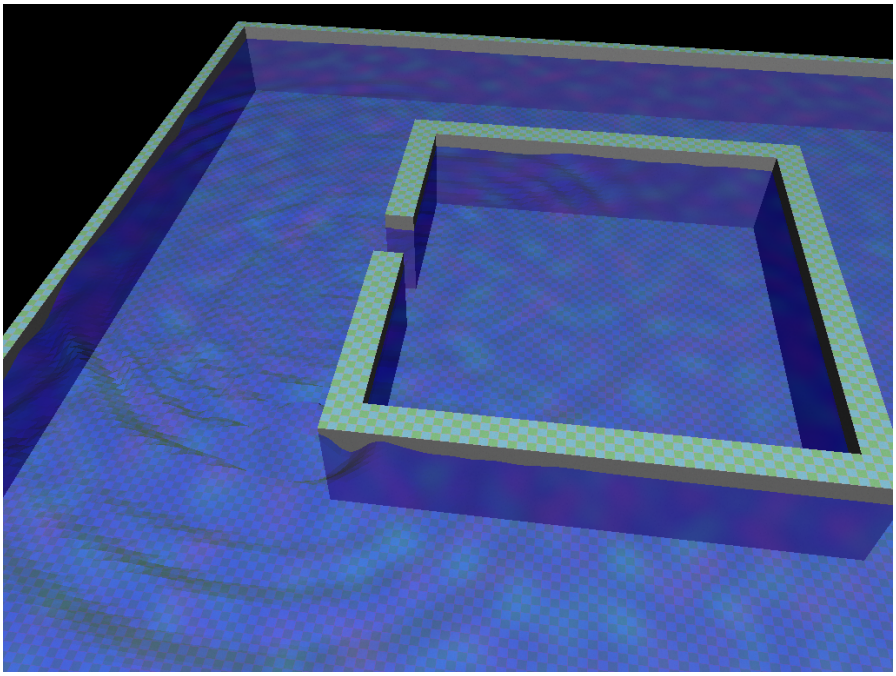
With the limit of 16ms to 33ms per frame in mind one can clearly see that it is out of the question to use grids larger than 512x512. Anything up to this limit is fine though. Without multithreading the use of a 512x512 grid is questionable but still possible depending on the complexity of the rest the application. When using just one thread on average 89% of the simulation time was spent on the surface simulation and consequently 11% was spent on the liquid-object interaction. The relative time spend on the surface simulation is reduced to 78% when using a second thread. Multithreading reduced the time spent on computing the new heights by 48% on average showing that the simulation can really easily be parallelized. For most games 30 objects interacting with the liquid at the same time is quite a large number and most of the time is spent on simulating the surface anyway. So even with more complex collision detection the simulation would probably still be fast enough to use in a real game. The memory consumed by the five floating point number arrays and the one boolean array necessary for the simulation can easily computed as up to 21 MB for a 1024x1024 grid considering that single precision floating point numbers were used. For a modern computer this might pose no problem but for an Xbox 360 or PlayStation 3 this is a significant amount of the 512 MB memory those consoles have. This is especially bad for the Xbox 360 because games running on this console can not rely on the presence of a hard disk for swapping. The memory consumption of smaller grids is no issue though. For a 512x512 grid the memory requirement is reduced to 5.25 MB already not accounting for possible optimizations.

## 3.3. Realism

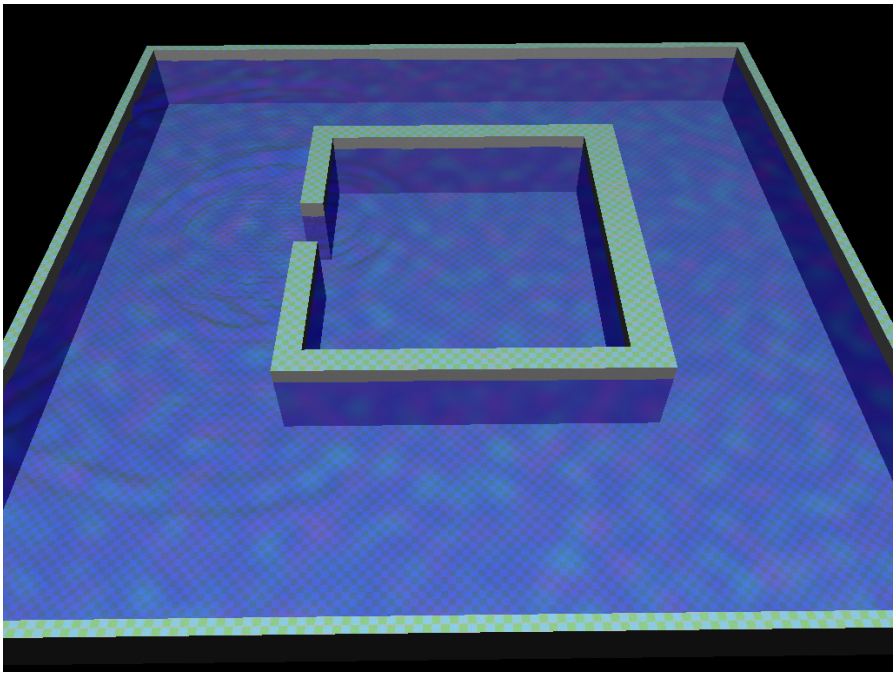
The simulation worked mostly as expected. The various parameters like wave speed had the effect one expected, meaning that behavior of the liquid could be adjusted in such a way that it does not necessarily looked like water. Simulating waves with a low amplitude worked quite well including reflections. While already looking nice with a grid size of 128x128 for a surface covering nearly the whole screen the look could still be improved by increasing the grid size. But unfortunately waves with high amplitudes looked unnatural due to being too smooth. They lacked the sharp tops one expects from such waves. Object interaction worked as expected as well. The interactions looked nicer than what most games offer. This is especially true in comparison with games using texturing tricks instead of modifying triangle meshes when applying the effects to liquid surfaces. The object-to-liquid interaction did not look convincing in every case though. Since the velocity of objects colliding with the surface is not considering when spreading the liquid around to create waves, the generated waves were too small. Another problem was dragging objects, which are far below the liquid surface, around. They affected the surface in just the same way as floating objects being dragged around. One would expect that the effect of the replaced liquid on the surface is dependent on the depth of the object inside the liquid.

Stability was a major concern. In games the frame rate usually determines the time steps used for any kind of simulation. With 30 frames per second as a lower limit this means that the time steps could be up to 33ms. For 1024x1024 grids this time step was not small enough to allow for sufficiently fast waves for simulating water. This limitation (the limit on the wave speed because information can only travel on grid cell at a time) was discussed in chapter 2. In addition to that problem numerical explosions occurred sometimes. This means that at some grid cells the height become unrealistically high or low causing visible spikes on the surface. These artifacts appeared more often with larger grids or when many objects interacted with the surface in close proximity to each other or close proximity to borders.

Figures 3.1 and 3.2 show screenshots of the simulation in action. The look of liquid surface could easily be improved by adding better rendering with surface normals interpolated over the mesh instead of using one fixed normal per triangle and adding effects like reflection and refraction.

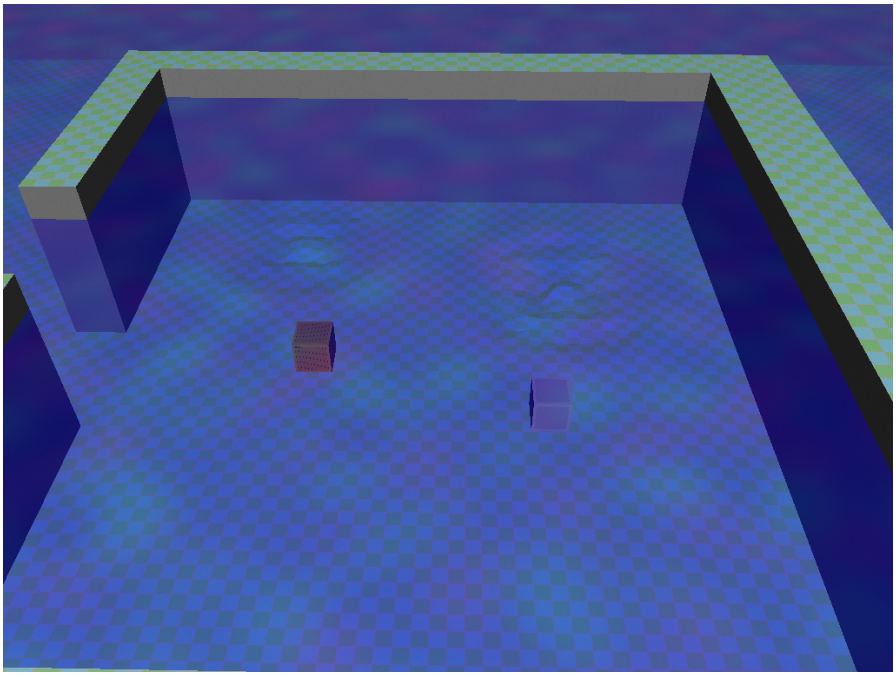


(a) 128x128 grid

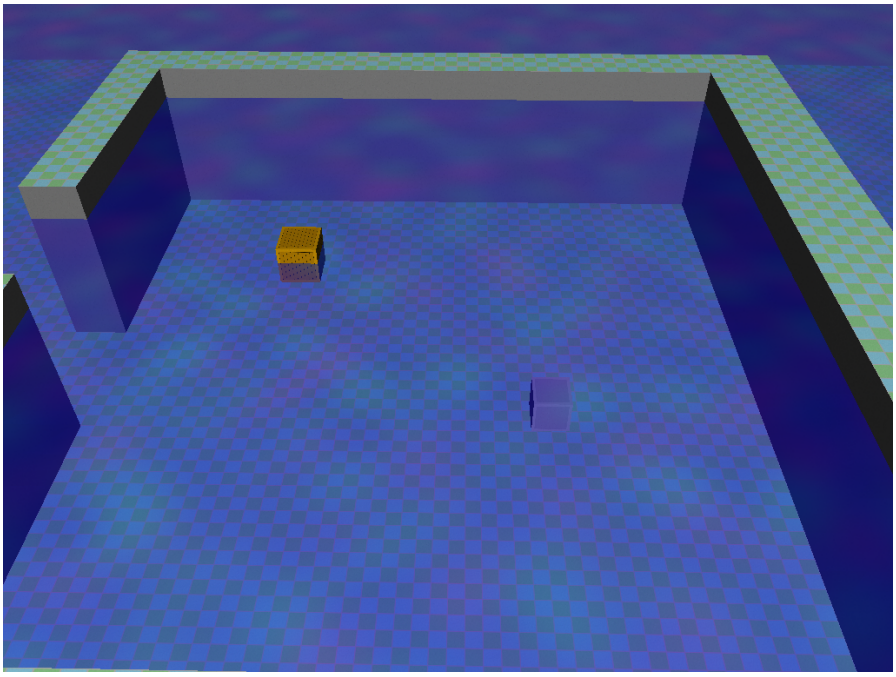


(b) 256x256 grid

Figure 3.1.: Simulation results. Starting with a pool of quiet water waves have been caused by adding a sufficiently large pulse near the entrance to the inner section. The waves can be seen to spread beyond obstacles and being reflected by the borders on the left side.



(a) The crates have just been dropped into the water



(b) Situation after a few seconds

Figure 3.2.: Further simulation results. Two crates made of wood and metal have been thrown into the water. The insufficient wave amplitudes can be seen. After a few seconds the waves have dissipated and the lighter wooden crate has risen back to the surface and floats on the water due to buoyancy.

# 4. Conclusion

This chapter concludes this thesis by discussing points to improve on in future works and summarizing the work done in this thesis.

## 4.1. Suggestions for future works

Obvious points one could work at are the limitations discussed throughout this thesis. Stability is an issue which not only puts limitations on the simulation but also requires careful tuning of parameters to avoid artifacts, e.g. the spikes mentioned in the evaluation. Different stability issues could be solved in different ways. The limit on wave speed could be relaxed by taking more than just the direct neighbors into account when deriving the force acting on each grid cell. To solve the issue with the spikes one would probably need a better physical model like the shallow water equations or even the full Navier-Stokes equation. A better model would also help making waves with high amplitudes looks more natural. Additionally it would allow taking more forces into account like wind or real damping improving the realism further. Thus other physical models are an interesting direction for future works.

Another big limitation is that only "water in tank"-like liquids can be simulated. This comes mainly from the fact that the simulation is based on a grid. While one could try to work without assuming that the grid cells are evenly spaced this would basically turn the simulation into a particle-based one. While interesting in itself it does not make sense to use this thesis as a basis for a particle-based simulation. As Matthias Müller-Fischer already mentioned one could try to add special cases like breaking waves or a change in the size of the surface (e.g. waves arriving at a beach) by identifying when they occur and using specialized implementations for their simulation.

To improve performance and memory consumption (five floating-point number arrays) one could look into using multiple small grids instead of one large grid. E.g. two 128x128 grids could be used with one covering the entire liquid surface and the other covering just a small fraction of the surface surrounding the player increasing the resolution and thus realism near the player. The size of the surface covered by grid surrounding the player would not need to change, only it's location would have to be dynamic. This means that a future work could look into how to efficiently find out which parts of the coarser grid do not need to be updated because of the finder grid around the player. Also a problem to be solved in this situation is how to efficiently transfer heights from one grid to another considering that one grid is moving around. This technique is sometimes referred to as "level of detail" (LOD) or sleeping and is already used in games for other problems most notably when rendering graphics.

Aside from full fluid dynamics simulation little has been done to simulate liquids other than water or even interactions between multiple liquids. While this might seem irrelevant for games, game developers adapt to their given possibilities, e.g. they adapted to the Nintendo DS and the Nintendo Wii which offered new control schemes. So given the ability to simulate liquid interactions an innovative game developer might came up with new interesting gameplay ideas.

## 4.2. Summary

This thesis has shown that there are multiple ways to simulate liquid surfaces like procedurally generated surfaces based on simply evaluating functions at the points covered by said surface. Another approach is using particles. The movement of the particles based on some model could be simulated and from their positions a surface could be derived. One could even go so far and try simulating full fluid dynamics using linear equations systems. In this thesis yet another approach was used namely height fields. Heights were stored on a grid and their change over time was simulated. Each of these approaches has different advantages and disadvantages. The most realistic simulations based on fluid dynamics or a large number of particles are usually slow. When using the seemingly fast and simple procedurally generated liquids it is difficult to do liquid-object interactions. Height fields are fast, simple and allow to more or less easily include interactions but suffer from stability issues.

The height fields used in this thesis were based on the simple physical model of a membrane. This model was used to derive a surface simulation which can be implemented with just a few lines of code (without considering interactions). This simplicity is one of the reasons for the high performance. It also means that adding interactions with objects and geometry inside the "water tank" (blocked grid cells) did not make the simulation overly complicated and thus overly difficult to implement or slow to compute. Yet that simulation still suffers from limitations. Aside from the stability issues like limited wave speeds or numerical problems there are also physical phenomena which can not be simulated with the model used in this thesis. The simulation is basically limited to ponds, lakes or pools of relatively quiet liquid with only small waves.

Various hints have been given on how future works might make height fields more usable. It has been suggested to consider more neighboring cells when calculating new heights, use a better physical model, use specialized implementations for phenomena which otherwise could not be simulated and to use a simulation which adapts to the movement of the player. Also interactions between liquids have been proposed to expand the possibilities game developers have.

# Appendices

# A. Benchmark results

These are the benchmark results as noted in the chapter Evaluation. Details on how they were determined can be found in said chapter. The results are shown in two tables. The first table shows the results when using just one thread while the second table contains the results of using two threads. Each table shows the time needed for computing liquid-object interactions, the time needed for liquid surface simulation and the sum of both. These times only include the implementation of the extended pseudo-code listed in chapter 2. Time needed for other things like rendering is not included. The dependence of the computation times on the number of objects and the grid size is shown.

| **no. of objects** | **grid size** | **time for interactions** | **time for liquid simulation** | **total time** |
|---|---|---|---|---|
| 0 | 128x128 | 0.06 ms | 0.49 ms | 0.55 ms |
| | 256x256 | 0.21 ms | 2.39 ms | 2.60 ms |
| | 512x512 | 0.83 ms | 9.79 ms | 10.62 ms |
| | 1024x1024 | 4.00 ms | 39.48 ms | 43.48 ms |
| 30 | 128x128 | 0.10 ms | 0.91 ms | 1.01 ms |
| | 256x256 | 0.31 ms | 2.59 ms | 2.90 ms |
| | 512x512 | 1.22 ms | 10.06 ms | 11.28 ms |
| | 1024x1024 | 5.58 ms | 40.70 ms | 46.28 ms |
| 60 | 128x128 | 0.15 ms | 0.54 ms | 0.69 ms |
| | 256x256 | 0.42 ms | 2.65 ms | 3.07 ms |
| | 512x512 | 1.63 ms | 10.69 ms | 12.32 ms |
| | 1024x1024 | 7.04 ms | 42.30 ms | 49.34 ms |
| 90 | 128x128 | 0.21 ms | 0.63 ms | 0.84 ms |
| | 256x256 | 0.54 ms | 2.83 ms | 3.37 ms |
| | 512x512 | 2.03 ms | 10.65 ms | 12.68 ms |
| | 1024x1024 | 8.59 ms | 42.48 ms | 51.07 ms |

Table A.1.: Single-threaded benchmark results

| no. of objects | grid size | time for interactions | time for liquid simulation | total time |
|---|---|---|---|---|
| 0 | 128x128 | 0.06 ms | 0.35 ms | 0.41 ms |
| | 256x256 | 0.21 ms | 1.33 ms | 1.54 ms |
| | 512x512 | 0.86 ms | 5.67 ms | 6.53 ms |
| | 1024x1024 | 3.98 ms | 21.55 ms | 25.53 ms |
| 30 | 128x128 | 0.16 ms | 0.32 ms | 0.48 ms |
| | 256x256 | 0.32 ms | 1.51 ms | 1.83 ms |
| | 512x512 | 1.23 ms | 5.87 ms | 7.10 ms |
| | 1024x1024 | 5.58 ms | 22.98 ms | 28.56 ms |
| 60 | 128x128 | 0.14 ms | 0.34 ms | 0.48 ms |
| | 256x256 | 0.43 ms | 1.46 ms | 1.89 ms |
| | 512x512 | 1.66 ms | 5.82 ms | 7.48 ms |
| | 1024x1024 | 7.05 ms | 23.58 ms | 30.63 ms |
| 90 | 128x128 | 0.17 ms | 0.68 ms | 0.85 ms |
| | 256x256 | 0.55 ms | 1.45 ms | 2.00 ms |
| | 512x512 | 2.06 ms | 6.02 ms | 8.08 ms |
| | 1024x1024 | 8.63 ms | 23.42 ms | 32.05 ms |

Table A.2.: Multi-threaded benchmark results

# Bibliography


[1] V. Belyaev. Real-time simulation of water surface. In *International Conference Graphicon*, 2003.

[2] R. Bridson. *Fluid Simulation for Computer Graphics*. A K Peters, Ltd, 2008.

[3] C. Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 2004.

[4] Association for Computing Machinery: *Fluid Simulation*. WWW document, 09.08.2010. (URL: http://portal.acm.org/citation.cfm?id=1281681).

[5] D. Meschede. *Gerthsen Physik*, page 115. Springer, 22th edition, 2004.

[6] M. Müller-Fischer R. Bridson. Fluid simulation: Siggraph 2007 course notes. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, pages 1–81, New York, NY, USA, 2007. ACM.

[7] J. Stam. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*, March 2003.

[8] R. Rivera: *Boost C++ Libraries*. WWW document, 08.08.2010. (URL: http://www.boost.org/).

[9] NVIDIA Corporation: *GPU Gems - Chapter 34. GPU Flow-Control Idioms*. WWW document, 08.08.2010. (URL: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter34.html).

[10] NVIDIA Corporation: *GPU Gems 3 - Chapter 30. Real-Time Simulation and Rendering of 3D Fluids*. WWW document, 08.08.2010. (URL: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch30.html).

[11] S. Lantinga: *http://www.libsdl.org/projects/SDL_ttf/*. WWW document, 08.08.2010. (URL: http://www.libsdl.org/projects/SDL_ttf/).

[12] M. Müller-Fischer: *Invited Talks / Courses*. WWW document, 08.08.2010. (URL: http://matthiasmueller.info/talks/talks.htm).

[13] Autodesk Inc.: *Maya 3D Animation, Visual Effects, and Compositing Software - Autodesk*. WWW document, 08.08.2010. (URL: http://usa.autodesk.com/adsk/servlet/pc/index?siteID=123112&id=13577897).

[14] Khronos Group: *OpenGL - The Industry Standard for High Performance Graphics*. WWW document, 08.08.2010. (URL: http://www.opengl.org/).

[15] Intel Corporation: *Real-Time Deep Ocean Simulation on Multi-Threaded Architectures - Intel® Software Network*. WWW document, 08.08.2010. (URL: http://software.intel.com/en-us/articles/real-time-deep-ocean-simulation-on-multi-threaded-architectures/).

[16] Intel Corporation: *Real-Time Parametric Shallow Wave Simulation - Intel® Software Network*. WWW document, 08.08.2010. (URL: http://software.intel.com/en-us/articles/real-time-parametric-shallow-wave-simulation/).

[17] R. Bridson: *Robert Bridson*. WWW document, 08.08.2010. (URL: http://www.cs.ubc.ca/ rbridson/).

[18] S. Lantinga: *Simple DirectMedia Layer*. WWW document, 08.08.2010. (URL: http://www.libsdl.org/).

[19] R. Truelsen. *Real-time Shallow Water Simulation and Environment Mapping and Clouds*, 2007.