TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Timo Kellomäki

Large-Scale Water Simulation in Games

Julkaisu 1354 • Publication 1354

Tampere 2015

Tampereen teknillinen yliopisto. Julkaisu 1354
Tampere University of Technology. Publication 1354

Timo Kellomäki

Large-Scale Water Simulation in Games

Thesis for the degree of Doctor of Science in Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB109, at Tampere University of Technology, on the 4th of December 2015, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2015

ISBN 978-952-15-3643-4 (printed)
ISBN 978-952-15-3654-0 (PDF)
ISSN 1459-2045

Abstract

Water is an important element in the nature. It is also often encountered in digital games and other virtual environments, but unfortunately interaction with it is typically very limited. The main reason for this is probably the immense computational cost of simulating water behavior. Simulating water and other fluids by numerically solving *Navier-Stokes equations* is commonplace for offline engineering applications such as bridge building, weather prediction, or aeronautics. Since the 1990s, these methods have also been applied to computer graphics, but the focus has been in offline applications such as movie special effects. Recent advances in programmable graphics hardware have facilitated real-time fluid simulation in a large enough scale to be applicable in games. This far these methods have been mostly used in games only for visual purposes. This thesis is motivated by the wish to see more games where also the gameplay is affected by water simulation.

The first part of this thesis studies the roles of interactive water in different kinds of games. Requirements for water simulation methods are formulated by examining those roles. The thesis then introduces some background theory and various methods for water simulation. The focus is in heightfield-based methods, which simplify the problem by assuming that the water surface can be represented as a vertical displacement from a neutral level. This assumption allows very large amounts of water to be simulated with the very limited resources available for this purpose in a typical game. Most of these methods work on a heightfield terrain and can be enriched with fully 3D effects such as splashes and waterfalls by adding a particle simulation system.

An important problem is coupling the water simulation with existing rigid body simulations that are largely used for the dynamics of game objects. The coupling includes effects such as floating, objects moving with the flow, and building dams out of bodies. The thesis introduces a new heightfield-based coupling method, which allows the building of dams from rigid bodies in the heightfield context, unlike the previous approaches. The proposed methods, including the underlying water simulation method and visualization, were implemented in parallel using

graphics processing units. The methods were found to be fast enough to be applicable in games.

Finally, the most promising current simulation methods are compared from a games point-of-view using the criteria set in the beginning: performance, simplicity, visual quality, richness of behavior, and rigid body coupling. Since quality of experience is a subjective matter, user tests are recommended for comparison. Included in the thesis is one of the first such studies, which found out that leaving out the velocity self-advection step of a shallow water equation solver had no statistically significant effect on any of the measured psychological impacts. Based on the analysis, recommendations for the choice of simulation methods are given for different kinds of games.

Preface

In 2011 I contacted Colossal Order to discuss potential topics for my thesis. This lead to enlightening discussions with Antti Lehto and Damien Morello, which made me realize the untapped potential of water simulation in games and select this intriguing and complex topic. I have also lately been honored to join Colossal Order to work with these great people.

I would like to thank professor Tommi Mikkonen for his continuing support of my work as the head of department, my current supervisor, professor Timo Saari for his research co-operation and comments on the manuscript, and the pre-examiners, professor Perttu Hämäläinen and professor Nils Thürey, for their valuable feedback. I also appreciate the financial support received from both KAUTE and Ulla Tuominen foundation.

Working without a research group has been a hard path that I cannot recommend to anyone. Luckily I have had the benefit of two people who share a great ability to color the margins of any draft red with detailed and insightful comments. Firstly, I am most grateful of the untiring work by my original supervisor, professor Antti Valmari, to equip me with the analytical thinking tools and philosophy necessary in scientific work.

Secondly, no words are adequate to describe the significance of my beloved wife, Tiiti. Besides just providing invaluable emotional support and running the family while I concentrated on my work, she has also fulfilled the roles of a peer, advisor, and a proofreader. Finally, my dear children, Teemu and Kerttu, have shown me a deeper purpose in life that can make a PhD thesis feel trivial.

Timo Kellomäki
Tampere, October 14, 2015

Contents

List of Publications

[P1] T. Kellomäki.
Fast Water Simulation Methods for Games.
In *Computers in Entertainment*. Accepted for publication (acceptance in October 2014, status last updated November 9, 2015).

[P2] T. Kellomäki, and T. Saari.
A User Study: Is the Advection Step in Shallow Water Equations Really Necessary?
In *Eurographics short papers*, 2014.

[P3] T. Kellomäki.
Interaction with Dynamic Large Bodies in Efficient, Real-Time Water Simulation.
In *Journal of WSCG*, Vol. 21, No. 2, pp. 117–126. 2013.

[P4] T. Kellomäki.
Two-Way Rigid Body Coupling in Large-Scale Real-Time Water Simulation.
In *International Journal of Computer Games Technology*, 2014.

List of Figures

List of Tables

Author's Contribution to Publications

The author is the sole and independent contributor to [P1], [P3], and [P4].

For [P2], professor Saari was responsible for designing and implementing the psychological questionnaires. The analysis of variance was a collaboration of the author, professor Saari, and Katja Laine. The rest of the publication, including but not limited to, setting the research question, the regression analysis, interpreting the results, implementing the simulation software, administering the user tests, and reporting the results, is the work of the author.

List of Symbols and Abbreviations

1D, 2D, 3D	1-dimensional, 2-dimensional, 3-dimensional
$\partial f/\partial x$	partial derivative of f with respect to x
∇x	gradient of x
$\nabla \cdot \mathbf{x}$	divergence of x
$\mathbf{u} \cdot \nabla$	advection operator
$\mathrm{D}/\mathrm{D}t$	material derivative operator
$\mathrm{d}/\mathrm{d}t$	differential operator
Δt	time step of a numerical method, s
Δx	grid spacing of a numerical method, m
a	acceleration, $\mathrm{m/s^2}$
b	a heightfield terrain
d	water depth, m
F	force, N
g	acceleration due to gravity, approx. 9.81 $\mathrm{m/s^2}$
$\mathbf{g}$	acceleration vector due to gravity, $(0, 0, -g)^T$
h	a heightfield water surface
Kib, Mib, Gib	kibibit (2^{10} bits), mibibit (2^{20} bits), gibibit (2^{30} bits)
k, M, G	10^3, 10^6, and 10^9 times a quantity
m	mass, kg
$O(f(n))$	the set of functions that grow asymptotically at most as fast as a constant times $f(n)$

p	fluid pressure, $\mathrm{N/m^2}$
q	a generic quantity of interest
q^i	the value of quantity q at time i
q_i	the value of quantity q at grid point i
t	time, s
$\mathbf{u}$	velocity vector
u	velocity in the direction of the x axis
v	velocity in the direction of the y axis
$\mathbf{x}$	position vector
ν	fluid kinematic viscosity, $\mathrm{m^2/s}$
ρ	fluid density, $\mathrm{kg/m^3}$
ANOVA	analysis of variance
CFD	computational fluid dynamics, a branch of computational physics
CFL condition	Courant-Friedrichs-Lewy condition
FFT	fast Fourier transform, an algorithm to convert between discrete space and frequency domains
FLIP	fluid implicit particle, a fluid simulation method
FPS	first-person shooter, a game genre
fps	frames per second
GPU	graphics processing unit, a computer circuit for parallel processing
LBM	Lattice-Boltzmann method, a fluid simulation method
LOD	level of detail
NSE	Navier-Stokes equations
PDE	partial differential equation
PERF	primary egocentric reference frame
PIC/FLIP	particle-in-cell/fluid implicit particle, a variant of FLIP
SAM	self-assessment manikin, a pictorial assessment technique
SPH	smoothed-particle hydrodynamics
SSM	spatial situation model

Chapter 1

Introduction

1.1 Background

Water is one of the most interesting elements in nature. It is soothing and beautiful to look at, fun to play with, and even creates its share of catastrophes. It would seem, then, that it would also be an essential element in computer games and other virtual worlds that seek to model nature. Yet a look at the bulk of current game worlds reveals that in most cases, water only has the role of a beautiful decoration. There are, of course, games with ships or the possibility to swim, but the water almost always occupies a static area and has little to no meaningful gameplay associated with it. Of course, some kind of simulation would be needed to achieve interactivity.

Physical simulation is definitely not something foreign to games. The advent of physics engines has raised the bar on the interaction level in game worlds. Fully simulated rigid bodies are now trivially easy for the programmer to add in their game, and this has created an explosion of new game genres. What could be achieved if water simulation was as easy and commonplace?

Like all fluids, water motion is described by the Navier-Stokes equations, which are hard to solve or even numerically simulate. Especially turbulent flows are extremely chaotic, and capturing the whole richness of water behavior would need a very detailed simulation. Indeed, water simulation is notorious for being computationally very hard, and games rarely have any extra resources available. This is hardly surprising, since some 30–100 high fidelity images of an increasingly large and complex game world need to be produced per second. This far, in the

computer graphics field, water simulation has mostly been utilized in offline applications, such as movie special effects.

Luckily, in games we are usually not interested in the exact modeling of the complexities of the real world. It can be surprising how much the laws of physics can be simplified and even ignored to still create an illusion that the players fail to notice as unplausible, as long as the result is wrapped in a beautiful visualization. Knowing this, we are more prepared to solve the performance problem by simplifying the physics simulation considerably.

One of the most common tricks used in computer graphics is to replace the physically-based simulation with something more static that looks similar. For example, we constantly use photographic images as textures instead of actually simulating the materials on an atomic level. This non-physical approach has proved to work well for water modeling, because the details are fleeting, almost random, not important to interaction, and do not systematically affect the large-scale behavior of the body of water. Therefore we only need to coarsely simulate the large-scale flows, such as where a river flows, using, e.g., a simulation resolution of one meter. The visual richness of the smaller details can simply be an illusion that does not affect the large-scale simulation.

Unfortunately, it turns out that even a coarse simulation becomes a burden on the computational resources if large bodies of water are to be simulated. If we want to model one square kilometer of water with a vertical extent of one hundred meters even with a coarse resolution of one meter, a 100 million cells need to be updated dozens of times per second.

The essential optimization is based on noticing that, in games, what happens under the surface can often be ignored. In the big picture, only the height of the water surface is important. Restricting to a *heightfield* removes the vertical simulation dimension. While the simulation domain thus becomes two-dimensional (2D), the game world itself still stays three dimensional (3D). It is important to differentiate this from games with actual 2D worlds. Updating the one million remaining cells is well within reach of our current hardware, provided that each update is simple and can be implemented in parallel. Truly three-dimensional details, such as splashes, are lost, but they can be added back locally where needed.

The traditional solution in computer graphics to modeling this kind of large areas is to use different levels of detail (LOD), i.e., use a simpler simulation model, or a coarser resolution far away from the viewer. This has some problems, though. Coupling the different simulation resolutions is difficult, and especially so when trying to maintain the massive parallelism needed. Another complication is that if gameplay is to depend heavily on the simulation, the areas where the player is

not looking might still affect the game, and therefore need to be simulated at least in some detail. As a worst case, a player might even exploit the LOD system by altering water behaviour simply by looking at it! Some games might benefit from LOD techniques despite these issues, but this thesis concentrates on methods that do not use such an approach.

Another central topic besides performance is how the player is to interact with water. Because rigid bodies are currently the main way to physically interact with game worlds, it is essential to model the interaction between rigid bodies and water. The two simulations need to be coupled in two directions: the bodies float and are pushed by the water (water-to-body effects). On the other hand, water cannot flow through the bodies, which causes various body-to-water effects. Unfortunately, implementing this kind of coupling in real time turns out to be very difficult.

As will be seen in the next chapter, there already exist several water simulation games that take place in a 2D world. Almost all of these games use a particle-based approach, which is indeed well-suited for this problem in 2D worlds. The particle-based approach is also well-suited for 3D environments when small amounts of water suffice. However, the situation becomes more interesting when large 3D environments need to be modeled, since particle methods are not always computationally efficient enough. Therefore, this thesis concentrates on large 3D environments, where no single all-encompassing solution is yet apparent.

1.2 Research Questions and Methods

This thesis strives to lower what we find to be the most important barriers preventing water simulation from becoming commonplace in 3D computer games and virtual environments that model large areas of terrain: performance problems and implementation complexity. Another area of interest is finding ways to achieve richer interaction, where water is lifted from its usual role in games as a decoration to something that can actually be played with. Some of the reasoning behind these goals is presented in Chapter 2. The following main research questions of this thesis arise from this goal:

RQ1 Which of the available methods are best suited for large-scale 3D water simulation, taking into account the strict performance limits and limited development budgets of games?

RQ2 How can the methods be improved to ease the constraints, especially to allow richer interaction with water in games?

Our current best answer to RQ1 can be found in Section 6.7 of this thesis, and to RQ2 in [P4].

The design science paradigm seeks to create utility by building and evaluating artifacts, such as concepts, methods or software implementations. The created artifacts need to solve a relevant problem, be evaluated rigorously using metrics, and contribute to the knowledge base. This is in contrast to the behavioral science paradigm that tries to understand different phenomena by positing and validating theories. However, Hevner *et al.* suggest that these two approaches can and should complement each other in a research cycle. [23]

Most of this work falls under the design science paradigm. We analyze the potential usage of water simulation in games and use that to find relevant evaluation metrics. Simulation methods are then implemented, and evaluated according to these metrics.

However, we also argue that the behavioral science aspect is currently underrepresented in the computer graphics branch of water simulation research. The roots of water simulation in computer graphics are in the science of computational fluid dynamics (CFD), in which the end products typically solve a technological problem, such as building a dam that does not break. In contrast, the end products of computer graphics, be it movies, games, or other virtual worlds, are built for human users. Because of this, besides utilizing only the typical design science metrics, such as performance, we also need to understand how and why these technologies affect the users. The tools of behavioral science are well suited for this purpose. The user study included in this thesis, [P2], falls partially into this category.

1.3 Summary of Main Contributions

To answer RQ1, the current methods were first compared to each other in [P1]. The fastest available method, *pipe method*, seemed to suffer from a bad reputation, because the family of simulation methods it represents was often dismissed as not believable enough. To be able to include it in the comparison, we implemented the pipe method on a graphics processing unit (GPU). Evaluating the results turned out to be very promising: Most of what was lost in realism was gained in simplicity and speed, and the method was able to convincingly simulate flow over any heightfield terrain, which already enables many interesting gameplay features.

The results of [P1] and discussions with the game company *Colossal Order* led them to use a variant of the pipe method in their hit game *Cities: Skylines*.

However, because the believability of different methods is hard to evaluate objectively, user studies were called for. We implemented a small game that was used to conduct such a study in [P2]. No difference between a method very similar to the pipe method and a more complex method based on the shallow water equations was found in any of the psychological impacts that were measured.

When analyzing RQ2, we noted that gameplay is often based on how the player can affect the game world. Because much of the interaction in current games is done via manipulating objects represented as rigid bodies, it was natural to study how rigid bodies can affect water in heightfield methods. It turned out that all of the available coupling methods in the heightfield context concentrated on modeling small, floating bodies in a large and steady mass of water. They were based on letting water flow right through the bodies, which is not a problem in their context, but severely limits more general usage of these methods. The only object-to-water effects in these previous methods were some surface waves created for visual believability purposes.

A desire to richer interaction lead us to investigate the possibility of heightfield methods that truly block flow with the objects, as is typical in the offline full 3D simulations. Some simplifying assumptions allowed for a new method based on this approach. The method was originally implemented and evaluated in [P3], and further improved upon in [P4]. This method enriches the interaction possibilities in games by enabling rigid bodies to dynamically block the water flow in large-scale heightfield simulation.

To summarize, this thesis studied the requirements for large-scale water simulation in games, and then evaluated, compared, and improved the available methods in light of these criteria. Specifically, the main contributions of the publications comprising this thesis are:

- In *Water Simulation Methods for Games* [P1], a comparison of water simulation and water-object interaction methods suited for games was presented. This included a fast, GPU-parallel implementation of the pipe method, which was shown to be clearly faster than the competing methods. The water was also visualized interactively with small scale details created by texture advection. The work was originally published in 2012 in the *Academic Mindtrek* conference, but an extended journal version has been accepted for publication in *Computers in Entertainment*.

- *A User Study: Is the Advection Step in Shallow Water Equations Really Necessary?* [P2] used psychological measurements and statistical analysis to study the difference in user experience of two water simulation methods: the shallow water equations and the simpler wave equation. No statistically significant difference was found, suggesting that the simpler method may suffice for some applications. The work was published as a short paper in the *Eurographics* conference in 2014.

- *Interaction with Dynamic Large Bodies in Efficient, Real-Time Water Simulation* [P3] introduced a novel approach to rigid body coupling in the heightfield context. The method provides object-to-water coupling of a new kind, so that the rigid bodies can block flow. This enables richer interaction with water, such as building dams out of the rigid bodies. The work was published in *Journal of WSCG* in 2013.

- *Interaction with Rigid Bodies in Large-Scale Real-Time Water Simulation* [P4] further improved and generalized the method of [P3], e.g., enabling water-to-object coupling in the same framework, fixing some rendering problems, and making the method more physically-based and compatible with multi-layer models. The work was published in *International Journal of Computer Games Technology* in 2014.

- In addition to the publications, Chapter 2 of this thesis analyzes the potential usage of water simulation in games. Chapter 6 significantly updates and extends the comparison presented in [P1].

1.4 Structure of the Thesis

The rest of the thesis is structured as follows.

Chapter 2 discusses the role of water and water simulation in games in more detail. The role of water in games is analyzed, some psychological background is given, and the criteria for evaluating different methods are established.

Chapter 3 starts from the Navier–Stokes equations and introduces some simplifying assumptions to arrive in a form more feasible for real-time simulation. Numerical methods for implementing the actual simulation are then introduced. Finally, some of the most commonly used water simulation methods are shortly reviewed.

Chapter 4 describes in more detail some of the heightfield-based simulation methods that are central to this thesis: the shallow water equations, the pipe method,

and the wave equation. Our particular implementation of the pipe method and its visualization are also presented.

Chapter 5 addresses the problem of coupling rigid-body solvers with water simulation, for example, how to make bodies float on the water or to form dynamic obstacles that stop the flow.

Chapter 6 compares some of the simulation methods that are best suited for games, based on the criteria set in Chapter 2.

Finally, Chapter 7 summarizes and concludes the work, looking back at what was accomplished and laying out a roadmap for the future.

Chapter 2

Water in Games

2.1 Background

Most games take place in some game space, be it text-only, a visually extremely realistic 3D environment, or something in between [81]. Most of these worlds represent a virtual world of some kind, and reference the environment and culture that we as players live in [57]. Because water is such a common and interesting element in our environment, it is only natural for it to also exist in these game worlds. This chapter analyzes the roles of water in games both using the concepts of *flow* and *presence*, and by examining those roles in existing games. These roles are then used as a basis for establishing criteria for the simulation methods that could be used to implement water in games.

At least two distinct roles for water are easily identified: Making the environment seem real or believable, and providing an interactive element to enrich gameplay. These roles are not orthogonal, because it is a good practice to combine presentation with the game mechanics as closely as possible, but it will still be beneficial to discuss these topics separately.

Throughout the text, the terms *static* and *dynamic* are used to refer to two separate things: sometimes the whole water surface consists of a static geometry, e.g., a simple plane, and the appearance of water is created using various visual tricks that color the pixels in interesting ways. This is in contrast to dynamically moving the actual drawn triangles according to the water surface. The other meaning relates to the fact that in many current 3D games the surface triangles do move, but the area covered by the water cannot change during the game (or only changes

in predetermined ways, e.g., an event that always floods a certain part of a game level), in contrast to a mass of water that is able to freely run over the terrain. Some kind of water simulation is needed to achieve dynamic behavior in the latter sense.

2.2 Spatial Presence and Flow

To understand the role of water in creating believable virtual environments, the concept of *spatial presence* (or simply *presence*) is essential. Presence has been described as the feeling of "being there," or forgetting that one is only using media instead of having a real, physical experience [36]. Feeling presence is a human reaction to immersive media, but can also be felt using imagination [79]. Feeling presence has been suggested to contribute to high media enjoyment [37].

According to the theory of presence proposed by Wirth *et al.*, constructing a spatial situation model (SSM, a mental image of the perceived space) is an important precondition for presence. The SSM is constructed by observing spatial cues provided by the media, and is stronger when there many cues that are plausible and consistent with what the user expects based on their previous experiences. [79]

However, a strong SSM alone is not enough. In order for presence to occur, the user also has to accept the mediated space (e.g., a game world) as their primary egocentric reference frame (PERF). In other words, the media has to continuously compete with the real world about which space the user believes he or she is located in [79]. According to the theory of perceptual hypothesis, the user selects a hypothesis and looks for confirming cues. On the other hand, perception is biased by a strong hypothesis so that the user tends to see what is predicted by the hypothesis [8].

A plausible and rich SSM, therefore, gives rise to a stronger hypothesis, which helps the user's imagination to fill in the missing details that are expected to be in the scene. On the other hand, contradictory information that breaks expectations weakens this virtuous cycle. [79]

Interactivity strongly supports seeing the media as PERF [79]. Getting enough feedback and having a wide variety of possible interactive actions is important [65, 80]. Other factors that contribute to seeing the media as PERF are using multiple coherent modalities (such as audio cues synchronized to the visual cues) and persistence of the spatial structures [79]. All of these are quite typical to games.

Additionally, the user's motivation may help accepting a weaker SSM as PERF. This is helped by *involvement* and *suspension of disbelief*. Involvement happens when the media resonates with the user's values, experiences, and beliefs [33], but also when the user is provided with rich interactive capabilities for acting in the environment, for example, exploring [79]. Involvement helps presence in that it directs the user's concentration to the media instead of the cues from the real world [79]. Suspension of disbelief happens when the user wants to be entertained and therefore consciously or subconsciously ignores the conflicting cues, such as feeling the keyboard and mouse, or a plot hole in a movie [79].

Both involvement and suspension of disbelief are the more important the less immersive the media is. For example, books need to heavily draw on these factors, while a more immersive virtual reality system may achieve presence completely without them. [79]

Another important concept is *flow*. It is an experience acquired by deep concentration on a task that can be completed. Its prerequisites are clear tasks and goals, immediate feedback, controllable actions that affect the task, and full concentration on the task at hand. In a state of flow awareness of everyday life and self is weakened, and the sense of time is distorted. It is an immensely gratifying experience that is itself worth striving for even without other rewards. [13]

Much of the positive effect of presence on media enjoyment may be due to it helping achieve flow [45, 78]. Whether or not this is the case, it seems clear that encouraging both presence and flow are highly important for successful computer games [78, 79].

2.3 The Believability Aspect

Because water is so common in nature, with rivers, ditches, oceans, lakes, puddles, rain, etc., occurring everywhere in our everyday surroundings, we have strong expectations on how it behaves and also expect to see it everywhere in the game world. It is thus hardly possible to create a believable natural environment without water and other liquids. On the other hand, both the visual and dynamic nature of water include some mind-blowingly interesting, beautiful, and complex phenomena (Figure 2.1). It is to be expected that modeling and rendering water is essential for sustaining the believability in any virtual natural environment. In this light, the huge effort poured into into developing visually believable water modeling for games is not surprising.

Figure 2.1: Water exhibits complex behavior in the Tammerkoski rapids (photo: Visa580).

Water was already present in the visually simple 3D games of the early 1990s such as the *Doom* series (Figure 2.2) or *Ultima Underworld*. The surface was represented as a simple, static plane with some kind of a texture, often animated. Since these early days, rendering has progressed tremendously, and water surfaces are now often represented almost photorealistically with effects such as reflection, refraction, and even caustics taken into account. On the other hand, the static plane with no dynamic geometry whatsoever can still be found from many games. It is only very well decorated.

With the current, very advanced rendering, game worlds are starting to visually resemble the real world. However, as we have seen, presence is not a simple function of visually mimicking reality. Because water can be interacted with in many complex and interesting ways, creating the right feel when a game character or object meets water should also be essential for believability. For example, the WWW is full of videos of players walking in shallow water and shooting their weapons at it to compare the splashes created by the bullets. This is entirely in accordance with the theory of presence, which emphasizes the role of interaction and getting expected feedback in seeing the media as PERF.

Though much emphasis is still put on creating visually realistic water rendering, the importance of the quality of interaction has not been completely ignored, either. This is where many water simulation systems in current games come into play. For example, several modern games incorporate a method for creating visually believable ripples from user interaction. This is important, because it helps provide the visual cues players expect from interacting with water.

Figure 2.2: *Doom 2* (1994) used simple animated textures for water.

The techniques typically used in current games range from clever use of textures and normal mapping to FFT-based methods such as iWave [70]. These techniques have been used to enrich the look and feel of water and interaction with it in, just to name a few examples, *F.E.A.R.*, *Crysis*, *Resistance 2*, *Just Cause 2*, *HALO 3*, *Assassin's Creed 3* (Figure 2.3), and *Battlefield 4*. See [60] for some details on Assassin's Creed 3. Uncharted 3 (Figure 2.4) went a step further towards actual simulation by using wave particles [20, 82].

However, we argue that the bottleneck for immersiveness in current game worlds is typically not in the visuals, but in the overly simplistic interaction possibilities. Unlike in most other media, the player is allowed and expected to interact with the world, but instead of the endless possibilities of subtly affecting things as in the real world, the player is met with a mostly static environment with interaction possibilities that are often binary and scripted: pull this lever down, do the predetermined, always similar jump, or place a predesigned building here. This in stark contrast to, say, a movie, where every subtle interaction of the characters and the world can be preplanned.

A similar thought is brought forward by Salen and Zimmerman, when they explain that the player is often aware of the game, its rules, the interface, the social context, and several other aspects that are not authentical reproductions of reality. Even more, they argue that these artificial aspects actually provide important meaning to the play experience, and failing to recognize these aspects makes it difficult to create meaningful play. They call this overconcentration in creating a

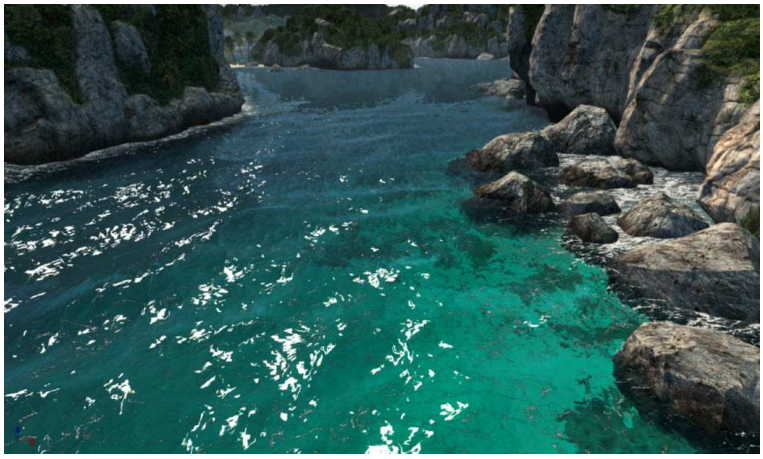

Figure 2.3: Visually realistic rendering in *Assassin's Creed 3* (2012).

Figure 2.4: *Uncharted 3* (2011) utilizes water simulation and advanced rendering for its water.

sensory illusion *the immersive fallacy* [57, pp. 450–455]. This argument can also be understood as stressing the meaning of involvement and suspension of disbelief.

A case in point is the hit game *Minecraft*, which employs pixelated and unrealistic graphics, yet succeeds in evoking a strong feeling of a living world by providing much richer interaction possibilities than most other games of its era. The success of Minecraft seems to have started a revolution, where games are made with the goal of enabling richer interaction than traditionally, instead of concentrating on mere visual believability.

The two goals of interactivity and visual realism have often been in contradiction, since static worlds are typically easier to make visually good-looking. For example,

each new state of an object often requires almost as much work from the 3D artist as making the original object. Another example is the possibility of precalculating extremely realistic lighting in a completely static environment.

Due to the reasons stated above, we find it extremely important to analyze the often neglected interactive aspects of water in games, and how they are connected to the water simulation technology.

2.4 The Gameplay Aspect

The specialty of games compared to other media is interaction. Sicart defines game mechanics as methods invoked by agents for interacting with the game world (agents could be either the human players, or something controlled by the computer) [61]. The mechanics are closely connected to game rules, which describe how the game moves from a state to another. One difference is that rules are neutral, because they are automatically evoked, while game mechanics are used to forward the intentions or agency of a player or another agent in the game [61].

A useful dimension in game analysis is the narrative versus system-based gameplay [29]. Games that are focused on the narrative (e.g., *The Walking Dead*, *Gone Home*) are closer to interactive movies, where the player only subtly interacts with a pre-written plot. Game mechanics are not very important. On the other hand, system-based games need the game mechanics to be rich enough to provide interaction that is interesting or entertaining for the player. This could be via challenges providing learning opportunities and thus the socially stirring possibility of proving these skills in a competition (e.g., *Counter Strike*). Another dimension is through creative self-expression, e.g., allowing the player to build something personal in the world (e.g., *Minecraft*).

Often, systems that can provide interesting gameplay opportunities are *emergent*, i.e., they are built from simple rules that interact and combine to form something endlessly complex and surprising [66]. Simplicity, or at least naturality or familiarity is important to make the game easily understandable to the audience [31]. This fits the flow theory well. A natural system helps make it clear for the user how the task works and what the possible actions and their results are. On the other hand, emergent, complicated systems provide ample opportunities for creating challenging tasks that can be solved with correspondingly high skills, another essential prerequisite of flow.

It should be noted that in games that are also virtual worlds, much of the gameplay is typically directly connected to the spatial features of the game world. For example, rivers are often efficient positions for defenders in strategy games, and effective tactics in first person shooter (FPS) games require flanking opponents using cover and alternative routes around the game world.

Unfortunately, there seems to be a persistent lack of such emergent and dynamic systems that function as a basis for interesting game mechanics. The few systems that have been invented are often repeated from game to game, only changing the theme and narrative. Such a phenomenon is, for example, clearly visible in the FPS genre, where the gameplay is based on a few recurring elements: moving an avatar in a 3D world, taking cover, and quickly aiming at the opponent.

In the last decade or so, physical simulation has proved itself to be an important source of game mechanics. It is also connected to the physical space. So far it has mostly taken the form of rigid body physics, the use of which has exploded with the availability of off-the-shelf physics engines. Additionally, examples of game mechanics based on more complex physical simulations have begun to appear, e.g., the mud simulation in the off-road game *Spintires*.

Even games where most of the mechanics are traditional have nowadays added physical simulations, because in addition to believability, simulation often provides new gameplay possibilities. For example, in the context of an FPS, a few simple crates lying around in the game world can be used as movable cover, makeshift stairs to reach a high ground, or even for some creative building fun without the game designer actually having to plan these mechanics. At best, this can be as simple for the developer as adding the rigid body property to the objects in the game engine. On the other end of the spectrum, many games are completely built on game mechanics based on physical simulation, e.g., *Angry Birds*.

Water fits well in the picture, because it demonstrates complex yet very familiar behavior, especially in interaction with moving bodies. It is clearly a very emergent element in nature, providing endless possibilities for play. Just imagine the real-life events of floating boats down creeks, letting water erode sand castles, surfing, or even whole cities being destroyed by floods. We believe that there is an enormous untapped pool of such gameplay possibilities emerging from even the simplest water simulation.

2.5 Water-Based Game Mechanics

Let us now analyze in more detail how water has already been employed in games to create game mechanics. We also give examples of how simulation can enrich those mechanics by making the water dynamic. These examples are then used as a basis for mapping needs for water simulation methods from a gameplay point of view.

A common usage of water is as an obstacle. In many, especially older, games the characters either cannot enter water at all, or are drowned upon entering. In many of these games, the role of water is purely thematic: gameplay-wise, the game could work just as well without any references to water. Any other barrier, such as walls, could be used for similar gameplay effects, but water is often the most believable, natural obstacle. For example, since the play area often needs to be limited, some kinds of borders are needed. A common solution is to make the levels islands, naturally limited by an unending ocean around them (see e.g., the *Grand Theft Auto* series). In some games, the roles of water and land are reversed: the player controls a boat moving on water, and the shores become natural obstacles.

Simulation could significantly enrich this usage, since it allows the player to manipulate the area occupied by the water to open up new routes, or block enemies from reaching an area.

Of course, in many games, it is possible to enter the water, which often interacts with the character. Usually at least the character's movement properties are changed: movement is slower, and buoyancy counters gravity. Slower movement might mean vulnerability due to inability to escape. Buoyancy opens up movement in the vertical direction more freely than mere jumping. This can be used to create game mechanics dealing with jumping above the surface. Some arcade-style games are based on balancing the different control schemes on both sides of the surface, e.g., game genre where the player is controlling a fish that is able to jump out of the water [18].

Additionally, a drowning rule is sometimes implemented, limiting the time the characters can spend underwater, which causes a potentially interesting challenge if climbing out of the water is difficult. Sometimes the characters can also hide by staying below the surface, but the hiding time is limited due to the drowning rule, increasing tension.

If water in the game level is static, these mechanics are limited to voluntarily going in to water, and pushing or luring other characters in. Simulation again opens up new possibilities, because the water level becomes dynamic and can even

Figure 2.5: *Sprinkle* has 2D water-based gameplay, where a water spray is used to extinguish fires.

be controlled by the player. Examples of such possibilities are drowning or slowing down enemies by unleashing a mass of water, controlling water to either make high locations reachable, protecting low locations from the water, or making water reach a specified area to trigger a positive effect.

A different kind of gameplay space is enabled by utilizing the flow of water. This includes waves, advecting properties such as color or temperature, and floating objects. Traditional childrens' play with bark boats is a good example. This last category practically requires some kind of a dynamic simulation to become interesting, since a static, precalculated flow would make the game rather repetitive.

An example of reaching new areas can be found from *Munin*, where the player is able to rotate parts of the level. This causes water to flow into new areas, and buoyancy can then be used to reach areas otherwise unreachable. The idea of protection is the basis of the game mechanic of the prototype game made for this thesis, which is used in [P2]. In the game, the player modifies terrain to save houses from drowning. Examples of directing water to a goal can be found in *Where's My Water*, or various fire extinguishing games, such as *Sprinkle* (Figure 2.5).

Two examples of games that utilize advection are *Liquidsketch*, where the player controls water flow in order to create target colors by mixing differently colored liquids, and the city builder *Cities: Skylines*, where flow simulation is used, e.g., to model where the sewage water of a city ends up (Figure 2.6). In *Cities: Skylines*, the player is also able to build dams for creating power, which affects the water

Figure 2.6: In *Cities: Skylines*, heightfield water simulation is used in a 3D world to see where the waste water ends up.

flow. Removing the dams afterwards can also cause interesting, dynamic flooding of the city.

The mechanics mentioned above can be further enriched if the simulation allows for more detailed interaction with objects (usually implemented as rigid bodies). This requires a versatile and robust coupling method between the water and the bodies. Both kinds of coupling can be used as game mechanics: examples of water-to-object coupling include capsizing ships, sailing, or the player pushing around objects with a controllable water spray. Examples of the object-to-water direction could include building or breaking dams. Combining these mechanics could be used to create game play systems that are varying and emergent. However, as will be discussed in Chapter 5, this kind of interaction is a technically challenging problem in large-scale 3D fluids. Our work in [P3, P4] aimed at finding solutions to this problem.

Since game mechanics are defined as methods of interacting with the game world, it is useful to think how the player actually interacts with water. In some cases, the player controls the water rather directly by inserting or removing it from some location. The water velocity could also be affected by the player. Examples include inserting water sources (either simple omnidirectional sources, or particle sprays that can be directed), or affecting flow either directly by, e.g., dragging gestures in the water, or in more creative ways, such as changing gravity.

In some games, the player can also control the terrain or objects that are then coupled to the water. This class includes the visual ripples caused by a moving character, but also richer interaction such as altering the terrain to push the water

around, driving a car into the water, or using several rigid bodies to build a dam. As is typical in games, the interaction does not have to obey any natural laws. For example, think about a magic spell that divides the sea.

Based on the above discussion, we propose a taxonomy for classifying the role of water in games. The first class is the **effect on gameplay**, and consists of the following subclasses:

G1 being an obstacle

G2 altered movement properties

G3 being directly useful at some location

G4 being harmful at some location

G5 pushing objects

G6 advecting properties (pollution, color, etc.)

The second class is **the control method**, with the following subclasses:

C1 no control at all

C2 adding at a location

C3 removing from a location

C4 applying forces or impulses

C5 altering terrain or objects

The no control subclass is included because interesting gameplay does not require the player to be able to control the water. Simply coping with the natural flow caused by gravity or other external factors is often enough.

For example, in the firefighting game *Sprinkle*, water is useful in certain locations, where it extinguishes fires (G3). The control method allows player to select at what height water is added and give an angle for the initial impulse for the inserted water (C2, C4). The challenge becomes interesting, because physical simulation is then used to resolve whether the water ends up in the correct location. The water could, e.g., hit an obstacle in the way, be bounced to an area that is otherwise unreachable, or first fill a cavity and then overflow to the desired location.

In *Munin*, puzzles are solved by using buoyancy to make the avatar reach new platforms, which we classify as altering movement properties (G2). The interesting part is the very unique control scheme, which is a combination of all of our subclasses C2–C5: rotating a part of the level moves water to another position, changes forces by affecting gravity, and alters the terrain as the whole structure of the level is changed (Figure 2.7). We would like to note that it is often clever new combinations like this that makes games interesting.

Figure 2.7: In *Munin*, parts of the game world can be rotated, which also affects water.

We conclude that of the gameplay effect subclasses, being an obstacle (G1) and altered movement properties (G2) do not require water simulation, but are clearly enriched by it. For G3–G6, simulation is somewhat essential, because they either require the area under water to change, or some notion of water flowing. Only pushing objects requires coupling with dynamic objects, and the water-to-body effects are enough. The control methods naturally require a simulation, because static water provides very little to control. Note that the altering terrain or objects category (C5) often requires a simulation method which is able to handle body-to-water effects. There are varying levels of interaction that can be achieved in this subclass, as will be discussed in Chapter 5.

2.6 Evaluation Criteria for Water Simulation Methods

The main purpose of this thesis has been to take technology forward so that gameplay-affecting, interactive water in games would become more commonplace. Some roadblocks preventing such games from being implemented were analyzed

Figure 2.8: Much of the gameplay in *From Dust* deals with simulated heightfield water in a 3D environment.

in [P1]. Based on issues identified in that paper and the above analysis on gameplay, we set our criteria for evaluating various water simulation methods in light of their suitability to games:

- performance
- simplicity
- visual quality
- richness of behavior
- coupling

Performance has traditionally been the main hurdle for water simulation. However, the explosive growth of available parallel processing power with the advent of the modern GPU is about to change things. While fully 3D water simulation is still far out of reach for large amounts of water, the heightfield methods discussed in Chapter 4 are already fast enough for many kinds of 3D games as exemplified by *From Dust* (Figure 2.8) and our research prototype game used in [P2].

Simplicity is a factor because it makes the technology easier to adopt, and creating tools faster and cheaper. It is often the case that a state-of-the-art method is so complex that only few people in the world have the knowledge to properly implement it despite publications dealing with the topic. In such cases, the availability of the method is then often dependent on single implementations that eventually become abandoned and finally obsolete as technologies change, with nobody left

with the expertise to update the implementation. An example of this is the once widely used, but now deprecated D3DX graphics programming library: implementations of some of the most complicated (and useful) features of it are still not publicly available years after its deprecation [76].

In general, using an opaque library implementation of a complex method is usually a less flexible approach than using such a simple approach that it can be implemented in-house. On the other hand, it should be noted that even a complex method can sometimes be successfully hidden behind a good interface.

Water simulation is not trivial to implement, not least because of the necessary parallelization. Many of the water simulation methods and the required GPU programming skills are too complicated to be learned by the busy game developers, at least in smaller game studios. A lot of resources were spent for developing the technology behind *From Dust*, instead of getting to focus on the content. As was the situation with physics engines, only the proliferation of off-the-shelf solutions made large-scale adoption of rigid body physics possible. Generic and easy-to-use water simulation methods and libraries are called for.

Quality of user experience is here divided into two classes based on the two roles of water in games: visual quality, and richness of behavior. As discussed above, a visually believable game world is essential for presence, but the end goal of games typically to try and provide an enjoyable experience.

However, media enjoyment is hard to measure objectively without user studies. Measuring valence, flow, or enjoyment directly from the users probably give much more useful information than, for example, calculating how closely a water simulation emulates some feature of reality. However, since user studies are somewhat difficult to conduct, richness of behavior can be used as a rough proxy for the immersion and thus presence. We can note classes of water behavior occurring in nature (e.g., waterfalls or wave dispersion) that can or cannot be modeled by the method in question.

However, since presence in games is also heavily affected by the interaction quality and involvement, it should not be the sole priority to strive for realistic and natural effects, if they do not contribute to the believability or some gameplay feature. Richness of behavior can thus also be characterized in terms of which gameplay elements the method allows. For this, our analysis above can be used.

For example, some popular methods, such as FFT or wave particles, can create very believable waves by displacing the water surface up and down, but do not allow the area occupied by water to change dynamically. This mostly rules out the mechanics of water being useful or harmful at some location (G3, G4) and severely

limits the usefulness of water as an obstacle or altering movement properties (G1, G2).

As seen in the analysis of water-induced gameplay, rich and robust coupling with the terrain and rigid bodies are essential for many new kinds of gameplay. This criterion can also be partly evaluated by listing types of coupling that can be achieved, such as interaction with a 3D or heightfield terrain, floating rigid bodies, rigid bodies blocking flow, or the ability to cope with difficult cases such as animated, soft, or thin objects. This topic is further discussed in Chapter 5.

Chapter 3

Water Simulation

In this chapter, we introduce the reader to the theoretical background of water simulation and some of the most common simulation methods. For a more comprehensive introduction to the topic, see Bridson's excellent book [6].

3.1 Navier-Stokes Equations

Fluid flow is often described using the *incompressible Navier-Stokes equations* (NSE), which are a set of partial differential equations:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} + \frac{1}{\rho} \nabla p = \mathbf{g} + \nu \nabla \cdot \nabla \mathbf{u}, \tag{3.1a}$$

$$\nabla \cdot \mathbf{u} = 0, \tag{3.1b}$$

where $\mathbf{u}$ is velocity, t is time, ρ is density, p is pressure, $\mathbf{g}$ is acceleration by gravity, and ν is viscosity.

A large variety of fluid phenomena can be modeled by solving a version of these equations. Examples include weather prediction, air flow around planes and cars, and water flow in a pipe or an ocean.

The NSE are notoriously ill-behaved. For example, a question regarding some basic properties of their solutions, the *Navier-Stokes existence and smoothness* problem, is one of the unsolved Millennium prize problems in mathematics [27]. Numerical

methods are needed to solve even simplified versions of the NSE, and are often plagued with stability problems. [6]

Turbulence is the small-scale, chaotic swirliness that is characteristic to fluid flow. Turbulence manifests itself especially in fluids with small viscosity and high speed flows. However, since turbulence is difficult to model numerically due to numerical damping [15], the methods in this thesis concentrate on modeling *laminar* (layered) flow, where turbulence is not present. Turbulent details can be added using non-simulated effects such as FFT [55] or noise [7], or special algorithms such as wavelet turbulence [32] or vorticity confinement [15]. However, turbulence is more important to the visual appearance of smoke than water. In large-scale real-time water simulation, most of the flow is laminar and many solvers ignore turbulence completely.

3.1.1 The Lagrangian and Eulerian Viewpoints

Let us think of measuring some property of a fluid in a continuum, for example, air temperature. The observer can either move with the fluid flow (a weather balloon), or be fixed to some location (a weather station). The former situation is the Lagrangian viewpoint, which typically leads to a particle-based discretization when modeling a fluid. The latter is the Eulerian viewpoint, which is connected to using a grid-based discretization.

The advantage of the Lagrangian approach is that the velocity field of the fluid moves both the quantities of interest and the observer in the same way. This leads to a simulation, where we follow a certain number of small particles. The forces affecting each particle are evaluated to get the acceleration of the particle. Quantities of interest, such as density or temperature, are stored in the particles and are thus automatically moved, or advected, by them. These quantities can then be evaluated in any point of the continuum, e.g., as a weighted average of the corresponding properties of the particles.

In contrast, in the Eulerian approach the simplest numerical solution is a regular grid. In this framework, the fluid properties change in time because of two reasons: the properties themselves can be changing (such as the whole fluid cooling off), but also because of advection, i.e., the fact that the fluid itself moves along the velocity field of the fluid. This means that in the next instant of time, different portions of fluid with possibly different properties appear at the location of the observer. For example, imagine a river with first warm water and then cold water flowing past. Even if the water itself did not change its temperature, the water would seem to get colder for a stationary observer in the river.

Even though the advection step is an additional burden compared to the Lagrangian methods, the regularity of the grid has some numerical benefits, such as the relative easiness of approximating spatial derivatives [6, p. 7]. Also, Eulerian methods can often employ longer time steps than what is possible in the Lagrangian approach [P1]. Intuitively, the particles can get arbitrarily close to each other, which causes a large pressure force pushing them apart. The force should quickly diminish as the particles get pushed apart, but a large time step combined with a simple time integration scheme causes the large force to be applied for too long. For a discretization based on a regular grid this obviously cannot happen, because the forces are always calculated from quantities evaluated at the same distance, Δx, from each other.

To understand how these approaches are related to the NSE, imagine a fluid that consists of particles with positions $\mathbf{x}(t)$, velocities $\mathbf{u}(t) = \mathrm{d}\mathbf{x}/\mathrm{d}t$ and some property q (e.g., temperature). Let us consider how q changes in time for a particle that is currently at some position $\mathbf{x}$. Since q is a function of both time and position, and position is a function of time, we can use the chain rule to obtain [6, p. 8]:

$$\frac{\mathrm{d}}{\mathrm{d}t}q(t, \mathbf{x}(t)) = \frac{\partial q}{\partial t} + \mathbf{u} \cdot \nabla q \equiv \frac{\mathrm{D}q}{\mathrm{D}t}. \tag{3.2}$$

The D notation is called the *material derivative*. Its first component term is $\partial q/\partial t$, the measure of how the quantity q itself changes in time with all else equal, and the second term $\mathbf{u} \cdot \nabla q$ takes into account how q changes because the fluid is flowing past the point (i.e., the fluid is advected). The advection term is exactly what is automatically kept track of in the Lagrangian approach, because the observation point (the particle) is also advected along the velocity field, but the Eulerian methods need to employ a specific step to account for it. [6, pp. 4–8]

We can now use the material derivative to form more concise equations without explicitly repeating the advection term. This kind of equations are often called *advection equations*. For example, setting the material derivative to zero gives the simplest possible advection equation, which describes a situation where the property is not changing by itself, but is only being advected by the flow [6, p. 8]:

$$\frac{Dq}{Dt} = 0 \Leftrightarrow \frac{\partial q}{\partial t} = -\mathbf{u} \cdot \nabla q. \tag{3.3}$$

Equation 3.3 should further help us see that if we wish to model the fluid from the Eulerian viewpoint, just to reach the neutral situation with only advection and no other changes, we need to evaluate the advection term. The research question

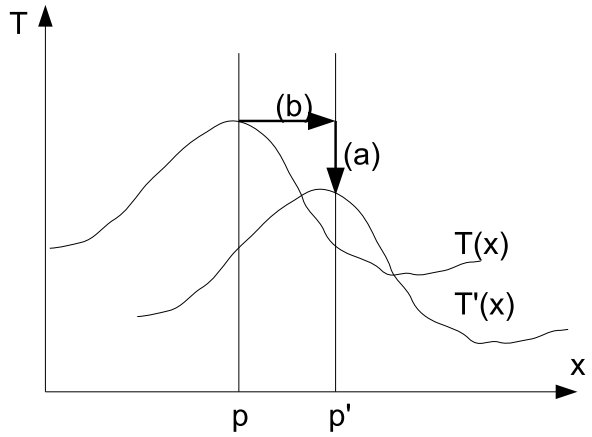

Figure 3.1: A 1D example of the material derivative in Lagrangian and Eulerian contexts. Both methods need to account for the change in general fluid temperature (a). The advection part (b) is handled by the particle itself moving in the Lagrangian viewpoint, but needs to be specifically accounted for in the Eulerian viewpoint.

of [P2] was whether we can drop the advection term from the velocity equation, and still reach a believable simulation in the context of shallow water equations (discussed in Section 4.2), and the result of subjects not noticing this simplification should be rather surprising in light of the current discussion.

To further clarify the issue, consider the example in Figure 3.1. In it, we consider how the temperature $T(x)$ in a 1D fluid changes during some time interval. The fluid itself is cooling uniformly (a) and also moving with uniform velocity so that it moves the distance $p' - p$ to the right in a single time step (b). The new temperature is $T'(x)$, which is the original temperature shifted right and lowered. Now, if we follow a particle starting from p, the particle itself is moved, using its own velocity, to p' during the time step. All we need to do is lower the particle temperature. If we instead use an Eulerian viewpoint, to find out the new fluid temperature at the grid point p', we need to combine the effect of the shift because of the velocity (b), and the effect of the cooling (a). These two terms correspond to the two terms in the material derivative.

3.1.2 The Momentum Equation

We are now in position to understand the first equation of the NSE (3.1a), often called the momentum equation. It describes how the fluid velocity **u** evolves over

time. Consider a small particle of fluid, which has mass m, and velocity $\mathbf{u}$. Its behavior is governed by the forces F acting on it according to Newton's law

$$a = \frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} = F/m, \tag{3.4}$$

where the material derivative is used because the velocity itself is advected by the velocity field, just as all other fluid quantities (i.e., this is an advection equation).

Most of the components of the momentum equation 3.1a represent different accelerations caused by forces affecting our water particle: gravity g, pressure difference $-\frac{1}{\rho}\nabla p$, and viscosity $\nu\nabla \cdot \nabla\mathbf{u}$. However, the viscosity term is often dropped in computer graphics, because the numerical methods used to solve the equations create more than enough viscosity in the scales we are interested in [6, p. 13]. If extra viscosity is needed, real-time methods typically resort to simply scaling the velocity on each time step by a time step dependent constant $\mu < 1$ (e.g., [42,P3]).

Substituting the remaining accelerations into Newton's equation 3.4, we arrive at a simplified version of the momentum equation, which should be compared to the originally presented version (3.1a):

$$\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} + \frac{1}{\rho}\nabla p = \mathbf{g}. \tag{3.5}$$

The only difference besides the now dropped viscosity term is that Equation 3.1a has the material derivative written out. This creates the additional advection term $\mathbf{u} \cdot \nabla\mathbf{u}$. Notice that the term contains $\mathbf{u}$ twice, since the velocity field is advected by the velocity field itself (i.e., the generic quantity q in the above discussion on advection is replaced by $\mathbf{u}$, because the velocity of the fluid moves with fluid itself).

3.1.3 The Incompressibility Equation

Equation 3.1b is called the *incompressibility equation*. In reality, fluids are not incompressible, but close enough that we treat them as such in computer graphics and games. The equation can be understood intuitively by noticing that if the divergence of the velocity is zero, water is flowing out as fast as it is flowing in from other directions.

If a vector field is incompressible, it is called *divergence-free*. One of the most complicated steps of the traditional fluid solvers is to ensure that the velocity field

stays divergence-free. The discretization typically leads to a sparse linear system that needs to be solved. However, as we shall see in the next section, the heightfield methods are able to skip this step. Since this thesis almost exclusively deals with heightfield methods, the incompressibility equation is not discussed further. For a more detailed derivation of the equation and its relation to pressure, see [6, pp. 10–16].

3.2 Numerical simulation

To simulate water using a computer, numerical simulation methods are applied. The topic is vast, but we will go through some of the basics here.

3.2.1 Discretization

The equations need to be discretized in both time and space. To solve the value of $q(t)$ in a differential equation of the form $\partial q/\partial t = f(q)$, the equation obviously needs to be integrated over time. In most numerical methods, this is approximated by finding the value of $q(t)$ at a finite number of time instants $q^1, q^2, \ldots, q^n$, with Δt between each instant, denoted by the superscripts. There exist a plethora of more and less accurate methods for achieving this, but for the most of this thesis, we use the *forward Euler* method, which simply evaluates the next value as

$$q^{i+1} = q^i + \Delta t f(q^i). \tag{3.6}$$

This is only accurate if $f(q)$ stays constant for the duration of the time step. Many more sophisticated time integration methods exist, such as the Runge-Kutta family [9]. However, these are typically much more costly to evaluate than forward Euler, and games often do not benefit from the additional accuracy, since they do not typically aim for a realistic simulation. It is possible, though, for a more sophisticated integration method to permit using a longer time step, which might more than make up for the slower evaluation.

To solve a differential equation of the form $\partial q/\partial t = f(q) + g(q)$, an approximate solution is often found using *splitting*: q^i is first updated with the effect of $f(q^i)$ to get an intermediary value q^{i+}, and then $g(q^{i+})$ is used to calculate the effect of g to finally obtain q^{i+1} [6, pp. 17–20].

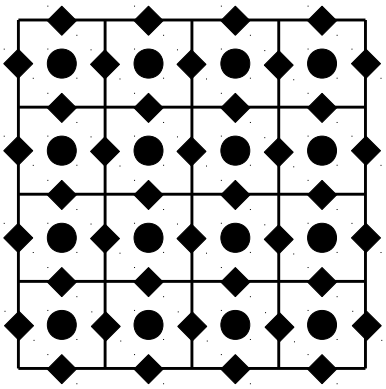

Figure 3.2: A staggered 2D grid similar to what was used in our water simulation implementations in [P2, P4]. Some quantities are stored at the centers of the square cells marked by the circles (typically pressure in traditional methods and water depth in heightfield methods), while others are stored in the cell faces marked by the rhombi (typically one component of velocity per face).

The spatial discretization is typically done using a grid in the Eulerian methods and by using some finite number of particles in the Lagrangian methods. The grids used in real-time applications are almost always regular but come in many variations.

When a quantity q is stored in a grid, the values are defined only in grid points q_i. The necessary differential operators also need to be replaced by their discrete versions. As an example in 1D, the gradient of q at grid point i can be estimated by the *central difference* $(q_{i+1} - q_{i-1})/(2\Delta x)$, where Δx is the grid spacing.

Notice that the central difference gradient operator ignores the value of q_i, which causes problematic behavior in water simulation. As an example, take a pressure field such as $p_i = 1 + (-1)^i$: there are obvious pressure differences that should cause forces toward the zero-valued cells, but the previous operator estimates the gradient to be zero everywhere, and no forces are created. In practice, this would mean that pressure differences between neighbors quickly grow very large and the whole system will start oscillating. On the other hand, using the left or right difference is obviously biased.

A common solution is to use a staggered grid (see Figure 3.2), where different variables are stored in different locations [22]. This brings some numerical advantages. For example, if water pressure is stored at the center of each cell and a single component of the velocity at each face (i.e., u is stored halfway between two pressure values that are adjacent in the x-axis), the ∇p that is needed in velocity updates can be evaluated using the values of p in the surrounding two cell centers:

$\nabla p_{i+\frac{1}{2}} = (p_{i+1} - p_i)/\Delta x$ (and similarly for each direction in a 2D or 3D situation). This avoids the problems described in the previous paragraph. Because the gradient is always calculated between two adjacent values, any difference between the values is guaranteed to affect the gradient. The downside is that interpolation is needed to evaluate velocity at grid points.

3.2.2 Stability

Numerical integration methods provide only approximations of the true solutions for the corresponding PDEs. It is usually the case that when $\Delta t \to 0$, the numerical solution converges to the actual solution, but as we use larger and larger timesteps, many methods become *unstable*, i.e., start to exponentially deviate from the true solution. Being true to reality is not a requirement in games as such, since we are not really interested in the actual solution, because any believable result will suffice. However, unstable simulations often gather huge amounts of energy and blow up, which is naturally critically important to be avoided if gameplay is to depend on the simulation.

An important feature of the simplistic forward Euler method used is that it is an *explicit* method, meaning that q^{i+1} is only a function of the state at the previous time instant, q^i. In contrast, *implicit* methods need to solve an equation that includes the future state. They are usually more stable, but also more expensive to calculate.

The most common tool for analyzing the stability of an explicit numerical fluid solver is the CFL condition (after Courant, Friedrichs, and Lewy) [12]. The idea is that the solution of a partial differential equation at $(\mathbf{x}, t)$ depends on the initial conditions inside some part of the whole domain, the so-called *domain of dependence*. This domain of dependence typically gets larger as t grows. Many grid-based explicit numerical methods are based on finite differences, where information is only transmitted between neighboring cells on each time step. If information in the actual solution moves faster than this, it is often not theoretically possible for the whole domain of dependence to affect the numerical solution. The CFL condition quantifies this at the limit of $\Delta x \to 0$ and $\Delta t \to 0$. [6, pp. 33–35]

Using the same idea, a necessary (but not sufficient) stability condition for the convergence of most explicit time integration methods based on finite differences can be formed:

$$\frac{\Delta x}{\Delta t} \geq |\mathbf{u}|, \tag{3.7}$$

where $|\mathbf{u}|$ is the fluid speed, which equals the speed of information in the true solution [6, p. 34].

As we can see, the stability condition sets an upper bound for the time step, given Δx and $|\mathbf{u}|$. Smaller time steps allow for larger velocities without becoming unstable, but using too small a time step wastes computational resources. On the other hand, the smaller the simulation scale, the shorter time steps are needed.

Using the forward time discretization and the central difference in space together is often called the *FTCS scheme* (for Forward-Time, Central-Space). It works sufficiently well for solving many problems, such as the heat equation. Unfortunately, it turns out that for some problems, including the advection equation and the wave equation, the FTCS scheme requires an even stricter limit on the time step to be stable. As the grid spacing $\Delta x \to 0$, the time step needs to be reduced $O(\Delta x^2)$ [35]. Since this would usually require too small time steps to be practical, it is typical to include some artificial viscosity by multiplying the velocities after each time step by some value $\mu(\Delta t) < 1$. This could easily, however, cause the water to look too viscous. Another simple option is to give up the FTCS scheme and use, e.g., the more stable first-order upwind scheme described in Section 4.3.

3.2.3 Numerical Simulation on the GPU

GPUs were developed for the massively parallel task of transforming the vertex points of 3D models in space, and later for evaluating lighting models for individual pixels. Current GPUs execute up to thousands of threads in parallel, enabling several orders of magnitude more floating point operations per second compared to CPUs. The downside is that many of the threads are limited to run a single program on multiple data. Additionally, execution control and synchronization is somewhat limited, and more importantly, quickly eradicate the advantages of parallelism. For example, it is typical that both branches of an if clause get evaluated for all threads running the same program. These features necessitate a completely different way of thinking compared to traditional algorithmics.

The programmability of the modern GPUs has enabled their use in various tasks other than the original purpose of rendering computer graphics. This field is called *general-purpose computing on graphics processing units*, or GPGPU for short. GPUs are useful when solving any problem that has a very large number of operations that can be executed independently of each other. The various

GPU programming interfaces allow short programs, so-called kernels, to be run in parallel for a specified size of 2- or 3-dimensional arrays.

The grid-based numerical water simulation is almost a perfect match for the GPU hardware. For example, the author's first naïve implementation of the pipe method was sped up some 20 times even on an old GPU with 96 parallel threads, when compared to a single-core CPU implementation.

Unfortunately, there are still several barriers to GPU programming. Development tools are not as advanced as the tools for CPU programming, and efficient use of the GPU requires using low-level programming languages and understanding complex concepts, which necessarily has an adverse effect on productivity [41]. Hardware differences are not always transparently handled, and the execution model makes implementing many non-trivial algorithms and data structures much more complicated. Additionally, in modern games the GPU is often overworked, while game developers struggle to employ all of the CPU cores. Despite these factors, in such a performance-critical field as water simulation, the performance gains are too large to be ignored.

One way to implement the basic Eulerian water simulation approach on the GPU is to use two grid-sized arrays per quantity of interest. One copy is used as a read-only input to the computation, while the results are written to the other copy. Each thread processes a single simulation cell, and is able to read the previous state freely, but only allowed to write to the single position corresponding to its cell. After each step, the role of the buffers is reversed. This common technique is called *double buffering*. It ensures that the order of processing does not affect the results, enabling a fully parallel implementation with the only synchronization happening between algorithm phases. GPUs also have a so-called *texture memory*, which is optimized to work quickly when used in this way instead of enabling random write access.

Naturally, this approach limits what the programs can do, since they cannot write to random memory positions. However, surprisingly many problems can be mapped to this approach using some creativity. All methods implemented in this thesis use this double-buffering model of execution, including the relatively complicated coupling methods of [P3,P4]. This has benefited their execution speed.

Implementing the pipe method on the GPU is further discussed in Section 4.6.

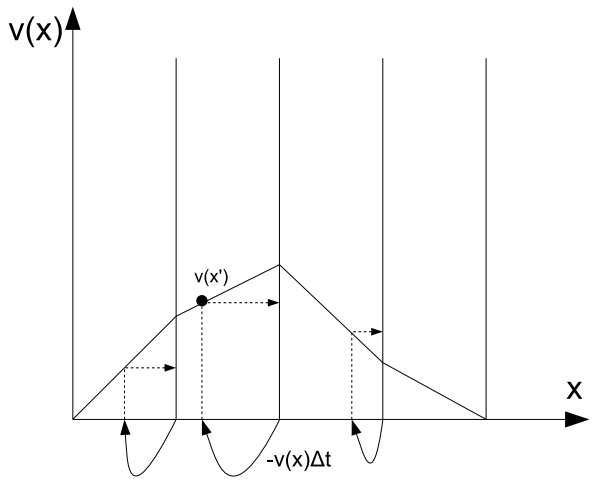

Figure 3.3: A 1D example of advecting the velocity field using the semi-Lagrangian method. For each grid point x, the new velocity $v(x, t + \Delta t) = v(x', t)$, where $x' = x - v(x)\Delta t$. x' is an estimate of the position from which a particle would end up at x during a single time step.

3.3 Fluid Solvers

This section introduces some methods that have been used to model water behavior, focusing in methods that are applicable in real-time.

3.3.1 The Eulerian Approach

The traditional Eulerian fluid solvers are based directly on solving the NSE in three dimensions, typically using a staggered grid. The NSE are split into three parts: advection ($Dq/Dt = 0$), forces ($\partial u/\partial t = g$), and incompressibility, which finds a pressure distribution so that the incompressibility equation is satisfied [6, pp. 19–20].

The advection step could be solved using the forward Euler method, but as already mentioned, it is very unstable for this problem. The standard procedure instead is the *semi-Lagrangian method* introduced to computer graphics in 1999 by Stam [63]. It is based on a backward trace of the quantity. We are trying to find $q^{t+\Delta t}(\mathbf{x})$, where $\mathbf{x}$ is some grid point. Let us consider the position $\mathbf{x}' = \mathbf{x} - \mathbf{u}^t(\mathbf{x})\Delta t$, which is a linear estimate of the position from where a hypothetical particle would need to begin at t to end up exactly at $\mathbf{x}$ at $t + \Delta t$. Notice that we only use the known

value of $\mathbf{u}^t$ at the grid point $\mathbf{x}$. Now we simply let $q^{t+\Delta t}(\mathbf{x}) \coloneqq q^t(\mathbf{x}')$. Since $\mathbf{x}'$ is typically not a grid point, the value needs to be interpolated from the surrounding values (which turns out to be an essentially free operation when using GPU texture memory). An example of advecting the velocity field itself in 1D can be found in Figure 3.3.

The semi-Lagrangian method is unconditionally stable. In light of the CFL condition, this is intuitive, because for large velocities, information might be fetched from several grid cells away if needed. The downside is that the necessary interpolation can be thought of as a smoothing filter, which means that some small details are lost, creating a less lively look. The loss of energy is also visible in the simplified example of Figure 3.3. This phenomenon is a prime example of numerical methods causing unwanted dissipation. There are also more advanced methods, such as the MacCormack method and its variants, to combat the deficiencies of this simple approach [59].

The incompressibility, or *pressure projection*, step is typically the most complicated and time-consuming step in these solvers. In it, a sparse linear system needs to be solved. Luckily, this step is avoided by the heightfield methods that are the focus of this thesis, and is thus out of scope in this short introduction. Instead, the heightfield methods typically directly include a force caused by the pressure gradient ∇p. This force can be integrated together with gravity using forward Euler, as will be shown in Chapter 4.

Unfortunately, the 3-dimensional Eulerian fluid solvers are typically too slow for real-time applications with a large grid such that we would need in a game [P1]. To model an area with a given spatial resolution, n cells are needed in each dimension, giving $O(n^3)$ cells in total. Despite this, impressive small-scale effects can be achieved with these methods also in real-time, such as the Dragon demo of NVIDIA's Flameworks [47].

There are several approaches to make the Eulerian solvers faster. The grid need not be fixed and equispatial. Some research has been done on moving grids [54], making cells larger or smaller based on the situation. A promising example of the latter method is the *tall-cell method*. In it, the water surface is modeled in full 3D, but the volume underneath the surface is represented by a single tall cell per cell column [10, 28].

For water and other fluids with a free surface, an additional step is needed in the full 3D context to find the surface. This can also be somewhat time-consuming. The standard approach keeps track of a *level set* [6, pp. 85–94], but since this is another issue avoided by the heightfield methods, it is not discussed further here.

3.3.2 SPH and Other Lagrangian Methods

Smoothed-particle hydrodynamics (SPH) is currently one of the most common of the particle-based methods [26]. The basis is a Lagrangian simulation, where the advection term is automatically handled by the movement of the particles. All that is left is to integrate the various forces affecting the particles to update their velocity, and then to update the position based on the velocity. The forces typically consist of pressure, gravity, and possibly viscosity.

The problem is that to find out the necessary forces, we need to evaluate the necessary fluid quantities and their derivatives, such as pressure gradient, at the particle locations. SPH is a method used for interpolating these quantities as a distance-weighted sum from the values stored in the surrounding particles. The weighting function (*kernel*) resembles a gaussian, but is usually selected to have a compact support to avoid having to iterate through all other particles.

Since a very large number of particles is needed to get a good result with the method, it is necessary to use a spatial acceleration data structure to search for the particles that reside inside the kernel's support. Uniform grids are currently among the most commonly used [26]. This approach makes the method practically run in linear time in the number of particles. However, $O(n^3)$ particles are required to get a similar resolution as in a grid with n cells per dimension. Even with acceleration structures, the neighborhood search can be time-consuming, and balances the comparative advantage SPH gets over Eulerian methods from the Lagrangian handling of the advection term.

The resulting surface is not trivial to construct and render. For real-time methods, a screen-space approach is commonly used (e.g., [1]). To avoid single particles from being distinguishable, various smoothing methods have been proposed [26].

There is a vast literature of variants and extensions of SPH that include, e.g., GPU implementations, different methods for enforcing incompressibility and handling boundaries, combining particles with varying sizes or time steps, and multiphase fluids. However, most of the methods are not designed for a real-time context. For a recent survey on SPH in computer graphics, see [26].

The position-based method is an interesting recent development [38]. It is a Lagrangian method, but instead of working on particle velocities, the particle positions are directly manipulated. Much longer time steps can be achieved using this method instead of SPH [39]. As this method matures, it will probably become very competitive when simulating relatively small amounts of water.

SPH has also been applied to simulating only the water surface, i.e., a heightfield [62]. This method is naturally much faster than a full 3D solution, since much fewer particles are needed for large masses of water. It also retains the good rigid body coupling properties of SPH (see Chapter 5). However, it is not currently competitive with the performance of grid-based heightfield methods due to very short time steps, as will be seen in Chapter 6. If a position-based version of this approach could be devised, the result might be very interesting for large-scale simulations.

3.3.3 Other Real-Time Fluid Simulation Approaches

The Eulerian and Lagrangian viewpoints can also be combined. The typical approach is to simulate the bulk of the water with an Eulerian solver and add visual small-scale details using a particle system. This was already done by O'Brien and Hodgins [49], who added a simple particle system on top of their heightfield simulation.

Another family of methods that combines the two viewpoints is FLIP (*fluid implicit particle*) [5] and its variants, such as PIC/FLIP [84]. These methods can be seen as particle-based methods that utilize a grid for some phases of the solving. These methods are very popular in the visual effects industry, but not as commonly used in the real-time context.

Yuksel *et al.* [82] introduced wave particles, which are also a Eulerian/Lagrangian hybrid method. The particles represent disturbances in the surface, and the heightfield is interpolated from nearby particles. One of the strengths of this method is the interaction with floating objects. New particles are created by collisions, which enables the modeling of ship wakes and waves reflecting from various objects in the water. However, this method still assumes an open-water scene where the surface is at a fixed level, which limits its use in richer scenes such as rivers or dam breaks.

A large literature has also tackled the problem of representing (deep) ocean waves. Instead of the physically-based approach, the methods used are often based on empirical data of ocean behavior. These methods focus on modeling the periodic behavior of open-sea scenes. Parametric methods model the water surface as a sum of evolving periodic functions, while spectral methods model the ocean in the frequency domain, using an inverse fast Fourier transform (FFT) to construct the surface [69]. These methods can provide visually excellent results compared to the computational resources invested, but typically cannot model interaction with a terrain. They have already been used in several commercial games for open sea scenes. For a review on these methods among others, see [14].

It is also possible to use the parametric or spectral methods to create small-scale visual details on top of an Eulerian simulation [10]. An even cheaper method for creating similar details is to use normal mapping with precalculated textures that are advected along the flow [75]. The latter is the approach adopted in the visualizations of the results of this thesis.

Another approach is the *Lattice-Boltzmann method* (LBM), which is based on statistical physics. The method is applicable both in 3D and 2D, and is computationally relatively efficient, simple, and able to handle boundary conditions well. Its shortcomings include the strict limit on the time step to make the method stable, and large memory consumption [50]. The method is commonly used in computational fluid dynamics (see, e.g., [74, 83] for references), but also occasionally in computer graphics [73]. Recently, 2D versions of the method have also been used for shallow water simulations on arbitrary terrain [51]. Ojeda's thesis provides a more detailed treatment of the method with a real-time focus similar to ours [50].

Chapter 4

Heightfield water simulation

4.1 Heightfields

This chapter discusses heightfields, which are a common way of modeling large-scale water areas in a 3D world. Throughout the chapter, we assume a right-handed coordinate system with the horizontal axes x and y and the vertical axis z pointing upwards.

For large-scale water simulations in real-time, full 3D simulations are often too slow. A common solution is to assume that the water forms a heightfield, i.e., the surface z-coordinate is a function $h(x, y)$ with water below and air above the surface. This allows the simulation to take place on a 2D grid. If n cells are required per dimension, the time complexity is typically lowered from $O(n^3)$ to $O(n^2)$. The regular 2D grid can be imagined as a group of adjacent water columns, where the water is defined by an $n \times m$ 2D array of depth values. An $n \times m$ heightfield is often visualized by creating $(n-1) \times (m-1)$ squares that are rendered as two triangles each. Figure 4.1 depicts a 3×3 heightfield drawn as 8 triangles.

The downside of this approach is that many interesting phenomena are lost. There can be no strictly vertical waterfalls, splashes, air bubbles, or breaking waves. However, some of these effects only occur in limited areas in the simulation and can be added back using, e.g., a local small-scale particle simulation or cheap procedural methods such as animated textures, while the bulk of the water is simulated using 2D methods [10].

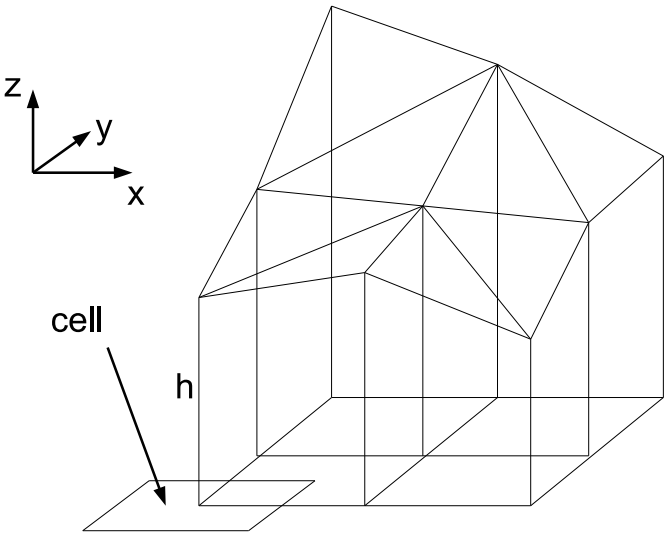

Figure 4.1: A 3×3 heightfield with one of the nine simulation cells drawn. The centers of the cells are displaced in the vertical direction by the values of the heightfield h, and connected to form triangles for visualization.

Some games only need oceans or other static water areas. In these cases, the problem is to create believable waves, especially in interaction with floating objects such as ships. It is usually enough to use heightfield-based deep water models, which assume no terrain exists below the water. Many actual water simulation implementations in current games are based on such models (e.g., [20, 60]). The FFT [69] and wave particle [82] methods introduced in the previous chapter are examples of this approach.

On the other hand, in many 3D computer games, terrain is also represented as a heightfield. It is therefore interesting to solve a different problem, where the water flows on top of the terrain and interacts with it. Here we focus on this case. We assume that the water flows on top of an arbitrary heightfield terrain $b(x, y)$, allowing the water to freely spread to dry areas. In these cases, the water can also be represented by the depth $d(x, y) \geq 0$, with surface $h = b + d$. Most simulation methods for this setup use the depth instead of the surface as the main simulation variable.

4.2 Shallow Water Equations

A very common approximation used to create a heightfield simulation on top of a heightfield terrain is to ignore the vertical velocity of the water, resulting in

shallow water equations (SWE). The horizontal velocities u and v (corresponding to the x and y directions) represent the averages in the whole column of water.

Another important assumption of the SWE is that the vertical movement of the water is dominated by gravity and pressure, which is an agreeable approximation if the water is only moving very slowly. Taking the vertical component of the momentum equation 3.1a, dropping all other terms except the pressure and gravity, and integrating from some coordinate z to the surface h results in the hydrostatic equation [6, p. 171]

$$p(z) = \rho g(h - z). \tag{4.1}$$

Notice that in this model, there is no more any need (or even possibility) to store or solve pressure, because it is fully determined by the depth. With these assumptions, the shallow water equations can be derived from the NSE:

$$\frac{\partial d}{\partial t} + \mathbf{u} \cdot \nabla d = -d(\nabla \cdot \mathbf{u}), \tag{4.2}$$

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -g \nabla h. \tag{4.3}$$

The proof is left out for brevity, but can be found, e.g., in [6, pp. 170–173]. A *conservation law form* of Equation 4.2 can also be obtained by applying simple vector arithmetic (notice that d is a function of position and time):

$$\frac{\partial d}{\partial t} = -\nabla \cdot (d\mathbf{u}). \tag{4.4}$$

This form is more useful for numerical simulation, because it is easier to discretize in such a way that keeps the water volume constant [10].

Equation 4.2 describes the rate of change of the fluid depth. As usual, the depth is advected by the term $\mathbf{u} \cdot \nabla d$. On the right side, the negative divergence of the velocity represents the volume density of the inwards flux to the point, which is multiplied by the depth, which represents the vertical direction that has been integrated out.

Equation 4.3 describes how the velocity changes, and also has two components: the velocity self-advection term on the left, and the term describing acceleration

caused by the pressure difference on the right. Because the hydrostatic pressure profile was assumed, this acceleration is simply the acceleration by gravity times the surface gradient.

According to linear wave theory, the propagation speed of gravity waves depends both on the water depth and wavelength. Long waves on deep water propagate faster than short waves on shallow water. This dependency on wavelength is called *dispersion*, which is an essential feature contributing to the look of an ocean.

Dispersion is not modeled correctly by the shallow water equations, which therefore are often claimed not to be well suited for modeling deep oceans. Bridson writes that even if the physics is not consciously understood by the viewer, the water will look shallow. However, it turns out that when the water actually is shallow, the dependency on wavelength becomes insignificant, and this simplification is less of a problem. [6, pp. 186–187]

4.3 Discretizing the SWE

The SWE can be solved using numerous different approaches. Here we briefly describe an explicit Eulerian approach using a staggered grid as described in Section 3.2. This presentation roughly follows Chentanez *et al.* [10], and the approach was also adopted for [P2].

First, Equation 4.4 is advanced for a step (depth integration), and then equation 4.3 is split into two steps: velocity advection and velocity integration. These steps are repeated in this order after each other. Each of these phases can be implemented as a single GPU kernel. The velocity equation is updated component-wise, with the updates of u independent of v, and vice versa.

The height equation 4.4 can be discretized in space to get

$$\frac{\partial d_{i,j}}{\partial t} = -\frac{(\bar{d}u)_{i+\frac{1}{2},j} - (\bar{d}u)_{i-\frac{1}{2},j} + (\bar{d}v)_{i,j+\frac{1}{2}} - (\bar{d}v)_{i,j-\frac{1}{2}}}{\Delta x}, \tag{4.5}$$

where the subscripts denote the grid location where a variable is evaluated, $\bar{d}$ is the depth evaluated at the *upwind* direction, i.e., $\bar{d}_{i+\frac{1}{2},j} = d_{i,j}$ if $u_{i+\frac{1}{2},j} > 0$ and $d_{i+1,j}$ otherwise, and similarly for the y direction. This upwind discretization is more stable than a symmetrical scheme [10]. The usual forward Euler integration can now be used to advance the equation forward in time.

To intuitively understand equation 4.5 and the upwind scheme, consider that the flow between two adjacent cells is coming from the cell in the upwind direction. The volume of water incoming per second is equal to $\Delta x \bar{du}$. The rate of change of depth in a cell is then equal to the sum of these volumes through all four cell faces divided by the cell area Δx^2, which is exactly equation 4.5. Other upsides of this formulation include that it is guaranteed to conserve volume unlike some other schemes. It is also easy to see that the scheme automatically keeps the depth non-negative as long as $max(|u|, |v|) \leq \Delta x / \Delta t$. This condition should already be familiar to the reader, because it is a version of the stability condition in Equation 3.7 implied by the CFL condition.

The velocity advection is typically solved using the stable semi-Lagrangian method of Stam [63]. The method is practically identical to the 3D version, as described in Section 3.3.1. More sophisticated methods also exist, for example the MacCormack method and its modifications [10,59], but the basic version is relatively simple and fast. Its downside is the excessive smoothing, resulting in a loss of detail.

The velocity update is straightforwardly discretized as

$$u_{i+\frac{1}{2},j} \mathrel{+}= -\frac{g}{\Delta x}(h_{i+1,j} - h_{i,j})\Delta t, \tag{4.6}$$

and correspondingly for v.

The border conditions at the edges of the simulation area also need to be taken into account. Reflecting borders are easy to implement by simply setting the velocity to zero at the border. This can also be used to reflect from obstacles that are above the water surface. As in all numerical solvers, a completely absorbing border is very difficult to achieve. Chentanez *et al.* use the *perfectly matched layers* method, which is based on damping the waves approaching the borders [10]. Such an approach is especially important when modeling an open water scene.

On the other hand, in the author's experience with simulations on a complex terrain, it is usually sufficient to simply set the water depth at the border to zero in order to absorb most of the water. This causes some reflections, but it is hardly noticeable in a gaming environment.

4.4 The Wave Equation

A further simplification of the SWE is to drop the advection terms completely, as was done already by Kass and Miller [30]. Equations 4.2 and 4.3 become

$$\frac{\partial d}{\partial t} = -d(\nabla \cdot \mathbf{u}), \tag{4.7}$$

$$\frac{\partial \mathbf{v}}{\partial t} = -g\nabla h. \tag{4.8}$$

Deriving equation 4.7 assumes that the depth varies only slowly [30]. This is reasonable in some scenarios. There seems to be no justification for dropping the velocity advection term to reach Eq. 4.8, except that the result has been found to work well in practice. As shown in, e.g., [30], these linearized shallow water equations are actually equivalent to just the 2D wave equation

$$\frac{\partial^2 h}{\partial t} = gd\nabla \cdot \nabla h, \tag{4.9}$$

where $\nabla \cdot \nabla$ is the 2D Laplacian operator.

Bridson points out that the wave equation has solutions that move with the speed $\sqrt{gd}$. Because waves thus move faster in deeper water, and water is typically deeper at the top of the wave than at the front, the top tends to catch the front. This eventually causes the wave to break. [6, p. 176]

Even though the catching-up behavior is included in the solver, the breaking of course cannot be modeled by the heightfield approach. Breaking waves were added to a shallow water simulation using a particle system in [10].

If D is the maximum depth value in our simulation, the previous wave speed can be plugged into the stability condition of Equation 3.7 to get a more precise stability condition for the time step of this method [6, p. 177]:

$$\Delta t \leq \frac{\Delta x}{\sqrt{gD}}. \tag{4.10}$$

It implies that the stable time step actually depends on the maximum water depth in the simulation.

We would like to note that the conservation law form of the depth equation (4.4) is already as simple to discretize as the linearized version of Equation 4.7, as explained in section 4.3. It is thus also possible to use an alternative method where only the velocity self-advection term is dropped, which should work better than the fully linearized version in situations where the depth changes rapidly.

Figure 4.2: An example of the difference between linearized (left) and full SWE (right). The whirlpool is not created in the linearized version. From [P2].

Dropping the velocity advection step removes some interesting phenomena, such as whirlpools. An example of such a difference can be seen in Figure 4.2.

Several versions of the linearized SWE method were implemented by the author. Most of them used the conservation law form of the depth equation (4.4), and only needed to drop the velocity advection term. These implementations were used for the water simulation in [P2] and in experiments related to [P3].

The main research question of [P2] was whether the difference affects the user experience. No statistically significant difference was found in any of the categories measured (valence, flow, presence, realism) [P2].

The findings of [P2] may be due to the difference being altogether indiscernible, the users being indifferent to the changes, or the statistical power of our test being too small to notice any effect. Our methods cannot differentiate between the two first cases, but if either of them is true, it is reasonable to say that the advection step had no significant effect on the quality of user experience. However, there is more to be said about the third case than what fit in the short paper, so let us supplement the publication here. First of all, box plots of the average effects of advection to different impacts per user are presented in Figures 4.3 and 4.4 for the games and the videos, respectively. The data is approximately normally distributed, though there are a few more outliers than expected. Removing them did not affect the conclusions.

Additionally, we ran some simple simulations to further estimate the statistical power of our tests. Our target was to find the minimum effect size needed for the rate of false negatives to be under 0.2 for each impact, which gives a 1:4 ratio between type I (false positive) and type II (false negative) errors. For this purpose, we repeatedly drew simulated data with our sample size from a multivariate normal distribution with the parameters estimated from our data, but artificially inserting

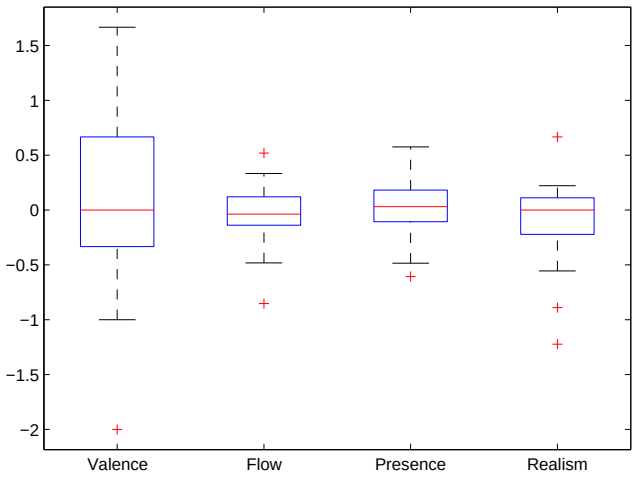

Figure 4.3: Distributions of advection effect, averaged over 3 levels, on measured impacts per subject in the games of [P2]. Advection effect is the difference of the measurement with and without advection. The scale is units of the impact scale (1 to 9 for valence, 1 to 7 in the others). The whiskers are drawn at the last non-outlier data point. Outliers are more than 1.5 box widths away from the box (25th to 75th percentile), which translates to approximately 0.7% of normally distributed data points.

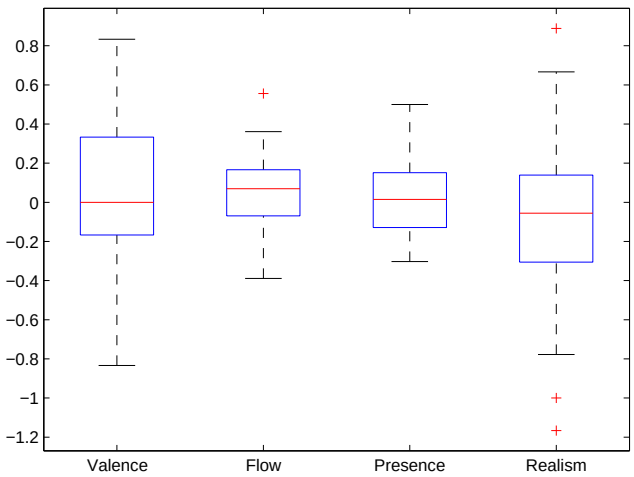

Figure 4.4: The distributions of advection effect, averaged over the six videos used in [P2] (see Fig. 4.3).

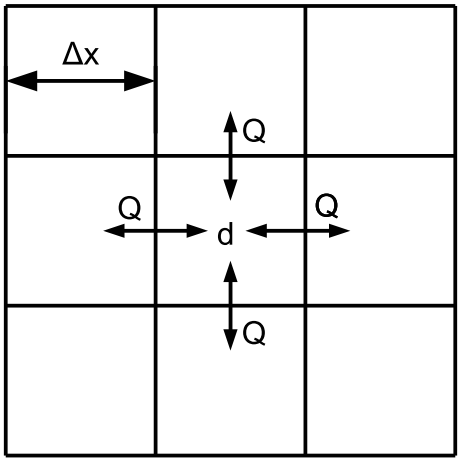

Figure 4.5: The variables of the pipe method in a Δx-spaced grid of 3×3 simulation cells. The depths are stored at cell centers and the flows at the faces between them. In our model, each (non-border) cell is connected to four bidirectional pipes.

effects of various sizes. The same ANOVA tests as in the study were then conducted for the simulated data and the rate of false negatives counted (10 000 repetitions per run). The simulations were repeated until the search resulted in the desired false negative ratio.

The least powerful of the tests was the one for valence in the games, where an effect of about 0.3 points on the 9-step SAM scale was required before the desired false negative rate was reached. The other required effect sizes were estimated to be between 0.1 and 0.2 points of the respective scales of each impact. This is admittedly not a full-blown, careful power analysis, and it is dangerous to read too much into such numbers, because no objective meaning can be assigned to a unit of valence or other psychological impacts based on self-reported data. Despite that, it should at least give some approximate information on the size of the effect that could have gone unnoticed in the study.

4.5 The Pipe Method

A different approach that, however, leads to a method similar to the wave equation, is to think of the heightfield water as a group of adjacent water columns, and start from the hydrostatic equation (4.1). In the original formulation due to O'Brien and Hodgins [49], the columns were thought to be connected by virtual pipes,

which leads to the name *pipe method*. There are several variations of the method, and we again briefly describe one of the approaches.

In the model, each pair of neighbors is connected by a pipe. The original formulation by O'Brien and Hodgins also had two unidirectional pipes between each pair of neighbors, but it is enough to use a single bidirectional pipe instead. The *von Neumann* neighborhood (the four neighbors sharing a common edge with the cell) is usually enough, but some authors also used the *Moore* neighborhood (which also includes the four cornering cells).

The structure is a staggered grid, but instead of storing velocity in between the cells, as in previously introduced methods, the *flow* Q (cubic meters of water per second) is stored. This turns out to make the implementation of our coupling algorithm simpler [P4]. Each pipe is thought to span the interval of the two cell centers, thus having a length of Δx.

Each flow Q is updated by the following formula, derived by O'Brien and Hodgins using the hydrostatic pressure and Newton's law [49]:

$$Q \mathrel{+}= A\frac{g}{\Delta x}\Delta h \Delta t, \tag{4.11}$$

where A is the cross-section of the pipe, and Δh is the surface height difference between the two columns connected by the pipe, e.g., $\Delta h_{x+\frac{1}{2},y} = h_{x+1,y} - h_{x,y}$.

Because the depth must not become negative, the pipe method now employs a limiting step, where the total outflow of water during a time step is calculated. If this would exceed the depth in the cell, all outflows are scaled before the depth update so that the remaining depth will become exactly zero.

Additionally, the flows Q are optionally scaled by some time-step dependent multiplier $\mu(\Delta t) < 1$ to improve stability. This issue was already discussed in more depth in Section 3.2.2.

The water depth is then updated by summing the total flow in all four pipes connected to each cell, dividing by the horizontal area of the cell, $(\Delta x)^2$, to convert volume change into depth change, and applying Euler integration:

$$d \mathrel{+}= -\Delta t\frac{\sum Q}{(\Delta x)^2}, \tag{4.12}$$

where the sum is over all four pipes connected to the cell, such that the outgoing flows have a positive sign.

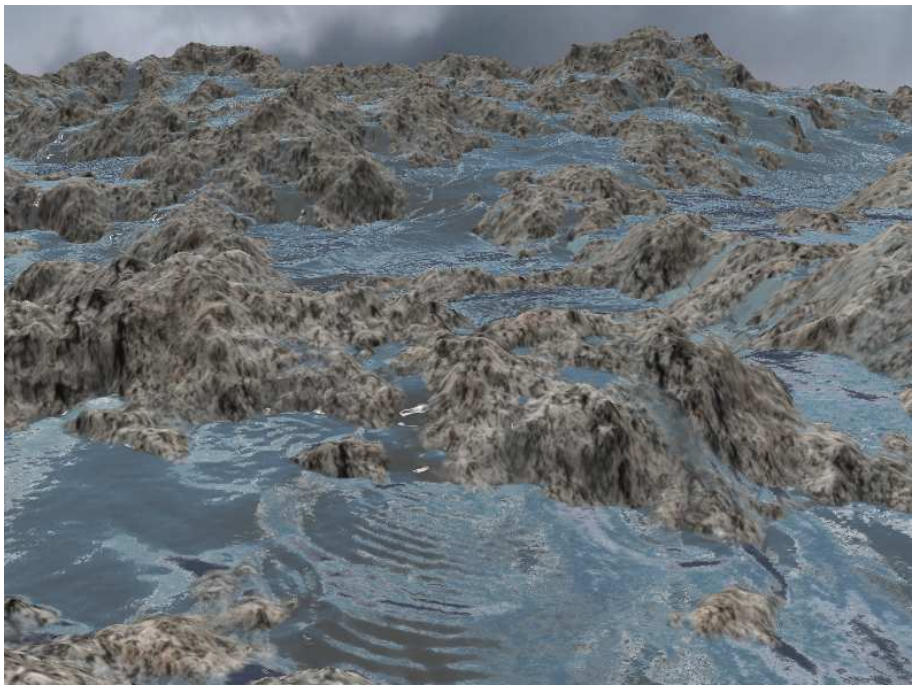

Figure 4.6: An example scene simulated using the pipe method on a 1024×1024 grid. From [P1].

Many implementations do not mention whether they use a constant A or let it vary based on the common interface of the two water columns. We calculate the cross-section as $A = \bar{d}\Delta x$, where $\bar{d}$ is the upwind depth as in Equation 4.5. We have also found out that artificially varying A leads to an approximate method for modeling viscosity (larger values make the water more lively). Exposing this parameter to creative game developers could provide opportunities for some fascinating non-realistic gameplay.

Another useful addition here is to limit the spreading of water uphill by taking the geometry of the situation into account. The basic idea is to only allow flow when the water surface is above the terrain at the dry target cell. There is really no one correct way to do this. The terrain can be considered to be formed from column-like steps, or the average of the two surrounding terrain height and surface values can be used, or the variables can be evaluated at upwind/downwind cells. This blocking can also be done for the velocity step, for the depth step, or both. Combined with the other parameters of the method the user has a lot of freedom to tweak how spreading wavefronts will look like. One example of taking this into account in the SWE context can be found from [10]. Also, our blocking methods in [P3, P4] are based on a generalization of this idea.

It is often claimed that the pipe method is equal to solving the wave equation, and thus only has a vertical velocity and is only suited for modeling local oscillations of the surface and not the global flow of a river or advecting floating objects. Because of this, some authors recommend to use SWE instead. See e.g., [10,34,62]. However, the pipe method stores the flow, which serves a very similar purpose as the velocity in SWE, to store the momentum gathered in previous simulation steps (similarly, some of the wave equation solvers use the two last heights instead, but the flow can be calculated from those). We have not noticed a significant visual difference between the pipe method and the linearized SWE, which was in turn found to have no statistically significant difference for various measures of user experience, when compared to the full SWE in [P2].

Indeed, it is trivial to calculate the velocity field from the flow. Since Q represents the volume of water that passes through a pipe per second, and the pipe cross-section is $\bar{d}\Delta x$, the velocity $u = Q/(\bar{d}\Delta x)$ (u is used for velocity here, though the pipe might as well be aligned to the y-axis). If the velocity at a grid point is required, it can be interpolated from the velocities at the pipe locations (which is a very cheap operation for GPU texture memory). This calculation was used by [42] for erosion simulation. We have also often advected floating bodies using variants of the pipe method with a quality that should be quite satisfying for most games. See, e.g., the floating objects in [P4].

The pipe method has been extended and applied in various ways. Mould and Yang generalized the model to multiple layers to combat vertical isotropy [43]. A single layer is usually sufficient, but this idea was used in [P3,P4], which used a two-layer pipe method to let water flow both over and under a body in the water. Holmberg and Wünsche also use the column-based approach and aim to model turbulent water with an added particle system [24]. The advent of programmable GPUs made the pipe method somewhat popular, because very large areas could now be simulated in real time using a GPU implementation [40, 42, P1]. The method was also extended for modeling hydraulic terrain erosion [42, 64]. A variant of the model was also used by Colossal Order for their game *Cities: Skylines*. An example scene simulated on a heightfield terrain using the pipe method can be seen in Figure 4.6.

4.6 Pipe Method on the GPU

As an example of the double buffering based GPU implementation described in Section 3.2.3, let us consider our implementation of the pipe method.

Let the grid be $n \times m$. We allocate two $n \times m \times 4$ arrays of 32-bit floating point type in the texture memory. The four values per cell are terrain height, water depth, and a flow value for both of the two dimensions. Depending on the hardware, these can be a single 4-channel texture or several textures with fewer channels, whichever is more efficient. The indexing is staggered so that position (i, j) contains the terrain height and depth of the cell (i, j) and the flows in the pipes leading to $(i + 1, j)$ and $(i, j + 1)$. Thus accessing the four pipes connected to a cell requires access to information stored in the cell itself and its top and left neighbors. We use 32-bit floating point numbers, because in our experience, 16-bit (half precision) values are not accurate enough, and using double precision is unnecessary.

Now the following processing kernels are executed repeatedly:

1. Flow update: read terrain, depth, and old flow. Calculate the new flow according to Eq. 4.11 and apply the limiting described below the equation.
2. Depth update: read the new flow and old depth, write the new depth according to Eq. 4.12, adding any water from sources (e.g., rain).

In each cell, the limiting step in the flow update kernel requires all four updated flows from the surrounding cells. To avoid adding another kernel, we need to recalculate the flow values in the "up" and "left" pipes of each cell even though the new flows are only written for the "down" and "right" pipes. This does not guarantee that velocities are only limited when necessary, but the method is conservative, i.e., it errs on the side of limiting the velocities too much in order to never produce a negative depth.

After each kernel, the role of the buffers is changed. Since the update rules only affect a single channel, if a single 4-channel texture is used, the kernels need to copy the previous values of the channels that are not updated, after which the roles of the buffers are switched. If single-channeled textures are used, it is enough to only switch the roles of the two copies of the updated buffer. This procedure is completely parallel with no need for synchronization inside a kernel. Each cycle takes the simulation forward one time step. The method naturally produces two most recent depth and velocity values to the GPU memory, from which the visualization can interpolate to produce a smooth animation of, e.g., 60 Hz.

Since each kernel consists of only few calculations, our implementations of these methods have always been limited by the memory bandwidth instead of processing. To get a rough idea, let us briefly analyze the worst-case texture reads per cell, assuming that each value is stored in a four-channel 32-bit texture. In real life, these calculations are naturally complicated by cache arrangements, memory

transaction granularity, and many other implementation details. For this calculation we assume the 128 bit memory transaction granularity that we had in our test hardware. This coincides with the cell data structure size.

The flow shader updates two pipes leading from (i, j) to $(i + 1, j)$ and $(i, j + 1)$, but due to the limitation step described above, it needs information on the whole neighborhood of five cells (640 bits, or five memory reads). The depth update writes the new depth at (i, j), and thus needs information on the cell itself and the flow in the four pipes connected to it. Two of those pipes are stored in the same 128 bits as the depth information itself, so it only needs to read the data of cells $(i - 1, j)$ and $i, j - 1)$ (384 bits). Since the time step is typically 25 ms, using, for example, a 1024×1024 grid of data, this translates to 40 Gib/s of memory traffic.

As the information is used in 3–5 cells per pass in a quite regular and localized pattern, in practice some of the bandwidth could be saved by caching. Actual measurements show the global memory bandwidth usage of our method to be about 28 Gib/s for this grid size with the NVIDIA Quadro 1000M GPU. For comparison, the specifications of a high-end desktop GPU, NVIDIA GTX Titan, list its memory bandwidth at 2149 Gib/s [48].

The coupling methods described in [P3, P4] have also been designed to follow this same execution model, with additional kernels for rigid body coupling added in the mix. The coupling methods were designed to minimize the number of kernels and the amount of data that needs to be read for each update, and the phases were planned to avoid random write access. Precise timings and some other details of our implementations are presented in the publications [P1–P4].

4.7 Visualizing Heightfield Water

Finally, for completeness' sake, we shortly describe the main features of the water visualization that we used, though it it is not the focus of this thesis. The necessary computer graphics background has been left out of the thesis, but any standard textbook on the topic should suffice to understand the following.

In our implementation, the terrain and possible rigid bodies are drawn first. The water surface heightfield $h(x, y)$ is then drawn by creating a single triangle mesh as in Figure 4.1, i.e., an $n \times m$ simulation grid translates to $2(n-1)(m-1)$ triangles. Since all calculations are done on the GPU, the heightfield data is readily available with. We also want to decouple the simulation step from the rendering frequency, so the drawn heightfield is interpolated from the last two simulation results to

create a smooth 60 Hz animation even though the simulation frequency is typically much lower.

Triangle normals are needed for lighting. The normals are calculated per pixel by evaluating $h(x, y)$ at the four points $(x \pm \Delta x, y \pm \Delta y)$ using the hardware interpolation available for texture memory. The normal is then perturbed according to two normal map textures with alternating weights. The textures are stretched by the flow velocity and reset when their weight is zero, exactly as described in [75].

The lighting is based on dividing the light to reflected and refracted portions according to Schlick's approximation [58]. The refracted portion is not actually refracted, but instead the color of the background (terrain or a rigid body previously rendered) is mixed with a blue-grayish water color. The mixing weights and transparency are based on the distance to the background, which is calculated from the depth buffer. The reflected portion uses the standard cube map reflection from a skybox.

The two most important further enhancement would probably be adding foam textures and using tessellation hardware to control the number of triangles.

For images and videos of the method in action, see the included publications and the videos associated with them (e.g., `http://downloads.hindawi.com/journals/ijcgt/2014/580154.f1.avi`, which is the supplementary material for [P4]).

Chapter 5

Rigid-Body Coupling

5.1 Background

As was discussed in Chapter 2, coupling the water simulation with the rigid body simulation is very important both from the believability and the gameplay points of view. To recap, the interaction between water and bodies is two-way: water causes drag and buoyancy forces on the bodies (water-to-body coupling), and on the other hand, rigid bodies provide dynamic border conditions to the water (body-to-water coupling).

Realistic coupling is a challenging problem that is very important for achieving the quality required in, e.g., movie special effects. It has therefore been studied by many researchers interested in the full 3D methods, especially since the early 2000s. Both Lagrangian and Eulerian methods have their own problems, but very convincing results have been achieved by offline methods using either approach. During the previous few years, convincing new real-time Lagrangian methods have also been demonstrated [39]. However, this chapter concentrates on the Eulerian, and especially heightfield, approaches that are more relevant to the thesis.

Some sophisticated coupling methods take both coupling directions into account simultaneously (*strong coupling*, e.g., [56]). It is, however, more feasible in real-time methods to employ so-called *weak coupling*, where the water and rigid body simulations are stepped forward in time alternatingly.

5.2 Object-to-Water Coupling

In fluid simulation, we are typically concerned with completely solid obstacles: water and the object are not allowed to coexist in the same place. This means that at no boundary point is water allowed to flow inside the body. In the Eulerian context, the early approach of Foster and Metaxas was to voxelize the obstacles into the simulation grid, so each cell was marked either as water, solid, or free. The results were used as boundary conditions for the fluid simulation [17]. However, for moving obstacles the velocity of the obstacle should be taken into account. This is often formulated as the *no-stick condition* [6]:

$$\mathbf{u} \cdot \mathbf{n} = \mathbf{u}_b \cdot \mathbf{n}, \tag{5.1}$$

where $\mathbf{u}$ is fluid velocity at the point, $\mathbf{n}$ is the body normal at the boundary, and $\mathbf{u}_b$ is the body velocity at the point. It makes the velocities equal in the normal direction, but does not restrict tangential velocities in any way. In reality, the equality should be an inequality (allowing separation), but traditional methods are only able to solve the condition as an equality. Pressure is typically used as the variable that enforces the no-stick condition. On each time step, a pressure solving step finds such pressure values that cause velocities to agree with the condition. This approach was brought to computer graphics by Takahashi *et al.* [67], though it was known in the field of computational fluid dynamics at least since the 1960s [3]. More complex border conditions become relevant when viscosity is modeled [6, pp. 107–124], but they fall outside of the scope of this thesis.

There are many options for representing the moving obstacles themselves. For rigid bodies, the boundaries of the object can be represented by simplified implicit surfaces (say, a sphere around the object), or a triangle mesh. However, in fluid simulation, another common approach is to model the bodies as particles connected by constraints, such as springs [19, 56], which also allows generalization to deformable bodies [21].

One of the most obvious problems in the voxelization approach is that boundaries unaligned with the grid are handled poorly. Grid artifacts become visible, and even worse, a very small movement of the body can suddenly change a whole cell to completely blocked, causing disturbing artifacts. There are yet other problems with the approach. For example, water will flow through very thin objects (e.g., cloth) that do not completely fill a grid cell.

There are two main approaches to combat the problems with voxelization. Some authors use an irregular grid near the objects (e.g., [16,28]), but this is not feasible in real time. Another option, based on the *immersed boundary method* [53], is to

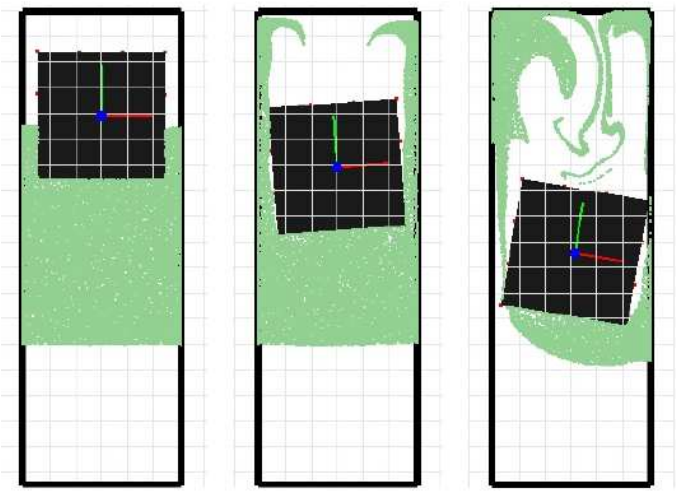

Figure 5.1: Sub-grid details captured by the fractional cell approach used by Batty et al. for gas simulation [2].

average the solid properties in each cell. Batty et al. calculate the fraction of the cell that is obstructed, and calculate fluid mass and other properties based on it. They also pose the pressure solving step as a kinetic energy minimization problem. Besides getting rid of the grid-related artifacts, their approach produces sub-grid details as seen in Figure 5.1. They are also able to efficiently generalize the no-stick condition to an inequality, which allows water to flow outward from the boundary, removing another common artifact of water crawling along walls [2]. Very thin objects can be handled using ray casting [21].

The heightfield methods were derived from the full 3D equations by some assumptions that allowed us to use mostly the same techniques as in 3D by simply dropping a dimension. However, the pressure term is not included in the model anymore, because the hydrostatic pressure is assumed. Therefore we cannot directly use pressure as something that couples the fluid and the objects as in the methods described above.

More importantly, the whole idea of treating each grid cell as having a simple fraction of solid material cannot be used in a heightfield simulation, since the fluid is simulated in a 2D domain, while the bodies exist necessarily in a 3D domain. Water might flow over a low barrier or under a floating body. There might even be an arbitrary number of complex bodies on top of each other, with water in each gap between them. Clearly, the heightfield assumption is broken. Typically, the approaches taken in the heightfield context, then, are much simpler than in full 3D simulations, and do not even try to enforce the no-stick condition.

Figure 5.2: A visual comparison of the difference between a traditional method (left, [72]) and a blocking method (right, [P4]). From [P4].

In some cases, the object-to-water effects can simply be ignored or replaced with visual effects, such as some non-simulated waves, based on the object movements, e.g., to create a ship wake. This is often sufficient for very small and/or light objects that do not have a large effect on the water. This approach is common especially in the methods designed for open-water scenes. In such scenes dense objects simply sink to the bottomless ocean and floating objects have no significant effect on the flows in the ocean besides creating some surface waves. Wave particles [82] and the iWave system [70] are examples of such an approach.

A simple heightfield body-to-water coupling method was proposed by O'Brien and Hodgins. They allow the objects to apply an external force to the surface [49]. The force pushes the surface down, causing waves to emanate to the surroundings. They also applied a particle system to model the splashes caused by an object hitting the surface. Such a system could be, and has often been, added to augment almost any heightfield method.

Thürey *et al.* introduced a more advanced method that keeps track of the submerged height of the body in each cell [72]. Some proportion of the changes of this height are propagated to the fluid, creating waves. The method allows an object to drop into the water, cause waves, submerge, and finally be pulled back up again to cause more waves. However, it does not prevent water flowing through objects. For example, if a house lies on a dry surface and a flood wave approaches, it will simply flow through the house almost unaffected. In other words, in this and all other traditional methods, the water surface goes through the objects, with both substances coexisting in the same place.

As explained in chapter 2, it would be interesting from the gameplay perspective to really affect where the water flows by interactively controlling rigid bodies. This

motivated us to create a fast and simple object-to-water coupling method based on not letting water inside the bodies [P3].

The key idea was to only allow a single, continuous solid vertical interval per cell. This approach has its limitations, but is still surprisingly applicable in many practical cases. After this assumption, the water can be divided into two layers, one above and one below the possible solid. Bodies can then truly block flow on exactly the vertical extent they cover, but water is also able to flow over or under them. This allows various new scenarios, such as building dams out of rigid bodies, without sacrificing as much performance as moving to a fully 3D simulation would. The difference to the traditional approach is illustrated by Figure 5.2. However, our original method [P3] is not able to implement water-to-object coupling, has some rendering problems, and utilizes some ad hoc solutions.

The work in [P4] is a continuation of [P3], providing several object-to-water coupling improvements. Many ad hoc solutions are removed by a more sophisticated handling of flow blocking. It is also shown how the method could be generalized to a multi-layer model, where the assumption of vertically contiguous blockers could be relaxed, though this was not implemented due to the performance cost. The method was combined with a more traditional object-to-water coupling method of [72] in a single scene: floating objects are simulated using one method and heavy objects using another.

5.3 Water-to-Object Coupling

Water affects the objects via pressure and viscous stress. The viscosity-induced forces are rather complex and are typically not solved realistically even in the offline methods [6, p. 13]. Therefore some simplifications are made, and the coupling can be divided into three components: buoyancy, drag, and lift.

Buoyancy is caused by the difference of the lifting forces by the pressure under the body and the sinking forces by the pressure over the body. If the hydrostatic pressure profile of equation 4.1 is assumed, this difference can also be expressed as Archimedes' law: the buoyancy force is equal to the weight of the water that the body has displaced. For a floating body, this implies some kind of a virtual level inside the body, where the surface would be if the body did not exist.

Buoyancy is obviously very important for games, because many interesting game objects need to float. The simplest possible model would be to only apply a vertical force at the center of mass, but it is also desirable to model the torque caused by

different buoyancy at different points in the object, for example, to model a ship that is rocking or even capsizing. The problem is that simply summing the forces in a coarse grid easily causes an unstable situation, where the body starts to spin. This can be limited by some artificial dampening, for example, scaling the angular velocity of a body during each time step with a multiplier that gets smaller as the object gets more submerged. A more detailed calculation is also possible, but the performance cost might be prohibitive.

To find out the buoyancy force in 3D methods, the pressure can simply be summed over each cell which has both a water and a solid. In heightfield methods, pressure is typically not modeled at all and this approach is not available. Archimedes' law is usually used instead. In most methods, this is very straightforward to implement. The weight of the displaced water in a cell is simply based on the submerged volume: $F_{buoyancy} = \rho g(\Delta x)^2 \max(0, \min(s, z_+) - z_-)$, where z_+ is the top and z_- is the bottom of the body's extents at that cell (see e.g., [72]). Instead of per cell, this can also be done per mesh triangle (breaking large triangles to as many as necessary), increasing upwards force if the normal points downwards and decreasing otherwise [10, 82].

Drag is required to make a body follow the flow, since it forces the velocity of the object to equalize with the flow velocity, which also has interesting gameplay possibilities. Lift has less apparent significance, and is sometimes ignored in simple heightfield simulations. Both of these forces are more complex to model than buoyancy, and they are taken into account in various ways depending on the method.

Stokes' law, assuming laminar flow, states that for a small spherical object in a fluid, the viscous resistance force is linearly proportional to the relative velocity between the object and the water, and to the radius of the sphere. A simple model then, is to calculate the vertical extent of the body that is underwater in each cell, and multiply it by the velocity difference and some user-defined (i.e., arbitrary) constant [72]. Again, another option is to calculate the forces triangle by triangle based on the velocity differences and triangle normals [10, 82].

In [P3] and [P4], unlike in other heightfield-based coupling methods, bodies actually displace the water. Because of this, information on where the surface would be located without the body is not readily available, and thus the displaced volume cannot be directly calculated. In other words, if buoyancy would be calculated in the usual way based on the difference of the bottom of the body and the surface, the result would always be zero for a floating body, because the surface is just below the body. Therefore the method introduced in [P3] only implemented object-to-water coupling. An additional problem is that due to the limited reso-

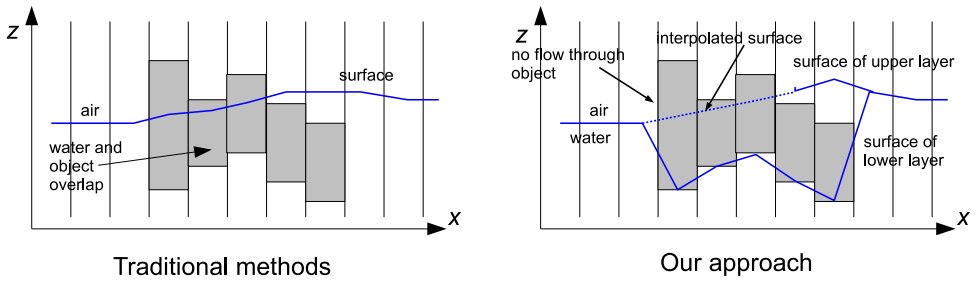

Figure 5.3: Left: in traditional methods (e.g. [72]), the water surface goes through a floating body. Buoyancy is simple to calculate, but water flows through the bodies, which leads to very limited body-to-water coupling. Right: in our approach [P4], two layers are used and the flow is calculated by not letting water flow through the body. The final surface is robustly interpolated for drawing and to calculate buoyancy.

lution of the grid, the rendered surface tends to curve down already around the bodies, which looks very unnatural.

Further work in [P4] solved the mentioned issues by introducing a robust method for interpolating the displaced surface from the surrounding area. Buoyancy can then be calculated in the standard way, and the interpolated surface can also be used for drawing. An alternative approach could be to directly keep track of the pressure under the body, but our experiments showed the result of the simpler interpolation to be just as good. Figure 5.3 compares the traditional approach to that introduced in [P4].

5.4 State of Coupling in Heightfield Methods

There seems to currently be no single method for achieving believable results in both water-to-object and object-to-water coupling in a heightfield water simulation. The usual methods of, e.g., [10, 72] work very well for small floating objects, if the water flow mostly consists of small waves. On the other hand, if a large and heavy object should push the water out from an area or block the flow completely, these methods leave much to be desired.

In contrast, the blocking-based method of [P4] is applicable in the latter situations. We find the results acceptable in most cases, but the method is far from perfect. The main limitation is due to the problematic voxelization of the solids to the

grid. This causes especially bad spatial aliasing when the method is used to handle floating objects in a steady flow, as analyzed further in the publication.

As mentioned above, the voxelization issue has been thoroughly researched in the fully 3D case. The most promising solution is applying a fractional method similar to [2] or [70]. However, due to the 2D fluid simulation being coupled to a 3D rigid body simulation, the previous methods are not directly applicable, and further work is needed to find a solution. Additionally, the handling of the blocking is not yet very physical. For example, instead of the no-stick condition, we use a simplified condition where the body velocity is not taken into account.

The two approaches can be combined by selecting which method to use on a case-by-case basis, but this is only a workaround. While we doubt that the traditional methods can be extended to handle situations like building dams, we are confident that further work on the blocking approach will be able to make the method more generally applicable in games.

Another viable option for the future might be using a hybrid method, where the water around objects is simulated using particles, and a heightfield is used for the rest of the world. This combination could achieve the best of both worlds with robust interaction between water, smoke, rigid and non-rigid bodies as in [39], but still retains the possibility of including large amounts of water. The main hurdle would be how to seamlessly and robustly couple the two water simulations. This approach would not necessarily introduce much complexity, because it is already a good idea to include a particle system in a heightfield simulation to represent splashes and other truly 3D effects.

Chapter 6

Comparison of Water Simulation Methods

6.1 Background

We have now seen an assortment of methods that could be used for modeling water behavior in digital 3D games. This chapter summarizes the most important differences between some of the methods using the criteria set in Chapter 2: performance, simplicity, visual quality, richness of behavior, and coupling.

The methods included in this comparison have been introduced in Chapters 3 and 4. Fully 3D methods include the 3D Eulerian method, and the family of 3D Lagrangian-based methods (e.g., SPH, FLIP, position-based fluids). The latter family is mostly considered a single method in this chapter. The tall cell method can be considered a hybrid of 2D and 3D. The rest of the methods are based on heightfields: the pipe method, SWE, 2D LBM, wave particles, FFT, iWave, and 2D SPH.

While there have been some survey articles dealing with water simulation methods [14, 25, 26, 52, 68], we are not aware of a comprehensive comparison of the methods from a games' point of view. The comparison in most categories is necessarily subjective until more data and comparable reference implementations become available. However, many of the differences in, e.g., performance or classes of behavior modeled are large enough to make our comparison useful at least as a starting point to guide the reader when selecting a technique to use.

In the computer graphics community, it is customary to create standard reference scenes rendered, e.g., with an offline ray tracer, to which the real-time results can then be compared. To our knowledge, there is no such systematic work in the field of real-time water simulation. Each researcher has simply created some varying scenes of their own. Floating semi-complex objects, such as the Stanford bunnies are often shown in various scenes.

In the field of computational fluid dynamics, there exist more rigorous scenarios even with analytical solutions to compare to, e.g., several kinds of dam break scenes, but these have not been systematically adapted in the computer graphics literature. Most methods have no publicly available reference implementations that could be used as the basis of comparison. Also, because in games believability is more important than strict realism, user studies would be very informative. Sadly, the only such study in the field so far seems to be [P2].

This chapter is based on our earlier work in [P1], with a more thorough selection of candidate methods, refined comparison criteria, and several other improvements.

6.2 Performance

Some timings of a few different water simulation implementations were presented in [P1]. Revised and expanded results are collected into Table 6.1. The results are based on comparing reported statistics instead of implementing the methods in a single software and hardware environment. They at least provide an approximate comparison.

Many authors report the performance of their method as simulation frames calculated per second (fps), i.e., the reciprocal of the time a single simulation step takes (which is called *timing* in Table 6.1).

The fps measure is typically used when comparing the smoothness of game visuals, so it may seem like a natural measure of a simulation method aimed to provide a smooth animation. However, the fps figure is not enough for comparing simulation methods, because the maximal stable time steps greatly vary between methods. If the time step is small, water will seem to flow very slowly. For example, from Table 6.1 we find out that the time step of a certain 2D SPH implementation is 2 ms, while the time step of our pipe model implementation is 25 ms. Imagine that the resulting frames from these simulations were simply shown to the viewer at, say, 60 fps. Our pipe simulation would progress 60 Hz * 25 ms = 1.5 seconds per

Method	Theory	Cells	Δt	timing	rtkc
Pipe model [P1]	$O(n^2)$	65k	25 ms	0.55 ms	3000
SWE [10]	$O(n^2)$	122k	20 ms	2.2 ms	1100
2D LBM [51]	$O(n^2)$	16k	16 ms	0.35 ms	730
Wave particles [82]	$O(n^2 + m)$	65k/100k	?	6 ms	?
FFT [69, 77]	$O(n^2 \log n)$	65k	?	10 ms	?
iWave [70]	$O(n^2)$*	–	–	–	–
3D Lagrangian [39]	$O(m)$*	100k	4 ms	4.3 ms	100
Tall cells [11]	$O(n^2)$*	16k	33 ms	33 ms	16
2D SPH [62]	$O(m)$*	128k	2 ms	50 ms	5
3D Eulerian [2]	$O(n^3)$*	0.4k	?	500 ms	?

Table 6.1: Comparison of the performance of some methods. "Theory" is the asymptotic speed of the method with a given horizontal resolution of n cells, and/or m particles (* marks a significantly large constant multiplier). Cells is the number of horizontal cells or particles as appropriate. Timing is the time used to advance the simulation a single step. *rtkc* or "real-time kilocells" is the number of cells that can be simulated in real time (1000s). '?' signifies missing data and '–' means that no particular implementation was selected for this comparison. Partially based on [P1]

second (i.e., faster than real-time), while the SPH simulation would only advance 120 ms per second.

It is also trivial to interpolate as many extra frames as needed from two last states of the simulation. In our experience, for a relatively calm scene, simulation steps of hundreds of milliseconds or even several seconds can provide good-looking animations if a simple interpolation of the water surface between two last calculated simulation steps is used. Therefore, the simulation time steps are not usually limited by the need to get a smooth animation (i.e., a high fps figure) but by stability.

Another factor that makes the comparison difficult is that all of the methods compared do not necessarily use the same grid scale. Many authors do not even mention the scale used. To understand why the scale is important, consider the CFL condition (see Section 3.2.2): The smaller the scale, the smaller time steps are required. Thus simulating, e.g., a 100×100 heightfield in real time takes much more computing power if it represents an area of 1 m $\times$ 1 m than 100 m $\times$ 100 m. Intuitively, in a small scale, the gravity force is large compared to the cell size, which quickly causes large velocities (measured in cells per second).

Luckily, for virtual terrains, there is no need to have a grid denser than a few tens of centimeters to a few meters, because details smaller than that do not affect the big picture, and can thus be created purely visually (e.g., normal maps, FFT). On the other hand, in most game worlds the resolution cannot be allowed above a few meters if human-sized features are to be modeled. It therefore makes sense to compare the methods aimed at roughly a horizontal resolution of one meters. Thus the size of the modeled area is thus almost directly proportional to the number of cells the method is able to handle.

To help compare different methods, we introduced a measure called *rtkc*, or "real-time kilocells" in [P1]. It is defined as the number of cells (in thousands to make the figure more accessible) that can be simulated in real-time, i.e., the ratio of the time step to the timing. For a person implementing a game, this measure directly gives an idea of how large a terrain can be simulated in real time, and should make the performance of different methods much more comparable than using a simple fps figure. For example, assuming a grid resolution of 1 m, the *rtkc* value of 3000 for our pipe model implementation means that roughly an area of 3 km^2 can be simulated in real time on the hardware where the figure was measured.

There are still several problems remaining with the comparison. Performance depends on the processing power, memory bandwidth, and other hardware details in a complex way. Hardware differences thus cannot be ruled out, since most of the implementations are research prototypes that are not publicly available, so each timing is measured with whatever hardware the research team happened to use. Another issue is the varying grid size, because the fixed costs of the methods can be quite large and become significant when using small grids, which makes the comparison unfair if different grid sizes are used. The grid size can also affect cache efficiency. For this reason, we have tried to select similar grid sizes for the comparison within the data provided by the authors.

According to [P1], the pipe method is at least an order of magnitude faster than the other methods. However, when comparing with the SWE implementation of [10], the cost of the particle system was also included. Here we only compare to the time reported for the heightfield simulation part, which gives more even results (a difference of 3x, though this does not take hardware differences into account). Both the SWE and the linearized SWE (which is equal to the pipe method in performance) are implemented in the same context in [P2]. We found that the full SWE only adds 38 % to the execution time. We concur that while the pipe method is definitely faster than the SWE and the difference may be significant for some games, it is probably not a decisive difference.

Both the tall cells method [11] and SPH/SWE hybrid of [62] are certainly much slower than the 2D Eulerian heightfield methods, though in an asymptotic sense

there is no difference [P1]. Particle-based methods (SPH/SWE and SPH) seem to suffer from short time steps and needing more particles than grid-based methods need cells to reach a comparable result, but the relative performance gets better as the dry area increases. As the hardware continues to improve, these methods are becoming more and more viable. One should also remember that the SPH/SWE implementation in our comparison was published in 2011 [62], and research on SPH and related methods has progressed considerably after that [26]. For example, the position-based approach could possibly be transferred to the SWE context.

Wave particles, LBM, and FFT-based methods were not included in the comparison of [P1]. We shall briefly address the performance of those methods below.

Yuksel *et al.* provide some data on the performance of their method implemented on an NVIDIA GeForce 7900 GPU [82]. Because their method concentrates on the waves created by objects interacting with water, the performance is mostly dependent on the triangle count of the objects. It is therefore difficult to compare to the other methods, where the grid size or particle count dominates the complexity. Additionally, their heightfield rendering step timing apparently includes both building the heightfield from the particles and actually rendering the result, which differs from the approach of [P1], where rendering times were ignored. Time step and heightfield scale are also not given, which makes performance comparison impossible. As an example, with a heightfield resolution of 512×128, up to 100 000 wave particles, and a single boat, the whole method takes about 6 ms. They demonstrate coupling with a large number of objects in real-time.

Tubbs and Tsai implemented the SWE using 2D LBM on the GPU [74]. As an example somewhat comparable to the scenarios in [P1], they provide a partial dam break scenario with a 2001×2001 grid with a 0.1 m spacing. Calculating a single time step takes about 9 ms using an NVIDIA Tesla C1060 computing processor (comparable in performance to a relatively typical current desktop GPU). Since the LBM method is partially iterative, the time step is not given, but from their extended partial dam break scenario, an average time step of about 5 ms can be inferred. With these values, a rough estimate of the performance of this method would be about 2000 *rtkc*, using the terminology of [P1]. Ojeda and Susin implemented a similar scenario using LBM on the GPU [51]. Their LBM simulation runs several times iteratively during each time step of 16 ms. This took 0.35 ms for a 128×128 grid. The hardware was NVIDIA GTX 280. Based on these two implementations, the method seems to perform approximately as well as the pipe method or Eulerian SWE simulations despite the small time steps. However, the comparison does not take into account that faster hardware was used for the LBM simulation than for our SWE and pipe simulations.

For the FFT method, the slowest part is probably the $n \times n$ inverse FFT [69]. As the inverse FFT has a time complexity of $O(n^2 \log n)$, we estimate that the method should be maybe an order of magnitude slower than the pipe method or SWE for large areas. We refer to an implementation by Wang *et al.* [77]. They also include a particle system besides the FFT, and there is unfortunately not enough information to deduce how fast the pure FFT system would be. From the data provided, we estimate that their 256×256 result could be created in about 10 ms. There is no information on the time step.

The iWave method, on the other hand, is based on continuously applying a convolution kernel, which needs some tens of coefficients [70]. This probably results in a performance somewhat comparable to the FFT implementation. We are not aware of a comparable GPU implementation of the method.

For the fully 3D methods (Lagrangian and Eulerian), their $O(n^3)$ time complexity makes them very slow for large areas. Comparison between them is an ongoing debate in the movie effects world [26]. SPH has the large advantage of only having to simulate areas where water is present, which allows it to already be used in games for small-scale effects. For comparison, we have selected a relatively fast Eulerian simulation [2] and the very fast and promising position-based Lagrangian approach [39]. Both of the methods include coupling.

For the Lagrangian methods it should be noted that the *rtkc* figure is not directly comparable to the grid-based methods, since filling a 3D area with water requires considerably more particles than the number of cells used in a heightfield method. However, the figure for the position-based fluid suggests that if this new development could be combined with the 2D SPH approach, the performance might be competitive with the Eulerian heightfield methods.

6.3 Simplicity

The pipe method (or the linearized SWE) is, without a doubt, the simplest of the methods in this comparison. A non-parallel implementation can be created using only a few lines of code by an unsophisticated programmer, and versions of it were already used decades ago by the hobbyist demoscene programmers. The method is also an excellent fit to the GPU, making the implementation relatively easy.

We added the semi-Lagrangian advection to reach the full SWE in [P2], resulting in approximately a 70% increase in the number of lines in the simulation code. The advection is relatively simple to implement even on the GPU, but more com-

plex than the pipe method itself. Neither the pipe method nor SWE require any advanced data structures, since simple 2D arrays are all that is needed.

A basic version of 2D SPH is also easy to understand and implement. In practice, the neighbor search requires some kind of an acceleration structure, which can be somewhat complex to implement efficiently on the GPU. Solenthaler *et al.* do not mention using an acceleration structure [62]. The performance figures might improve considerably with such an addition. LBM, FFT, wave particles, and iWave all have their complexities, but seem relatively straightforward to implement also on the GPU.

The tall cells method and the full 3D Eulerian method need to solve a large sparse linear system for the pressure. This can be done in linear time with iterative solvers such as preconditioned conjugate gradient, or multigrid methods [4]. These solvers are clearly more complex to implement efficiently (especially on the GPU) than what is needed by the methods mentioned previously. Some GPU implementations are available for the task of solving sparse linear systems [44, 46], but they are not necessarily optimized for the special case of a certain water simulation method.

6.4 Visual Quality

From the images produced by most of the methods included in this comparison in Figures 6.1 through 6.8 it is easy to note that the different visualizations make comparing the simulation methods themselves difficult. The issue of fair comparison is thus even more difficult than in the other categories, because implementation details, such as the utilization of textures painted by artists or different additional effects such as foam modeling, play such a huge role in the results. It is also typical to add small-scale details using, e.g., advected textures [75] or FFT [72]. Differences in the detail system might well be more important than the simulation method to the visual quality.

Another issue is the varying scenes, since one method might be well suited for a certain scale or type of water motion, where another is not. Since objective criteria are hard to set, more empirical work using user studies is called for.

The full 3D approaches can both create scenes that are virtually indistinguishable from reality, as seen in recent movies. Of the more limited methods, the tall cells method of [10] is clearly the most believable, since unlike the heightfield methods, it is able to retain most of the interesting 3D behavior near the surface.

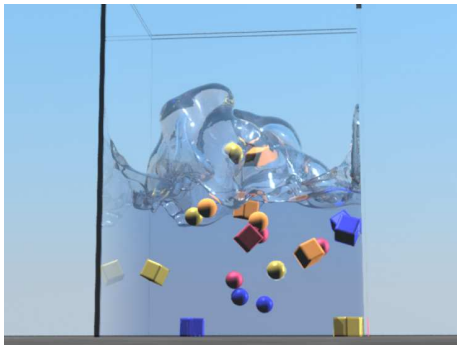

Figure 6.1: Full 3D Eulerian simulation with physics coupling on a $100 \times 150 \times 100$ grid (not real-time) [56].

Figure 6.2: Tall cell based simulation with additional visual effects including particles and foam on a $128 \times 34 \times 128$ grid [11].

Figure 6.3: SWE with a particle system and additional details created using FFT on a 256×256 grid [10].

Figure 6.4: 2D LBM simulation with a coupled particle system on a 128×128 grid [51].

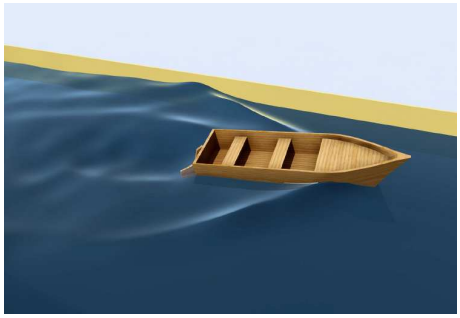

Figure 6.5: Wave particles with 100k particles [82].

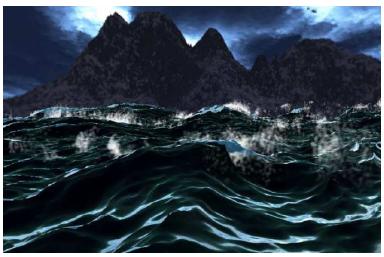

Figure 6.6: FFT ocean with particle splashes [77].

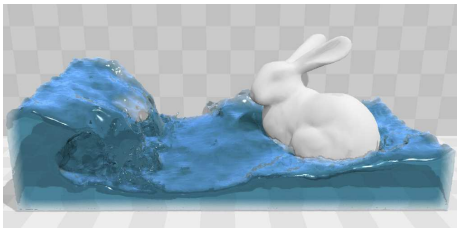

Figure 6.7: 3D Lagrangian simulation (position based fluids) with 128k particles [39].

Figure 6.8: 2D SPH with 100k particles [62].

Of the methods that produce a heightfield, the FFT method has also been used in movie effects. It captures many intricate phenomena, such as wave dispersion, and can create seascapes that are hard to distinguish from reality [69]. 2D LBM and the Eulerian SWE are both used also in CFD, and can be very true to reality in the shallow water cases [10, 74].

The SPH/SWE method is difficult to compare to the others, since the visualization technique is different from that of the Eulerian methods [62]. However, since it is a heightfield method, the result quality is closer to SWE than tall cells.

Our work in [P2] aimed to provide a fair comparison of the visual quality between the SWE and its linearized version. The implementations, visualizations, scenes, and simulation parameters were identical and the comparison was done with actual users and a playable game in addition to just videos. The study found no statistically significant difference between the user experience created by the methods as measured by the self-reported experiences of valence, flow, presence, and realism. A supplementary power analysis to this study was presented in Section 4.4. It should also be noted that only a single simulation resolution and a single type of a game were used in our study. We still conclude that while there may be some loss of quality caused by the linearization, it may not be as obvious a problem for games as sometimes thought to be [P2].

6.5 Richness of Behavior

Some methods limit the classes of water behavior that can be modeled, e.g., because of simplifying assumptions. Possibly the largest such difference takes place when limiting to a heightfield surface, which precludes modeling situations where the water surface is not a function of horizontal location. Such cases include, e.g., waterfalls, breaking waves, splashes, and air bubbles. Many of these can be added back using a particle system (e.g., [10, 49, 72]) or a generalized heightfield (e.g., [82]), but this brings a need of coupling the two simulations and/or makes the overall system more complicated.

Many of the methods also do not try to solve the complete NSE. Practically all of the methods being compared drop the viscosity term, even though the problem is usually too much (numerical) viscosity instead of too little.

Another simplification is the assumption of very large water depth in the FFT, iWave and wave particle methods, which makes these methods incompatible with terrain. Even if border conditions are set so that waves are reflected from the

shores, this is a far cry from dynamically varying the flooded area or the ability of a lake to get filled until it overflows from the lowest point. This is a major drawback, considering many of the gameplay categories discussed in Chapter 2. On the other hand, LBM, SPH, and methods based on SWE typically handle these cases well. On the other hand, SWE does not model dispersion correctly for deep water, which causes non-convincing ocean waves [6, p. 187].

The wave equation based methods further simplify the situation by dropping the advection terms, which removes vortices and other complex behavior [30]. However, according to [P2], the difference might not affect the user experience adversely.

Regarding the gameplay perspective, it is noteworthy that it is possible to implement all of the gameplay categories outlined in Chapter 2 with the simple pipe method augmented by a particle system and a coupling method such as that introduced in [P4].

6.6 Coupling

As was seen in Chapter 5, interaction with moving bodies is difficult to implement in all heightfield methods. Even a simple submerged body breaks the assumption of the water always being vertically continuous.

A method for handling floating bodies with some kinds of reactive waves has been proposed for all methods considered in this chapter. These solutions are on an acceptable level for many games. However, when a stronger object-to-water coupling is needed, all current solutions have problems.

3D SPH researchers have, in general, demonstrated several excellent and robust coupling methods, usually based on sampling the rigid bodies as groups of particles. The coupling for 2D SPH in [62] also seems to function rather well, even implementing a dam made out of logs, for example. On the other hand, in Eulerian methods the fixed grid resolution often causes problems such as aliasing [6,P4]. Our work in [P3, P4] should be applicable to SWE almost as well as the pipe method, but is not yet on the level of the SPH-based coupling where the interaction with moving bodies can be truly dynamic.

	Perform.	Simplic.	Visual	Richness	Coupling
Pipe [P1, P2]	+ +	+ +	− −	+	+
SWE [10, 72, P2]	+ +	+	−	+	+
LBM [74]	+ +	+	−	+	+
FFT/iWave [69]	+	+	+ +	−	−
Wave particles [82]	+ +	+	−	−	−
Tall cells [11]	+	− −	+ +	+	−
SPH/SWE [62]	−	−	−	+	+ +
Full 3D Lagrangian	−	+	+ +	+ +	+ +
Full 3D Eulerian	− −	− −	+ +	+ +	+ +

Table 6.2: A qualitative comparison of different large-area water simulation methods for 3D games, comparing performance, simplicity, visual aspects, richness of behavior and coupling with objects.

6.7 Summary

After all this discussion, which simulation method should a game programmer choose for their game? Naturally, the answer depends on the requirements. First of all, not all games with water need any kind of simulation. Even a static 2D plane with visual trickery might well be sufficient to create a convincing illusion. If water is to play a more important role in the game, it might be worth considering using the computational resources for a more dynamic solution.

The strengths and weaknesses of the methods have been collected into Table 6.2, which tries to answer Research Question 1 of the thesis, as posed in Section 1.2. To more easily interpret the table, consider the following questions:

- *Are you making a 2D game?* SPH and related particle methods work well for 2D games, since they provide the possibility of almost unlimited water topology changes unlike most other 2D methods, and the number of particles needed is typically low enough even for mobile devices. The rest of the questions assume a 3D game.
- *How large is the area and what is the amount of water?* For very small 3D grids, such as 32^3, a full 3D solver might be fast enough, and will provide the best results. For small amounts of water, such as a leaking pipe or a player-operated firehose, 3D particle methods (e.g., SPH) are recommended. For larger areas such as whole rivers and lakes, heightfield methods are typically needed to get acceptable performance. The tall cell method is an in-between

case that provides great quality without the need for a full 3D simulation. However, it is probably still too slow for most games for now.

- *Are you planning to implement the method yourself with a limited time budget?* The pipe method is simple to implement, even on the GPU, yet provides most of the interactive possibilities. We recommend the description of Mei *et al.* as a starting point [42]. It is then easy to expand to the full SWE by adding semi-Lagrangian advection, if more physical accuracy is desired. For this we refer the reader to Chentanez and Müller [10]. SPH is another method that is easy to start with, but is suited to different situations. LBM is also rather simple, but there seems to be less literature available on implementing it for computer graphics or games applications than for SWE. Each of these is also relatively straightforward to implement on a GPU.

- *What kind of a visual quality is needed?* The full 3D methods are naturally in a class of their own in this regard. The tall cell method also preserves the important surface details while not being all too heavy for resources, though it is currently constrained to smallish areas or tech demos that are allowed to reserve a whole top end GPU just for the water. A cheaper option in terms of performance might be to use a heightfield method as the base solution, but also implement a particle system for splashes and other effects as in, e.g., [72]. A lot can also be achieved by simple visual tricks, such as advecting normal and foam textures as in [75]. SWE is more physically-based than the pipe method, but the difference is small both in terms of performance and visual quality [P2].

- *Are you modeling an ocean, or rivers and lakes? Do you need the area covered by water to be dynamic?* Most of the heightfield methods are designed either for deep or shallow water. FFT, iWave, and wave particles can model the wave dispersion in oceans. These methods are based on an assumption of a zero water level with deep water under it, and cannot thus handle dynamic spreading over an uneven terrain. On the other hand, pipe, LBM, SWE, and SPH work best when applied to shallow water. All of these methods are based on a model with depth included, and can thus have a depth equal to zero to model dynamic changes between dry and wet areas as the water flows over uneven terrain.

- *What kind of gameplay and interaction is needed?* If we consider the gameplay effects listed in Chapter 2, most of the categories can be handled by all of the methods to some extent. In all of the methods, a velocity field can be calculated and used for advecting objects and water properties. Buoyancy is also handled by all of the methods. Being useful or harmful at a location, and adding or removing water becomes much more interesting from a gameplay point of view if the area covered by water is dynamic (see the previous question). Forces can be applied in all of the methods (except pure FFT)

at least to cause some reactionary waves. Again, the methods with dynamic flow have an upper hand for most of the gameplay categories. Finally, handling interaction with rigid bodies by seeing them as dynamic borders for the water is by far the most developed for the full 3D methods, but solutions also exist for the 2D SPH [62] and the pipe method [P3, P4].

In short, full 3D methods are the best in most ways other than performance. It is an eternal battle whether one should use an Eulerian or a Lagrangian approach, or some combination thereof. For most games, one of the Lagrangian methods is probably the best option. However, it is only feasible for small amounts of water. Of the cheaper methods, the ocean methods (FFT/iWave, wave particles) are great for a static ocean with beautiful and believable waves, but offer limited interaction possibilities.

Considering the fast heightfield methods, 2D SPH has the best interaction especially with rigid bodies, but offers somewhat disappointing performance due to the small time steps required. However, if the point-based approach could be incorporated to this 2D approach, it could become competitive again. Pipe, SWE, and LBM work well for large amounts of dynamic water flowing across a terrain. It would be interesting future work to compare the properties of the SWE and LBM methods. For now, the choice between them seems more like a matter of taste than anything else.

Most of the heightfield methods can be augmented with a particle simulation for splashes [49], breaking waves [72], and other effects (e.g., foam [71]). As the particle methods are simple to implement and already included in many physics engines, it is recommended to take advantage of them as a simple way to mitigate the limitations of heightfields, as in, e.g., [10] for SWE and [51] for LBM.

Chapter 7

Conclusion

This thesis has explored the application of water simulation techniques for large-scale terrains with a focus on the interactive role of water, but not forgetting its more typical, purely aesthetic role. The main application is to enrich gameplay, but any virtual environment that needs more interactive water could benefit from the methods discussed.

In Chapter 2, water was seen to help enable spatial presence in games both by creating a visually believable environment and richer interaction. Water is also a complex and interesting element that provides many opportunities for novel game mechanics, enabling emergent and dynamic gameplay systems, which are immediately understandable but hard to master. Both of these should help achieve flow and therefore media enjoyment.

In most games interactive water would be just a small part of the game, with both development and computer resources spread thin over dozens of different features. Considering this, some game developers view, perhaps justifiably, water simulation as hard to understand and computationally too expensive for games. Additionally, players expect a very high visual fidelity from current games. It is difficult to find a method that is able to provide the results developers are looking for with the limited resources available.

The performance issue has been somewhat alleviated by the advent of the modern programmable GPUs. Compared to CPU programming, however, GPUs are still more difficult to use due to the increased parallelism and less advanced tools. More software suites similar to the now widely available, easy-to-use rigid body simu-

lation libraries will be needed before large-scale dynamic water can truly become commonplace in games.

For simulating small amounts of water, recent developments in the position-based Lagrangian methods are very promising [39]. However, since the number of particles still needs to scale in three dimensions, no matter how much the hardware advances, heightfield methods will always be able to represent larger areas than fully 3D methods.

To answer the visual quality requirements, it is useful to notice that the visual details need not affect the water flow in a large scale, and they can thus be created using spectral or other non-simulation-based methods. However, even the best visual effects become meaningless if the immersion is broken when the player tries to interact with the game environment. While spectral methods can create very convincing visuals, they typically function much worse whenever interactivity is required.

In [P1], we found the pipe method to be the fastest and simplest of the heightfield methods. Despite its simplicity, it is able to model the flow in a way that is sufficient for all our gameplay categories described in Chapter 2. The visual details lost by it can be added back using a particle system where needed. Additionally, a user study showed that the user experience induced by a linearized method did not differ statistically significantly from that achieved by a more complex simulation that used the full shallow water equations [P2]. We therefore recommend game developers interested in adding large-scale flowing water to their game to at least experiment with the pipe method.

For interactivity, it is important to model how water flows over the terrain, and give the player an ability to affect the flow, either by modifying the terrain or via interaction with rigid bodies. Interaction with changing terrain is very possible even with the pipe method, as demonstrated by, e.g., our prototype game in [P2] or *Cities: Skylines*.

However, coupling with rigid bodies is more challenging, because heightfield simulation happens in a 2D grid, but the rigid bodies are 3D objects. The traditional methods are based on letting the water flow through bodies, which works well for small objects floating on a large and calm body of water, but restricts the usage of the methods to such scenarios.

This thesis introduced novel heightfield coupling methods that aim for richer interaction, concentrating especially on the body-to-water effects [P3,P4]. The methods are based on redirecting the flow around the bodies. The proposed solution allows,

for example, building completely dynamic dams. However, future work is needed to solve the grid aliasing issues and reaching more physically-based results.

An alternative avenue for the future is to combine the particle-based simulations with the heightfields, since the particle-based methods currently boast the best two-way coupling results. Ideally, such a hybrid method could use particles near the camera and any interacting objects, but fall back to a faster Eulerian heightfield simulation wherever possible. This would require solving the challenge of seamlessly coupling the two approaches. Further research into this area is needed.

We observed that there is a lack of empirical research in the field of real-time water simulation. There are very few comparisons of different simulation methods, or discussion on which metrics should be used for such comparisons. Indeed, not all publications report even the most basic necessary information, such as the time step or scale of the simulation. Because the visual quality or believability of the methods cannot be objectively measured, user studies are also called for.

This work tries to do its part of this empirical work. We introduced a measure called real-time kilocells to compare the performance of different grid-based methods, and used it to compare the performance of some methods. In another empirical dimension, possibly the first user study in the field was presented in [P2].

Much more empirical work is still needed, for example to create and use standard test scenarios for real-time water simulation. Different methods should be compared in the same environment using a similar visualization. For example, there is very little information on how LBM-based methods compare to SWE both in performance and believability of the results.

Despite the large amount of future work still needed to further these topics, this thesis argues that GPU-based heightfield water simulation is already mature enough to be used as the basis of several kinds of new game mechanics, facilitating the birth of possible new game genres in the near future. Implementing the simplest water simulation methods such as the pipe method should not be a major undertaking even for the smaller companies.

Ludography

Angry Birds (2009). Rovio.

Assassin's Creed III (2012). Ubisoft Montreal / Ubisoft.

Battlefield 4 (2013). EA Digital Illusions CE / Electronic Arts.

Cities: Skylines (2015). Colossal Order / Paradox Interactive.

Counter-Strike (2000). Valve.

Crysis (2007). Crytek / Electronic Arts.

Doom (1993). Id Software.

F.E.A.R. (2005). Monolith Productions / Vivendi.

From Dust (2011). Ubisoft Montpellier / Ubisoft.

Gone Home (2013). The Fullbright Company.

Grand Theft Auto series (1997–2014). Rockstar Games.

HALO 3 (2007). Bungie / Microsoft Game Studios.

Just Cause 2 (2010). Avalanche Studios / Eidos Interactive.

Liquidsketch (2012). Neukom, Tobias.

Minecraft (2011). Mojang.

Munin (2014). Gojira / Daedalic Entertainment.

Resistance 2 (2008). Insomniac Games / Sony Computer Entertainment.

Spintires (2014). Oovee Game Studios.

Sprinkle (2011). Mediocre Games.

Ultima Underworld: The Stygian Abyss (1992). Blue Sky Productions / Origin.

Uncharted 3: Drake's Deception (2011). Naughty Dog / Sony Computer Entertainment.

Walking Dead, the (2012). Telltale Games.

Where's My Water (2011). Creature Feep / Disney Mobile.

Bibliography

[1] Florian Bagar, Daniel Scherzer, and Michael Wimmer. A layered particle-based fluid model for real-time rendering of water. *Computer Graphics Forum*, 29(4), 2010.

[2] Christopher Batty, Florence Bertails, and Robert Bridson. A fast variational framework for accurate solid-fluid coupling. *ACM Transactions on Graphics (TOG)*, 26(3), 2007.

[3] David J Benson. Computational methods in Lagrangian and Eulerian hydrocodes. *Computer methods in Applied mechanics and Engineering*, 99(2), 1992.

[4] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Transactions on Graphics (TOG)*, 22(3), 2003.

[5] J Brackbill and H Ruppel. FLIP: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. *Journal of Computational Physics*, 65(2), 1986.

[6] Robert Bridson. *Fluid Simulation for Computer Graphics*. A K Peters/CRC Press, 2008.

[7] Robert Bridson, Jim Houriham, and Marcus Nordenstam. Curl-noise for procedural fluid flow. *ACM Transactions on Graphics (TOG)*, 26(3), 2007.

[8] Jerome S Bruner and Leo Postman. On the perception of incongruity: A paradigm. *Journal of personality*, 18(2), 1949.

[9] John Charles Butcher. *The numerical analysis of ordinary differential equations: Runge-Kutta and general linear methods*. Wiley-Interscience, 1987.

[10] Nuttapong Chentanez and Matthias Müller. Real-time simulation of large bodies of water with small scale details. In *2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10. Eurographics Association, 2010.

[11] Nuttapong Chentanez and Matthias Müller. Real-time eulerian water simulation using a restricted tall cell grid. *ACM Transactions on Graphics (TOG)*, 30(4), 2011.

[12] Richard Courant, Kurt Friedrichs, and Hans Lewy. Über die partiellen Differenzengleichungen der mathematischen Physik. *Mathematische Annalen*, 100(1), 1928.

[13] Isabella Selega Csikszentmihalyi. *Optimal experience: Psychological studies of flow in consciousness*. Cambridge University Press, 1992.

[14] Emmanuelle Darles, Benoît Crespin, Djamchid Ghazanfarpour, and Jean-Christophe Gonzato. A survey of ocean simulation and rendering techniques in computer graphics. *Computer Graphics Forum*, 30(1), 2011.

[15] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01. ACM, 2001.

[16] Bryan Feldman, James O'Brien, and Bryan Klingner. Animating gases with hybrid meshes. *ACM Transactions on Graphics (TOG)*, 24(3), 2005.

[17] Nick Foster and Dimitri Metaxas. Realistic animation of liquids. *Graphical models and image processing*, 58(5), 1996.

[18] gamesgames.com. Dolphin jumping games. `http://www.gamesgames.com/games/dolphin-jumping`. Accessed: Feb 5, 2015.

[19] Olivier Génevaux, Arash Habibi, and Jean-Michel Dischler. Simulating fluid-solid interaction. *Graphics Interface*, 2003, 2003.

[20] M. Gonzalez Ochoa. Water technology of Uncharted. Presentation in Game Developers' Conference 2012.

[21] Eran Guendelman, Andrew Selle, Frank Losasso, and Ronald Fedkiw. Coupling water and smoke to thin deformable and rigid shells. *ACM Transactions on Graphics (TOG)*, 24(3), 2005.

[22] Francis H Harlow, J Eddie Welch, et al. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of fluids*, 8(12), 1965.

[23] Alan Hevner, Salvatore March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, 28(1), 2004.

[24] Nathan Holmberg and Burkhard C Wünsche. Efficient modeling and rendering of turbulent water over natural terrain. In *2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*. ACM, 2004.

[25] Andres Iglesias. Computer graphics for water modeling and rendering: a survey. *Future generation computer systems*, 20(8), 2004.

[26] Markus Ihmsen, Jens Orthmann, Barbara Solenthaler, Andreas Kolb, and Matthias Teschner. SPH fluids in computer graphics. In *Eurographics 2014-State of the Art Reports*. The Eurographics Association, 2014.

[27] Clay Mathematics Institution. Millennium problems: Navier-stokes equation. `http://www.claymath.org/millenium-problems/navier%E2%80%93stokes-equation`. Accessed Oct 3, 2014.

[28] Geoffrey Irving, Eran Guendelman, Frank Losasso, and Ronald Fedkiw. Efficient simulation of large bodies of water by coupling two and three dimensional techniques. *ACM Transactions on Graphics (TOG)*, 25(3), 2006.

[29] Jesper Juul. A clash between game and narrative. Master's thesis, University of Copenhagen, 1999.

[30] Michael Kass and Gavin Miller. Rapid, stable fluid dynamics for computer graphics. In *17th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '90. ACM, 1990.

[31] Tadhg Kelly. The seven constants of game design, part two. `http://techcrunch.com/2014/07/27/constants-of-game-design-2/`. Accessed Feb 5, 2015.

[32] Theodore Kim, Nils Thürey, Doug James, and Markus Gross. Wavelet turbulence for fluid simulation. *ACM Transactions on Graphics (TOG)*, 27(3), 2008.

[33] Herbert Krugman. The impact of television advertising: Learning without involvement. *Public opinion quarterly*, 29(3), 1965.

[34] Hyokwang Lee and Soonhung Han. Solving the shallow water equations using 2D SPH particles for interactive applications. *The Visual Computer*, 26(6-8), 2010.

[35] Randall LeVeque. *Finite volume methods for hyperbolic problems*, volume 31. Cambridge University Press, 2002.

[36] Matthew Lombard and Theresa Ditton. At the heart of it all: The concept of presence. *Journal of Computer-Mediated Communication*, 3(2), 1997.

[37] Matthew Lombard, Robert Reich, Maria Grabe, Cheryl Bracken, and Theresa Ditton. Presence and television. *Human Communication Research*, 26(1), 2000.

[38] Miles Macklin and Matthias Müller. Position based fluids. *ACM Transactions on Graphics (TOG)*, 32(4), 2013.

[39] Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Unified particle physics for real-time applications. *ACM Transactions on Graphics (TOG)*, 33(4), 2014.

[40] Marcelo M. Maes, Tadahiro Fujimoto, and Norishige Chiba. Efficient animation of water flow on irregular terrains. In *Computer graphics and interactive techniques in Australasia and Southeast Asia*, GRAPHITE '06. ACM, 2006.

[41] Maria Malik, Teng Li, Umar Sharif, Rabia Shahid, Tarek El-Ghazawi, and Greg Newby. Productivity of GPUs under different programming paradigms. *Concurrency and computation: practice and experience*, 24(2), 2012.

[42] Xing Mei, Philippe Decaudin, and Bao-Gang Hu. Fast hydraulic erosion simulation and visualization on GPU. In *15th Pacific Conference on Computer Graphics and Applications*, PG '07, 2007.

[43] David Mould and Yee-Hong Yang. Modeling water for computer graphics. *Computers and Graphics*, 21(6), 1997.

[44] Eike Mueller, Xu Guo, Robert Scheichl, and Sinan Shi. Matrix-free GPU implementation of a preconditioned conjugate gradient solver for anisotropic elliptic PDEs. *CoRR*, abs/1302.7193, 2013.

[45] Thomas Novak, Donna Hoffman, and Yiu-Fai Yung. Measuring the customer experience in online environments: A structural modeling approach. *Marketing science*, 19(1), 2000.

[46] NVIDIA cuSPARSE library. `https://developer.nvidia.com/cusparse`. Accessed Oct 24, 2014.

[47] NVIDIA FlameWorks. `https://developer.nvidia.com/flameworks`. Accessed Aug 22, 2014.

[48] NVIDIA GeForce GTX Titan specifications. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan/specifications`. Accessed Aug 28, 2014.

[49] James O'Brien and Jessica Hodgins. Dynamic simulation of splashing fluids. In *Computer Animation '95*, 1995.

[50] Jesus Ojeda. *Efficient algorithms for the realistic simulation of fluids*. PhD thesis, FIB, 2013.

[51] Jesus Ojeda and Anton Susín. Enhanced lattice Boltzmann shallow waters for real-time fluid simulations. In *Eurographics 2013-Short Papers*. The Eurographics Association, 2013.

[52] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose computation on graphics hardware. *Computer graphics forum*, 26(1), 2007.

[53] Charles S Peskin. The immersed boundary method. *Acta numerica*, 11, 2002.

[54] Nick Rasmussen, Doug Enright, Duc Nguyen, Sebastian Marino, Nigel Sumner, Willi Geiger, Samir Hoon, and Ron Fedkiw. Directable photorealistic liquids. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA'04, 2004.

[55] Nick Rasmussen, Duc Quang Nguyen, Willi Geiger, and Ronald Fedkiw. Smoke simulation for large scale phenomena. *ACM Transactions on Graphics (TOG)*, 22(3), 2003.

[56] Avi Robinson-Mosher, Tamar Shinar, Jon Gretarsson, Jonathan Su, and Ronald Fedkiw. Two-way coupling of fluids to rigid and deformable solids and shells. *ACM Transactions on Graphics (TOG)*, 27(3), 2008.

[57] Katie Salen and Eric Zimmerman. *Rules of play: Game design fundamentals*. MIT press, 2004.

[58] Christophe Schlick. An inexpensive BRDF model for physically-based rendering. *Computer graphics forum*, 13(3), 1994.

[59] Andrew Selle, Ronald Fedkiw, Byungmoon Kim, Yingjie Liu, and Jarek Rossignac. An unconditionally stable maccormack method. *Journal of Scientific Computing*, 35(2-3), 2008.

[60] Mike Seymour. Assassins creed III: The tech behind (or beneath) the action. `http://www.fxguide.com/featured/assassins-creed-iii-the-tech-behind-or-beneath-the-action/`. Accessed: May 14, 2014.

[61] Miguel Sicart. Defining game mechanics. *Game Studies*, 8(2), 2008.

[62] B. Solenthaler, P. Bucher, N. Chentanez, M. Müller, and M. Gross. SPH based shallow water simulation. In *Workshop in Virtual Reality Interactions and Physical Simulation*, VRIPHYS'11, 2011.

[63] Jos Stam. Stable fluids. In *26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99. ACM, 1999.

[64] O-Beneš Šťava, B Brisbin, and M Křivánek. J.: Interactive terrain modeling using hydraulic erosion. In *Eurographics Symposium on Computer Animation 2008 (SCA'08)*, 2008.

[65] Jonathan Steuer. Defining virtual reality: Dimensions determining telepresence. *Journal of communication*, 42(4), 1992.

[66] Penny Sweetser. *Emergence in games*. Cengage Learning, 2008.

[67] Tsunemi Takahashi, Heihachi Ueki, Atsushi Kunimatsu, and Hiroko Fujii. The simulation of fluid-rigid body interaction. In *ACM SIGGRAPH 2002 conference abstracts and applications*. ACM, 2002.

[68] Jie Tan and XuBo Yang. Physically-based fluid animation: A survey. *Science in China Series F: Information Sciences*, 52(5), 2009.

[69] Jerry Tessendorf. Simulating ocean water. *Simulating Nature: Realistic and Interactive Techniques. SIGGRAPH 2001 course notes*, 2001.

[70] Jerry Tessendorf. Interactive water surfaces. In *Game Programming Gems 4*. Charles River Media, 2004.

[71] N. Thürey, F. Sadlo, S. Schirm, M. Müller-Fischer, and M. Gross. Real-time simulations of bubbles and foam within a shallow water framework. In *2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '07, 2007.

[72] Nils Thürey, Matthias Müller-Fischer, Simon Schirm, and Markus Gross. Real-time breaking waves for shallow water simulations. In *15th Pacific Conference on Computer Graphics and Applications*, PG '07, 2007.

[73] Nils Thürey, Ulrich Rüde, and Marc Stamminger. Animation of open water phenomena with coupled shallow water and free surface simulations. In *2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '06, 2006.

[74] Kevin R Tubbs and Frank T-C Tsai. GPU accelerated lattice Boltzmann model for shallow water flow and mass transport. *International Journal for Numerical Methods in Engineering*, 86(3), 2011.

[75] Alex Vlachos. Water flow in Portal 2. In *ACM SIGGRAPH 2010 courses*, SIGGRAPH '10. ACM, 2010.

[76] Chuck Walbourn. Living without D3DX. `http://blogs.msdn.com/b/chuckw/archive/2013/08/21/living-without-d3dx.aspx`. Accessed Oct 24, 2014.

[77] Chin-Chih Wang, Jia-Xiang Wu, Chao-En Yen, Pangfeng Liu, and Chuen-Liang Chen. Ocean wave simulation in real-time using GPU. In *International Computer Symposium*, ICS 2010, 2010.

[78] David Weibel and Bartholomäus Wissmath. Immersion in computer games: The role of spatial presence and flow. *International Journal of Computer Games Technology*, 2011, 2011.

[79] Werner Wirth, Tilo Hartmann, Saskia Böcking, Peter Vorderer, Christoph Klimmt, Holger Schramm, Timo Saari, Jari Laarni, Niklas Ravaja, Feliz Ribeiro Gouveia, et al. A process model of the formation of spatial presence experiences. *Media Psychology*, 9(3), 2007.

[80] Bob Witmer and Michael Singer. Measuring presence in virtual environments: A presence questionnaire. *Presence: Teleoperators and virtual environments*, 7(3), 1998.

[81] Mark Wolf. Space in the video game. In *The medium of the video game*. University of Texas Press, 2001.

[82] Cem Yuksel, Donald H. House, and John Keyser. Wave particles. *ACM Transactions on Graphics (TOG)*, 26(3), 2007.

[83] Jian Guo Zhou. *Lattice Boltzmann methods for shallow water flows*. Springer, 2004.

[84] Yongning Zhu and Robert Bridson. Animating sand as a fluid. *ACM Transactions on Graphics (TOG)*, 24(3), 2005.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland

ISBN 978-952-15-3643-4
ISSN 1459-2045