REAL-TIME WATER WAVES WITH WAVE PARTICLES

A Dissertation

by

Cem Yuksel

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

August 2010

Major Subject: Computer Science

REAL-TIME WATER WAVES WITH WAVE PARTICLES

A Dissertation

by

Cem Yuksel

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Co-Chairs of Committee,	John Keyser
	Donald H. House
Committee Members,	Scott Schaefer
	Ergun Akleman
Head of Department,	Valerie E. Taylor

August 2010

Major Subject: Computer Science

ABSTRACT

Real-time Water Waves with Wave Particles. (August 2010)

Cem Yuksel, B.S.; M.S., Bogazici University, Turkey

Co–Chairs of Advisory Committee: Dr. John Keyser
Dr. Donald H. House

This dissertation describes the wave particles technique for simulating water surface waves and two way fluid-object interactions for real-time applications, such as video games.

Water exists in various different forms in our environment and it is important to develop necessary technologies to be able to incorporate all these forms in real-time virtual environments. Handling the behavior of large bodies of water, such as an ocean, lake, or pool, has been computationally expensive with traditional techniques even for offline graphics applications, because of the high resolution requirements of these simulations.

A significant portion of water behavior for large bodies of water is the surface wave phenomenon. This dissertation discusses how water surface waves can be simulated efficiently and effectively at real-time frame rates using a simple particle system that we call "wave particles." This approach offers a simple, fast, and unconditionally stable solution to wave simulation. Unlike traditional techniques that try to simulate the water body (or its surface) as a whole with numerical techniques, wave particles merely track the deviations of the surface due to waves forming an analytical solution. This allows simulation of seemingly infinite water surfaces, like an open ocean.

Both the theory and implementation of wave particles are discussed in great detail. Two-way interactions of floating objects with water is explained, including

generation of waves due to object interaction and proper simulation of the effect of water on the object motion. Timing studies show that the method is scalable, allowing simulation of wave interaction with several hundreds of objects at real-time rates.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

LIST OF FIGURES

FIGURE Page

FIGURE Page

FIGURE Page

CHAPTER I

INTRODUCTION

In computer graphics, fluid simulation, particularly of water, is one of the major areas of research focus. Simulating large bodies of water (such as seas, lakes, and pools) has been especially important since they appear in various virtual environments and it is difficult to achieve realism in such environments without proper simulation of water behavior.

With improvements over the past decade, offline water simulation in computer graphics has reached a remarkable level of realism. Existing techniques for offline simulation are powerful enough to simulate a wide variety of fluid behavior and they can handle simulations in large environments as long as sufficient computation power is provided. Nowadays such simulations are commonly used in the visual effects and feature animation industries. When simulating large bodies of water, however, effects artists often employ ad-hoc techniques or unrealistically simplified methods whenever possible, since a full 3D fluid simulation can be too expensive to be used everywhere. Moreover, full 3D fluid simulations are not easy to control and it may be very difficult, if even possible, to drive the simulation towards an expected result. Therefore, one

---

This dissertation follows the style of *ACM Transactions on Graphics*.

may have to run many simulations to achieve a single acceptable result that would be close enough to the desired outcome. For these reasons, there is still room for improvements in offline fluid simulations in computer graphics. In particular, better distribution of computational resources to reduce simulation time while minimizing the loss in visual quality is still an important area of research.

While offline simulations of water can produce stunning visuals, achieving a similar level of realism at real-time frame rates remains an open challenge. This is because real-time graphics applications have some additional challenges over offline graphics. These challenges can be summarized as follows:

- **Speed Requirements:** The most obvious challenge is to make fluid simulations fast enough so that the simulation can run at real-time frame rates. While the definition of *real-time frame rates* varies (some researchers consider 15 fps real-time, others require at least 60 fps), for research purposes the exact definition is often unimportant, since we expect any implementation to run significantly faster on the new hardware that will come out within a few years. However, it is important to note that in many environments water plays a secondary role, so we cannot dedicate all our computational resources to the simulation of water. In that sense, we not only need to satisfy the minimum requirements of real-time frame rates, but also go beyond these requirements, so that these simulations can be used in practical interactive applications.

- **Stability Requirements:** The need for stability of the simulation is more pronounced in real-time environments compared to offline simulations. Offline simulations in computer graphics are primarily used to generate a sequence of images for a particular shot in a film, which is often a few seconds long and

almost always much less than a minute. Therefore, the minimum expectation from an offline simulation is that it should work properly at least once with a certain set of parameters to produce the desired result. The parameters of the simulation can be adjusted until the offline simulation produces acceptable results. On the other hand, real-time simulations used in video games or other interactive systems may run for hours, and these simulations are expected to be stable for the entire time. Note that the stability requirement here is not simply the stability of the software, but the stability of the underlying mathematical and computational methods that are expected to produce visually acceptable fluid behavior.

- **User Interaction:** User interaction is an inherent component of all real-time graphics applications. The need for supporting arbitrary user interactions with the simulation in real-time systems makes it particularly difficult to fine tune the parameters of the simulation. Real-time simulations of water are expected to work with a wide range of possible user interactions using predetermined simulation parameters. On the other hand, offline scenarios are mostly fixed for a particular shot and the parameters of the simulation can be adjusted according to the desired water behavior for that shot, as opposed to having one set of parameters for handling any possible interaction.

- **Hardware Limitations:** For many fluid simulations involving large bodies of water, offline simulations can run on a large cluster of computers. Real-time water simulations, however, are expected to run on an average PC or a game console with limited memory and other computational resources. While the parallel power of the graphics hardware provides significant acceleration to all graphics related computation, hardware parallelism of a modern PC or

a game console is not sufficient to provide the necessary computational power to overcome the demands of realistic water simulations. Furthermore, in real world applications, the computational resources that can be allocated to water simulation alone is often only a small portion of the whole hardware system. Therefore, expecting the same level of realism from a real-time implementation of a computationally demanding simulation is simply not reasonable.

For all these reasons, achieving realistic real-time water simulations that would match the quality of their offline counterparts remains (and perhaps will remain) an open challenge. However, many researchers and graphics programmers have implemented different techniques to enable better water simulations in real-time environments. Such implementations mainly employ one or more of the following approaches:

- The simplest approach is to replace water simulation with some ad hoc mathematical formulation to animate a 3D surface such that the resulting motion resembles water behavior. These techniques are extremely limited in terms of the possible scenarios they can handle, and often fail to produce realistic results. However, these techniques are still used to avoid the computational expense of fluid simulations in various scenarios. While ad-hoc formulations can be acceptable for certain applications, such approaches alone cannot come close to achieving the quality of offline simulations.

- Another common approach is to use the same water simulation techniques as for offline graphics but with parameters chosen to reduce the computation requirements. Such approaches often generate low-resolution solutions with higher error ratios than their offline counterparts. It is also possible to replace certain steps of the water simulation with less accurate but more efficient alternatives.

As expected, these simulations produce faster but significantly lower quality results.

- Employing the parallel power of the graphics hardware is another approach. With the new advancements in real-time graphics hardware, many researchers and graphics programmers have modified existing offline fluid simulation techniques such that they can be executed on the GPU. While very impressive results can be achieved with this approach, the scenarios that can be simulated in real-time frame rates are very limited. Furthermore, the parallelism provided by modern GPUs can only account for a few orders of magnitude improvement in the best case. Therefore, we cannot rely on the hardware parallelism that is available on a modern PC or a game console alone to achieve desired performance levels with high quality results.

- A promising alternative is to use precomputation, which is a technique commonly used for many other problems in real-time graphics. Precomputation can be a very powerful tool when used in combination with other approaches to replace computational requirements of certain tasks with additional memory requirements. While such an approach has been investigated for fluid simulation of gases and smoke, whether or not precomputed water simulation, where the water surface needs to be represented, can be achieved using similar approaches is an open question.

- Another alternative approach is to provide solutions for a subset of water behavior, rather than aiming for a complete solution that would cover all water effects. Such methods can concentrate only on water waves, splashes, bubbles, and various other water related natural phenomena. When needed, a more complete solution can be achieved by combining a number of such algorithms.

It is the premise of this dissertation that this latter approach of limiting the solution domain of the simulation, is the key to achieving real-time results comparable to offline fluid simulations. In real-time environments, a particular water simulation is often employed to imitate a certain water behavior only. In that sense, a real-time water simulation can be considered successful as long as it provides visually realistic solutions to the desired behavior, and other water phenomena the simulation can offer are irrelevant. Concentrating on only a subset of water behavior makes it possible to introduce simpler physical models for water simulation that can be computed more efficiently.

The wave particles technique described in this dissertation is an example of such an approach, as it is designed to simulate only surface waves on large bodies of water. Even though the water behavior that wave particles can simulate is limited to surface waves, within its solution domain wave particles manage to produce high quality results with very high computational efficiency. Therefore, the wave particles method is ideal for simulating dynamic water waves in many real-time scenarios.

1.1. Motivation of Wave Particles

The main motivation behind the wave particles approach is to propose a water simulation technique that captures important visual components of a subset of water behavior, while producing high quality visuals with very efficient computation within its solution domain. The targeted water phenomenon is the wave behavior induced by dynamic interaction of water with fixed or floating objects. In that sense, water-object interaction is an inherent part of the wave particles approach.

While existing full 3D fluid simulations are powerful enough to handle almost any water behavior, their high computational requirements make them too expensive for real-time applications. Especially when the task is to simulate large bodies of water, full 3D simulations can be too expensive even for offline applications. Furthermore, while it is possible to compute water-object interaction within some modern 3D fluid simulators, doing so significantly increases the computational demands.

While examining water-object interaction with large bodies of water, one can observe a number of visually important components of water behavior. These are splashes, bubbles, foam, and waves. Another component that is often not visually apparent is the 3D fluid flow and turbulence under the water surface. The first simplification that can be introduced at this stage is ignoring water behavior that is not visually apparent. In that sense, 3D fluid flow under the water surface, which is generally rather expensive to compute, can be eliminated. One important advantage of introducing such simplifications at an early stage of building the theoretical simulation model is that the limitations introduced by this elimination become easy to identify. In this case, we should not expect a method that ignores 3D fluid flow to properly handle scenarios where the 3D fluid flow is a significant factor in driving visually identifiable water behavior. An example of such a scenario would be simulating vortices formed by a sinking ship. On the other hand, the method might be very good at representing the persistent waves generated by the same event.

As for the visually apparent components, for the most part splashes, bubbles, and foam are mainly the result of chaotic dynamics. For this reason, statistically derived ad-hoc formulations used to emulate splashes, bubbles, and foam can produce reasonable approximations to these phenomena. On the other hand, the wave behavior

of water due to interaction with floating objects is not only visually significant, but also generally predictable. For this reason, ad-hoc formulations that emulate wave behavior often fail to produce visuals with acceptable realism. Therefore, the wave behavior of water should be simulated properly to achieve high quality visuals.

If we are to consider surface waves only, we can build our simulation model based on the wave equation, which is much simpler than the Navier-Stokes equations that formulate general fluid behavior. Since the wave equation is a differential equation, the first idea that comes to mind for solving the wave equation would be finite difference techniques, which are commonly used in fluid simulations to solve other differential equations, such as variants of the Navier-Stokes equations. However, the simplicity of the wave equation makes it possible to envision simpler computational models to estimate the result of the wave equation solution. The wave particles approach is one such model.

To be able to construct such a model, we need to examine the wave equation closely. After careful examination, one can see that the wave equation inherently formulates a moving waveform. Therefore, the solution of the wave equation for water surface waves can be approximated by moving deformations on the water surface. This is exactly what wave particles aim to do.

1.2. Summary of Wave Particles

Based on the idea that a collection of moving waveforms provide a solution to the wave equation, the wave particles technique uses a particle system to track the mo-

tion of such waveforms. This simple particle system is composed of particles that move on the flat water surface. Each particle is associated with a certain surface deformation, and the total deformation caused by a collection of wave particles produces a waveform traveling over the water surface. Wave particles move on the water surface, reflect from boundaries, and subdivide into smaller wave particles. While a single wave particle on a flat water surface does not constitute a valid solution to the wave equation, the collective behavior of wave particles provide an analytical approximation to the wave equation.

Our careful formulation of wave particles permits considering each wave particle independently from all other wave particles that co-exist in the particle system. This is a very important property of wave particles that distinguishes it from other particle based water simulation approaches in computer graphics.

Since the behavior of each wave particle can be computed independently, the wave particle system can be simulated very efficiently and can be parallelized very easily. By employing the parallel computation power of modern GPUs, wave particles can provide a fast simulation method for surface waves on large bodies of water. More specifically, simulation speeds above 300 frames per second can be achieved with interacting objects and complicated wave behavior.

Wave particles provide an approximate analytical solution to the wave equation rather than a numerical solution. Therefore, the wave simulation with wave particles is unconditionally stable and it does not include any stability issues numerical approaches suffer from. The properties of any wave particle, at any given time, can be computed directly and accurately without the need for numerical iteration.

In addition to computational efficiency and stability, wave particles also have other important features that make them highly desirable for both offline and real-time water simulations. Perhaps the most important additional contribution of wave particles is that they enable the simulation of wave behavior on extremely large bodies of water. Since wave particles represent dynamic deformations due to dynamically induced surface waves, as opposed to the water body itself, the volume of water is not a limiting factor for wave particles. The performance of wave particles is determined by the number of concurrent interactions with water that generate waves, but the size of the water surface has no effect on the performance. This makes the wave particles technique a desirable choice for simulating waves on an open ocean or a large pool.

Furthermore, wave particles permit high level control over the simulation result, which is a very desirable property when the surface waves are expected to behave in a certain way for artistic reasons. Since the wave particles essentially form a 2D particle system on a flat water surface, all one needs to do to modify the result of the simulation is to move particles around or change their properties as desired. Unlike other fluid simulation techniques, the properties of wave particles directly shape the water surface. Therefore, it is very easy to predict the outcome of any modification to the wave particle system.

1.3. Overview

This dissertation explains how the wave particles method can be used for real-time water simulation with two-way object interactions using a simple 2D particle system. After an overview of the related work in computer graphics and the theory of water

waves discussed in Chapter II, we present the theory of the wave particles method in Chapter III. Handling the interaction of water with dynamic objects in a wave particle simulation system is discussed in Chapter IV. One important property of wave particles is that it permits highly efficient implementation for a water simulation system. The details of our implementation is provided in Chapter V. After presenting our results in Chapter VI, we discuss advantages and limitations of the wave particles method along with possible future extensions in Chapter VII.

CHAPTER II

RELATED WORK ON WATER SIMULATION

There is a large body of work on simulating water and other fluids in computer graphics. In this chapter we briefly overview some of these methods.

2.1. Explicit Surface Modeling

Early work on water simulation concentrated on directly modeling the water surface and its animation due to surface waves. These methods are mostly based on trigonometric formulations, noise functions, or Fourier synthesis. One of the first attempts for mathematically modeling the water surface was made by Schachter [1980], representing the water surface using narrow-band noise waveforms. Fournier and Reeves [1986] used a simple parametric formulation of ocean waves that could produce breaking wave shapes. Another parametric model was proposed by Peachy [1986] for water waves approaching a sloping beach. Ts'o and Barsky [1987] introduced wave tracing, a method for tracing wave propagation directions. Using a Fourier synthesis method based on an empirical spectrum model of real sea waves Mastin et al. [1987] generated water surfaces for ocean scenes. More recently, Schneider and Westermann [2001] used graphics hardware for real-time simulation of waves using a noise function [Perlin and Hoffert 1989]. While these methods achieve realistic results, they are not applicable for dynamic simulations of two-way interactions of water with floating

objects.

The ocean wave simulation method in Tessendorf's SIGGRAPH course notes [2001] was also based on Fourier synthesis that depicts the behavior of real ocean waves. This method is one of the most popular methods for simulating ambient ocean waves. For handling interactions with the water waves and obstacles in waves, Tessendorf [2004] proposed a simple convolution filter that simulates waves reflecting off of arbitrary obstacles. This approach permitted fast computation of one-way interactions between the water waves and obstacles in water.

2.2. Height Field Simulations

A height field is essentially a 2D image, each pixel of which keeps a floating point value that determines the height of the water surface for the horizontal position that corresponds to the pixel. Height field simulations of water are commonly used for fast simulations of the water surface. Kass and Miller [1990] used finite differences to solve simplified 2D shallow water equations over a dynamic height field. Chen and Lobo [1995] used a pressure-defined height field arising from a 2D solution of the Navier-Stokes equations. They proposed a way of handling two-way object to fluid coupling; however, their implementation is limited to one-way coupling only. Extending the approach of [Kass and Miller 1990], O'Brien and Hodgins [1995] added a particle system to simulate splashing liquids. An interesting use of height field fluid simulation was proposed by Baxter et al. [2004] for interactive watercolor painting.

Height field simulations provide a relatively fast way of simulation water as compared

to full 3D water simulations. On the other hand, they are limited to height field representations of the water surface, so they only handle the vertical motion of the water surface. Furthermore, numerical simulations based on height fields are prone to either instability or numerical dissipation, which introduces excessive damping to the simulation.

2.3. Lagrangian Fluid Simulations

Lagrangian fluid simulations use 3D particle systems [Reeves 1983] for simulating fluids. Miller and Pearce [1989] used a particle system for simulating deformable objects and viscous fluids by applying interaction forces to nearby particle pairs. These forces are strongly repellent when the two particles are too close to each other, and they are weakly attractive when the particles are some distance apart. Terzopoulos et al. [1989] proposed a method for simulating molecular dynamics for melting solids with Lagrangian particles.

Müller et al. [2003] used the smoothed particle hydrodynamics (SPH) method [Lucy 1977; Monaghan 1977] that was introduced to computer graphics by Stam and Fiume [1995] for simulating water. The SPH method models the fluid volume as a collection of particles that interact with each other through hydrodynamic forces. While the SPH method is very flexible and can be applied to simulating various fluid related phenomena, it cannot enforce incompressibility; therefore, it is not ideal for incompressible (or nearly incompressible) fluids like water. Premoze et al. [2003] used the moving particle semi-implicit method [Koshizuka S. 1996] for introducing incompressibility to particle based simulation of water. To achieve incompressibility

with , Sin et al. [2009] proposed a Voronoi diagram based projection step for keeping the velocity field divergence free. Recently, Solenthaler and Pajarola [2009] proposed a predictive correction method for SPH that accounts for a large portion of incompressibility, thereby allowing weakly compressible SPH simulations. An interesting recent approach was proposed by Lenaerts et al. [2008] for simulating porous flow in deformable solids with SPH.

Lagrangian methods can produce high frame rates when a relatively small number of particles are used for simulating a relatively small volume of water. On the other hand, for simulating relatively larger bodies of water, a large number of particles are needed and computing the interactions among a large number of particles can be computationally demanding. Adams et al. [2007] and Hong et al. [2008] proposed adaptive models for reducing the number of particles with minimal quality loss. These models merge neighboring particles inside the fluid volume to reduce the number of particles, and split the particles wherever more detail is needed.

2.4. Eulerian Fluid Simulations

There is a large body of work on Eulerian grid-based solutions to fluid simulations in computer graphics. These methods offer a numerical solution to the Navier-Stokes equations that define the fundamentals of fluid motion. Eulerian methods offer some form of spatial discretization or partitioning of the computation volume into a collection of cells. Using this discretization, the fluid velocity is computed at discrete locations in this space based on the Navier-Stokes equations and this velocity field is used for advecting the fluid volume at each time step. Accurate numerical so-

lutions to the Navier-Stokes equations appear in computational fluid dynamics. In computer graphics, accuracy is often less important than computational efficiency; therefore, Eulerian fluid simulations in computer graphics concentrate on efficient and approximate solutions to the Navier-Stokes equations. Nonetheless, even these highly simplified solutions to the Navier-Stokes equations are computationally demanding.

Eulerian fluid simulations were first introduced to computer graphics by Foster and Metaxas [1996]. In this method they included a one-way water-object interaction by advecting the floating objects using the fluid velocity. Foster and Metaxas [1997] also proposed a method for controlling water animation within an Eulerian simulation.

The seminal work in this area was the stable fluids method [Stam 1999] that enabled a relatively efficient and unconditionally stable solution to the Navier-Stokes equations using semi-Lagrangian advection. Most Eulerian fluid simulation methods proposed afterwards use some variation of the stable fluids approach. Unfortunately, the stability and efficiency of the stable fluids method also leads to excessive dissipation, resulting in highly damped simulations.

Foster and Fedkiw [2001] proposed a similar Eulerian simulation technique that permitted moving objects to properly affect the fluid simulation. Enright et al. [2002] introduced the particle level set method that uses massless tracking particles for deforming an implicit representation of the water surface. Extending the particle level set method, Losasso et al. [2006] developed a representation for simulating multiple interacting liquids. Selle et al. [Selle et al. 2005] proposed a vortex particle method for reintroducing small scale detail lost due to numerical dissipation. Zhu and Bridson [2005] developed a hybrid Eulerian-Lagrangian method for simulating sand as fluid.

Researchers also proposed methods for reducing the number of cells to accelerate the Eulerian fluid simulations. Losasso et al. [2004] proposed an octree structure for using smaller cells only when needed, thereby reducing the total number of cells for achieving the same level of detail. Houston et al. [2006] proposed an Eulerian simulation with run length encoding to reduce the memory footprint of the computation. Irving et al. [2006] presented a technique for simulating large bodies of water with object to fluid coupling using the combination of a Navier-Stokes based fluid solver and a height field formulation. This structure produced tall grid cells to reduce the computational requirements for simulating large bodies of water with Eulerian methods. In addition to these grid-based discretizations of the computation volume, tetrahedral meshes have been used for fluid simulation of gases [Feldman et al. 2005; Klingner et al. 2006] as well as liquids [Chentanez et al. 2007].

One-way fluid to object coupling is used by many researchers is in the form of boundary conditions to the Navier-Stokes equations [Foster and Metaxas 1996; Foster and Metaxas 1997; Fedkiw et al. 2001; Enright et al. 2002]. A number of researchers proposed different methods for handling two-way interactions between water and objects. Takahashi et al. [2002] developed an impulse based collision system for simulating two-way interactions between rigid objects and the fluid surface. Fedkiw [2002] proposed the ghost fluid method to couple deformable solids with compressible fluids, and Takashi et al. [2003] proposed the immersed boundary method for coupling deformable solids with incompressible fluids. Genévaux et al. [2003] also proposed a method for allowing deformable solids to interact with fluids using fluid marker particles, which are attached to nearby solids via springs. The rigid fluid method proposed by Carlson et al. [2004] handled two-way interactions between rigid objects and fluids by advecting rigid objects as fluids and maintaining the rigidity of interacting objects

by constraining the velocity of the cells that are inside the rigid objects. Guendelman et al. [2005] proposed a method for handling two-way interactions of thin shells and liquids, which prevents leaking of the liquid through the thin shell. Another method for simulating two-way interactions of thin shells and fluids is proposed by Robinson-Mosher et al. [2008], which resolves the stability issues of [Guendelman et al. 2005] in a more general framework. More accurate tangential fluid flow near fluid-object boundaries is computed by Robinson-Mosher et al. [2009] by decoupling normal and tangential fluid velocities and constraining the normal velocity. Eulerian fluid simulations with tetrahedral meshes are also used for two-way interactions of fluid and rigid bodies [Klingner et al. 2006; Batty et al. 2007] as well as deformable objects [Chentanez et al. 2006].

While the Eulerian simulations of water can produce impressive results, these methods are generally too computationally intensive for most real-time simulations. Real-time implementations of such techniques can only handle limited simulation volumes with limited simulation resolution and limited accuracy.

2.5. Real-time Fluid Simulations

There are also methods specifically targeted for real-time graphics applications. Jensen and Goiáš [2001] used grid based Eulerian methods to model deep ocean waves, including the effect of fluid on objects, but their computation of the effect of objects on the fluid involves numerical differencing and the addition of artificial damping, so we expect it to be highly sensitive to parameter tuning. Hagen et al. [2005] simulated nonlinear shallow-water waves on the GPU and achieved up to 30 times speed up as

compared to a CPU implementation. Kim et al. [2006] used the GPU to compute buoyant forces on arbitrary models, achieving one-way fluid on object interactions in real-time. They achieved 16 frames per second with 50 floating objects.

A CPU implementation of an Eulerian Navier-Stokes solver [Stam 1999] can easily achieve real-time frame rates for low resolution 2D simulations. The GPU implementations of Eulerian fluid simulation techniques can achieve about two orders of magnitude speed up, typically using less accurate advection schemes. Harris [2004] demonstrated how a 2D simulation of Navier-Stokes equations for incompressible flow can be implemented on the GPU. Crane et al. [2007] presented a full 3D simulation of water on the GPU, achieving around 120-180 frames per second at a grid resolution of $64 \times 64 \times 128$ on a GeForce 8800 GTX graphics hardware. Cords [2007] proposed a method for separating water surface simulation into two components: one of them simulates low-frequency fluid flow using a 3D fluid solver, while the other component simulates high frequency surface waves using a 2D height field simulation of surface waves. In this method, the low-resolution 3D simulation efficiently produces a rough water motion and the details of the water surface are introduced by the 2D wave simulation.

Lately, Lagrangian simulations of water has been popular for real-time applications. Following the SPH approach of Müller et al. [2003] for interactive applications, Clavet et al. [2005] implemented a Lagrangian simulation of viscoelastic fluids and achieved interactive frame rates. Using the parallel computation power of the modern GPUs, high frame rates can be achieved with Lagrangian fluid simulations [NVIDIA 2010].

One very interesting approach for real-time simulation of fluids is the model reduction

method of Treuille et al. [2006]. This method precomputes a large collection of Eulerian fluid simulations. Then, the solutions of all these simulations are projected onto a low-dimensional solution space using PCA. At run time the simulation works on this low-dimensional space and projects the solution to the desired resolution. This permits high resolution simulations of fluids at real-time frame rates. On the down side, the precomputation time of this method can be extremely long. Furthermore, it can only handle simulation scenarios that are "trained in" and can be represented by the low-dimensional model. Moreover, this method is developed for simulating gases, so it is questionable whether there is a way to develop a similar technique for liquid simulations as well.

CHAPTER III

WATER SIMULATION WITH WAVE PARTICLES

In this chapter we explain the wave particles technique for handling larger bodies of water in real-time virtual environments, which was first published in [Yuksel et al. 2007]. Before going into the details of the wave particles method, we will talk about the motivation behind this technique through a visual and technical analysis. These are presented in the next two sections and detailed explanations of wave particles are provided in the subsequent sections.

3.1. Visual Analysis of Large Bodies of Water

The aim of visual analysis is to observe the behavior of larger bodies of water and identify the visually significant components. Visual analysis is the first step of building an efficient technique for handling large bodies of water in real-time virtual environments. While it might be possible to develop a method to handle all water related phenomena, such comprehensive solutions would be complicated and inefficient. Since computational efficiency is of prime importance for any technique that is targeted for real-time graphics applications, it may be preferable to limit the scope of the water related phenomena that a method can depict to make it more efficient. Visual analysis helps us understand how the complicated overall behavior can be separated into various components and which of these components deserve more attention than

others.

Visually examining the behavior of large bodies of water, one can easily identify various distinct components. The ones we consider here are the following:

- Splashes
- Bubbles
- Foam
- Surface deformations

Of these four components, surface deformations have been the most difficult to adequately handle in real-time applications. While splashes, bubbles, and foam are chaotic in nature, and thus lend themselves to representation by techniques with stochastic formulations, surface deformations require more deterministic and computationally expensive methods. Furthermore, the most noticeable and recognizable large scale behavior of large bodies of water is due to surface deformation in one form or another. Therefore, it is important to have efficient algorithms to properly handle surface deformations.

Most surface deformations on large bodies of water are due to surface waves. In fact, all other deformations are highly unstable and they quite rapidly evolve into surface waves. Surface waves, however, are highly stable and they can travel very long distances if their amplitudes are high enough. Therefore, it is important to simulate surface waves for handling large bodies of water.

In a real-time virtual environment, surface waves can be classified into two groups:

ambient waves and interaction waves. Ambient waves are waves that exist in the environment, such as ocean waves in an ocean scene or subtle motion of water in a still pool. These waves are generally wind induced. Regardless of their actual source, ambient waves can be treated as the natural rest state of the system, and they can be properly modeled by statistical formulations and even precomputed. On the other hand, interactive waves are induced by locally applied ourside forces, such as those due to the interaction of water with floating objects. These are the waves that need to be simulated in real-time, since in an interactive environment these are often generated by direct or indirect user interaction, so that precomputation is not possible. The wave particles method described in this section aims to simulate these interactive waves only.

3.2. The Wave Equation

The wave particles method provides a discrete analytical solution to the wave equation. Before we begin a discussion of the details of wave particles, it is important to analyze and understand the wave equation itself. For a transverse wave, the wave equation can be written as

$$\nabla^2 Z = \frac{1}{v^2} \frac{\partial^2 Z}{\partial t^2} \ , \tag{3.1}$$

where $Z(\mathbf{x}, t)$ is the wave height at position $\mathbf{x} = (x, y)$ and time t, and v is the constant wave propagation speed for the medium. According to this equation, the second spatial derivative of the function Z is directly proportional to the second time derivative of Z by a constant factor determined by the wave propagation speed v. In other words, this equation means that the change of Z in time can be determined by the shape of the function Z.

Furthermore, it can be easily shown that the wave equation permits wave superposition. Let Z_1 and Z_2 be two functions that satisfy the wave equation above. Then, the superposition Z_3 of these two functions, such that $Z_3 = Z_1 + Z_2$, also satisfies the wave equation. This means that different groups of waves can be simulated independently, and the final result can be obtained as the superposition of all groups. We will use this superposition property for completely separating ambient waves from interactive waves, and the final result of the simulation will be the superposition of these two groups of waves.

A differential equation in this form can be easily solved using one of several numerical integration techniques. However, our aim is to provide an analytical solution, which calls for a better understanding of the wave equation. In the following two subsections we first discuss the wave equation in 1D, and then we discuss the wave equation in 2D, which corresponds to surface waves in 3D.

3.2.1. 1D Wave Equation

A transverse wave in 2D is essentially a 1D wave, since the other dimension is used for the wave amplitude. Figure 3.1 shows a 1D wave on the $x - z$ plane that is centered at point x_0 at time t_0 with amplitude a and moving in the positive x direction. In this case the wave equation 3.1 can be written as

$$\frac{\partial^2 Z}{\partial x^2} = \frac{1}{v^2}\frac{\partial^2 Z}{\partial t^2} \, . \tag{3.2}$$

By closely examining this equation, it is easy to see that any waveform function Z with a constant shape and moving in either the positive or negative x direction with

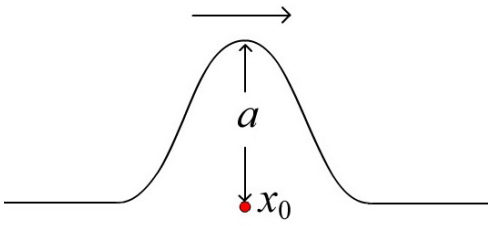


Figure 3.1. A 1D transverse wave in 2D.

constant speed v is a solution to this equation. The general solution to this function is in the form

$$Z(x,t) = F(x - vt) + G(x + vt) \ , \tag{3.3}$$

where F and G are two traveling functions[1] with constant speed v in negative and positive x directions respectively. Note that F and G themselves can in turn be the linear sums of other functions travelling with the same velocity as F and G. This means that any traveling function with any shape is a solution to the wave equation, as well as linear combinations of such functions.

3.2.2. 2D Wave Equation

In 3D, water surface waves propagate on the 2D $x - y$ plane and the wave equation takes the form

$$\frac{\partial^2 Z}{\partial x^2} + \frac{\partial^2 Z}{\partial y^2} = \frac{1}{v^2}\frac{\partial^2 Z}{\partial t^2} \ . \tag{3.4}$$

[1]A traveling function is any arbitrary function f_1 of position x and time t such that it can be written in the form $f_1(x,t) = f_2(x + vt)$, where $f_2(x)$ has a constant value for all x.

This extra dimension over 1D waves makes the wave propagation significantly more complicated. Unlike 1D waves that could only propagate in two distinct directions, surface waves can propagate in an infinite number of directions on the water surface. Surface waves can not only travel in an infinite number of directions, but also can be expanding and contracting. Thus, surface waves can form much more complicated and perhaps more interesting functions as compared to the traveling constant waveforms of 1D waves.

Wave particles are essentially used for tracking wave propagation on the 2D water surface as a discrete solution to the wave equation. The following sections provide a detailed explanation of the wave particles technique.

3.3. Representing Waves with Particles

The wave particles method uses a particle system for representing the dynamic deviation of the water surface. Each wave particle is assigned a *local deviation function* D that determines the deviation of the water surface around the wave particle. For transverse waves we can represent the wave height function Z as the sum of all local deviation functions, such that

$$Z(\mathbf{x},t) = z_0 + \eta_z(\mathbf{x},t) \text{ , and} \tag{3.5}$$

$$\eta_z(\mathbf{x},t) = \sum_i D_i(\mathbf{x},t) \text{ ,} \tag{3.6}$$

where z_0 is the rest height of the water surface when there are no waves, η_z is the deviation caused by waves, and D_i is the deviation represented by the i^{th} wave particle. The position and propagation direction of the wave particle, along with a number of

other wave particle properties, define the shape and behavior of its corresponding local deviation function. This formulation converts the wave simulation to a simple 2D particle system. Note that the motion of the surface deviation caused by each individual D_i does not have to satisfy the wave equation on its own; however, the wave height function Z should be a valid solution to the wave equation.

For the sake of simplicity we begin explaining wave particles in 2D and then we discuss how this 2D definition can be extended to 3D.

3.4. Wave Particles in 2D

In two dimensions (one dimension plus height), we formulate the local deviation function such that it corresponds to a finite wave form traveling with constant speed v, thus satisfying the wave equation. Letting a_i be the amplitude, W a constant *waveform function* and $x_i(t)$ the particle's position at time t, the local deviation function for particle i is

$$D_i(x,t) = a_i \, W\left(\, x - x_i(t) \, \right) \ . \tag{3.7}$$

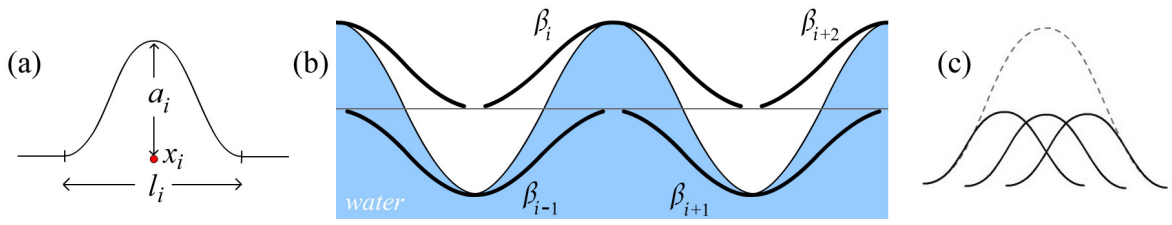


Figure 3.2. (a) Shape of waveform function, (b) Continuous waves constructed from local deviation functions, (c) Grouping waves

As we discussed earlier, satisfying the wave equation in 2D is not difficult at all; any constant waveform that travels with constant speed (i.e. $x_i(t) = x_i(0) + vt$) satisfies Equation 3.2. Therefore, the choice for the waveform function is somewhat arbitrary. However, some functions can be more beneficial than others in terms of implementation and easily producing natural looking waves. The natural choice for the waveform function is sinusoidal, giving a shape similar to the vertical deviation of most water surface waves. Therefore, we use

$$W_i(u) = \frac{1}{2}\left(\cos\left(\frac{2\pi u}{l_i}\right) + 1\right)\ \Pi\left(\frac{u}{l_i}\right)\ , \tag{3.8}$$

where l_i is the wavelength and Π is a rectangle function[2]. Besides shape, there are several other reasons behind the choice of this particular waveform function (Figure 3.2a):

- It is non-zero only in a finite range with the first derivative becoming zero at the endpoints. Thus C^1 continuity is maintained when waveforms are summed.
- It is very easy to create continuous waves with wavelength l_i by placing a number of local deviation functions with positive and negative amplitudes that are l_i distance apart (Figure 3.2b).
- Wave shapes with higher wavelengths can be approximated by grouping these local deviation functions as in Figure 3.2c. This property will be especially important when defining radial wave particles in Section 3.7.
- Finally, this waveform function will be useful when it comes to inducing the circular motion of the water surface, as discussed in section 3.10.

[2]Rectangle function $\Pi(x)$ is 1 for $|x| < \frac{1}{2}$, $\frac{1}{2}$ for $|x| = \frac{1}{2}$, and 0 otherwise.

In 2D, wave particles form a 1D particle system. Each particle moves with constant speed v toward the positive or negative x direction. The length l_i, amplitude a_i, and position x_i of the wave particle i defines the deviation of the surface D_i caused by the wave particle. The sum of all deviation caused by all wave particles in the system defines the final shape of the surface curve.

3.5. Wave Particles in 3D

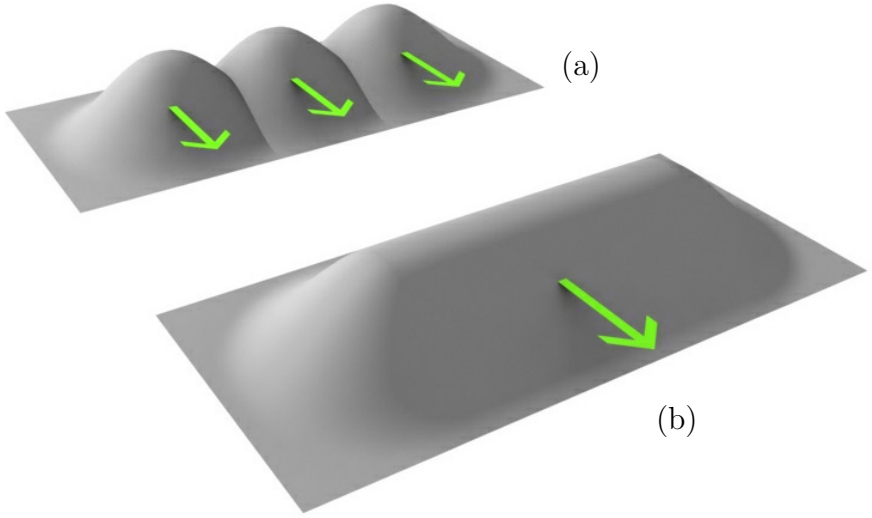


Figure 3.3. (a) Individual wave particles (b) Wavefront formed by these wave particles

In 3D, we have waves traveling on a 2D surface. Unfortunately, the wave behavior of surface waves is not as simple as for 1D waves. In three dimensions, water surface waves take the form of continuous wavefronts. These wavefronts can take rather complicated shapes depending on the propagation of a wave. Therefore, instead of formulating our local deviation functions to represent a whole wavefront, we model wavefronts by placing local deviation functions (wave particles) side by side as shown

in Figure 3.3. In this form a single local deviation function alone does not satisfy the wave equation, but that a collection of local deviation functions can be used to produce a wavefront that satisfies the wave equation.

For the sake of simplicity, we begin our description of wave particles in 3D by assuming that the wavefront is linear. Then, we discuss how this formulation can be extended to curved wavefronts.

3.5.1. Linear Wavefronts

On a linear wavefront we would like the shape of the deviation functions in the wave propagation direction to be the same as for the 2D case defined in Equation 3.8 for the reasons explained above. However, we need to convert this 1D function to a 2D function to be able to represent surface waves. We achieve this using a tensor product of two functions: the function in the direction of motion is the waveform function in Equation 3.8 and the perpendicular function is the *blending function* that is used for blending neighboring wave particles of a wavefront, such that the wavefront can be properly represented. As a result, the deviation function of a wave particle can be written as

$$D_i(\mathbf{x}, t) = \; a_i \; W_i(u) \; B_i(v) \; , \tag{3.9}$$

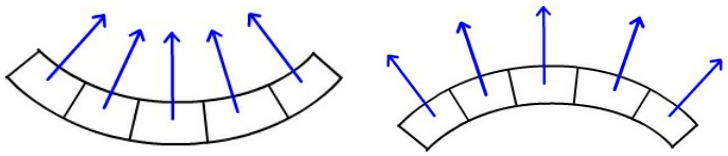
where $u = \hat{\mathbf{u}}_i \cdot (\mathbf{x} - \mathbf{x}_i)$ and $v = \hat{\mathbf{u}}_i^{\perp} \cdot (\mathbf{x} - \mathbf{x}_i)$ are the local coordinates of the local deviation function such that $\hat{\mathbf{u}}_i$ is the propagation direction and $\hat{\mathbf{u}}_i^{\perp}$ is a horizontal direction perpendicular to propagation, and B_i is the blending function. Note that the choice of blending function is somewhat arbitrary: any function that has finite support and whose translates sum to one is acceptable, yielding local deviation functions that

are non-zero only over a finite quadrilateral area determined by the non-zero extents of W_i and B_i.

In this formulation wave particles provide a discrete approximation to the continuous wavefront. Each wave particle carries a number of properties that define the position and the shape of its deviation function. The collection of all wave particles form a 2D particle system that travels on the planar water surface. Just like water surface waves, wave particles do not interact with each other, and their speed is determined by the water medium. The superposition of all wave particles (i.e. local deviation functions) gives the total deviation of the water surface.

3.5.2. Expanding and Contracting Wavefronts

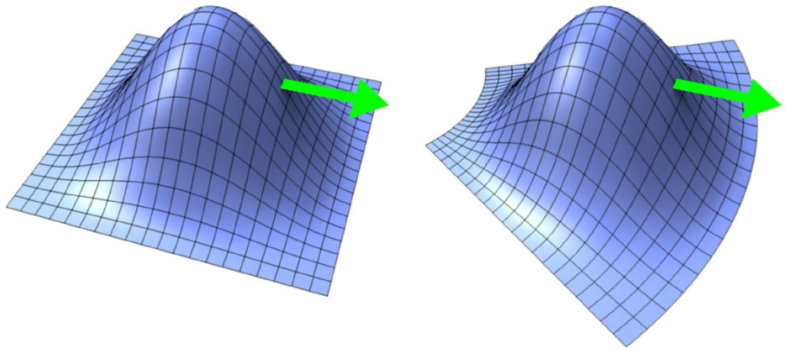
Wavefronts on a surface do not have to be linear. In fact, most wavefronts have a curved shape and they either *expand* or *contract* depending on the direction of wave propagation. Simple examples of expanding and contracting waves represented with wave particles are shown in Figure 3.4. It is very important to be able to handle these cases properly. One way to represent this behavior using wave particles is to bend the local u-v coordinates of the deviation functions to give the wave particle a curved



(a) (b)

Figure 3.4. (a) Contracting wavefront, (b) Expanding wavefront

shape, effectively warping the quadrilaterals (Figure 3.5a) over the non-zero domain (Figure 3.5b).



(a) (b)

Figure 3.5. (a) Wave particle, (b) Warped wave particle

As one would expect, the width of an expanding wave particle increases in time. Similarly, the width of a contracting wave particle decreases as the particle moves. A contracting wave particle eventually turns into an expanding wave particle when its width passes zero and the wave particle effectively "flips over." Note that the width property of a wave particle merely defines the distance between the centers of the two sides of the wave particle, so an expanding or contracting wave particle (with non-parallel sides) that has a zero width still has non-zero base area. Therefore, the amplitude does not become infinite when the width is zero.

To keep wave particles simple, we assume that the part of a wavefront curve that corresponds to a single wave particle has constant curvature. With this assumption, the bending of the wave particle on an expanding or contracting wavefront can be represented by a single curvature value. However, curvature is not a good choice for representing this bending, since the curvature of the wave particle changes as the

wave particle propagates. Instead, we represent an expanding or contracting the wave particle with an angular property that we call the dispersion angle. This property does not change with wave particle propagation.

Dispersion angle is the angle between the two side edges of a curved wave particle. The dispersion angle of the wave particle is shown in Figure 3.6 as α.

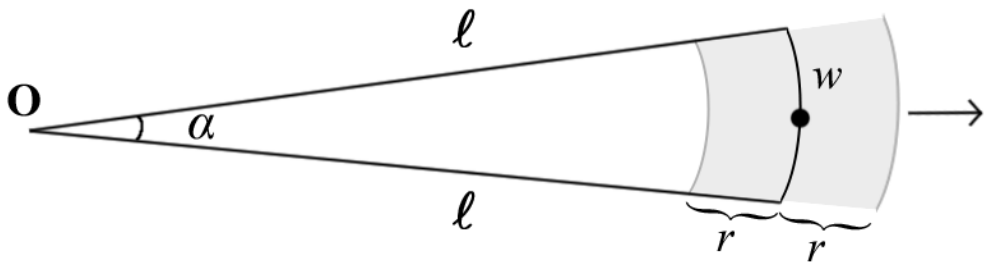


Figure 3.6. Wave particle dispersion angle α and wave particle origin **O**.

Furthermore, the width of a wave particle w changes as the wave particle propagates; thus, it is expanding or contracting. However, we would like to be able to represent the wave particle width in terms of constant parameters. For that we use wave particle origin.

The **Wave Particle Origin** is defined as the intersection point of the two lines aligned with the two sides of an expanding or contracting wave particle. The wave particle origin does not change as the wave particle propagates, because the side edge of an expanding or contracting wave particle moves along the same line. The wave particle origin also intersects with the line that is defined by the wave particle position and the propagation direction. When the wave particle is expanding, the wave particle

origin is behind the object; and when it is contracting, the wave particle origin is in front of the object.

We denote the distance between the wave particle position (center) and its origin as ℓ. Note that this distance increases with constant speed v as an expanding wave particle propagates.

A curved wave particle is bent along the arc defined by its origin $\mathbf{O}$, dispersion angle α, and the distance ℓ; and its length along this arc is defined by the wave particle width w. The relationship between the curvature κ of the wave particle and wave particle properties can be written as

$$\kappa = \frac{1}{\ell} = \frac{\alpha}{w} \, . \tag{3.10}$$

The position of the wave particle $\mathbf{x}$ can also be represented as $\mathbf{x} = \mathbf{O} + \ell\hat{\mathbf{u}}$, where $\hat{\mathbf{u}}$ is the direction of the wave particle. In Chapter V we discuss how these properties can be used for building a highly efficient wave particle system.

While an expanding wave particle is propagating, its amplitude decreases and its width increases linearly according to its dispersion angle. Amplitude may also be decreased to account for energy loss due to viscosity or other damping. On the other hand, the amplitude of a contracting wave particle increases while its width decreases in time. Since contracting wave particles turn into expanding wave particles when their width reaches zero, the amplitudes of all wave particles eventually decrease to near zero.

When the amplitude of a wave particle falls below a certain threshold, its effect on the

total deviation can be ignored and the wave particle can be removed from the system by *killing* the wave particle. Note that damping in this system is optional. Even with no damping, the amplitude of each wave particle decreases if the represented wavefront is expanding, and finally falls below the threshold. However, damping can be introduced to simulate fluids with different viscosities. Damping causes a wave particle to loose energy, which effectively reduces its amplitude in addition to the reduction caused by the expansion of the wave particle.

3.6. Diffraction and a Valid Solution to the Wave Equation

Diffraction is a well known wave behavior that is an inherent part of the wave equation. Figure 3.7 shows the diffraction of water waves going through a wide and a narrow slit.

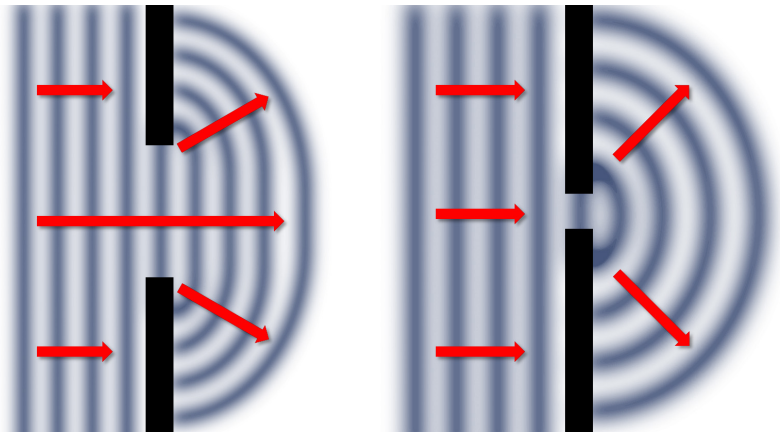


Figure 3.7. Diffraction of water waves going through wide and narrow slit.

In the case of simulating water waves with wave particles, the narrow slit (along with the two walls on either side) separates the wave particle that goes through the slit

from its neighbors on the wavefront. As a result, the wave particle must change its dispersion angle to immediately respond to this change. Otherwise, the wave equation is not satisfied.

This shows a rather extreme case of a wave particle's dependency on its neighbors on the same wavefront. Note that a wave particle alone is not a valid solution to the wave equation and it needs two identical neighbors on either side to be able to form a valid solution locally around it. If the two neighbors are not identical, The solution is locally incorrect, since the wave propagation direction dictated by the wave equation does not align with the propagation direction of the wave particle. When a wavefront diffracts such that only a small portion of the wavefront passes through a slit, the wave particles on either side of that small portion would not have two identical neighbors each; therefore, the simulation would not produce a valid solution to the wave equation.

Diffraction is not the only thing that may violate the condition that each wave particle must have two identical wave particles on either side on the wavefront. Whenever one or both of these wave particles are missing or have a different amplitude than the wave particle, the wave particle needs to respond by changing its properties accordingly to be able to produce a valid solution to the wave equation. We refer to this behavior as the *diffraction effect*, even though it does not have to correspond to a full diffraction behavior.

Handling the diffraction effect properly requires that each wave particle tracks the behavior of its neighbors and changes its properties accordingly to make sure that a valid solution to the wave equation can be produced. Unfortunately, this breaks our

initial desire to make each wave particle as independent as possible. Therefore, we choose to ignore the diffraction effect by assuming that each wave particle always has two identical neighbors on either side of it. The solution produced by wave particles is valid as long as this assumption holds. As we will see in the following sections and chapters, in many scenarios this is a reasonable assumption and wave particles can produce a plausible solution without the need for modeling the diffraction effect. We will discuss possible ways of extending the wave particles system to include diffraction in Chapter VII

3.7. Radial Definition of Wave Particles

We can further simplify the wave particle system using a radial definition for wave particles that approximates the shape of a wavefront. First, notice that Equation 3.8 can also be used as the blending function B_i in Equation 3.9. For radial wave particles, we use a radial definition for D_i rather than the tensor product definition in Equation 3.9, formulating our radial local deviation functions as

$$D_i^R(\mathbf{x},t) = \frac{a_i}{2}\left(\cos\left(\frac{\pi|\mathbf{x}-\mathbf{x}_i(t)|}{r_i}\right)+1\right)\ \Pi\left(\frac{|\mathbf{x}-\mathbf{x}_i(t)|}{2\ r_i}\right)\ , \tag{3.11}$$

where r_i is the radius of the wave particle.

This radial definition makes the wave particle system even simpler. For example, we do not need to worry about bending the wave particle to be able to represent expanding or contracting wavefronts. On the other hand, unlike the generalized definition of wave particles given in the previous section, radial wave particle cannot exactly represent any wavefront. The wavefront produced by placing radial wave

particles side by side can only provide an approximation.

According to our error analysis the maximum difference between the height of the wave crest for a linear wavefront and its representation with evenly spaced radial wave particles is less than 0.8% of the wave amplitude, and the maximum difference between the shape of the wave and its wave particle representation is less than 7.1% of the wave amplitude, as long as the distance between two neighboring wave particles is less than or equal to half of the wave particle radius. We provide the details of our error analysis in Appendix A. Based on this analysis the error introduced by the radial definition by bounding the distances between the neighboring wave particles on a wavefront to half of the wave particle radius.

Using this radial formulation of wave particles we cannot handle expanding and contracting waves by simply changing the width parameter as above, since here r_i defines both length and width of the wave particle. On the other hand, handling expanding waves is actually simpler with this radial definition. First, note that in the wave particle system each wave particle represents a certain packet of wave energy. If there is no damping in the system, the energy of a wave particle must be preserved. Since the radial definition of wave particles does not permit altering the wave particle width independent from its length, we must keep the width constant. If the width of a wave particle remains constant (i.e. constant base area), its amplitude must also be constant for preserving energy. This means that the deviation function of a wave particle preserves its shape and it merely travels with the wave particle. Yet, we know that if a wavefront is expanding, the amplitude of the wavefront must decrease due to energy preservation. With radial wave particles this happens automatically. As the wavefront expands, the distances between neighboring wave particles on the

wavefront increase. As a result, the total amplitude of the wavefront decreases, even though each wave particle preserves its amplitude. Therefore, we can completely forget about adjusting the wave amplitude as the wave expands, since it is automatically taken care of with the radial definition of wave particles. Yet, we need to make sure that the distances between neighboring wave particles are always smaller than half of a wave particle radius, so that the representation error on the wavefront can be bounded. The next section explains the *wave particle subdivision* procedure, which bounds the distances between neighboring wave particles.

3.8. Subdivision

On an expanding wavefront the distances between neighboring wave particles increase as the wavefront travels. This not only reduces the spatial sampling resolution of the wavefront, but also produces problems with the radial definition of wave particles. Figure 3.8 shows an example expanding wave and what happens after the wave particles travel some distance. As can be seen in this example, using the radial definition of wave particles does not allow the shape of the wavefront to be properly represented when the distances between neighboring wave particles on a wavefront increase arbitrarily. Also, we need to make sure that the distance between two neighboring wave particles on a wavefront is always less than half of a wave particle radius, so that we can bound the error introduced by the radial definition of wave particles. The wave particle subdivision procedure helps us achieve this goal.

Wave particle subdivision occurs when the distance between two neighboring wave particles on a wavefront becomes larger than half of the wave particle radius. To

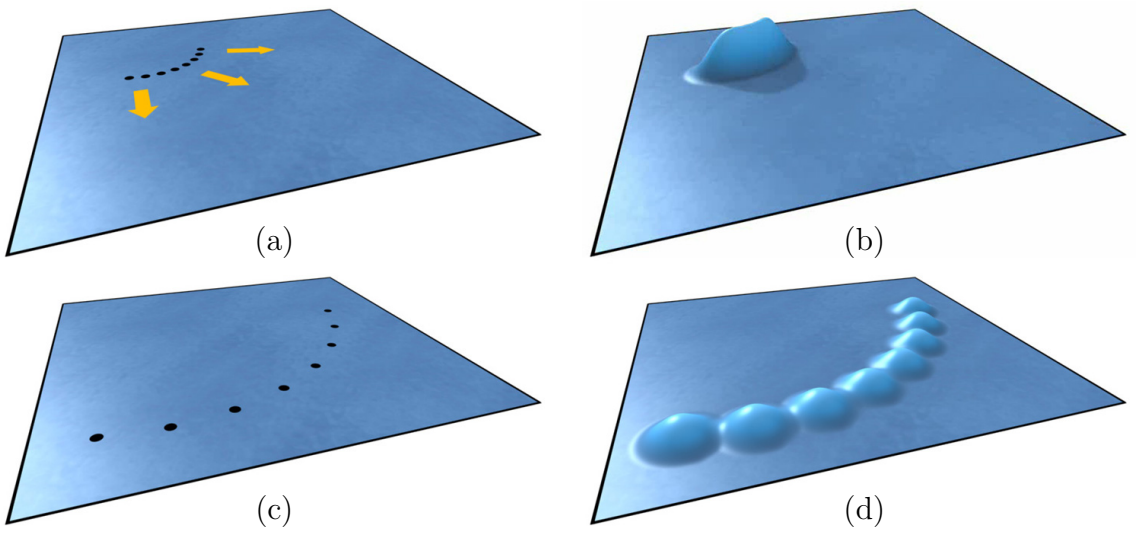


Figure 3.8. An expanding wavefront represented by radial wave particles with no subdivision. (a) and (b) are the initial positions of the wave particles and the wavefront they form, respectively. (c) and (d) are the positions of the wave particles and the wavefront shape they represent after wave particles travel some distance.

reduce the distance between the two wave particles, we simply introduce new wave particles in between these two wave particles. The new wave particles take their energy (i.e. amplitude) directly from the existing wave particles; thereby reducing the amplitudes of the existing wave particles. As a result, the overall amplitude of the wavefront remains unchanged.

To be able to build an efficient particle system that can be simulated as fast as possible, it is important to keep each wave particle independent. Therefore, we define the wave particle subdivision procedure such that each particle can subdivide on its own without having to coordinate with the neighboring wave particles on the same wavefront.

As we mentioned earlier, one of the properties that each wave particle carries is the dispersion angle. In the generalized formulation of wave particles the dispersion angle is a measure of curvature and it tells us how the width of the wave particle should change as the wave particle travels. In the radial formulation of wave particles the dispersion angle tells us when to subdivide the wave particles.

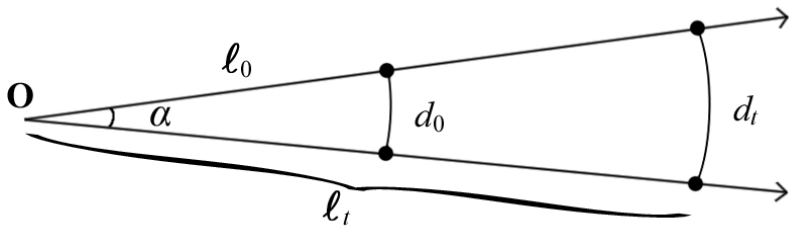


Figure 3.9. Calculation of the distance between two neighboring wave particles of a wavefront that share the same dispersion angle.

We assume that the dispersion angle of the wave particle that is subject to subdivision is the same as the dispersion angles of the neighboring wave particles on either side of the wave particle. When the two neighboring wave particles have the same dispersion angle, they are exactly on the same circular arc and their origin is the same point. Moreover, the angle between the two wave particles is equal to the dispersion angle of the wave particles as shown in Figure 3.9. Using this assumption, if we know the distances between the neighboring wave particles at a previous time step, we can compute the distance at any given time using the dispersion angle property. Let the angular distance between the two wave particles be d_0 at time t_0. The distances of both particles to the point **O** at time t_0 is ℓ_0. Assuming that α is small, the distance between the two wave particles can be approximated with the length of the arc that connects the two wave particles. Using this notation we can write the dispersion angle

as

$$\alpha = \frac{d_0}{\ell_0} = \frac{d_t}{\ell_t} = \frac{d_t}{\ell_0 + \upsilon(t - t_0)} \ . \tag{3.12}$$

By rearranging the terms of the equation above we can write the angular distance between the two wave particles d_t at time t as

$$d_t = d_0 + \alpha \ \upsilon \ (t - t_0) \ . \tag{3.13}$$

Using this equation we can tell exactly when the distance between the wave particles d_t will be greater than half of the wave particle radius r_i. When $d_t > \frac{r_i}{2}$ for the wave particle i, all we need to do is to add new wave particles between this one and its neighbors.

The simplest solution would be adding a wave particle in the middle of two subdividing wave particles and adjust the amplitudes of the subdividing wave particles accordingly. However, this means that we need to subdivide the two neighboring particles concurrently, which makes the two wave particles computationally dependent on each other. As stated earlier, we would like each wave particle to be completely independent. Therefore, instead of inserting a single wave particle in between the two wave particles, we insert two new wave particles. Each one of the two new wave particles comes from one of the subdividing wave particles making the subdivision of the two wave particles computationally independent.

Since in the wave particle system we assume that each wave particle has two identical neighbors on either side, when a wave particle subdivides it generates two new wave particles on either side as shown in Figure 3.10. Note that the subdividing wave particle is not removed. These new wave particles are placed $\frac{d_t}{3}$ away from the subdividing

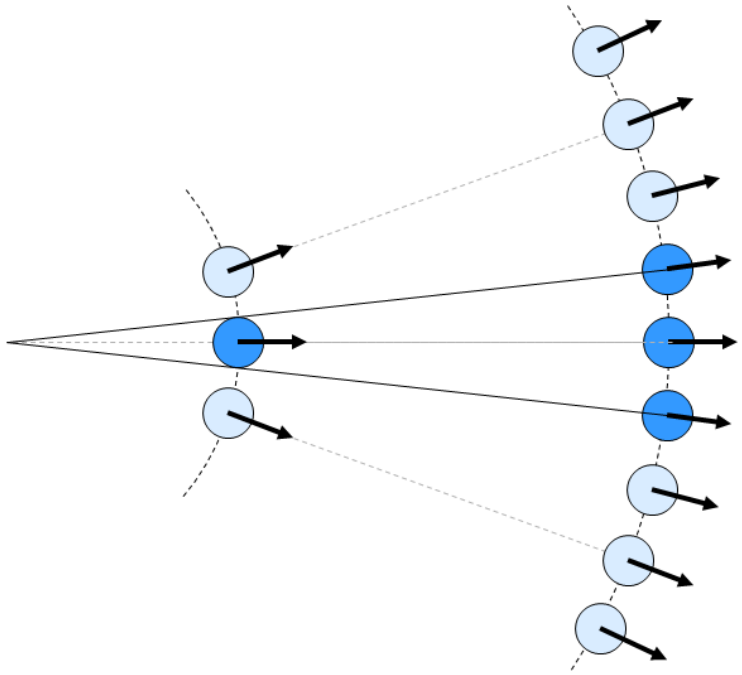


Figure 3.10. Calculation of the distance between two neighboring wave particles of a wavefront that share the same dispersion angle.

wave particle. The amplitudes of the subdivided wave particle and the two new wave particles become one third of the amplitude the wave particle before the subdivision. Similarly, the dispersion angles of the subdivided and new wave particles become one third of the dispersion angle before the subdivision. Finally, the angle between the directions of the new wave particles and the subdividing wave particle is one third of the dispersion angle before the subdivision.

Using this subdivision procedure each wave particle subdivides independently and the shape of the wavefront is automatically preserved. The only assumptions we use to achieve this are that the dispersion angle of a wave particle is the same as its neighbors' and that the neighboring two wave particles are the same distance away from the wave particle. Note that regardless of initial conditions, after a wave particle

goes through one subdivision operation these assumptions are precisely correct for the subdivided wave particle that is at the center.

3.9. Boundary Behavior

Boundaries are the edges of the container that holds the simulated water. Wave particles bounce back from the boundaries to simulate reflecting waves.

If the boundary that a wave particle is reflecting off of is linear, the reflection operation only changes the propagation direction of the wave particle, exactly like a mirror reflection. When a wavefront composed of multiple wave particles hits a linear boundary, each wave particle gets reflected one by one as it hits the boundary and the wave propagation continues with the updated wave particle directions.

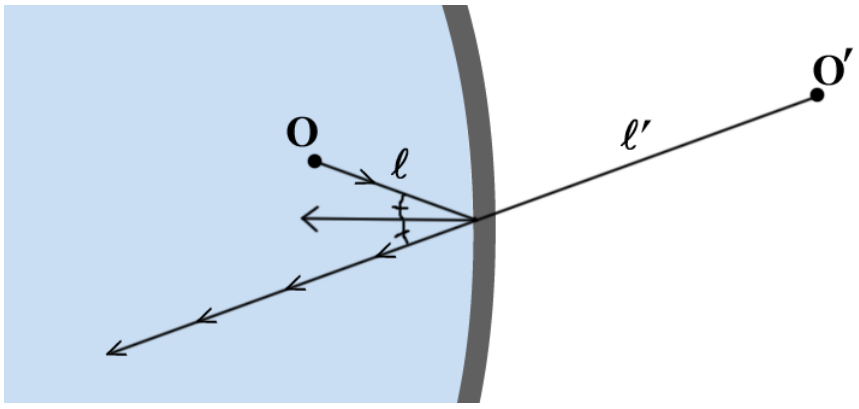


Figure 3.11. Reflection of a wave particle from a curved boundary.

When the boundary is curved, this reflection becomes more complicated, since the reflection operation also changes the dispersion angle of the wave particle based on

the curvature of the boundary. Figure 3.11 shows an example wave particle reflecting off of a curved boundary. In this figure **O** is the origin of the wave particle and ℓ is the distance of the wave particle to the origin when the particle hits the boundary. The width of the wave particle w at the hit position can be written as $w = \alpha\,\ell$, where α is the dispersion angle before the reflection. After the reflection the origin of the wave particle $\mathbf{O}'$ becomes the mirror reflection of the origin before the reflection **O**. Note that in case of a reflection from a curved boundary, the wave particle origin after the reflection can be on either side of the boundary depending on the curvature (i.e. the focal point) of the boundary and the position of the wave particle origin before reflection. Since the width of the wave particle immediately before and immediately after reflection are the same, the dispersion angle of the wave particle after reflection is

$$\alpha' = \frac{w}{\ell'} = \alpha\frac{\ell}{\ell'}\ . \tag{3.14}$$

This formulation of wave particle reflection is valid both when the shape of the container holding the water (i.e. pool) is convex or concave. However, if the container is concave, there is no guarantee that the line segment that connects two arbitrary points within the boundaries will not intersect the boundary. As a result, while a wavefront propagates, it is possible that only a portion of a particular wavefront gets reflected by a part of the boundaries, while the rest of the wavefront does not hit the boundary and continues to propagate without changing. If a portion of a wavefront gets reflected, the wavefront is effectively divided into two wavefronts. At the separation point of these two wavefronts the assumptions about the wave particle system are no longer valid. Since each wave particle requires two neighboring wave particles on either side to form a valid solution to the wave equation, if one of the neighboring wave particles gets reflected and moves away, the result is no longer a solution to the

wave equation. In reality, if a portion of a wavefront is reflected, the wave equation is satisfied by wave diffraction. However, since we do not include the diffraction effect in our formulation, we cannot allow wavefronts to be partially reflected off of a boundary.

If the edges of the container that holds the simulated wave volume form a convex curved boundary that is smooth everywhere, we do not need to handle diffraction to properly handle wave reflection. However, if the convex boundary has at least one sharp corner, the same problem can arise. Consider a wavefront moving towards a sharp corner. A part of the wavefront will reflect off of the boundary on one side of the corner and move in one direction, while the other part of the wavefront will reflect off of the boundary on the other side and move in a different direction. As a result, the wavefront will separate into two disconnected pieces. Therefore, the result will not be a valid solution to the wave equation.

One special case is a boundary corner making a right angle. Consider a wave particle moving towards the corner. Just after the wave particle hits the boundary on one side of the corner, it will hit the boundary on the other side. This will effectively invert the initial wave particle direction, no matter which side of the corner the wave particle hits firsts. Therefore, a corner with a right angle will not separate a wavefront and introduce discontinuity to the solution. As a result, a container with a rectangular boundary can be handled without the need for diffraction.

However, for all other boundary shapes, simulating diffraction is needed. Otherwise, the end result of the simulation may not be a valid solution to the wave equation.

On the other hand, wave particles are ideal for scenes where there are virtually no

boundaries, such as open ocean scenes. When there are no boundaries, there is no wave particle reflection; therefore, there is no need to worry about diffraction for handling partially reflected wavefronts. In that sense, open ocean scenes are the easiest scenarios for the wave particles method. Contrary to the common perception of water simulation techniques, open ocean scenes do not have higher computational demands with wave particles. They indeed require less computation, since wave particle reflection is eliminated.

3.10. The Circular Motion of Water Waves

Until now, we have considered only transverse waves that move the water surface up and down in vertical direction. However, it has been known for centuries that water waves are composed of both transverse and longitudinal waves [Gerstner 1802]. The transverse component accounts for the vertical motion of the water surface, while the longitudinal component refers to the horizontal motion due to waves.

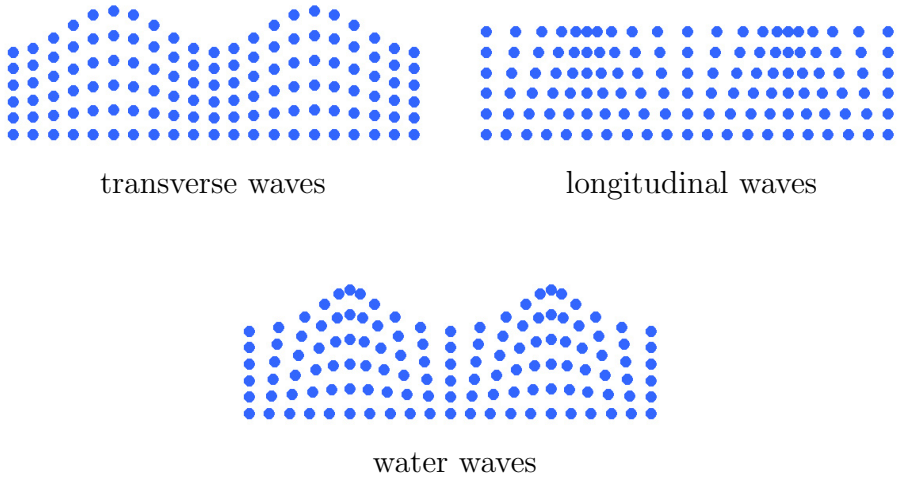


Figure 3.12. Transverse and longitudinal components of water waves

Figure 3.12 illustrates the transverse and longitudinal components of water waves. The transverse and the longitudinal components that form water surface waves have a constant phase relationship with each other. They not only co-exist, but are also aligned. This figure illustrates how the peak positions of the transverse component (wave crest) are aligned with the position where the water particles are closest due to the longitudinal component. This is why water waves can travel without translation of considerable water volume. As the water particles come together due to the longitudinal component, the water level rises, keeping the density constant while forming the transverse component.

As a result of these transverse and longitudinal components of water waves, the water particles near the surface of the water undergo circular motion as a wave passes by. The radius of this circular motion is the largest on the water surface and this radius exponentially decreases deeper into the water volume. Figure 3.13 shows this circular motion of water as presented in Gerstner's illustrations [Gerstner 1802].

When simulating water waves, it is important to properly account for both the transverse and the longitudinal components of the surface wave motion. In computer graphics, it is possible to come across wave simulation techniques and implementations that completely ignore the longitudinal component of water waves, resulting in rather unrealistic wave behavior. A few important reasons why the longitudinal component should not be ignored are the following:

- The longitudinal component is necessary for giving water waves a realistic shape. Is is a common misconception that water waves have a sinusoidal shape. In fact, due to the longitudinal wave motion, the wave crest takes a sharper shape, while

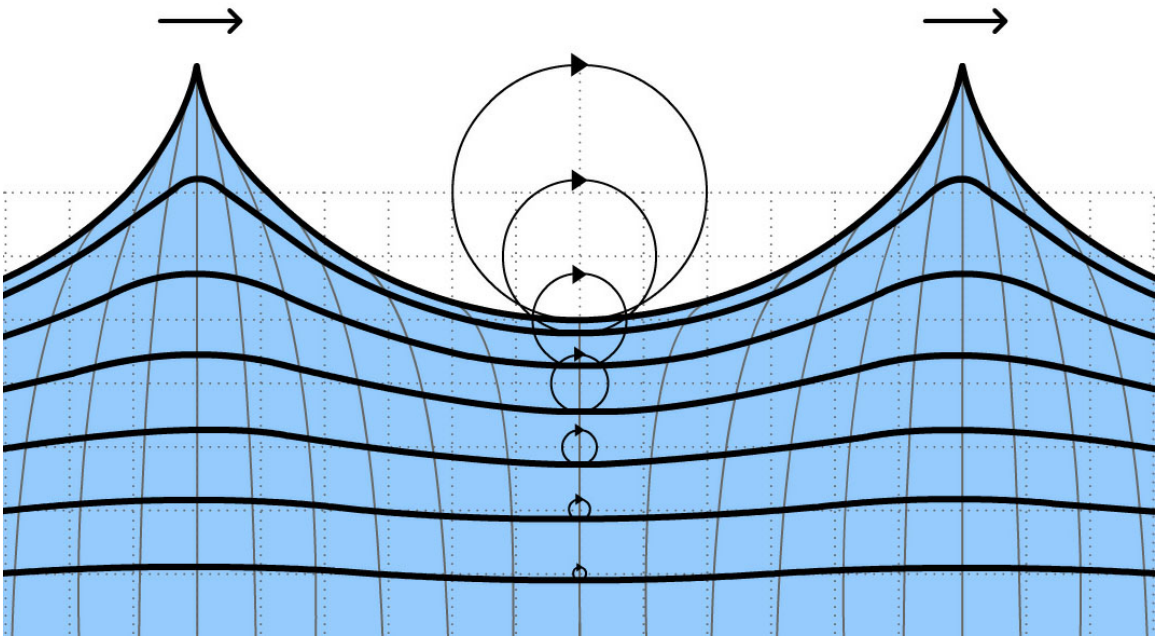


Figure 3.13. Isopressure lines showing the circular motion of water due to surface waves.

the wave trough expands horizontally and becomes smoother.

- The superposition of waves looks rather unnatural when the longitudinal component is ignored. Consider two wave crests traveling towards each other as in Figure 3.14. When the two waves collide, the superposition affects both the transverse and the longitudinal components. As a result, when the two waves collide, not only does the vertical amplitude of the surface deviation become the sum of the two waves, but the resulting wave crest becomes sharper than either one of the original waves because of the longitudinal superposition. If the longitudinal component is ignored, the superposition takes the shape of the dashed curve in Figure 3.14, which forms unnatural wave shapes in 3D.

- The longitudinal component is important when computing the velocity of the

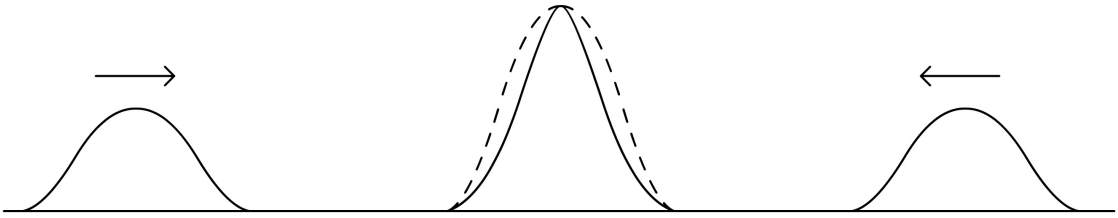


Figure 3.14. Superposition of two water surface waves traveling towards each other. If the longitudinal component is ignored, the superposition in the middle takes the shape of the dashed curve.

water surface and the interaction of water with floating objects. When a water wave with a positive amplitude approaches a floating object, the longitudinal motion of the water pulls the object towards the wave. Similarly, as the wave moves away from the object, the longitudinal motion pushes the object away from the wave, back to its original position. In this way, the wave induces a circular motion on the floating object. This behavior cannot be accounted for when the longitudinal component is ignored.

For the sake of simplicity, while discussing wave particles in the next chapter, we will begin by considering transverse waves only. We will then talk about how the longitudinal component of the water waves can be incorporated.

3.11. Longitudinal Deviation

For incorporating the longitudinal component of water surface waves into the wave particle formulation, we extend the definition of the deviation function introduced in equations 3.5 and 3.6, which is formulated for transverse waves, such that the

deformation of the water surface happens along the vertical z direction. To be able to include longitudinal waves and the deformations caused by longitudinal waves we define a surface deformation field $\eta : \mathbb{R}^3 \to \mathbb{R}^3$, such that the final position $\mathbf{x}'$ of a point $\mathbf{x}$ on the water surface is

$$\mathbf{x}'(\mathbf{x}, t) = \mathbf{x} + \eta(\mathbf{x}, t) \ . \tag{3.15}$$

Here the vertical component of η is the same as Equation 3.6, but the horizontal component is

$$\eta_{xy}(\mathbf{x}, t) = \sum_i \mathbf{D}_i^L(\mathbf{x}, t) \ , \tag{3.16}$$

where $\mathbf{D}_i^L$ is a *horizontal local deviation function* (Figure 3.15). This can be formulated similar to Equation 3.9 as

$$\mathbf{D}_i^L(\mathbf{x}, t) = \ a_i \ \mathbf{L}_i(u) \ W_i(u) \ B_i(v) \ , \tag{3.17}$$

where $\mathbf{L}_i$ is a vector function describing the longitudinal waveform. We derive the longitudinal waveform that corresponds to our transverse waveform function (Equation 3.8) from the circular motion of continuous waves (Figure 3.13).

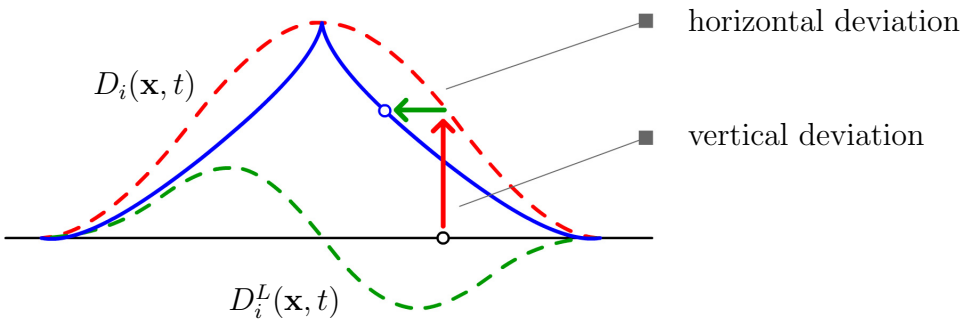


Figure 3.15. Components of the local deviation function in 2D

We find the longitudinal waveform that gives a circular motion when combined with

the transverse component to be

$$\mathbf{L}_i(u) = -\sin\left(\frac{2\pi u}{l_i}\right)\ \Pi\left(\frac{u}{l_i}\right)\ \hat{\mathbf{u}}_i\ , \tag{3.18}$$

where $\hat{\mathbf{u}}_i$ is the propagation direction of the i^{th} wave particle. For radial wave particles, note that the longitudinal component must only be created in the direction of motion. Thus, we define the horizontal local deviation by radially blending the directional deviation:

$$\mathbf{D}_i^{RL}(\mathbf{x}, t) = \mathbf{L}_i(u)\ D_i^R(\mathbf{x}, t) \tag{3.19}$$

Using this longitudinal formulation of wave particles each wave particle induces a part of the circular motion that water waves should generate. We can produce a perfect wave train with circular motion by generating wave particles with positive and negative amplitudes that are wave particle radius apart.

CHAPTER IV

WATER-OBJECT INTERACTION

In this chapter we discuss how the interaction between water and floating objects can be modeled. A simulation method that is designed to work in a real-time graphics application must permit some sort of user interaction. When it comes to simulating water waves, user interaction can be as simple as generating waves directly on the water surface based on user input. However, the useful scenarios supplied by this kind of interaction would be quite limited. Another form of user interaction could be coupling the wave simulation with a rigid body simulation, so that rigid bodies moved directly or indirectly by user actions can properly interact with the water body. This treats the water simulation as a component of a real-time rigid body simulation system.

Computation of physically accurate water-object interaction is a rather complicated and computationally demanding operation. A major part of the complication comes from the fact that one needs to compute the forces between the interacting object and the water at the boundary where the object touches the water. This boundary might have a complicated shape. More importantly, the fluid behavior can be rather complicated near this boundary. The motion of the object makes this computation even more complicated, since it forms a moving boundary between the object and water, as opposed to a static boundary, which would be significantly easier to handle. Accurately and efficiently handling fluid-object interactions is still an open research

area in both computational fluid dynamics and computer graphics.

For all these reasons, finding a reasonable simplification of physically accurate water-object interaction is challenging. On the other hand, physical accuracy is not really crucial for a simulation system designed for computer graphics applications. Especially for graphics algorithms that are intended for real-time simulation systems, computational efficiency is often more important than anything else, as long as one can find a way to achieve plausible animation results.

We begin simplifying water-object interaction by separating it into two components:

- Object to water coupling, and
- Water to object coupling.

Object to water coupling handles the effect of the object motion on water, and water to object coupling is responsible for computing the effect of water on the motion of the interacting object. Obviously, these two components are not independent of each other; therefore, a physically accurate water-object interaction system would solve for these two components concurrently. Instead, at each time step we handle object to water coupling independently from water to object coupling. As a result, the change in object motion caused by water to object coupling does not affect the motion of water until the next time step. The inherent assumption that comes with this separation is that the motion of water and the interacting objects do not change within a time-step. As the time step duration approaches zero, the error caused by this separation assumption approaches zero as well. On the other hand, when the time step is too large, this separation might cause significant differences between

consecutive frames, which might even cause visible fluctuations. The ideal setting for the time step depends on the properties of the interacting objects as well as the nature of the simulated motion.

4.1. Object to Water Coupling

The object to water coupling component of water-object interaction is responsible for computing the effect of the interacting object motion on the motion of the water. The object to water coupling technique presented here is designed to work with the wave particles method. Since wave particles represent the water motion in the form of surface waves, the object to water coupling technique is essentially about managing surface waves due to object motion. Other products of object to water coupling, such as splashes and bubbles, are ignored here as secondary effects. These secondary effects might be incorporated into object to water coupling using methods such as those discussed in Section 7.3.1.

Dynamic surface waves are very important for plausible object to water coupling. Almost any interaction of objects with water generates waves on the water surface. These waves not only affect the motion of water near the interacting object, but also can travel long distances over the water surface, affecting the shape of the water surface over a rather large area. Therefore, dynamic surface waves must be handled properly to achieve realistic looking water-object interaction.

4.1.1. Physical Wave Generation

Before we begin discussing the object to water coupling technique, it is important to understand the physical process of wave generation due to object interaction with water. When objects interact with water surface, waves are generated. However, the actual process of wave generation does not really take place immediately and this sort of wave generation is a rather complicated phenomenon.

Consider an object falling onto the water surface. As soon as the object hits the water surface, it induces some motion onto the water around the contact surface between the object and water. The motion induced by the object is often rather complicated and includes 3D turbulent behavior. Generally speaking, this motion does not correspond to a wave motion at all. However, as a result of this motion waves are formed on the water surface. Therefore, if we are to model physical wave generation due to object interaction, we can break the wave generation process into two components:

- Water motion induced by the object,
- Waves formed from this motion.

The object interaction with water often induces a rather complicated 3D fluid motion that potentially has some amount of turbulence, so it cannot be properly represented by efficient 2D structures or wave particles. Therefore, we can conclude that computing the first step as a part of a real-time water simulation system is rather ambitious for the capabilities of today's computer hardware.

While the first step of computing the 3D motion induced by the interacting object can

be computationally expensive, the second step of converting this motion to surface waves is in fact more challenging. To our knowledge, there is no method in computer graphics that would take an arbitrary 3D water motion as input and compute the surface waves that motion will eventually evolve into.

Due to the enormous difficulties of physical wave generation for our real-time water-object interaction system we have to settle for a merely physically plausible model. To make object to water coupling simpler, we assume that waves are generated immediately when an object hits the water surface. Following this assumption, all motion of an object inside the water volume immediately generates new waves. While this assumption may not be physically correct, it allows us to significantly simplify the object to water coupling computation. Using this assumption, we completely ignore the 3D turbulent motion caused by object interaction and we do not worry about converting this complicated motion to surface waves.

Obviously, assuming that surface waves due to object interaction are generated immediately affects the accuracy of the object to water coupling. More importantly, it presents challenges when it comes to building a physically based model for wave generation, since we know that we are not exactly following the actual physical process. Ideally, surface waves generated with this assumption should match the surface waves that would result from pure physical wave generation. In this ideal case the assumption of immediate wave generation only causes errors in close proximity to the interacting object, while the waves that affect a much larger area of water surface can still be represented properly.

4.1.2. Effect of Objects on Existing Waves

Objects that are floating on the water surface not only generate waves as a result of their motion but also affect the existing water waves. When there is an object floating in the water, it changes the behavior of the water medium directly below and above it as well as immediately around it. Therefore, when a surface wave arrives at the position of a floating object, it effectively enters a different medium. In simplest terms, some part of the wave continues through the new medium, while the rest of the wave is reflected.

However, the real world interaction of existing waves with floating objects is far more complicated than the brief summary above. The behavior of this interaction largely depends on the wavelength of the existing wave as compared to the size of the interacting object. If the wavelength is significantly smaller than size of the interacting object, the existing wave is strongly affected by the presence of the object. For example, it may reflect off of the object as it would a boundary. On the other hand, when the wavelength of the existing wave is significantly larger than the size of the object, the wave may be hardly affected at all. The most complicated interactions happen when the object size is comparable to the wavelength of the existing wave.

Based on this knowledge, one rather obvious simplification of modeling existing wave interaction with floating objects would be an ad hoc formulation to determine what portion of the existing wave gets reflected and what portion of the wave continues to travel without being modified. Yet, even with this simplified model, one needs to compute the intersections of dynamic objects interacting with the water and all existing waves on the water surface. Keeping in mind that a wave particle simulation

can have a large number of active wave particles at any given time, this intersection computation itself can be very slow for real-time graphics purposes. Furthermore, the actual interaction of existing waves with an object is far more complicated than this simple model. Especially in the presence of many existing waves interacting with the floating object at the same time, computing the interaction of each wave particle independently would not necessarily provide accurate results.

Instead, we propose a different simplification that is based on wave superposition. A modified wave due to object interaction can be represented as the superposition of two waves: the original unmodified wave and an *interaction effect wave* that is placed on top of the original wave as shown in Figure 4.1. This interaction effect wave can modify the original wave and even effectively cancel it out. In this way object interaction with existing waves is modeled by wave generation only. Therefore, assuming that the wave generation properly takes the existing waves into account, the computation of object interaction with existing waves can be completely eliminated, and it is inherently handled by the wave generation computation. With this assumption, we can avoid computing the intersections of existing waves with floating objects, as well as the complicated procedure of handling these intersections.

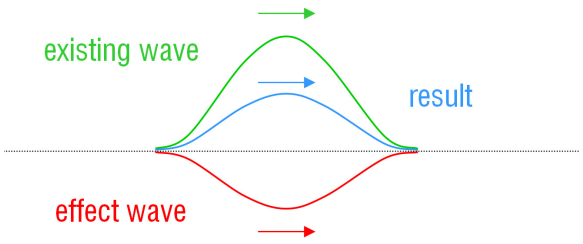


Figure 4.1. Modifying an existing wave by adding an interaction effect wave.

4.1.3. Energy Preservation vs. Volume Preservation

Wave propagation is highly efficient and for the most part it preserves energy. However, wave generation due to object interaction does not preserve energy in the form of waves. When an object hits the water surface a significant portion of the object's kinetic energy is converted into other forms such as turbulence, heat, and sound. The energy of generated waves due to the interaction of the object with water only corresponds to a portion of the initial energy of the object. Therefore, it is difficult, if possible at all, to derive a wave generation technique that is based solely on energy conservation.

On the other hand, since the compressibility of water is very low, water volume is preserved. Especially when computing the interactions of open water with floating objects, it is safe to assume that water is completely incompressible. (Note that in many computer graphics applications even gasses are assumed to be incompressible, while in fact they are highly compressible.) Therefore, we can base our wave generation method on volume preservation.

However, wave generation based on volume preservation of open water can only dictate the volume of waves generated by the interacting object motion; it does not provide any information about the directions or the shapes of generated waves.

4.1.4. Heuristics for Wave Generation

The volume preservation principle only tells us that for all waves generated with positive amplitude there must be a number of waves generated with negative amplitude

at the same time, such that the total volume of water displaced by generated waves with positive amplitude should match the volume of water displaced by the ones with negative amplitude. Other than this general principle, volume preservation does not provide guidance about the shapes of the waves to be generated or their placement.

Formally speaking, there are three questions that need to be answered to be able to build a wave generation system:

- Where should the generated waves be placed?
- What should be the directions of generated waves?
- What should be the sizes of generated waves?

We have developed heuristics for wave generation to attempt to answer these questions. These heuristics are based on experimental observations and analytical reasoning as well as convenience for implementing a highly efficient wave generation system. In the rest of this section we discuss these heuristics.

4.1.4.1. Wave Placement Heuristics

We know that when an object moves inside the water volume, it induces some motion to the water around it and this motion has some impact on the deviation of the water surface. Our aim with wave placement heuristics is to come up with a set of rules that would help use determine how these displacements should be distributed on the water surface.

Let us consider the case of an arbitrarily shaped object whose surface is in contact with

water moving inside the water volume. Therefore, as the object moves in a particular direction, some part of the object surface "pushes" the water around it, while some other part of the object surface "pulls" the water. Due to volume conservation, the total volume of water that is pushed by the object motion is equal to the volume of water that is pulled by the object. However, the flow of water around the moving object also affects the water surface, if the object is close enough to the water surface, and causes waves to be generated on the surface.

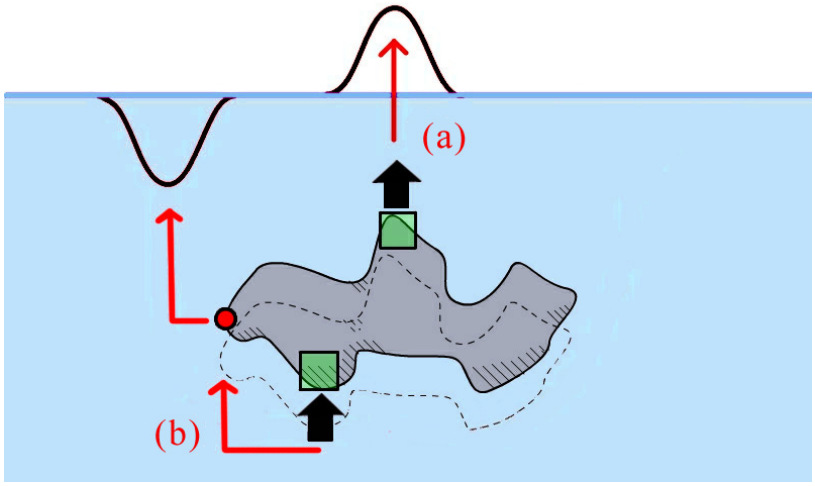


Figure 4.2. Wave placement of object faces. (a) Faces that have direct access to the water surface, (b) faces that do not have direct access to the water surface

Figure 4.2 shows an arbitrary object moving upwards inside a water volume. In this case, most of the top surface of the object pushes the water upwards, while most of its bottom surface pulls the water upwards. Generally speaking, this kind of motion generates 3D water flow around the object. However, since we are only interested in the water surface and we are generating waves immediately, we disregard this 3D water flow around the object. Instead, we try to determine how each part of the

moving object surface would ultimately affect the water surface and generate waves. As indicated in Figure 4.2, there are two distinct cases here:

- As shown in Figure 4.2(a), the object surface at the top of the object has a direct connection to the water surface, meaning there is nothing but water between the top object surface and the water surface. Therefore, we assume that the displacement of water caused by the motion of the top surface of the object is directly transmitted to the water surface directly above the object surface.

- Referring to Figure 4.2(b), we see that the object surface at the bottom of the object, does not have a direct connection to the water surface. In other words, the object itself lies between its bottom surface and the water surface. Therefore, the displacement of water caused by the motion of the bottom surface of the object cannot be transmitted to the water surface directly above the object surface. Instead, this motion of water must induce water motion around on either side of the object in this 2D case. As a result, the displacement caused by the bottom surface of the object is transferred to the sides of the object, and the water surface directly above either side of the object gets affected by this displacement.

Assuming that the object interacting with water is a polygonal mesh, the volume of water displaced by the motion of each face is used to generate waves

- directly above the face, if the face is on top of the object;

- around the object, otherwise.

In 3D we consider the silhouette of the object as seen from the top view and generate

waves around this silhouette for the faces that are not on top of the object. Transferring the displacement of water caused by a face to the points around the object silhouette is a bit more complicated in 3D. Since we do not use a 3D fluid simulation system, we do not have any information about the actual water flow around the object. Therefore, we cannot really tell which points around the object silhouette are affected by the water displacement caused by any face that is not at the top side of the object. Since we would like to avoid a full 3D fluid simulation around the object, we need a simple rule for distributing these displacement effects. One could transfer the whole displacement of a face to the nearest point around the object silhouette. Alternatively, the displacement caused by the face can be transferred to all points around the object silhouette evenly. It is safe to assume that the points around the object that are closer to a face of the object are more likely to be affected by the volume of water displaced by the face. Based on this reasoning, our heuristic is to distribute the water displaced by a bottom face around the object silhouette in an amount weighted inversely by their distances from the face.

Admittedly, the actual physical water motion induced by this object motion is far more complicated than our approach suggest. However, our aim is to provide an efficient wave generation system that produces plausible results, rather than a physically accurate one. Our simulation results have convinced us that this heuristic does, indeed, produce convincing water surface waves in response to object motion.

4.1.4.2. Wave Direction Heuristics

Wave direction heuristics are used for deciding on the propagation directions of generated waves. Our aim here is to propose a simple set of rules for assigning propagation directions to generated waves, such that the outcome can depict the general behavior of the waves generated by real water-object interaction. To achieve this purpose, we first conducted various experiments observing waves generated by real physical objects.

One of these experimental setups was particularly useful, since it allowed us to observe wave generation behavior in 2D. This setup consists of two glass walls that are placed very close to each other. We filled the region in between the two glass walls with water up to a certain height. For an interacting object, we used a cylindrical slice of candle cut slightly shorter than the distance between the two glass walls, so that it could slide easily between the walls, but could not rotate around an axis parallel to the plane of the walls. The candle was dropped onto the water and the interaction was recorded by a camera. Figures 4.3 and 4.4 show frames from some of our experiments with this setup.

In Figure 4.3 we show consecutive frames taken when the object hits the water surface. The first set of waves generated in the second frame on either side of the object begin to travel away from the object. While the object is inside the water volume and continues to sink, a negative deviation is induced on the water surface directly over the object and the water surface on two sides of the object slightly rise forming a positive deviation. In the following frames, the positive deviation moves towards the object and forms a sharp peak right above the center of the object. Even though

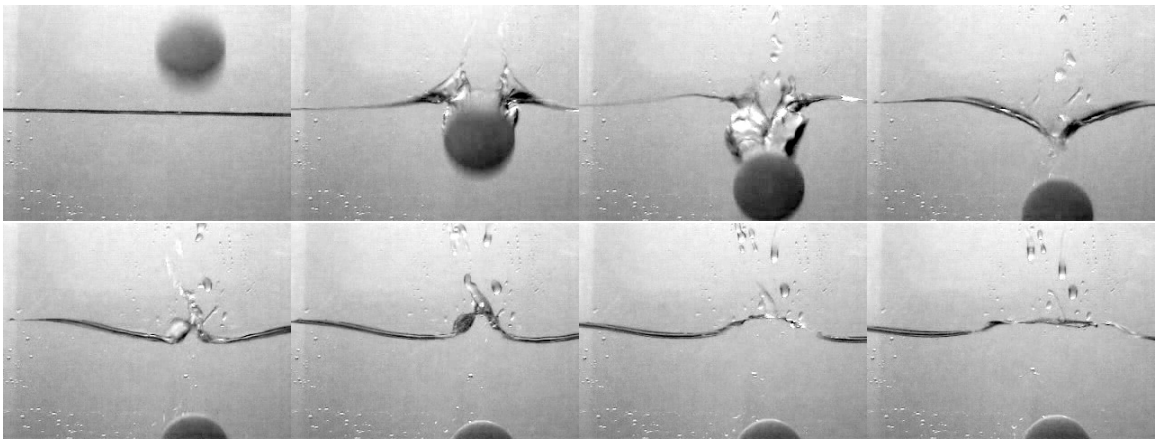


Figure 4.3. Frames taken from one of our 2D wave generation experiments showing an object falling into the water.

the actual motion of water is rather complicated, the motion of the surface can be approximated by surface waves. In this case, the initial impact generates positive amplitude waves that move away from the object. In the following frames, the dent above the object can be represented by negative waves that travel in both directions and the two bumps on either side of the object can be represented by positive waves that move towards the object. In the next few frames these positive waves that are formed on either side of the object travel towards the center of the object and form the peak.

Figure 4.4 shows consecutive frames from the same experimental setup taken while the object is coming out of the water. As the object moves up towards the water surface, a bump is formed above the object. The dents on either side of the object become clear when the object reaches the water surface. As the object moves up and down it emits waves that travel away from it on either side.

Following these observations we propose a simple heuristic model for assigning direc-

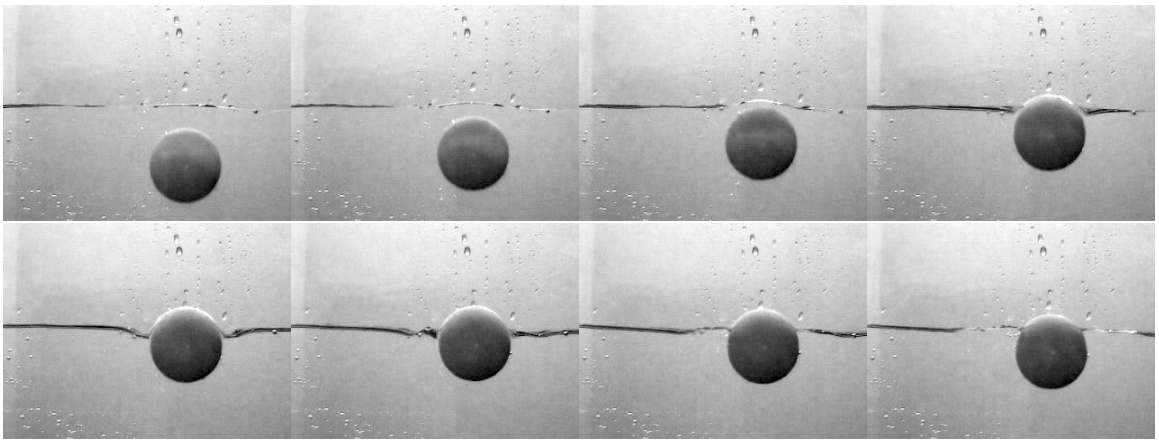


Figure 4.4. Frames taken from one of our 2D wave generation experiments showing an object coming out of the water.

tion to the generated waves. Figure 4.5 summarizes our wave direction heuristics. There are four cases:

- As shown in Figure 4.5a, when the object is on the water surface and it is moving up, it generates negative waves on either side, which is consistent with our wave placement heuristics described above. The motion of these waves are away from the object.

- Similarly, when the object is on the water surface and moving down, it generates positive waves on either side as shown in Figure 4.5b. The wave direction is away from the object in this case as well.

- When the object is below the water surface and moving up as shown in Figure 4.5c, negative amplitude waves are generated on either side of the object and positive amplitude waves are generated on top of the object. The positive amplitude waves in the middle move in both directions separating into two waves (similar to a ripple on a 2D surface). The negative waves, however, move

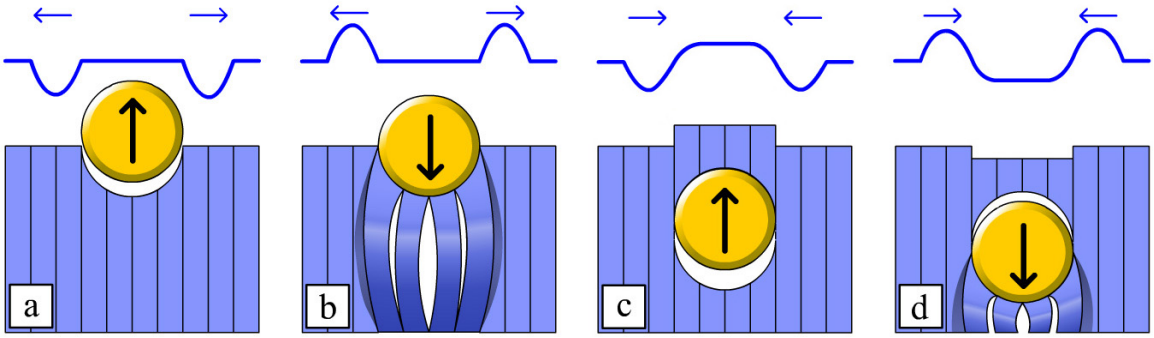


Figure 4.5. Cases of wave generation, (a-b) object is on the surface, (c-d) object is inside the fluid volume.

towards the object.

- Finally, when the object is below the water surface and moving down as shown in Figure 4.5d, negative amplitude waves are generated directly above the object and positive amplitude waves are generated on either side. Similarly, the waves right above the object separate and move in both directions, while the waves on either side move towards the object.

These four cases provide a unified procedure for assigning wave directions:

- If the wave is directly above the object, it moves in all directions forming a ripple,
- If the wave is on the side of the object and the object is on the water surface, the wave moves away from the object,
- If the wave is on the side of the object and the object is below the water surface, the wave moves towards the object.

Notice that based on these rules the directions of waves are independent of the direc-

tion of motion of the interacting object. This is a particularly useful property, since the object motion is not limited to being just up or down as in the four cases we considered. Furthermore, the object can have a rotation or even some deformation, which makes it impossible to talk about a single direction of the object motion.

4.1.4.3. Wave Size Heuristics

Wave size is the length of a generated wave at a certain time step. We do not use the term "wave length" to represent this value, because in our system wavelength refers to the collective length of multiple waves generated in consecutive time steps. You can see this in Figure 4.6. Wave size, on the other hand, determines the horizontal area on the water surface that is displaced by the motion of the object within a time step. Multiple waves generated at consecutive time steps can collectively represent a wave shape with a larger wavelength as compared to the wave size of each generated wave.

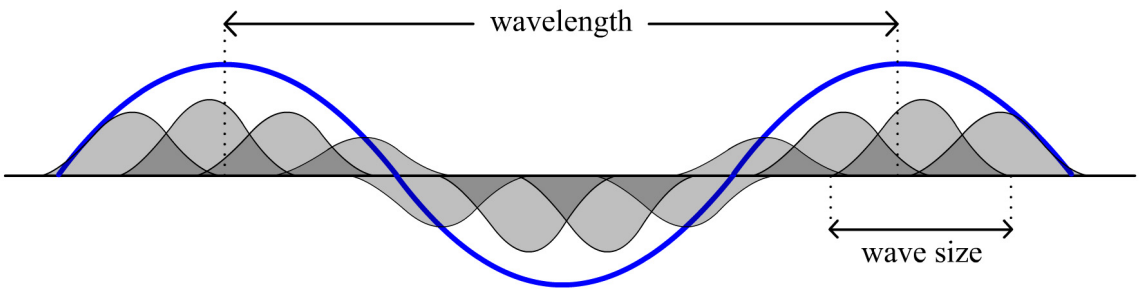


Figure 4.6. Difference between wave size and wavelength.

Using the principles of the wave placement heuristics, we can determine the volume of water displaced by each generated wave. Now that we know the total volume, we

need to find a way of using this information to determine the horizontal wave size and the amplitude of the wave. If wave size or amplitude can be determined, the other one can be computed using the wave volume.

Unfortunately, there is not enough information for determining the wave size or the wave amplitude independently from one another. Consider an object falling into water. We know that the object will induce water motion around itself and we assume that it will generate waves as a result of it. However, we do not know what the scale of this motion will be, i.e. how far this motion will stretch from the surface of the object. The scale of the induced motion is affected by various factors, including the shape of the object and the viscosity of water. Unfortunately, we do not have a good way of estimating this size, and without knowing it we cannot properly estimate the size of generated waves.

Since we do not have a proper way of estimating the wave size, we assume that it is constant for a constant time step size. Thus, we make the horizontal sizes of generated waves constant as a user defined parameter. This constant wave size assumption also helps when it comes to designing an efficient implementation as discussed in Chapter V.

One might expect to see waves with larger horizontal size when larger and heavier objects interact with water as compared to smaller and lighter objects. This behavior can be depicted using waves with constant size that are generated in consecutive time steps. Note that even though the wave size is constant, we can represent wave trains with different wavelengths as a superposition of multiple waves generated at consecutive time steps (Figure 4.6). Since heavier objects would have a larger mo-

mentum, the change in their velocity would be smaller as compared to lighter objects. Therefore, heavier objects would create similar waves at each time step and due to superposition would result in waves with larger wavelength.

4.1.5. Wave Particle Generation

Based on the principles explained above, when using wave particles for simulating water, the whole object to water coupling process boils down to wave particle generation. In this section we describe the wave particle generation process in detail.

The wave particle generation procedure generates wave particles around the interacting object based on the shape and the motion of the object and the shape and motion of the water surface around it at the time of computation. The existing wave particles are not directly used in this procedure. However, their presence changes the shape and the motion of the water surface, which in turn affects wave particle generation. The shape of the water surface is primarily used for determining which faces of the object are inside the water, while the motion of the water surface is used for determining the velocity inside the water volume due to wave motion. Since we do not have a full 3D fluid simulation system, this water velocity ignores the presence of the object and only accounts for the wave motion. The velocity of water on the surface can be directly taken from the wave particle simulation. Ignoring the presence of the object, the velocity inside the water volume is typically in the same direction as the surface velocity, but scales down exponentially deeper into the volume [Gerstner 1802].

For wave generation computations, we need to know the relative motion of the object

as compared to the motion of the water body. If the object is moving exactly the same as the water around it, it should not create any waves at all. Similarly, if the object is steady but the water around it is moving, the object should generate waves. Therefore, in the rest of this section when we talk about the object motion, we always refer to the relative motion of the object as compared to the motion of water.

The wave particle generation procedure is executed at each time step to generate wave particles around each interacting object. This procedure has three steps:

- Computing the volume of water displaced by the object,
- Finding the position and the net effect of this displaced volume on the water surface, and
- Generating wave particles on the water surface based on this net effect.

For the sake of simplicity, assume that the interacting object is represented by a triangular mesh. In the first step, we consider each face of the interacting object and compute the volume of fluid displaced by this face. This volume is equal to the volume traced by the motion of the face inside the water within the time step. We call this volume the *volume effect* of the face. This volume effect is positive when the face is *pushing* the water such that the motion vector of the object is on the front side of the face (i.e. the dot product of the motion vector and the surface normal of the face is positive). Similarly, the volume effect is negative, when the face is *pulling* the water such that the motion vector is on the back side of the face (i.e. the dot product of the motion vector and the surface normal is negative). A positive volume effect eventually translates to a wave with positive amplitude and a negative volume effect means negative amplitude. Assuming that the object velocity is constant within a

time step Δt, we can write the volume effect of a face as

$$V_{effect} = A_{face}(\mathbf{U}_{face} \cdot \mathbf{N}_{face})\Delta t \ , \tag{4.1}$$

where A_{face} is the area, $\mathbf{U}_{face}$ is the relative velocity inside the water, and $\mathbf{N}_{face}$ is the surface normal of the face.

Once the volume effect of each face is computed, the next step is to determine the positions on the water surface where each volume effect should be applied. For this task we use the wave placement heuristic described above. If a face is on the top side of the object, its volume effect is placed right above the face on the water surface. If the face is not on the top side of the object, we distribute its wave effect to the sides of the object. The sides of the object in this context are defined as the edges of the object silhouette as seen from a top view. As suggested in the wave placement heuristics, the amount of volume effect for a point on the silhouette boundary depends on the distance of the point from the face that generates the volume effect.

After we distribute the volume effect of each face onto the water surface, we know the volume of waves that should be generated at any point. A wave particle is generated for each wave effect on the water surface such that the volume of water displaced by the wave particle corresponds to the magnitude of the volume effect. The sign of the wave particle amplitude is the same as the sign of the volume effect. The radii of all the generated wave particles are the same based on the wave size heuristic. Using Equation 3.11 we can write the volume V of a wave particle as

$$\begin{aligned} V &= \int_0^r \int_0^{2\pi} \frac{a}{2}\left(\cos\left(\frac{2\pi u}{r}\right)+1\right) u \ \mathrm{d}\theta \ \mathrm{d}u \\ &= \frac{\pi}{2}\, a\, r^2 \ , \end{aligned} \tag{4.2}$$

where a is the amplitude and r is the radius of the wave particle. Note that this equation only considers the vertical deviation and ignores the change in wave particle volume due to horizontal deviation caused by the wave particle.

Now that we know the amplitude of the wave particles, we need to decide wave particle directions and dispersion angles. Consider the object silhouette as seen from a top view. The volume effects on the water surface are either inside this silhouette or on the boundary of the silhouette. If the volume effect is inside the boundary, we generate a wave particle in a random direction with dispersion angle $\alpha = 2\pi$, thus forming a ripple. Note that since $\alpha = 2\pi$, the direction of the generated wave particle makes no difference in practice. If a volume effect is placed on the boundary of the object silhouette, we can use the shape of the silhouette boundary to determine the wave direction and the dispersion angle. In this case the dispersion angle α comes directly from the curvature κ of the silhouette boundary at the position that the wave particle is generated. Let r denote the radius of the wave particle. By substituting $2r$ as the wave particle width w in Equation 3.10, we can calculate the dispersion angle as

$$\alpha = 2\, r\, \kappa\, . \tag{4.3}$$

The direction of the wave particle on the boundary of the silhouette depends on the vertical position of the object at the boundary. In other words, if a part of the object is above the water surface at the silhouette boundary point at which the wave particle is generated, the direction of the wave particle is assigned as the outward normal direction of the object silhouette at that point. If the object is below the water surface at that point, the wave particle gets the opposite direction, towards the object. Note that this procedure is consistent with the wave direction heuristics discussed in Section 4.1.4.2.

As one would expect, the effect of an object moving inside the water volume on the water surface depends on how close the object is to the surface. As the object goes deep inside the water, the magnitude of this effect on the water surface approaches zero. For emulating this behavior, we exponentially scale down the volume effect of a face based on how deep it is inside the water surface. As a result, when the object is closer to the water surface it generates waves with larger amplitudes than when it is deeper inside the water volume.

4.1.6. Limitations

The wave generation system described above is designed to be simple and efficient. While in practice it can generate plausible results, it is a vast simplification of a rather complicated phenomenon. Therefore, it has significant limitations.

First and foremost, it is questionable as to whether this wave generation system can produce a valid solution to the wave equation. As we discussed in Chapter III, the wave particle system is a valid solution to the wave equation only if each wave particle has two identical neighbors on either side. If the wave generation system can generate wave particles that obey this restriction, such that each generated wave particle has two identical neighbors on either side, the wave particle system preserves this condition assuming that there are no boundaries (waves on an open ocean) or the boundaries are rectangular (a rectangular pool). When this condition is violated, the wave particles do not provide an accurate solution to the wave equation. Unfortunately, a wave generation system that would satisfy this condition can only generate circular ripples, since any other wave shape would have to violate this condition at

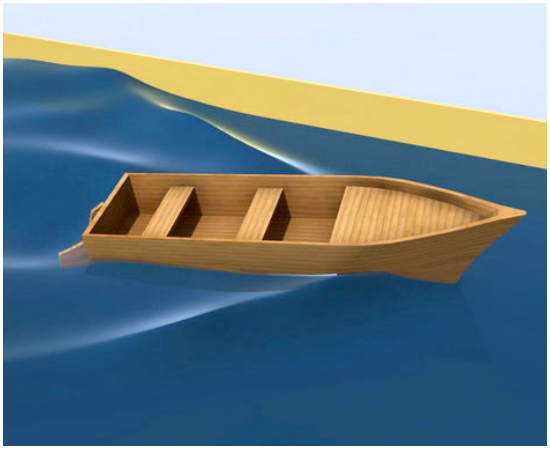
some point. The wave generation system described above can generate wave shapes other than circular ripples; therefore, the wave equation is violated as soon as the wave particles are generated. While the wave particle system is able to preserve the validity of the wave equation solution, if the wave equation is not satisfied at the time that the wave particles are generated, the wave particle system cannot correct this initial error.

On the other hand, places on the water surface where the wave equation is violated due to this wave generation process are expected to be relatively small. Waves that are generated due to object motion rapidly expand as they propagate. This expansion of waves is handled by wave particle subdivision as discussed in Chapter III. As a result, each generated wave particle turns into a finite wavefront with multiple wave particles as they propagate. The wave equation is violated only at the end points of these wavefronts, since a wave particle on one end point of a finite wavefront does not have two identical wave particles on either side, but one side only. However, for the rest of the wavefront, the wave equation is locally satisfied.

Furthermore, the wave generation system is rather limited when it comes to accurately depicting actual wave generation due to interaction of floating objects with water. This is an expected limitation, since the wave generation procedure is highly simplified, while the actual wave formation is a rather complicated phenomenon. Unfortunately, without a full 3D fluid simulation, we will always be limited in the accuracy of the waves created.

One limitation we observed in terms of depicting physical reality is that the wave patterns generated by moving boats in reality deviate from those produced with this

http://weblog.pell.portland.or.us/~orc/2005/08/17/000/



(a) (b)

Figure 4.7. Comparison of boat wakes generated by our simulation to real boat wakes. (a) Photograph of waves generated by a moving boat, (b) waves generated by our real-time wave generation technique

wave generation technique. Figure 4.7a shows a tugboat and wave generated by the motion of this tugboat. Notice the wave patterns produced by the motion of the boat. Unfortunately, we observed that the wave generation technique described here can only capture a rough shape of these waves (Figure 4.7b) and the repetitive wave patterns of the tugboat example do not form with our wave generation method. Please keep in mind that the wave generation system is intended to be general and it is not tailored for simulating boat wakes or any other specific situation. Yet, this example shows us the limits of the wave generation system as compared to reality. Since the wave simulation system has many significant inherent simplifications, it is difficult to determine which one (or ones) of these simplifications is a major factor for the wave generation system's inability to properly generate realistic boat wakes. We discuss this issue further in Chapter VII.

Finally, in the beginning of this chapter we argued that an ideal wave generation system can automatically handle the interaction of a floating object with the existing waves in the system. While this assessment may be true, our wave generation system is far from being ideal. The wave generation system takes the existing waves into account for determining the shape and the velocity of the water surface. However, since we know that the wave generation system does not produce physically accurate results, we do not expect it to perfectly handle the interaction of the floating objects with the existing wave particles.

4.2. Fluid to Object Coupling

Fluid to object coupling is the other half of water-object interaction and it handles the effect of the water body onto the motion of interacting objects. This effect is passed onto the interacting objects by applying forces based on the positions of the objects and the relative velocities of their faces as compared to the relative velocity of water. These forces are then used by a rigid body simulator to compute the motion of interacting objects.

The forces acting on the object due to water are buoyancy force, drag force, and lift force. We first discuss the buoyancy force in detail and then talk about drag and lift forces together.

4.2.1. Buoyancy Force

Buoyancy force is the force that keeps objects that have lower average density than water afloat on the water surface. The magnitude of the buoyancy force is proportional to the volume of the object inside the water.

The buoyancy force is applied only to the part of the object that is inside the water volume. While this concept of "part of the object inside the water" is quite natural for any person to comprehend, it rather misrepresents what actually happens in physical reality. When an object is partially submerged in the water, our general perception is that the part of the object that is "inside" the water intersects with the water volume. It is as if water and the submerged part of the object coincides together at the intersection of these two volumes. However, what actually happens is that when an object is partially submerged, it pushes the water around it, thereby elevating the water level. It is the volume of this elevation that the buoyancy force is proportional to, which perceptually can be interpreted as the volume of the object inside the water.

However, our aim here is to simulate the interaction of objects with large bodies of water. Therefore, we do not consider the elevation of water level due to a submerged object, and we compute buoyancy force using the volume $V_{inwater}$ of the object inside the water, such that

$$\mathbf{F}_{buoyancy} = -\mathbf{g}\,\rho\,V_{inwater}\ , \tag{4.4}$$

where $\mathbf{g}$ is the gravitational acceleration vector and ρ is the density of the fluid. This buoyancy force is applied at the centroid (center of the volume) of the part of the object inside the water.

Note that the buoyancy force has no dependence on object velocity. Therefore, the buoyancy force alone is never enough for water to object coupling. Note that the direction of the buoyancy force is always in the opposite direction of gravity and that it conserves energy. Therefore, when the buoyancy force is used alone without any other interaction force acting on the interacting objects, the objects oscillate in and out of the water without coming to a rest position. This is analogous to the harmonic motion due to springs.

4.2.2. Drag and Lift Forces

Drag and lift forces are the dynamic forces acting on an object due to its motion relative to the water. These forces are the result of fluid pressure on the surface of the object as well as the friction force between the water and the object. In fluid mechanics the drag force is defined as the component of the total dynamic force acting on the object in the direction opposite to the relative motion. Therefore, the drag force always acts to stop the object. The lift force is defined as the component of the total dynamic force acting on the object perpendicular to the direction of relative motion. Hence, the lift force effectively changes the motion direction of the object and its direction depends on the shape and the orientation of the object relative to its motion.

In fluid mechanics, the magnitudes of the drag and lift forces can be written as

$$F_{drag} = \frac{1}{2}\,\rho\,C_D\,A\,U^2 \quad \text{and} \tag{4.5}$$

$$F_{lift} = \frac{1}{2}\,\rho\,C_L\,A\,U^2\,, \tag{4.6}$$

where ρ is the density of the fluid, C_D and C_L are the drag and lift coefficients of the object, A is the effective area, and U is the magnitude of the velocity of the object relative to the fluid[1] [Munson et al. 2006].

Unfortunately, we cannot take these equations and use them to compute the drag and lift forces on arbitrarily shaped objects moving in water for several reasons:

- First of all C_D and C_L highly depend on the shape of the object as well as the orientation of the motion vector. Since the fluid motion around an object can be rather complicated, there is no closed form solution for computing C_D and C_L for an arbitrary object. These values are generally obtained via measurements.
- Moreover, C_D and C_L also depend on U. The values of these constants can change significantly when U is very large or very small.
- Furthermore, the effective area A also depends on the shape of the object and the direction of motion. While in many cases A is simply the projected area of the object in the direction of motion, this does not have to be the case for an arbitrarily shaped object.
- Finally, while the direction of the drag force is in the opposite direction of motion, the direction of the lift force is rather ambiguous. We know that the lift force is perpendicular to the direction of motion, which restricts the set of possible directions to the perpendicular plane only. We do not know which direction on that plane would be the correct direction for an arbitrarily shaped object.

[1]In fluid mechanics U is often used as the velocity of the fluid and the object is considered stationary

For having a general solution to water to object coupling, it is important that we can compute drag and lift forces on an arbitrarily shaped object moving in an arbitrary direction with an arbitrary orientation. However, in reality these forces are measured for carefully specified conditions. Even though it might be possible to estimate these forces using a full 3D fluid simulation, this would not be a practical solution and would not be appropriate for a real-time simulation system. For this reason we provide simplified approximations of drag and lift forces that are suitable for computer graphics purposes.

For the sake of simplicity, let's assume that the interacting object is represented by a triangular mesh. We begin our simplification by assuming that the drag and lift forces can be computed on each face of the object independently. The loss in accuracy due to this assumption can vary significantly depending on the shape of the object. If the object is convex, we make relatively little or no error with this assumption. However, for an arbitrarily shaped object, the error introduced by this assumption can be rather significant.

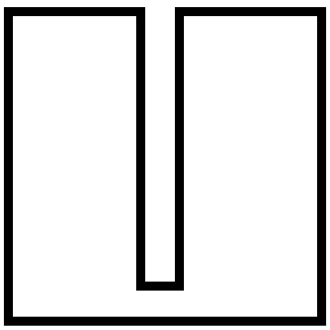


Figure 4.8. A box shaped object with a narrow cavity on one side. Such objects can introduce extra error if drag and lift forces are computed independently on each face.

For understanding the potential magnitude of the error caused by computing using each face of the object independently, consider the object in Figure 4.8. This box shaped object has a narrow cavity on one side. If this object is moving left or right, and if we compute the forces applied to each face of the object independently by ignoring the presence of all other faces, the faces on either side of the cavity would have a strong effect, especially on the drag force. However, in reality the fluid flow inside the cavity would be much different than what is suggested by this assumption; thus, we might actually get a more accurate drag force computation by completely ignoring the the cavity. While this counter example shows that computing drag and lift forces independently on each face of an object can introduce significant error under certain circumstances, the error is smaller for simpler shapes. We rely on this assumption in our approach, because treating faces independently is much simpler, it avoids the need for a full 3D fluid simulation, the forces can be computed quickly for arbitrary objects, and the error is low for simple shapes. Note that when an object has such an irregular shape, certain faces of the object (like the ones inside the cavity) can be ignored or their effect can be scaled down while computing lift and drag forces.

Our second simplification is assuming that drag and lift coefficients C_D and C_L are constant for each face. This is a somewhat less radical assumption than our first assumption above. Keeping C_D and C_L constant ignores their dependence on the orientation as compared to the direction of motion. To introduce the dependence on the direction of motion we use the effective area A, which is defined as

$$A = \left(\frac{\mathbf{N} \cdot \mathbf{U}}{|\mathbf{U}|} \, \xi + (1 - \xi) \right) \, A_{face} \, , \tag{4.7}$$

where A_{face} is the area of the face inside the fluid and $0 \leq \xi \leq 1$ is a user defined parameter. This ξ parameter determines the amount of motion direction dependency

for computing the effective area. When ξ is zero, the effective area becomes equal to the area of the face regardless of the direction of motion. Setting ξ as 1 makes the area equal to the projected area of the face on the plane perpendicular to the direction of motion; thus, the effective area becomes highly dependent of the direction of motion.

Considering that we are trying to compute the drag and lift forces on each face separately, it is possible to come up with a more accurate formulation for C_D, C_L, and A and their dependence on orientation of the face. Yet, since the faces are not really independent but rather are pieces of a larger object, it is unlikely that one can get higher accuracy with merely a more accurate formulation for an independent face. Therefore, we choose simplicity over additional work for questionable gain, and keep our formulation simple as stated above.

Finally, we need to determine the direction of the lift force, which can be in any direction on the place that is perpendicular to the direction of motion. Our assumption is that the direction of the lift force is in the plane defined by the velocity vector $\mathbf{U}$ and the surface normal $\mathbf{N}$ of the face, and that it is on the opposite side of the face from $\mathbf{U}$. Figure 4.9 shows the directions of drag and lift forces on a triangle.

Using the simplifications stated above we can write the drag and lift forces acting on a face as

$$\mathbf{F}_{drag} = -\frac{1}{2}\,\rho\,C_D\,A\,|\mathbf{U}|\,\mathbf{U}\,, \tag{4.8}$$

$$\mathbf{F}_{lift} = -\frac{1}{2}\,\rho\,C_L\,A\,|\mathbf{U}|\,\left(\mathbf{U}\times\frac{(\mathbf{N}\times\mathbf{U})}{|(\mathbf{N}\times\mathbf{U})|}\right)\,. \tag{4.9}$$

The total drag and lift forces acting on the object are the sums of all drag and lift

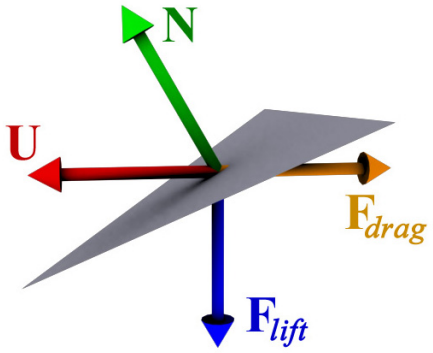


Figure 4.9. Directions of drag and lift forces on an object face moving with velocity **U** relative to the fluid's local velocity.

forces on all faces of the object. Each one of these forces are applied at the center of the face, in order to compute the torque induced by these forces.

Please note that the magnitudes of both drag and lift forces are linearly proportional to the magnitude of the velocity squared $|\mathbf{U}|^2$ as stated in equations 4.5 and 4.6 as well as equations 4.8 and 4.9. A rather common mistake that has been repeatedly made in computer graphics is formulating velocity dependent forces as linearly proportional to $|\mathbf{U}|$, instead of $|\mathbf{U}|^2$. This ad-hoc formulation results in rather unrealistic floating object motion: when the object is slow the drag force becomes too large, but when the object is fast the drag force becomes too weak to properly and realistically stop the object.

Once we have drag and lift forces along with the buoyancy force, we have a complete method for handling water to object coupling.

CHAPTER V

IMPLEMENTATION OF WAVE PARTICLES

The power of the wave particles technique lies in the fact that it permits very efficient implementations. Without such an efficient implementation, simulating a desired number of particles can be considerably slower. Unfortunately, many details that lead to an efficient implementation of wave particles are far from being trivial.

The first half of this chapter describes several tips and tricks used in the implementation of wave particles to achieve high frame rates. The second half of this chapter provides information on how to effectively render a water surface in a real-time environment.

5.1. Implementing the Wave Particle Simulation

While a straightforward implementation of wave particles is very easy, the performance of such an implementation can be less than desired. In this section, we first overview our implementation of the wave particles method, and then we provide details about nontrivial steps and discuss how parts of this implementation can be significantly accelerated.

At each time step the simulation system has to perform certain operations to compute

the shape of the water surface as well as the positions of the interacting objects. We handle these operations in five steps:

- **Wave Particle Iteration**: moves the wave particles to their new positions and handles subdivision and reflection events that occur within the time step.
- **Water to Object Coupling**: calculates the forces acting on each object due to water to object coupling.
- **Rigid Body Simulation**: moves the objects to their new positions, taking the water to object coupling forces into account.
- **Wave Particle Generation**: creates new wave particles based on the object motion.
- **Height Field Generation**: computes the new height field shape from the deformations represented by the wave particles.

In the subsequent subsections we discuss these steps in detail. Note that the ordering of these steps is not strict and some of them might be executed in parallel. At the end of this chapter, we explain how this system can be designed to make use of thread level parallelization.

5.1.1. Wave Particle Iteration

The task of wave particle iteration is computing the current positions of wave particles as well as handling all subdivision and reflection events that should take place within the time step. While a straightforward implementation of wave particle iteration can

be considered trivial, this step can be significantly accelerated with a more elaborate implementation. We first discuss accelerating the computation of new wave particle positions, and then we explain how the wave particle subdivision and reflection events can be efficiently handled in this framework.

5.1.1.1. Computing Wave Particle Positions

Wave particles form an exceptionally simplified particle system. First of all, wave particles move on a 2D plane and they do not interact with each other. Moreover, no external forces act upon the wave particle system, so all wave particles more with a constant speed and their directions do not change apart from the subdivision and reflection events. Therefore, computing the current state of the wave particle system from its state at the previous time step is a rather trivial operation, and it can easily be implemented with a single loop that visits all wave particles and updates their positions. Since the wave particles do not interact with each other, this operation can be carried out in parallel and computed on the GPU.

Furthermore, updating the positions of all wave particles at each time step can be completely eliminated. Since wave particles move with constant velocity, if we know the position of a wave particle $\mathbf{x}_0$ at any time t_o, we can easily compute its current position as follows:

$$\mathbf{x} = \mathbf{x}_0 + v\,\hat{\mathbf{u}}\,(t - t_0)\;, \tag{5.1}$$

where v is the constant wave speed and $\hat{\mathbf{u}}$ is the propagation direction of the wave particle. Therefore, we do not need to compute and store the value of $\mathbf{x}$ for each wave particle at each time step. Instead, we can easily compute this value from $\mathbf{x}_0$, $\hat{\mathbf{u}}$, and

t_0 when needed.

While we still have to perform the same calculation to find the current position of a wave particle, the fact that we do not need to store the current position of a wave particle makes a significant difference in terms of performance. Note that if we update the wave particle positions, the copy of the wave particle data in the CPU cache becomes "dirty," and it needs to be copied back to the main memory. Avoiding the unnecessary updates of the wave particle positions eliminates a large portion of memory write events regarding wave particle iteration. Furthermore, in a multi-core system, if multiple CPU cores are accessing the same memory cache block, having to update the data in the cache may result in further decrease in cache performance.

Note that the main performance cost of updating wave particle positions comes from memory read and write operations, rather than the few multiplication and addition operations we need for computing Equation 5.1. The size of the wave particle data in memory is typically many times that of the whole CPU cache, and it can be quite large if a large number of wave particles are used. By eliminating this update of wave particle positions, we avoid all cost related to this operation and the only penalty of doing so is a few additional multiplication and addition operations for computing Equation 5.1 whenever we need the current position of a wave particle.

Therefore, in our implementation of the wave particle system, we do not keep the current positions of wave particles in the wave particle data. Instead, we keep the origin of each wave particle as well as the time when the wave particle was or will be at the origin position. The fact that we also need to keep the time at origin information does not increase the size of the wave particle structure, since we also

use this information for computing the age of each wave particle, which is needed for the subdivision operation.

As we discussed before, damping in this system is optional. If we would like to add some damping, we can do so by computing the damped amplitude of the wave particle a_{damped} from its undamped amplitude a as

$$a_{damped} = a \exp\left(-\zeta\left(t - t_0\right)\right) \ , \tag{5.2}$$

where ζ is the damping coefficient, t is the current time, and t_0 is the time at the wave particle origin.

5.1.1.2. Subdivisions and Reflections

Unfortunately, we cannot eliminate subdivision and reflection operations, like we eliminated updating the wave particle positions. Therefore, we need to subdivide and reflect each wave particle as needed within the time step.

Fortunately, typically only a very small portion of all wave particles need to go through subdivision and reflection operations at an arbitrary time step. Therefore, if we can find a way to visit only those wave particles that have to go through the subdivision and reflection operations within the time step, we avoid accessing all of the wave particle data.

This can be done using time tables for subdivision and reflection events. The time table for subdivision keeps a list of all wave particles that need to subdivide for each discrete time step, and the time table for reflection keeps a similar list for reflections.

Note that when a wave particle is generated, the exact times that the particle will go through a subdivision and a reflection event can be computed, since the boundaries for reflection are stationary. If the particle should go through a subdivision before reflection, we place the wave particle in the subdivision list for the time step at which it much undergo subdivision; otherwise, we place it in the reflection list.

At each time step, we only visit the wave particles that are in the subdivision and reflection lists of the current time step. We perform the subdivision or reflection operation on each one of these wave particles, and then place the wave particle on another list based on when it will subdivide or reflect again. If a wave particle has to go through multiple subdivisions, reflections, or a combination of these two operations, we perform them all before placing the wave particle in a new list.

Efficiently implementing these time tables can be tricky. For high performance, we must avoid dynamic memory allocation at run time, so the memory for these time tables must be pre-allocated in the beginning of the simulation. Unfortunately, we have no way of knowing beforehand how much memory we will need for each list in the time tables. In practice, the sizes of these lists can drastically different. No matter how unlikely, it is possible that all wave particles in the system subdivide or reflect at the very same time, so one of these many lists might have to be large enough to contain all the wave particles, while most others might be empty. Yet, we know that the total memory we need for these lists is bounded by the maximum number of wave particles in the system, since each wave particle appears only once, and in only one list.

As a solution, we use a linked list structure for each list shown in Figure 5.1. Both

subdivision and reflection tables keep a number of pointers, each of which correspond to a discrete time step in the future. These pointers are either null (meaning that the list is empty), or point to the first wave particle in the list. Each wave particle has a next pointer, which is either null (end of the list) or points to the next wave particle in the list. In this way, we can access all wave particles at a given time and perform the subdivision or reflection operation. Note that we only need a single pointer for each wave particle, since a wave particle can be either in a subdivision list or a reflection list, but not both. Also note that on 64-bit systems, keeping a 32-bit integer index of wave particles instead of a 64-bit pointer would reduce the memory overhead for these lists.

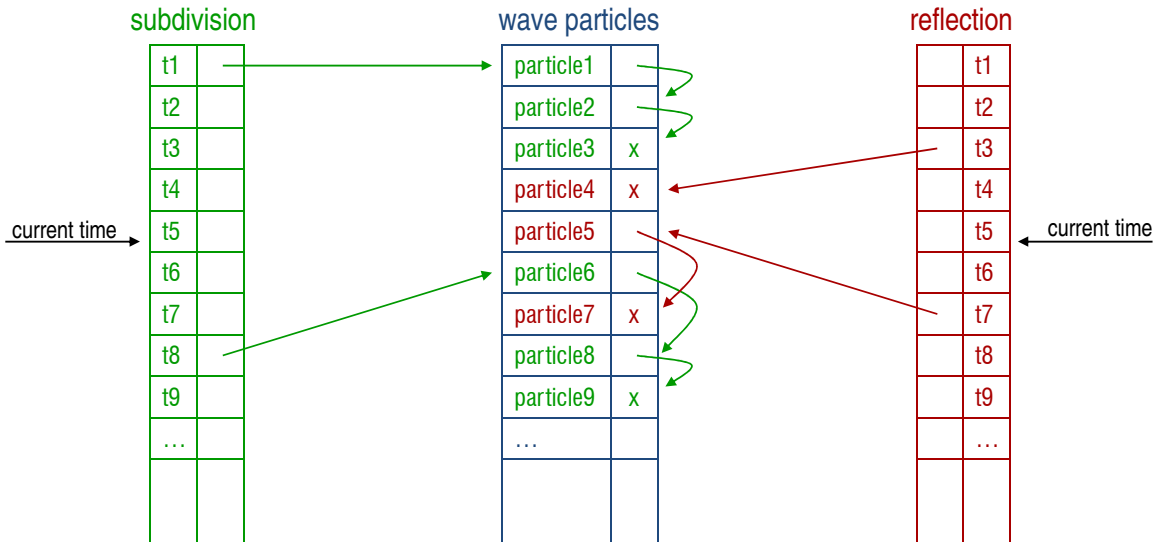


Figure 5.1. The linked list structure for accessing the subdivisions and reflections for a given time step.

Therefore, the wave particle iteration time is related to the number of subdivision and reflection events in the time step, but not directly related to the total number of wave particles in the system. This allows the iteration of millions of wave particles

with minimal cost.

Reflecting a wave particle is rather simple, since we only change the origin position and the propagation direction of the wave particle. For curved boundaries, we also change the dispersion angle and the origin time information. Note that the origin of a wave particle can lie outside the boundaries of the container. This does not produce any problems, since the origin information is merely used for computing the current position of the wave particle.

On the other hand, subdividing a wave particle is more complicated, because we spawn two more wave particles in addition to the subdividing wave particle. To avoid dynamic memory allocation at this step, we pre-allocate the memory for the maximum number of wave particles. To generate a new wave particle, we simply find an unused wave particle and flag it as "alive." Finding an unused wave particle itself might be a costly operation, if we have to examine each wave particle.

For handling this in a simple and efficient way, we keep a pointer to the next available wave particle. This pointer is initialized as the first wave particle in the beginning of the simulation, and we increment this pointer as we generate new wave particles. When this pointer reaches the end of the pre-allocated memory for all wave particles, we set it back to zero and the wave particle that this pointer points to is always assumed to be unused. The inherent assumption here is that when this pointer goes through all wave particles and comes back to the first one, the wave particle at this location must be already dead. Otherwise, we overwrite a "living" wave particle. While more elaborate solutions can be easily found, such as efficiently searching for next unused wave particle in the list, we prefer keeping this procedure simple for high

performance. If we end up overwriting a living wave particle, it means that the simulation has generated more wave particles that it can reliably handle as determined by the maximum wave particles that are pre-allocated in the beginning of the simulation. In our implementation, the possibility of overwriting living wave particles is reduced by the steps we take for improving the memory access pattern as explained below.

Another thing we must consider for high efficiency is the order of memory accesses for handling reflection and subdivision events. In general, memory accesses for handling reflection and subdivision operations can have a random pattern, since wave particles might appear in an arbitrary order in a subdivision or reflection list. Unfortunately, it is not simple to efficiently order wave particles in a linked list structure to improve the memory access pattern by making it closer to sequential. However, one thing we know is that when we subdivide a wave particle, we end up with three identical wave particles that will subdivide again all at the same time. Therefore, placing these wave particles sequentially in the memory would improve the memory access pattern, as compared to placing them in random order. Furthermore, since the two child wave particles subdivide exactly at the same time with the subdividing parent wave particle, we can also relocate the parent wave particle and place it right next to the new wave particles. By relocating the parent wave particle to an unused location together with the two new wave particles, we not only introduce some consistency to the memory access pattern, but we also use up the wave particle list somewhat more rapidly, meaning the unused wave particle pointer goes through the whole wave particle list somewhat faster. However, this actually reduces the possibility of overwriting an existing wave particle, since the wave particle is likely to be relocated during a subdivision operation before it is overwritten by another new wave particle. Note that this approach is heuristic, and does not provide any guarantee that no wave

particle will be overwritten.

5.1.2. Water to Object Coupling

We compute the water to object coupling forces for the parts of the objects that are inside the water. For determining what part of the objects are inside the water, we use the height field that is generated in the previous time step. Thus, we assume that the shape of the height field does not change much within a time step, which is not completely accurate.

We use the height field to determine the depth in water for any given position. However, the values that we read from the extended height field also have horizontal displacement. This means that the extended height field cannot directly give us the height for the point that we lookup in the extended height field. Finding the correct location on the extended height field that would give us the height at the position we want requires the inverse transformation of the final horizontal displacement field. To eliminate this complicated procedure, we convert the extended height field to a basic height field with no horizontal deformation. We can easily do so by rendering the fluid surface using the extended height field onto the basic height field texture. This overhead can be minimized by using a low resolution version of the surface for this conversion. In this basic height field texture, we also keep the water velocity at the surface that is directly taken from the external height field, as we need the water surface velocity while computing drag and lift forces.

In our implementation of the wave particles technique, we the GPU for computing water to object coupling forces. The parallel computation power of the GPU helps

accelerate this computation. Furthermore, we make use of the other features of the GPU hardware, such as the rasterizer while computing the buoyancy force and the texture unit for height field lookups. Moreover, since we compute the basic height field on the GPU from the extended height field, performing the force computations also on the GPU is convenient, as we can avoid the need to pull the basic height field data from the GPU to the main memory.

5.1.2.1. Computing the Buoyancy Forces

For computing the buoyancy force for an object, we need to find the total volume of the object inside the water and centroid (center of the volume) of this part inside the water.

We can efficiently compute the volume of the object inside the water on the GPU. For this computation, we render a low resolution image of the object as seen from a top orthogonal view (Figure 5.2b). For each fragment on each pixel, we output the depth of the fragment inside the water, or zero if the fragment is outside the water. Multiplying this depth value with the area that corresponds to a pixel gives us the volume of the water column above this fragment. For fragments that have a surface normal that is facing downward, we output a positive depth value. If the surface normal of the fragment is facing upward, we output a negative depth value. If the fragment is not inside the water, we either discard the fragment or output zero. We render the object using additive blending, so the value on each pixel in the end becomes the sum of all positive and negative depth values of all fragments that correspond to the pixel (Figure 5.2a). By outputting negative depth values for

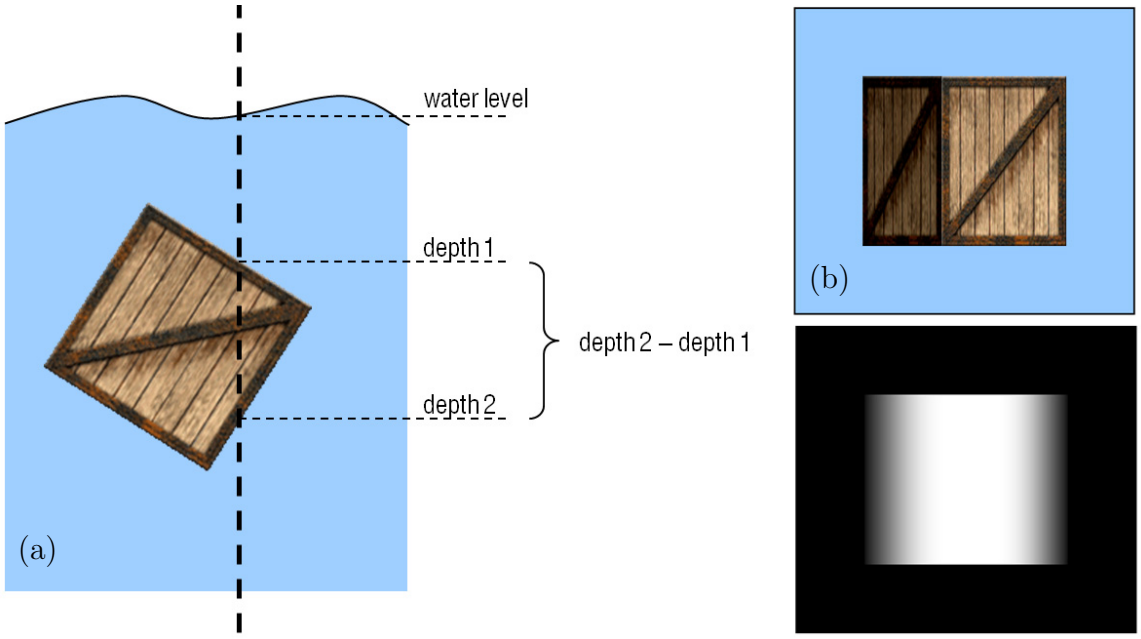


Figure 5.2. Computation of the buoyancy force on the GPU. (a) side view of the depth values that correspond to a pixel, (b) the orthographic top view of the object, (c) the final image of the volume computation on the GPU.

upward facing fragments, we discard the volume of the water column that is not inside the object. As a result, for each pixel we get the total volume of the object column inside the water (Figure 5.2c). We refer to this final image as the *buoyancy image*.

Once we have the image that shows the volume of the object inside the water for each pixel, we can compute the total volume and the horizontal position of the centroid for this volume. Note that the vertical position of the centroid cannot be computed from this image, because we do not know the vertical centroid of each pixel. However, since we know that the buoyancy force is always upward in the vertical direction, we do not need to know the vertical position of the centroid and we can apply the buoyancy force at any vertical position on the object.

Obviously, using a high-resolution buoyancy image would produce more accurate results, but it would also increase the amount of computation we need for calculating the total volume and the centroid of this volume, in addition to the extra cost of rendering a high-resolution buoyancy image. When a low-resolution buoyancy image is used, the computation becomes less accurate but more efficient. Moreover, in addition to the inaccuracy caused by the low-resolution buoyancy image, the orientation of pixels is likely to cause some bias. In other words, merely rotating the object some random amount around the vertical axis that goes through its center and then computing the buoyancy image, is likely to produce a different value for the total volume inside the water. A common technique for eliminating similar biases in computer graphics is applying a different random rotation to the object each time the computation is performed. Note that this rotation is only used for the computation purposes and the object is not actually rotated. Applying a random rotation for the buoyancy computation, replaces the bias caused by the low-resolution buoyancy image with noise. While noise is often more desirable than bias in many computer graphics algorithms, for computing the buoyancy force noise can be less desirable. This is because if we compute a slightly different buoyancy force at each time step due to noise, we might get a randomly vibrating object motion which would especially be visible when the object is about to come to a rest pose. Eliminating the random rotation and accepting the bias caused by the low-resolution buoyancy image, avoids this undesired vibration, even though the average buoyancy force applied to the object can be less accurate.

The total volume and the centroid of this volume can be computed on the GPU. One can easily write a fragment shader that reads this whole image and outputs the total volume and the centroid. Alternatively, this computation can be handled

in parallel by rendering lower and lower resolution images (multiples of 2) until the image becomes a single pixel, computing the total volume and the centroid at each step. Which one of these approaches would be more efficient depends on the specifics of the GPU hardware as well as the parameters of the simulation at hand, such as the number of objects and the resolution of the initial buoyancy images.

In our implementation, we perform this computation on the CPU by pulling the low-resolution buoyancy image we rendered to the main memory. This was convenient for us, since our rigid body simulation runs on the CPU.

5.1.2.2. Computing the Drag and Lift Forces

We compute the drag and lift forces for each face of each object separately and apply these forces at the center of the face. Again, we can use the parallel computation power of the GPU for this computation.

Obviously, if a face is outside the water, no drag or lift force is applied to the face. The simplest way of checking whether a face is inside the water is comparing the face centroid against the water height that corresponds to the position of the face centroid. This simple check would only produce a binary decision, such that a face would be considered either completely inside the water or completely outside. However, this binary decision is likely to produce problems when a face is partially in water. For example, if the centroid of a face is slightly below the water level at one frame, the whole face would be considered inside the water and we would apply drag and lift forces on this face. In the next time step, a minor motion of the object (or the water surface) might cause the centroid of this face stay outside the water, which

would eliminate drag and lift forces on this face. As a result, we might have a sudden change in the total interaction forces acting on the object, depending on the magnitudes of the drag and lift forces that correspond to this face. Such sudden changes in consecutive time steps are likely to cause undesired object motion with sudden inexplicable jumps, especially when the object has relatively large faces.

Therefore, for handling the computation of drag and lift forces properly, we need to calculate what fraction of each face is inside the water and scale the area of the face that is used for the force computation by this fraction. For simplicity, we look up the basic height field texture at the face centroid position to determine the water level for the face, and we assume that the water level is constant around the face. We provide a simple way of computing the fraction of a triangular face that is inside the water in Appendix B.

In our implementation, we perform the drag and lift force computation on the GPU and write the output to a force texture. We render each face of each object as a point primitive on this force texture. Each one of these points is written onto a different pixel of the force texture. We send the necessary information for computing drag and lift forces as vertex attributes. We compute the dynamic forces using equations 4.8, 4.9, and 4.7. Then, we copy the force texture to the main memory and use it to determine drag and lift forces on each face of each object. Note that a CUDA implementation can be more efficient for this task.

In our implementation, we apply the forces at the centroid of each face, regardless of what portion of the face is submerged in the water. A more accurate implementation would be applying these forces at the centroid of the part of the face inside the

water. While applying the drag and lift force always onto the face centroid produces reasonable accuracy with high resolution objects, when the object has large faces, this approach might provide undesired results. For example, consider a cube with 12 triangular faces (two triangles for each quadrilateral face). If we apply the drag and lift forces always at the centroid of each face, we may not get proper torque when the cube is partially submerged in the water. To avoid this, we simply used higher resolution objects. Therefore, the cube shaped objects in our simulations have 48 faces.

5.1.3. Rigid Body Simulation

The rigid body simulation is not an integral part the wave particles system. It is used for simulating the motion of the interacting objects, and most rigid body simulators can be used for this purpose. The communication between the rigid body simulation and the rest of the system is handled by applying water to object coupling forces on the simulated objects and requesting object positions and velocities from the rigid body simulator, which is used for both displaying the objects and generating waves as they move.

We used a rigid body simulator that works on the CPU in our implementation of the wave particles method. Therefore, all the forces we computed on the GPU had to be transferred to the main memory, so that they could be passed to the rigid body simulator. Since we perform most of the calculations on the GPU, a rigid body simulator that works on the GPU might have been more suitable.

Since the rigid body simulator is not an integral part of the system, it could also

be replaced by a more sophisticated simulation system that could handle deformable objects as well as rigid bodies. As long as this simulation can provide the positions and velocities of each face, and it permits applying forces to the simulated object faces, it can be used instead of our rigid body simulator.

5.1.4. Wave Particle Generation

The wave particle generation step handles the object to water coupling by creating new wave particles due to the object motion computed by the rigid body simulation. This is one of the most complicated parts of our implementation. It has three main components:

- Computing the wave effects of each face,
- Determining the positions and properties of the wave particles to be generated,
- Finally, generating wave particles.

In our implementation, we use the GPU for most of these computations. The wave effect of each face can be easily computed in parallel on the GPU. Note that the computation of the wave effects of each face has some common operations with the computation of the drag and lift forces. Therefore, for higher efficiency, we perform the wave effect computation while calculating the drag and lift forces.

For determining the positions and properties of the new wave particles, we render a low-resolution silhouette of each object inside the water as seen from an orthographic top view. Then, we place the wave effect of each face onto the pixel that corresponds

to the face centroid. We leave this wave effect at this position, if the face is on top of the object silhouette. If the face is below the top layer of the object, we distribute its wave effect to the boundary of the silhouette. While distributing the wave effects, we perform the necessary computations to determine the direction and the dispersion angle for the new wave particles.

Once we determine the properties of the wave particles to be generated, we pull this information from the GPU to the main memory, and generate wave particles on the CPU.

For simplicity and efficiency, in our implementation we handle each object independently. Because of this choice, even when nearby objects come close together and touch each other, their silhouettes are not joined in our computation. As a result, the wave effect of a face on one object cannot be distributed to the silhouette boundary of another object. Therefore, the wave particles generated on the perimeter of an object might be placed right on top of another object.

After testing various algorithms for implementing the wave generation based on the procedures provided in Section 4.1, we developed a fast but approximate method for generating waves, which we call the *silhouette pyramid* method. We first explain how this method works, then we describe how this method is integrated into our system and used for generating wave particles.

5.1.4.1. The Silhouette Pyramid Method

The silhouette pyramid method is used for distributing wave effects to the object silhouette boundary and determining generated wave particle directions and dispersion angles. This algorithm is designed for current GPU architectures, and it favors speed over accuracy.

We start with a low resolution silhouette of the interacting object as seen from an orthographic top view. The floating point color values in this image will represent the object silhouette, as well as wave effects of the faces of the object. The aim of the silhouette pyramid method is to distribute wave effects to the nearest silhouette boundary pixels and compute wave directions at these locations. The resulting image is used for generating wave particles due to object motion in the fluid.

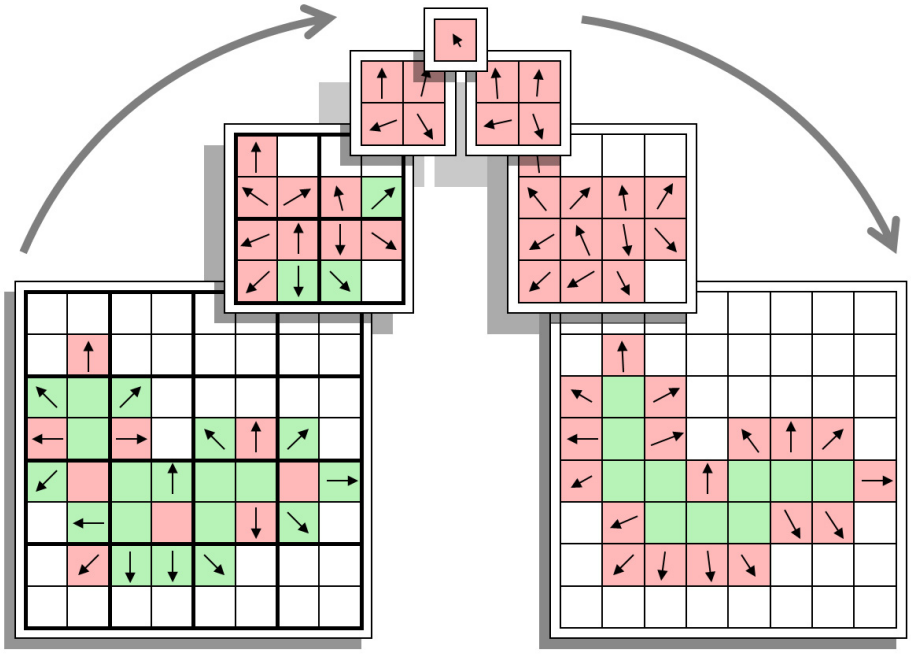


Figure 5.3. Overview of the silhouette pyramid method

The silhouette pyramid algorithm uses multiple iterations to convert the initial low-resolution silhouette image to lower and lower resolutions, averaging wave effects and directions. Then, using multiple iterations, the averaged values in the lower resolution images are used to smooth directions and distribute wave effects while moving back to higher resolutions. The above figure shows an overview of the algorithm. The steps of the silhouette pyramid method are explained in the following:

- **Step 1:**

  We begin by drawing a low resolution silhouette of the object on a floating point texture buffer. Since we will generate wave particles from each pixel of the silhouette boundary, we try to keep the texture resolution as low as possible (4×4 to 32×32 should be enough for a single object). Ideally, the size of a pixel of this low-resolution texture in world coordinates should be close to the diameter of a wave particle. We discard all the fragments outside the fluid, and write the depth in water and vertical component of the surface normal N_z onto two separate color channels. Note that if all fragments are inside the fluid volume (the object is fully submerged), all N_z values would be positive. We will use the sign of N_z to identify if the object is fully submerged at any pixel location.

- **Step 2:**

  We copy this texture onto another texture buffer with the same size, and draw each face as a point, writing the wave effect of the face. When writing the wave effect, we check the N_z and depth values against the corresponding pixel that are computed in the previous step, such that:

  - If the N_z is positive (this means that the object is fully submerged at

this pixel location), we compare the depth of the face point to the surface depth value at the pixel. We use a small bias for this comparison so that the depth values of a face recorded in the previous step do not occlude the face itself at this step. If the depth of the point is larger than the surface depth, the face is not on top of the object and the wave effect will be distributed (indirect wave effect), otherwise the face is on top of the surface and the wave effect will be direct.

- If the N_z is negative, the object is partially submerged at this pixel location and the wave effect is indirect, meaning it will be distributed to the boundaries of the silhouette.

We record the direct and indirect wave effects on two separate channels, and combine them with additive blending so that if two face points correspond to the same pixel, both values are added. The other two components of this texture will keep the depth and the N_z values of the previous texture.

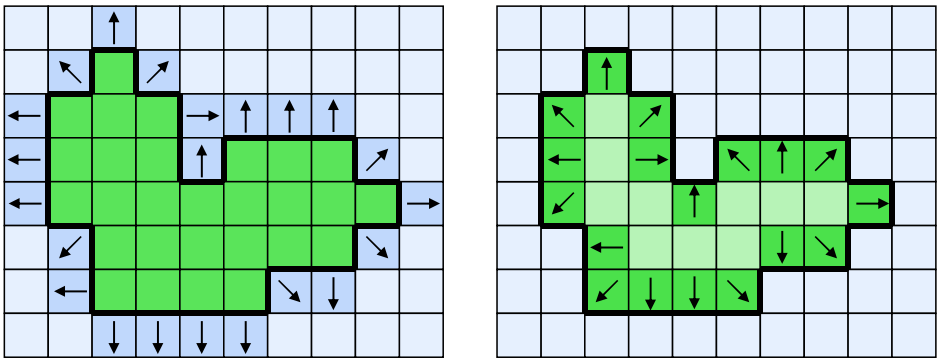
- **Step 3:**

  Now that our silhouette and wave effects are ready, we draw the texture from the previous step onto another texture buffer with the same size. This time our task is to identify boundary pixels and assign boundary directions. For each silhouette pixel, we copy the wave effect channels and check the four neighbors of the pixel. There are two alternatives for picking the boundary pixel:

  - outer boundary (pixels neighboring the object silhouette), and
  - inner boundary (pixels on the object silhouette).

  Figure 5.4 shows an example silhouette with outer and inner boundaries along with initial directions at each boundary pixel computed from four neighbors of

the pixel. Either one of these two alternatives boundary types can be used, as long as the choice of boundary type is consistent in the following steps.



Outer boundary Inner boundary

Figure 5.4. Outer and inner boundaries and initial boundary directions.

When using outer boundaries, the boundary pixels are empty pixels with a non-empty pixel as one of their four neighbors, and the boundary directions are the sum of all directions towards non-empty boundaries. Note that this direction can be a zero vector for some boundary pixels, if non-empty neighbors are on either side of the pixel.

When using inner boundaries, if any one of the four neighbors of a silhouette pixel is empty, this pixel is on the boundary and we assign a boundary direction. The boundary direction at this point is simply the sum of all directions towards empty neighboring pixels. Note that when using inner boundaries, the generated wave particles should be placed outside the silhouette using the final boundary directions, so that the generated waves do not coincide with the object silhouette.

If the N_z value at this pixel is positive (object is fully submerged at the pixel location), we invert the direction since the direction of the waves generated at

this point should be inwards from the boundary.

- **Step 4:**

  This step is repeated a number of times to smooth the directions and distribute the indirect wave effects. We draw the previous texture onto another texture buffer with half the width and height. At each pixel of this new texture, we look at the corresponding four pixels of the previous texture. We identify how many of these four pixels are on the silhouette boundary and write it on a color channel. If any of the four pixels is on the boundary, this pixel on the new texture is also considered a boundary pixel. The final direction vector is taken as the sum all four direction vectors, and the wave effect is the sum of the wave effects from all non-boundary pixels. We repeat this step a number of times (usually until the final texture size is 1x1).

- **Step 5:**

  We move back to higher resolutions by repeating this step a number of times. In this step we use two textures from the previous steps. The first texture is the immediate previous texture. Since we are moving to a higher resolution, the texture buffer of this step has twice the width and height of the first texture. The second texture is a previous texture from an earlier step, which has the same size as the current texture buffer (see Figure 5.5).

  At each pixel, we compute the directions and indirect wave effects using the two textures. The direction is simply the average of the corresponding directions in the two textures. The indirect wave effect is computed only on the pixels identified as silhouette boundary on the second texture. The value of the indirect wave effect is the sum of the wave effect value on the second texture and the wave effect value on the first texture divided by the number of boundary

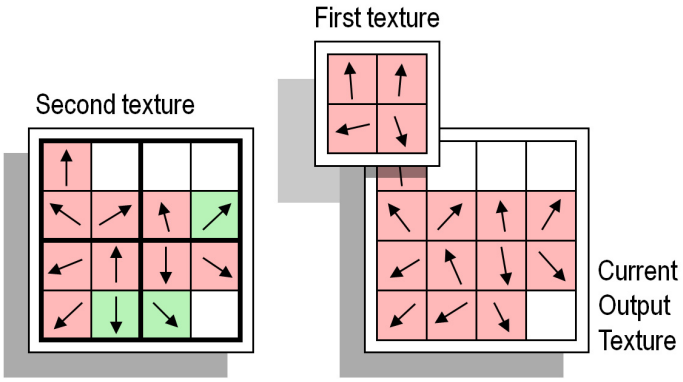


Figure 5.5. Previous textures used by the silhouette pyramid method while moving to higher resolutions.

pixels (one to four) recorded on the first texture. Through this procedure, each boundary pixel keeps half of its indirect wave effect, and distributes the other half to neighboring boundary pixels.

We repeat this step until we reach the original silhouette resolution. When we reach the original silhouette resolution, we also record the direct wave effects on to this texture by directly copying them from the texture generated at the end of Step 3.

The final texture we have at the end of these five steps has all the information we need for generating wave particles. It has four floating-point values at each pixel, of which two are the directions, one is the distributed indirect wave volume, and one is the direct wave volume. We copy this texture to main memory and generate wave particles as explained in the next subsection.

5.1.4.2. Generating Wave Particles

The silhouette pyramid method explained above renders a low-resolution silhouette image of the object such that each pixel keeps the direction for wave generation together with direct and indirect wave volumes. After we copy this texture to main memory, we examine each pixel and generate waves if the wave volume for the pixel is non-zero.

For direct waves, we create a wave particle with 2π dispersion angle at the position of the pixel. This wave particle corresponds to a circular ripple on the surface, and it immediately subdivides into multiple wave particles propagating in all directions. Therefore, the propagation direction we pick for this wave particle with 2π dispersion angle is not important, and it is randomly assigned.

For indirect waves, we assign the direction at the pixel as the wave particle direction. We compute the dispersion angle by checking the neighboring pixels that are on the boundary and calculating the angle between the directions on those pixels with the direction at the current pixel. We take the dispersion angle to be the average of these differences. For determining the wave particle center position $\mathbf{O}$, we use the following equation that is derived from Equation 3.10:

$$\ell = \frac{\omega}{\alpha} \,, \tag{5.3}$$

where α is the dispersion angle and ω is the wave particle width at the time of creation, which is taken to be the corresponding size of a pixel on the silhouette image in world coordinates. The wave particle center is computed as $\mathbf{O} = \mathbf{x} - \ell\,\hat{\mathbf{u}}$, where $\mathbf{x}$ is the corresponding position of the pixel and $\hat{\mathbf{u}}$ is the wave particle direction.

If we use damping, we adjust the wave particle amplitude such that when the damping is applied at the current time step, the amplitude corresponds to the amplitude of the wave to be generated.

Note that wave particles with zero dispersion angle never subdivide and continue propagating until they are overwritten. This might lead to unrealistic results. Therefore, we limit the dispersion angles such that when the computed dispersion angle is too close to zero (determined by a user defined interval), we assign the minimum positive dispersion angle in the permitted range.

5.1.4.3. Handling Wave Generation Bias

Both the silhouette pyramid technique and the fact that we use a low-resolution silhouette image tend to add some bias into the wave generation system. This produces undesired results, such that certain directions are favored over others by the wave generation method. To overcome this bias, we randomly rotate the silhouette image such that the orthogonal top view of the wave generating object is differently oriented at each time step. Note that the silhouette image is used for wave generation only and it is completely independent of the height field grid. Therefore, randomly rotating the orientation of which the silhouette texture is computed has no effect on any other part of the simulation.

5.1.5. Height Field Generation

The height field generation step is where the wave particle representation of the surface deformation is converted into an extended height field. The extended height field represents the vertical deformation as well as the horizontal deformation. Furthermore, in the extended height field structure we also keep the surface gradients (in x and y directions) for computing surface normals when rendering, and the water surface velocity induced by the wave particles.

Perhaps the simplest way of converting the wave particle representation to a height field is to render all wave particles onto a texture as circles. These circles are assigned texture maps that keep the local deformation function D^R for determining the wave amplitude at each pixel, as well as other textures that help us easily compute the deviations in surface positions and velocities caused by the wave particle. The final height field is generated by rendering all wave particles with additive blending. While this approach produces an accurate height field representation of the wave particles, it tends to produce too many fragments for blending. In our early tests we found that as the number of wave particles increases, height field generation with this method can quickly become too slow.

To overcome this performance limitation, we developed an approximate method that works significantly faster on current GPUs. This faster method begins with rendering all wave particles as point primitives on the height field texture with additive blending. Since point primitives are used, the number of fragments they generate for additive blending is minimal. Then, we perform two filtering passes, one of which filters the image in the horizontal direction and the other one in the vertical direction. At the

end of these two filtering passes we achieve an approximation of the extended height field representation of the wave particles in the system. While this approximate approach has a constant additional cost due to the filtering passes, it can perform orders of magnitude faster when there are a large number of active wave particles. In the remainder of this section we explain this method in detail.

5.1.5.1. Rendering Wave Particles as Points

We begin with an empty height field with the desired resolution, where all pixel values are initialized to zero. Then, we render each wave particle on this height field texture as a point primitive with additive blending.

At this step it is important to have some kind of antialiasing; otherwise, the wave particle motion that will be observed through the rendered height field will be aliased, which produces very unrealistic results. In the GPU hardware on which we implemented our system, we found that hardware antialiasing with floating point blending while rendering to a texture was too slow. Therefore, we rendered each wave particle as a point primitive with size 2, which creates a 2×2 cube. Then, we implemented antialiasing in the fragment shader such that it scales down the output values we compute by an antialiasing factor. In this way, we avoided the visual artifacts caused by aliasing.

The output of this step is the sum of the antialiased amplitudes of all wave particles rendered as point primitive, i.e. wave particle points.

5.1.5.2. Filtering Wave Particle Points

After all the wave particles are rendered on the extended height field texture, the next step is to filter this image in such a way that it will produce the extended height field from these point samples.

The proper way of doing this would be to apply some kind of a 2D convolution filter. This filter would check all the pixels around the computed pixel that are closer than the wave particle radius. For each non-empty neighboring pixel, the filter function would consider that there is a wave particle at that location with amplitude as recorded in the height field texture and then would compute the 3D displacement that is induced by that wave particle. The sum of all these displacements coming from all non-empty neighboring cells would be the total displacement from all wave particles around the computed pixel.

When simulating a rectangular pool, it is important to set the texture that we created in the previous step in mirrored repeat mode. This way, wave particles that are close to the edges of the pool will be partially reflected producing more realistic wave reflections off the rectangular boundaries.

The vertical deviation of the 2D filter can be easily handled by deriving the filter function d_z from Equation 3.11 as

$$d_z(\mathbf{p}) = \frac{1}{2}\left(\cos\left(\frac{\pi|\mathbf{p}|}{r}\right)+1\right)\ \Pi\left(\frac{|\mathbf{p}|}{2\,r}\right)\ , \tag{5.4}$$

where $\mathbf{p}$ is the vector from the filter center position to the computed point, and r is the wave particle radius.

The horizontal deviation is more complicated, because it depends on the propagation direction of the wave particle. However, we cannot reliably place the propagation directions in the height field computed in the previous step, because multiple wave particles can be on the same pixel of the height field overwriting the directions of each other. Fortunately, we can eliminate this dependency on the propagation direction using our knowledge about the wave particle system. We know that the horizontal deviation induced by a wave particle with positive amplitude moves the water surface in front of the wave particle in the opposite direction of the wave propagation, and the water surface behind it is moved in the direction of wave propagation. In that sense, a wavefront with positive amplitude pulls the water surface both in front of it and behind it towards itself making the wave shape sharper. A negative wavefront moves the water surface in the opposite direction, since its amplitude is negative. To be able to move the surface towards the wave crest, we need to know the propagation direction. On the other hand, we know that in a wave particle system that is a valid solution to the wave equation, we have two identical neighbors for each wave particle. Using this information, instead of moving the water surface along the wave propagation direction, we can pull the water surface directly towards the wave particle center. The neighboring wave particles would also pull the water surface similarly towards their centers, thereby correcting the error in the horizontal deviation direction. As a result, our filter function for horizontal deviation becomes

$$\mathbf{d}_{xy}(\mathbf{p}) = -\frac{\sqrt{2}}{2}\,\sin\left(\frac{\pi|\mathbf{p}|}{r}\right)\,\left(\cos\left(\frac{\pi|\mathbf{p}|}{r}\right)+1\right)\,\Pi\left(\frac{|\mathbf{p}|}{2\,r}\right)\,\frac{\mathbf{p}}{|\mathbf{p}|}\,, \tag{5.5}$$

which is derived from Equation 3.19. Using this formulation introduces some error to the horizontal deviation computation. Our numerical analysis concluded that the magnitude of this error in the direction of wave propagation is bounded by 7.7% of the wave amplitude, and in the perpendicular direction the error is bounded by 0.9%

of the wave amplitude. The error is larger when wave particles are further apart and the largest error happens when they are half a wave particle radius apart, which is the maximum permitted distance between neighboring wave particles on a wavefront.

5.1.5.3. Separable Filter Approximation

While the 2D filtering method explained above produces an efficient implementation of the height field generation, we can further reduce the cost of the filtering operation by approximating the 2D filter with two 1D filters. We achieve this by approximating the filter functions in Equations 5.4 and 5.5 as a tensor product of two 1D functions. In this case, the vertical deviation for point $p = \{x, y\}$ is approximated as $d_z(\mathbf{p}) \approx d_z^X(x)\, d_z^Y(y)$, where

$$d_z^X(x) = \frac{1}{2}\left(\cos\left(\frac{\pi x}{r}\right)+1\right)\ \Pi\left(\frac{x}{2r}\right)\ \text{, and} \tag{5.6}$$

$$d_z^Y(y) = \frac{1}{2}\left(\cos\left(\frac{\pi y}{r}\right)+1\right)\ \Pi\left(\frac{y}{2r}\right)\ . \tag{5.7}$$

Similarly, the horizontal deviations are approximated as

$$d_x^X(x) = -\frac{1}{2}\sin\left(\frac{\pi x}{r}\right)\left(\cos\left(\frac{\pi x}{r}\right)+1\right)\ \Pi\left(\frac{x}{2r}\right)\ , \tag{5.8}$$

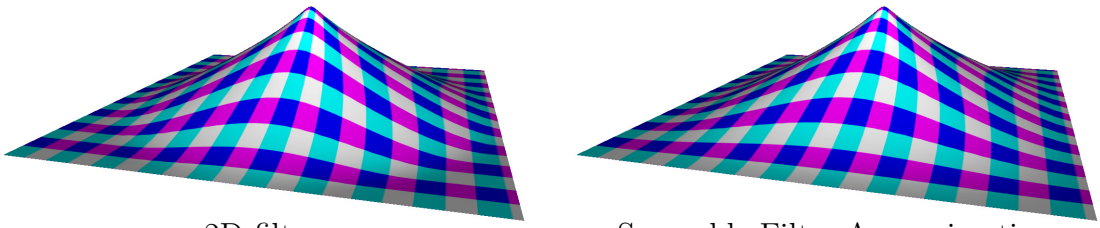
$$d_x^Y(y) = \frac{1}{4}\left(\cos\left(\frac{\pi y}{r}\right)+1\right)^2\ \Pi\left(\frac{y}{2r}\right)\ , \tag{5.9}$$

$$d_y^X(x) = \frac{1}{4}\left(\cos\left(\frac{\pi x}{r}\right)+1\right)^2\ \Pi\left(\frac{x}{2r}\right)\ \text{, and} \tag{5.10}$$

$$d_y^Y(y) = -\frac{1}{2}\sin\left(\frac{\pi y}{r}\right)\left(\cos\left(\frac{\pi y}{r}\right)+1\right)\ \Pi\left(\frac{y}{2r}\right)\ , \tag{5.11}$$

such that $d_x(\mathbf{p}) \approx d_x^X(x)\, d_x^Y(y)$ and $d_y(\mathbf{p}) \approx d_y^X(x)\, d_y^Y(y)$. Figure 5.6 compares the results of the 2D filter and the separable filter approximation for a single wave particle. As can be seen from this figure, the difference in perceived shape is minimal.

Our numerical analysis showed that the error introduced by the separable filter approximations are less than 6.4% of the wave particle amplitude for vertical deviation and less than 3.1% of the wave particle amplitude for horizontal deviation.



2D filter Separable Filter Approximation

Figure 5.6. Comparison of the filtering result produced by the 2D filter and two 1D filters using the separable filter approximation.

5.1.5.4. Additional Data in the Height Field

In our implementation, we also keep the surface gradient and the water surface velocity along with the vertical and horizontal deviations. The surface gradient is used for computing surface normals, which are used while rendering the water surface. Once we compute the vertical and horizontal deviations, we can easily calculate the surface gradients using finite differences. Similarly, the water surface velocity can be computed from the difference of the deviations on the height field at two consecutive time steps.

On the other hand, we can also compute the surface gradient and the water surface velocity using a separable filter approximation similar to the ones described above. In our implementation, we used these analytical approximations rather than the discrete approximations obtained by finite difference. Analytical approximations are especially

favorable when the surface deviation of the height field has high frequency changes. The GPU shaders we use in our implementation are included in Appendix C.

Finally, we add the surface deviations caused by ambient waves onto the deviations of the wave particles. For generating ambient waves we used the technique described in Tessendorf's SIGGRAPH 2001 course notes [2001]. This method produces a 2D texture of 3D surface deformations of ambient ocean waves at each time step. This 2D texture is seamlessly tileable and the animation of the ambient ocean waves forms a continuous loop. We precompute multiple time steps of ambient ocean waves and combine the 2D textures we gather from this computation into a 3D texture. This texture allows us to easily look up what the vertical and horizontal surface deviations are at any point at any given time. Since in our implementation we compute the surface gradient and the wave velocity using an analytical approximation, we also precompute these values for the ambient waves and store them similarly in 3D textures.

As a result, the extended height field we generate represents the final water surface to be rendered and to be used for water object interaction computations.

5.1.5.5. Projected Height Field

When simulating a pool, we can place the height field such that it represents the whole water surface of the pool. On the other hand, when simulating an ocean we cannot represent the whole water surface as a height field, because the simulated ocean water surface can be virtually infinite, while the height field is not. Therefore, when simulating an ocean scene, we generate a height field only for the part of the

water surface that is visible through the camera. This is accomplished by projecting a screen-space grid onto the part of the x-y plane that intersects with the view frustum of the camera. Figure 5.7 shows the part of the water surface that is used for height field generation based on the view frustum of a camera. Details of this approach can be found in Claes Johanson's M.S. thesis on real-time water rendering [2004]. We typically project the grid onto a slightly larger area than the view frustum of the camera to ensure that when the surface deformations of the height field are applied, the grid still covers the whole camera view.

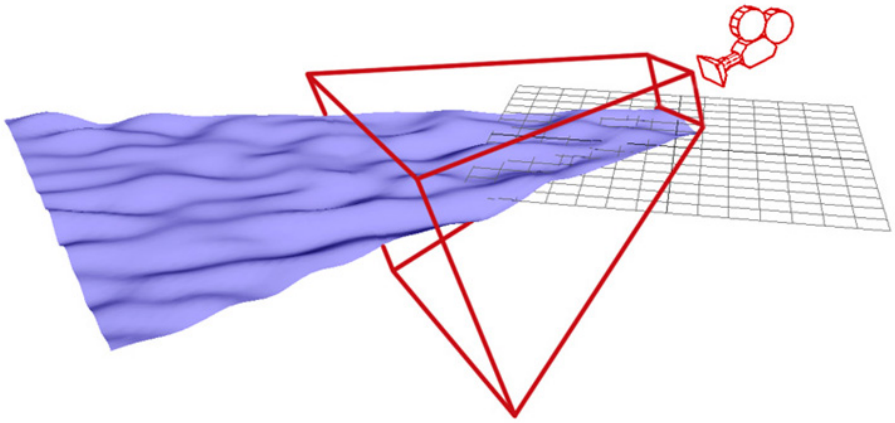


Figure 5.7. Camera attached grid for height field generation in ocean scenes.

In an ocean setting where the height field moves around with the camera, the corresponding size of a pixel in world coordinates can be different at different locations on the height field texture. Therefore, our filters get a different size at each pixel. In this case, it is useful to set a minimum size to the projected surface; otherwise, when the camera is too close to the water surface, the sizes of our filters can be too large for efficient computation.

Furthermore, the analytical approximations for the surface gradient are useful only when the size of a pixel on the height field texture is smaller than the size of the wave particles. When the pixel size is larger than the wave particle size, the filter size becomes smaller than a pixel and all the values come from a single pixel, which makes it impossible to compute the surface gradient analytically. This typically happens around gazing angles beyond a certain distance from the camera. As a solution, we switch to finite differences for pixels on the height field, wherever the corresponding size of a pixel in world coordinates is larger than the size of the wave particle. Alternatively, one can set a minimum limit to the corresponding size of a the wave particle radius on the height field, which would ensure that the surface gradient would be computed by considering multiple pixel locations on the height field texture.

5.2. Water Rendering

Properly rendering the water surface is crucial for achieving a plausible water animation. If the water surface we render does not really look like water, no matter how realistic the water animations are that we produce using our simulation system, the end result will not look like water.

Real-time rendering of water inherits certain challenges, because the water surface is highly refractive and reflective. For this reason, perhaps the best way of rendering water is ray tracing, which cannot be implemented with the desired efficiency even on modern graphics hardware that we have today. However, if real-time ray tracing becomes fast enough in the future, ray tracing will probably be the preferred way of rendering water. Until then, we need to employ various tricks to approximate

or “fake” certain visual components of water to achieve fast and plausible water rendering.

Our water rendering system has many similarities to techniques described in Claes Johanson’s M.S. thesis on real-time water rendering [2004], when it comes to handling refractions and reflections on the water surface as well as attaching the rendered water grid to the camera in ocean scenes. In our system, we also include a caustics generation method in scenes where the bottom surface underneath the water is visible through refraction. Figure 5.8 shows a frame from our real-time wave simulation and water rendering system. In the remainder of this section, we discuss how we handle different visual components of water rendering in our system.

5.2.1. Reflections

The water surface is highly reflective especially at near gazing angles. The aim of the reflection computation is to approximate the color of the reflected view ray that originates at a point on the water surface.

Since we cannot use ray tracing, we need to make some approximations to compute the reflection color. We consider two limit cases, in which the color of the reflection comes from an object at infinity and when it comes from an object at the water surface. If the reflected object is infinitely far from the origin of the reflection ray, the position of the ray origin can be ignored and the reflection can be determined from the ray direction only. On the other hand, when the reflected object is very close to the ray origin, the position of the ray origin becomes important and the ray direction can be approximated as the reflection direction from a flat surface. Based on this

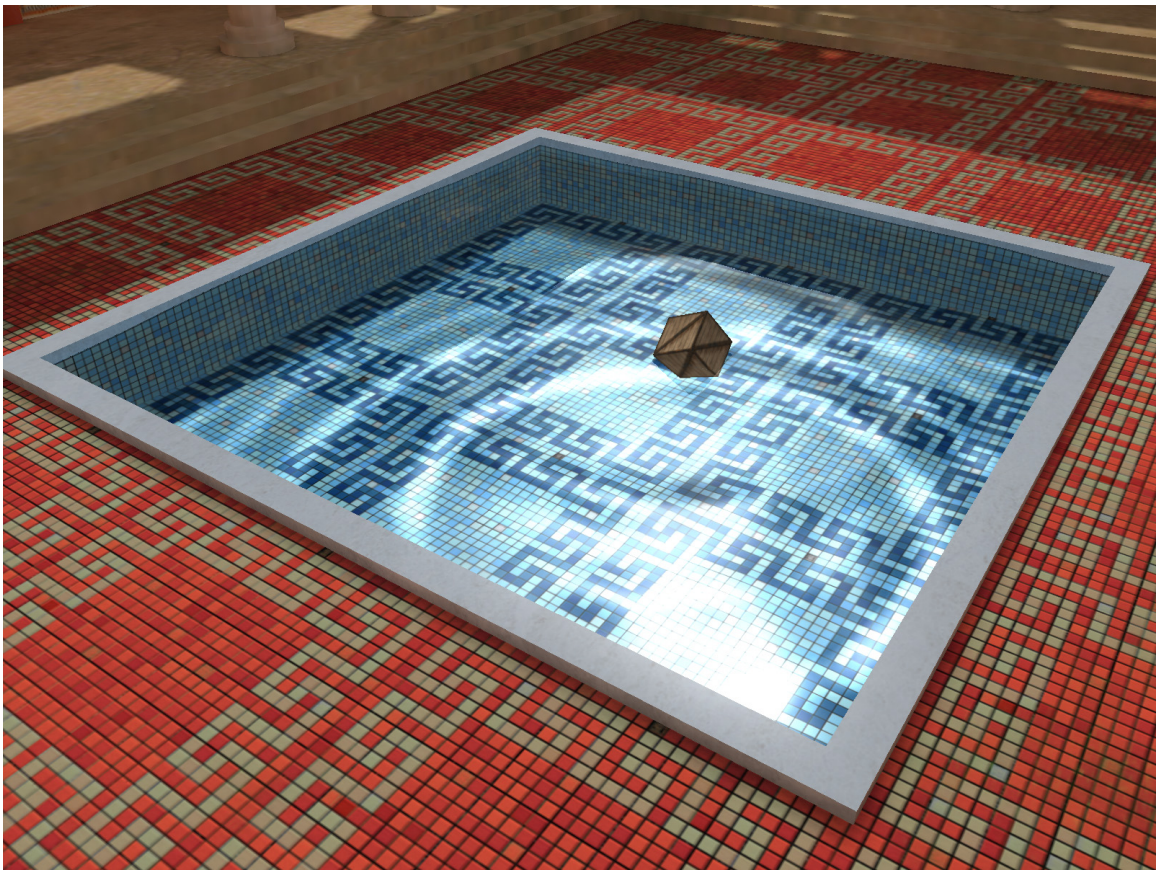


Figure 5.8. A frame captured from our real-time water rendering implementation that includes reflections, refractions, and caustics.

analysis, we separate reflections into two groups:

- Far reflections, which are reflections from distant objects, and
- Near reflections, which are reflections from objects that are close to the water surface.

Far reflections consist of the sky dome and distant objects that are far from the water surface. For determining the color of far reflections we use the reflection direction to look up color from a precomputed cube map. This cube map contains the color

of reflections for any given reflection direction. For generating this cube map, we render the six cube map faces one by one by placing a camera at the center of the scene and rendering the sky dome and all distant objects. In our implementation we perform this cube map generation only once at the beginning of the simulation, since our distant objects are stationary. If the scene has animating distant objects, the cube map for far reflections should be rendered at each frame.

Near reflections consist of objects in the scene that are interacting with water and other stationary objects that are close to the water surface, such as the edges of a pool. For computing near reflection texture, we assume that the reflection direction at any point on the water surface is equivalent to a flat water surface where the surface normal is pointing directly upward. Assuming that the water surface is flat, we can compute the near reflection texture by flipping the scene upside down, creating a mirror image of the scene as seen from the flat mirror surface, and rendering this mirrored scene from the current camera position. Reflections do not include the parts of the objects that are under the water surface. We replicate this effect by discarding objects that are underneath the water and discarding the below-surface fragments of the objects that are partially in the water.

For computing the near reflections while rendering the actual water surface, we look up the near reflection texture at the position of the pixel. If no objects are visible through the near reflection texture, we use far reflections instead.

Since the near reflection texture does not include the actual water surface normal, the generated near reflections are not affected by the surface normal changes and they look unrealistically flat. To overcome this we perturb the near reflection texture

lookup coordinate slightly by the horizontal component of the surface normal scaled with a user defined constant value, which determines the amount of this perturbation. While this perturbation overcomes the unrealistic flatness of the near reflections, it creates problems near the edges of the view where the perturbed texture coordinate can be outside the view, i.e. outside the computed near reflection texture limits. One way to overcome this is to slightly extend the view frustum while rendering the near reflections texture. Unfortunately, this approach introduces difficulties while aligning the computed reflection texture with the actual camera image and incorrect aligning can cause shifts in near reflections, which might be different at different parts of the image. Alternatively, one can scale the perturbation magnitude near the edges of the view to guarantee that the perturbed reflection direction can never fall outside the limits of the near reflection texture. In our implementation we simply changed the texture tiling option for the near reflection texture such that it uses mirrored repeat, so that even when the near reflection texture coordinate falls outside the limits, the computed near reflection color comes from some pixels on the near reflection texture that are close to the edge. While the artifacts of this simple solution can be visible, they are often difficult to notice when you are not specifically looking for them.

Another problem with the near refraction texture is that it does not include the deviation of the water surface, since it is computed assuming that the water surface is flat. As a result, cracks might be visible between objects on the water surface and their reflections. To avoid these cracks, we deform the objects while rendering the near reflection texture. For this deformation we use the basic height field that is generated for computing water to object coupling forces as explained in Section 5.1.2, so that the deviation we read from the height field does not include horizontal deviation and it corresponds to the actual vertical deviation of the computed horizontal position.

5.2.2. Refractions

Refractions are handled similar to near reflections. For producing the refraction texture, we render the part of the scene underneath the water surface, which includes the stationary objects like the bottom of the pool as well as the parts of the interacting objects that are inside the water. In the refraction texture we exclude all objects that are outside the water and fragments that are partially above the water surface. When rendering the water surface, we look up this refraction texture to find the refraction color at a given pixel in the view.

When a view ray, which originates at the camera position and goes through the shaded pixel, is refracted, its direction rotates towards the inverted normal of the water surface. As a result, the objects inside the water appear shorter, i.e. scaled down in the vertical direction. The amount of this scaling depends on the angle between the view direction and the surface normal based on Snell's law. In our implementation we account for this scaling with a constant scale factor.

Similar to the near reflections, the refractions computed at the pixel position look flat, since the refraction texture is prepared with the assumption that the water surface normal is in the upward direction everywhere. To account for the actual surface normal we can perturb the texture coordinate for refraction lookup, similar to the near reflection texture lookup.

However, when it comes to refractions, it is not as easy to come up with a user defined perturbation amount that would produce plausible results everywhere. For example, refractions of the objects that are close to the water surface should not be

perturbed too much; otherwise, they may not be aligned with the rest of the object that is outside the water. On the other hand, using a small perturbation would not be enough for objects deeper inside the water, such as the bottom of a pool.

To overcome this, together with our refraction texture, we also keep a refraction depth texture that keeps the vertical depth of each pixel in the refraction texture. While computing the refraction color at a pixel on the water surface, we first lookup the refraction depth texture without any perturbation. The depth value we read from this texture is used as an estimate for the actual depth of the refracted object. We scale the perturbation amount using this depth value, and then look up the refraction texture using the perturbed texture coordinate. As a result, refractions that are close to the water surface are perturbed a small amount, while the refractions coming from deep inside the water are perturbed more based on the water surface normal.

Just like rendering the near reflections texture, while rendering the refraction texture, we deform the objects using the height field to avoid cracks between the objects floating in water and their refractions.

5.2.3. Caustics

In water rendering, caustics are not only interesting visual elements that enhance the quality and realism of generated images, but also extremely important in providing necessary cues to perceive the shape of the rendered water surface. Figure 5.9 shows a height field surface rendered from top view with and without caustics. As can be seen from this image, caustics play a crucial role in visually defining the shape of the water surface. Without proper caustics, visual cues provided by refraction are often

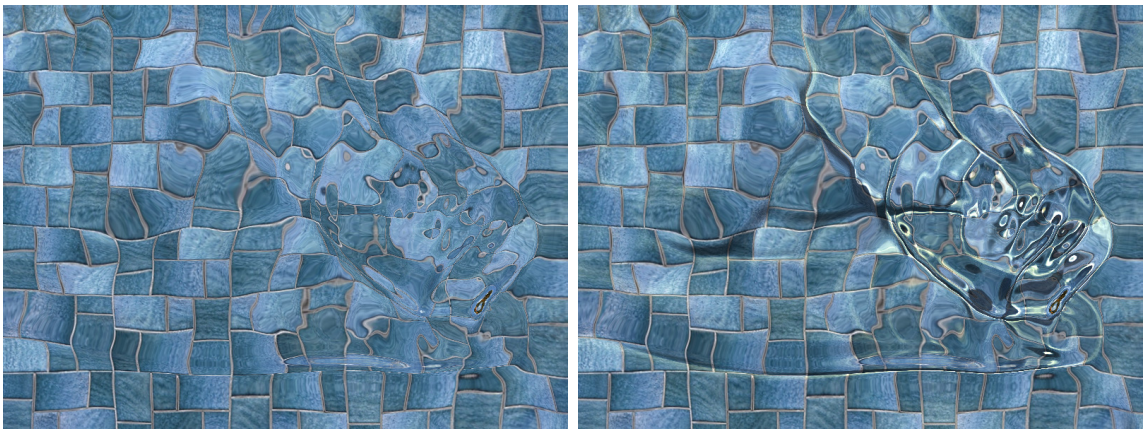


Figure 5.9. A height field water surface rendering with and without caustics. The image on the left only has reflections and refractions. The image on the right shows the same water surface with caustics computed using our fast caustics computation method for height field surfaces.

insufficient for a proper perception of the water surface.

In our system, we only consider refractive caustics that appear at the bottom of the pool due to the refractions of light through the water surface. In reality, caustics appear due to reflections of light from the water surface as well, but those caustics are not included in our system.

Since we only consider refractive caustics, they only appear underneath the water. Therefore, we use the caustics while rendering the refraction texture on objects underneath the surface. The caustics generation step produces a caustics map texture, which is used as an illumination map while rendering the objects for refractions. In our implementation, we ignore the caustics on the interacting objects and use the caustics map only for rendering the bottom of the pool.

Unfortunately, computing caustics can be computationally expensive. In our imple-

mentation of the wave particles method, we first used the real-time caustics computation technique proposed by Shah et al. [2007]. Afterwards, we developed a special caustics computation technique that is designed for caustics generated by refractions from height field surfaces onto a planar receiver. This special method is not as general as most previous real-time caustics computation techniques, but unlike previous methods it can reach extremely high frame rates making it an ideal choice for a wave particle simulation implementation. In the remainder of this section, we explain this fast caustics computation technique in detail.

5.2.3.1. Fast Real-time Caustics from Height Fields

For fast computation of caustics we follow light paths starting from the caustic-receiving surface instead of the light source. To simplify the computation, we assume that the caustic-receiving surface is a flat finite plane. The final result of our caustics computation is a caustics map that is mapped onto this plane. Figure 5.10 shows the caustics map of the frame in Figure 5.9 computed using this method. The grayscale value of each caustics map pixel represents the incoming light intensity of the corresponding pixel area on the caustic-receiving plane.

To produce this caustics map, we consider the refracted radiance from the height field water surface towards the caustic-receiving plane. For each pixel of the caustics map, we sum the refracted radiances towards the pixel from all points within a rectangular region R on the water surface. We refer to the center of this rectangular region as the *illumination center*.

Let $z = 0$ be the ground plane underneath the water surface and $\mathbf{P}_G$ be a point on

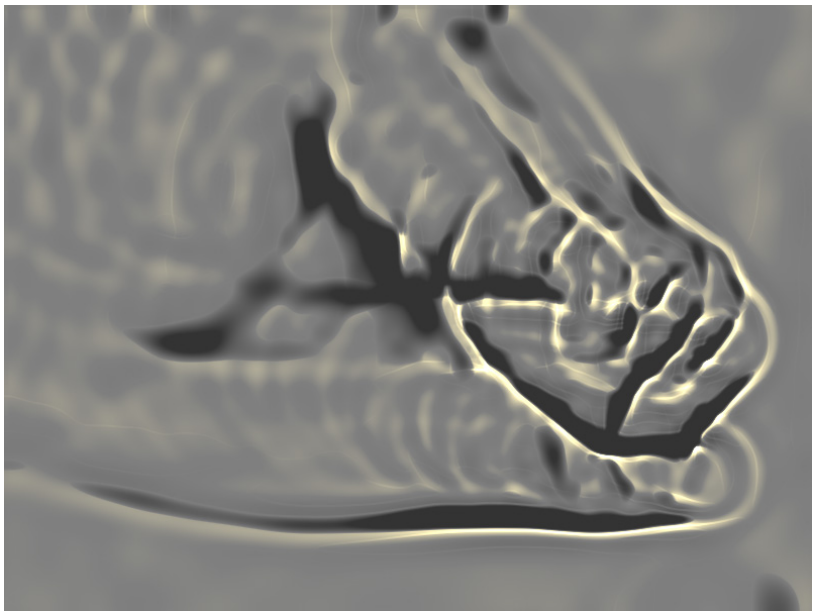


Figure 5.10. The final result of our caustics computation is this caustics map that includes bright and dark areas corresponding to caustics. This is the caustics map of the frame in Figure 5.9.

this plane (Figure 5.11). The *rest state*[1] of the height field is represented by the plane $z = h$, where h is the *rest depth* that corresponds to the distance between the ground plane and the water surface. The illumination center $\mathbf{P}_C$ that corresponds to the ground point $\mathbf{P}_G$ can be found by

$$\mathbf{P}_C = \mathbf{P}_G + h\mathbf{L}' \,, \tag{5.12}$$

where $\mathbf{L}'$ is the refracted light direction $\mathbf{L}$ from the rest surface with normal $\hat{z}$ in positive z-direction.

The size of the rectangular region R limits the part of the height field surface region

[1]In the *rest state*, the water surface is flat and all the values of the height field are equal to a constant value.

from which we can capture the illumination contribution to $\mathbf{P}_G$. In other words, our computation is accurate as long as the incoming radiance towards $\mathbf{P}_G$ through the water surface is confined within this rectangular region. The required size of R to capture 100% of the illumination is a complicated function of $\mathbf{L}$, h, and maximum surface normal deviation. When the height field water surface has high frequency deformations with large magnitudes, this size can be arbitrarily large and even cover the whole height field surface. However, in our simulations we mostly produce smooth water surfaces with low frequency deformations. Therefore, even a very small rectangular region can cover a significant portion of the incoming light, regardless of the magnitudes of the deformations.

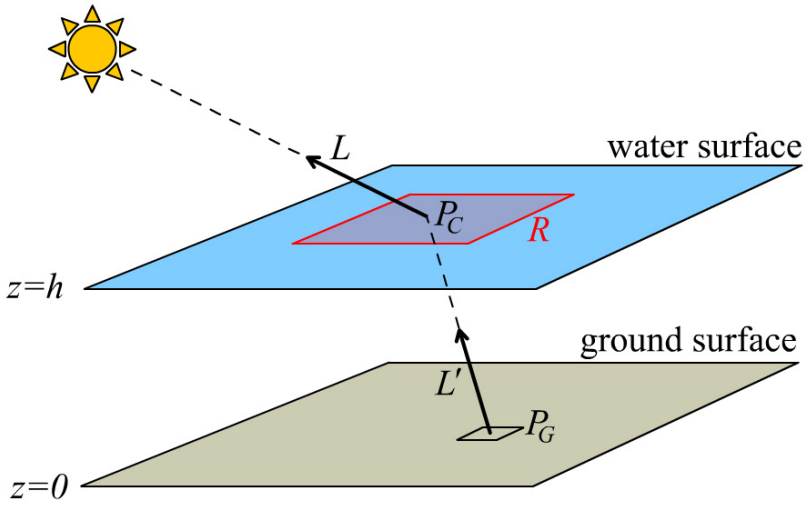


Figure 5.11. The illumination on point $\mathbf{P}_G$ comes from the refractions trough the rectangular area on the water surface.

Let A_G denote the area of a caustics map pixel on the caustic-receiving plane. Assuming that the rectangular region R is sufficiently large, the average light intensity

I_G over the area A_G can be written as

$$I_G = I_r(A_R, A_G)\frac{A_R}{A_G} \,, \tag{5.13}$$

where $I_r(A_R, A_G)$ is the reflected light intensity though the rectangular region R towards A_G, and A_R is the area of R. This equation can be written as an area integral over R,

$$I_G = \int_R \frac{I_r(A_w, A_G)\,\mathrm{d}A_w}{A_G} \,, \tag{5.14}$$

where A_w is an infinitesimal area on the water surface within R. To compute this integral we discretize this equation as

$$I_G = \sum_i I_r(A_i, A_G)\frac{A_i}{A_G} \,, \tag{5.15}$$

where A_i is the i^{th} sample area within R.

We approximate the reflected intensity I_r for the i^{th} sample by assuming that the surface normal is constant within the area A_i. Thus, $I_r(A_i, A_G) \approx \alpha I_r(A_i)$, where $I_r(A_i)$ is the average refracted light intensity through A_i and α is the fraction of the refracted area of A_i that intersects with A_G.

5.2.3.2. A Two-Pass Algorithm for Fast Caustics

To generate the caustics map efficiently we use a two-pass approach that minimizes the number of texture lookups (Figure 5.12). In the first pass we read the height field texture at multiple points along one direction (x-axis) storing the illumination contributions of these points in multiple textures. In the second pass we read the textures generated in the first pass along the perpendicular direction (y-axis) yielding

the final caustics map.

Figure 5.12. The two-pass algorithm: the first pass reads the height field texture and generates multiple outputs; the second pass reads the result of the first pass and produces the final caustics map.

In the first pass, for each pixel $\mathbf{P}_G^{i,j}$ of the caustics map, we find the corresponding illumination center on the height field and read N samples along the x-axis on either side of the illumination center. We place these samples on the height field such that the distance between two consecutive samples is equal to the width of a caustics map pixel, such that each sample represents an area on the height field surface that is equal to the area of a caustics map pixel. Note that the resolution or the orientation of the height field does not have to match the caustics map, since we base our sampling density and orientation only on the caustics map.

The aim of this first pass is not only to find the illumination contributions of these N samples on the pixel $\mathbf{P}_G^{i,j}$, but also on the neighboring M pixels of the caustics map along the y-axis, from $\mathbf{P}_G^{i,j-M/2}$ through $\mathbf{P}_G^{i,j+M/2}$. Therefore, the output of the first pass needs $M+1$ color channels, each of which correspond to a different pixel on the caustics map. On modern graphics hardware we can output up to 64 channels using multiple render targets (8 render targets with RGBA channels). However, in practice we found that as few as 8 channels can be sufficient since most of the illumination contribution comes from points close to the illumination center.

To compute the values of these $M+1$ channels, we calculate the refracted ray directions of each one of the N samples on the height field and find where these rays intersect the ground plane. Assuming the surface normal is constant within each sample area, a pixel sized square centered on each intersection point indicates the area illuminated by the refracted light through that sample. For each one of these square areas, we find the nearest two pixels between $\mathbf{P}_G^{i,j-M/2}$ and $\mathbf{P}_G^{i,j+M/2}$, then we compute the fraction of this square that overlaps with each one of these two pixels. The sum of all these fractions yields the total fraction of the refracted refracted intensity through these N samples on these $M+1$ pixels.

The pseudo code for the first pass is provided in Figure 5.13 for the case of $N=7$. Examining this this code one can easily see that a large portion of the computation in the first pass is repeated by multiple neighboring pixels. The computation of the refracted ray directions and their intersections with the ground plane are repeated multiple times. In our implementation we introduce an additional pass before the first pass to compute the intersection positions of the refracted light rays with the ground plane. The first pass reads the output of this additional pass, rather than the height field itself to reduce its computation load.

In the second pass, for each pixel $\mathbf{P}_G^{i,j}$, we simply sum the values from the previous pass that correspond to this pixel. These values are stored in different output channels of the first pass at the pixels $\mathbf{P}_G^{i,j-M/2}$ through $\mathbf{P}_G^{i,j+M/2}$. The resulting total values yield the fraction of incoming light at each pixel of the caustics map. The pseudo code for the second pass is provided in Figure 5.14 for the case of $N=7$.

The two-pass algorithm explained here enables efficient computation of caustics. Most

```
#define N      7
#define N_HALF 3

struct Pass1Out {
   float4 color0 : COLOR0;
   float4 color1 : COLOR1;
}

void CausticsPass1 ( out Pass1Out Out
                   ,  in float2 P_G : TEXCOORD0
                   ,  in float2 P_C : TEXCOORD1
                   , uniform sampler2D heightField
                   )
{
   // initialize output intensities
   float intensity[N];
   for ( int i=0; i<N; i++ ) intensity[N] = 0;
   // initialize caustic-receiving pixel positions
   float P_Gy[N];
   for ( int i=-N_HALF; i<=N_HALF; i++ ) P_Gy[i] = P_G.y + i;
   // for each sample on the height field
   for ( int i=0; i<N; i++ ) {
      // find the intersection with the ground plane
      float3 pN = P_C + ( i - N_HALF ) * xDirection;
      float2 intersection = GetIntersection( heightField, pN );
      // ax is the overlapping distance along x-direction
      float ax = max(0, 1 - abs(P_G.x - intersection.x));
      // for each caustic-receiving pixel position
      for ( int j=0; j<N; j++ ) {
         // ay is the overlapping distance along y-direction
         float ay = max(0, 1 - abs(P_Gy[j] - intersection.y));
         // increase the intensity by the overlapping area
         intensity[j] += ax*ay;
      }
   }
   // copy the output intensities to the color channels
   Out.color0 = float4( intensity[0], intensity[1], intensity[2], intensity[3] );
   Out.color1 = float3( intensity[4], intensity[5], intensity[6] );
}
```

Figure 5.13. The pixel shader pseudo code for the first pass.

of the computation is carried out in the first pass and the second pass merely combines the outputs of the first pass to produce the final caustics map. The computations of both of these passes take place in the fragment shader on the graphics hardware.

The efficiency of the algorithm comes from the fact that it does not require a high resolution water surface or a large number of point primitives to be rendered. The whole computation takes place in the fragment shader. It has a sequential texture

```
void CausticsPass2 ( out float4 color  : COLOR
                   , in  float2 P_G    : TEXCOORD0
                   , uniform sampler2D inColor0
                   , uniform sampler2D inColor1
                   )
{
   float val = 0;
   val += tex2D( inColor0, P_G + float2( 0, -3 ) ).r;
   val += tex2D( inColor0, P_G + float2( 0, -2 ) ).g;
   val += tex2D( inColor0, P_G + float2( 0, -1 ) ).b;
   val += tex2D( inColor0, P_G ).a;
   val += tex2D( inColor1, P_G + float2( 0, 1 ) ).r;
   val += tex2D( inColor1, P_G + float2( 0, 2 ) ).g;
   val += tex2D( inColor1, P_G + float2( 0, 3 ) ).b;
   color = val;
}
```

Figure 5.14. The pixel shader pseudo code for the second pass.

access pattern, which highly utilizes the texture cache on the graphics hardware.

This algorithm produces physically-based results as long as the caustics receiving surface is a flat finite plane. Therefore, using our method caustics on non-flat surfaces can only be approximated.

5.3. The Overall Water Simulation and Rendering System

In our implementation of the wave particle simulation and water rendering system, we make use of the parallel computation power of the GPU for computing various steps of the simulation system in addition to rendering. We also use the parallelization offered by the multi-core architecture of the CPU with multiple threads. In our implementation we used three CPU threads: main thread, rigid body thread, and wave particle thread. Figure 5.15 shows a diagram of our three CPU threads and their communication. The main thread is responsible for most of the computation and it includes all GPU related calls. The rigid body thread handles the rigid body

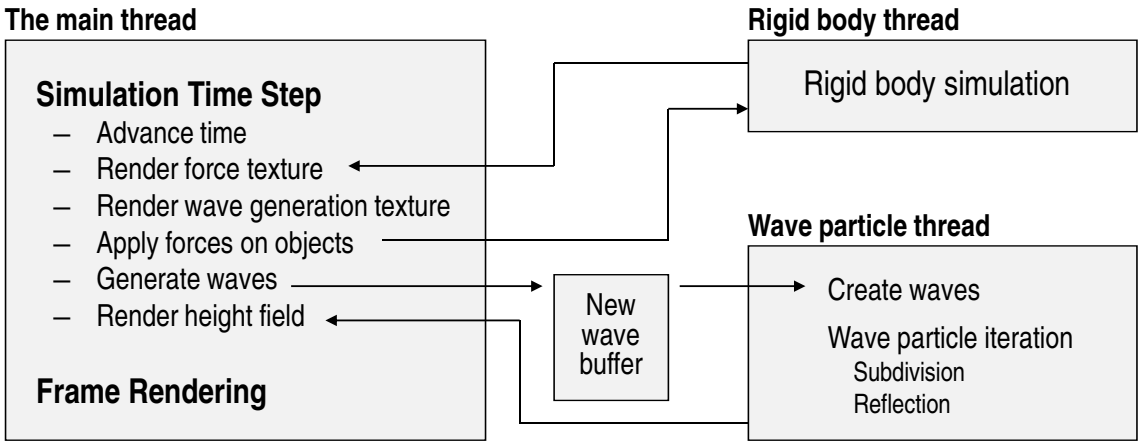


Figure 5.15. Overview of our implementation of the wave particle simulation.

simulation on the CPU. Finally, the wave particle thread performs all operations that change the wave particle system.

We place all GPU related calls into the main thread, so that different threads do not try to use the GPU at the same time. Note that depending on the architecture of the computer system, placing all GPU related tasks on one thread might increase or reduce the efficiency of the computation. Our early tests on the computer hardware on which we implemented our system indicated that having all GPU related tasks on a single thread could provide better performance. However, this may or may not be true for other hardware/software systems.

In our implementation, the main thread is responsible for a large portion of the simulation computation and all of the frame rendering computation. For each frame, the main thread first calls the simulation time step, and then renders the frame and displays it on the screen. The simulation time step begins by advancing the time value, which is read by all the threads for determining the current time. The next

step renders the textures for water to object coupling. For this step, the main thread requests the placements and velocities of all interacting objects from the rigid body thread. Afterwards, the main thread waits for the GPU to finish rendering the force texture and copies these textures to the main memory. The force textures include the drag and lift forces and their torques, but not the buoyancy forces. The buoyancy forces are computed at the next step together with the wave generation textures. Then, all these forces are passed to the rigid body simulation and the rigid body simulation for the time step begins. Afterwards, the main thread computes the new wave particles to be generated and places them in a new wave buffer. This buffer is read by the wave particle thread when creating new wave particles. Finally, the main thread renders the extended height field using the wave particles and continues to render the frame image to be displayed.

The rigid body thread is responsible for the rigid body simulation. The main thread triggers the rigid body simulation when the object forces are ready and waits for the rigid body simulation, when the rigid body thread can not complete its time step by the end of the simulation time step of the main thread.

The wave particle thread handles all operations acting on the wave particle system. It constantly monitors the new wave buffer that is written by the main thread, and places new wave particles that are in the buffer into the wave particle list. It also monitors the time value and performs all wave particle subdivision and reflection operations that should be completed until the end of the current time step. In our implementation, we do not have a mechanism to ensure that the wave particle thread completes its computation for the time step. This is because the wave particle thread does not actually perform any iteration as discussed in Section 5.1.1.1 and the current

positions of the wave particles can be computed at any time on the GPU while rendering the height field. All operations that are handled by the wave particle thread are very fast to compute and they can be completed in the background. Therefore, in our implementation we assume that the wave particle thread always has enough time to complete its operations for the current time step, and that the main thread does not have to wait for it.

CHAPTER VI

RESULTS

In this chapter we explore some of the results we have obtained with our implementation of the wave particles method. We first provide a qualitative analysis of our results and then we present the results of our performance analysis.

6.1. Qualitative Analysis

6.1.1. Analyzing the Wave Shape

To analyze the wave shape, we observe the waves generated directly from user interaction. Figure 6.1 shows the deformed water surface due to wave particles generated when the user drags the mouse pointer over the water surface. As the user clicks on the water surface and drags the mouse cursor, we create wave particles in the form of a circular ripple at the position of the mouse cursor. When the user drags the mouse cursor in one direction, the generated wave particles at different time steps superimpose and form a wave shape similar to that shown in Figure 6.1.

The superposition of wave particles includes the longitudinal component as well as the transverse component of waves, and the extended height field we generate from the wave particle system includes the effects of both vertical and horizontal deformations

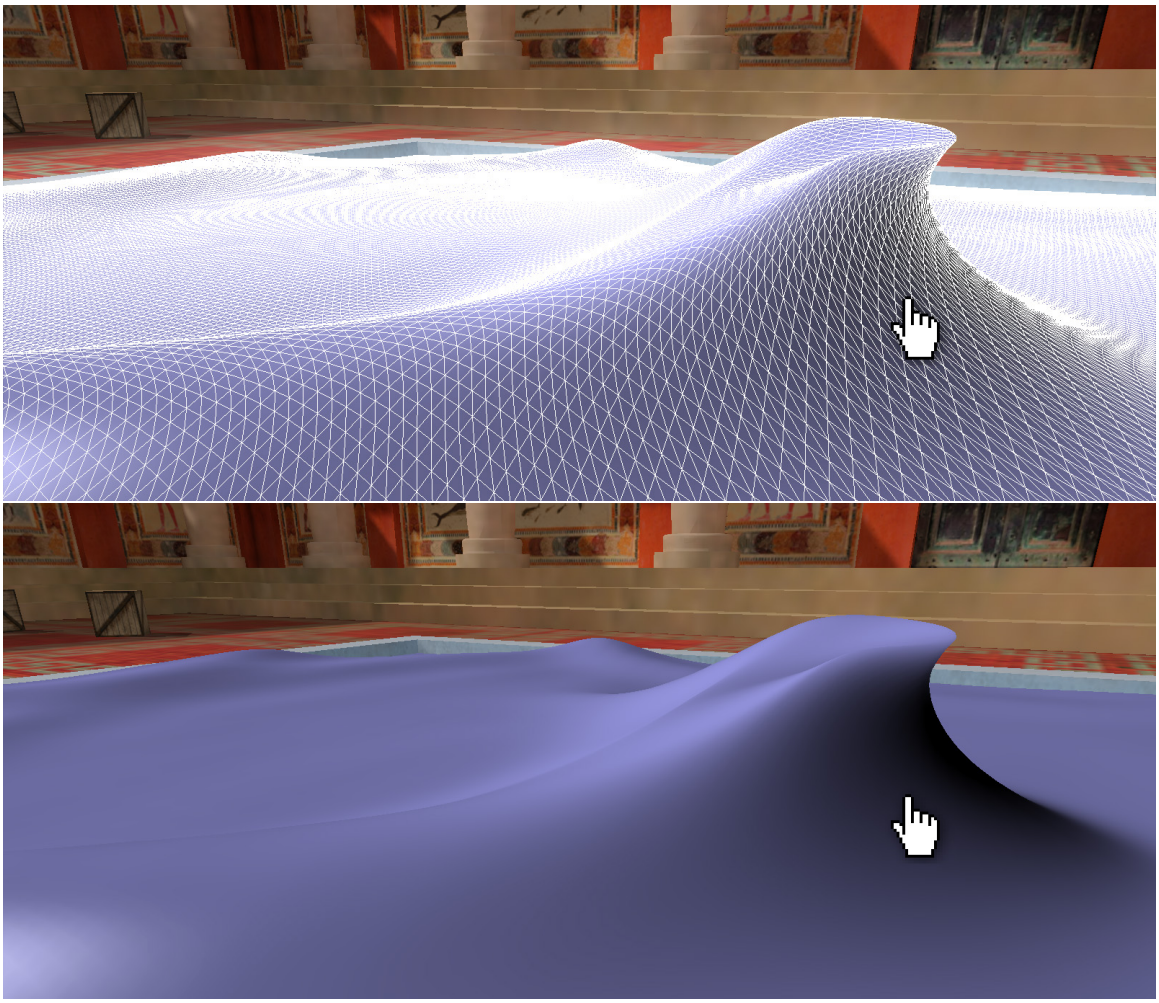


Figure 6.1. Waves generated by direct user interaction.

of the water surface. As the amplitude of the final wave shape increases, the wave crest becomes sharper as occurs with natural water waves. This demonstrates the visual importance of including the longitudinal wave component in the wave particle formulation.

Note that even though the shape of the particular wave front in Figure 6.1 makes it seem like the wave is about to break, it does not break, because breaking waves

are not supported by the wave particles method. This is because the breaking wave motion cannot be represented using the wave equation. We discuss breaking waves further in Section 7.2.2.

6.1.2. Analyzing Water to Object Coupling

For testing the accuracy of our water to object coupling, we compare our simulation results to real world experiments. The second row of Figure 6.2 shows frames from a video recording of a ball floating in a wave tank. The top row of this figure shows the scale of the wave on the right, and a composite of the second row frames on the left. As can be seen from these frames, and from the composite, the ball makes a circular motion as the waves pass. This is consistent with the formulations of Gerstner [1802].

The third row of Figure 6.2 shows corresponding frames taken from our simulation with wave particles, and the bottom row is organized like the top row. In this simulation, we have not modeled the waves as ambient waves that would match the shape of the waves in the experiment. Instead, we used a wave particle system for producing these waves. On one side of the simulated wave tank, we generate new wave particles at every time step that propagate towards the other side of the wave tank. For determining the amplitudes a_t of the generated wave particles at each time step t we use

$$a_t = a_{max} \sin\left(\frac{2\pi t}{T}\right) , \tag{6.1}$$

where a_{max} is the maximum amplitude of the generated wave particles and T is the period of the waves. We tuned the values of a_{max} and T such that the shapes of the generated waves in the simulation match the shapes of the waves in the recorded

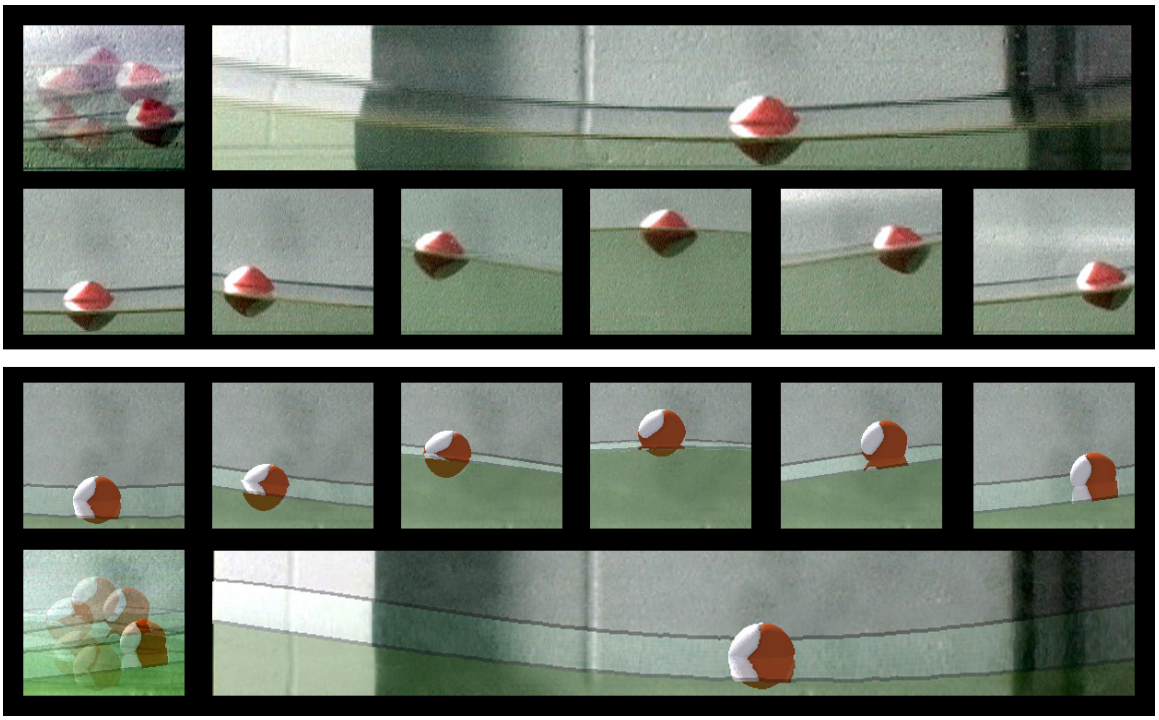


Figure 6.2. Circular motion due to waves moving from left to right. (top two rows) video capture, (bottom two rows) simulated using the wave particles method with water-object interaction.

experiment.

Comparing the results of the simulation to the results of the experiment, we see that the wave particle formulation of waves along with the water to object coupling computation can produce realistic results. The circular motion of the ball we observe in the experiment is properly simulated with our implementation of the wave particles method. Note that this motion cannot be simulated with simple height field formulations that do not include the horizontal motion of water.

Notice that the shape of the circle in the real experiment is horizontally elongated, which can be seen at the top left in Figure 6.2. Even though the motion looks like an

ellipse on this image, this is actually caused by the fact that each wave pushes the ball slightly to the right side in the wave tank. This is because the waves in the wave tank cause some minor mass transport at the water surface in the direction of the wave propagation. While ideal waves should cause no mass transport, waves generated in a wave tank do produce this effect.

For testing the behavior of our water to object coupling in more complicated scenarios, we prepared the offshore boat model seen in Figure 6.3. This boat model consists of two propellers and two rudders that are attached to the boat body with single-axis rotational joints. We apply motor torques for rotating the propellers and the rudders based on user interaction. These torques merely rotate the joints and do not, in themselves, exert any net force accelerating the boat.

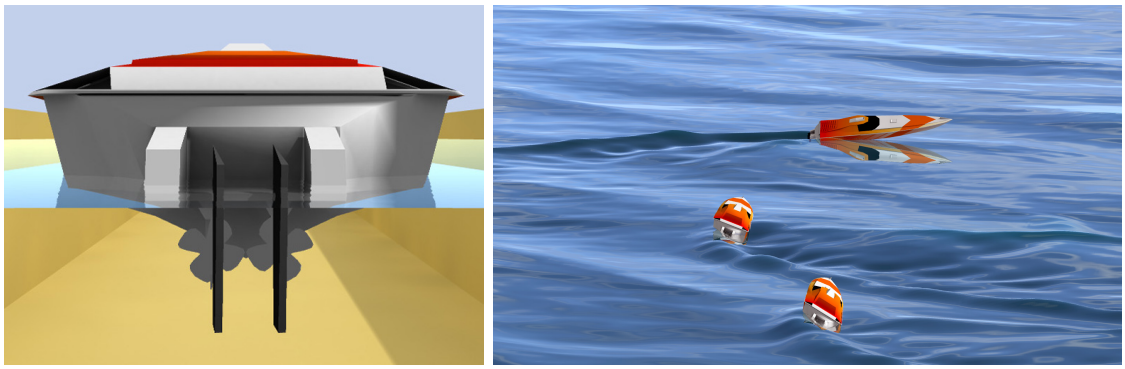


Figure 6.3. Propellers and rudders of offshore racing boat.

When the propellers are rotated, the lift forces on the blades of the propellers produce a net force applied to the propellers, which is translated to the boat body. As a result, the boat begins to move forward like a real boat. The lift force on the boat body pushes the bow of the offshore boat upward. Similarly, when the rudders are turned, the lift forces on the rudder blades as well as the lift force that is directly applied on

the boat body due to water to object coupling make the turn in the desired direction. Notice that in Figure 6.3, boats roll toward the direction that they are turning, just like boats do in reality. This motion is the direct result of the water to object coupling computation.

While the results we obtained with the offshore boat model show that our simulation system can properly handle complicated interactions of water with floating objects, the physical accuracy of the results we obtained is limited by the assumptions made in our method. Consider a real boat similar to the one in Figure 6.3. The rudder of such a boat is almost always placed right behind the propeller. In reality this provides better steering performance, since the motion of water induced by the propeller has its highest speed right below the propeller. However, in our simulation we do not compute the 3D water flow around the interacting object; therefore, placing the rudder at some other position than right behind the propeller does not make much of a difference in terms of the effectiveness of the rudder.

6.1.3. Analyzing Wave Generation

The wave generation method we describe in Section 4.1 is a significantly simplified model of the real world wave generation process. Although our method might not produce results with accuracy suitable for engineering or design purposes, our goal is simply to produce visually plausible waves based on water-object interaction.

Figure 6.4 shows 5 consecutive frames taken from a simulation of a box shaped object falling into the water. As can be seen in this figure, waves are generated around the object when it hits the water surface. The water to object interaction forces

(specifically the drag force) immediately slow down the object after it falls into the water. Therefore, the largest waves are generated when the object first hits the water surface. Afterwards, the motion of the object is significantly slower; thus, the object displaces a smaller volume of water, generating wave particles with significantly smaller amplitudes.

Figure 6.4. A box shaped object falling into water and generating waves.

In Figure 6.5 we show a more complicated example for wave generation. In this case, waves are generated around a boat due to its motion in water. This boat has a similar setup to the offshore boat shown in Figure 6.3. Unlike the offshore boat, it has a single propeller and a single rudder. Both the propeller and the rudder are used in wave generation along with the boat body; therefore, in terms of wave generation there is no difference between these three objects. Looking at the four frames in Figure 6.5 we see that the wave generation method can produce quite plausible results. Also notice that the wave crests get a sharp shape when the amplitudes of the waves are high due to the longitudinal component of waves and the wave crests get smoother as the wave amplitude decreases, forming natural looking wave shapes.

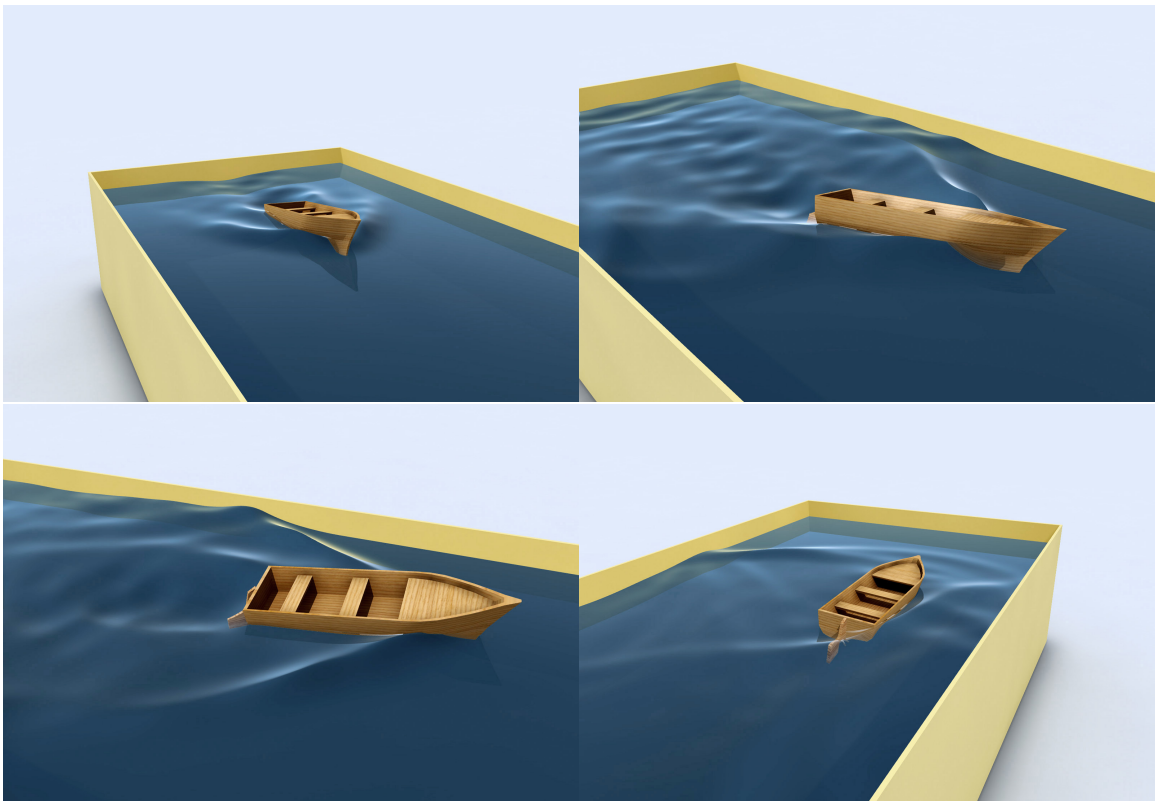


Figure 6.5. Boat in water tank, showing surface waves generated by the motion of the boat.

6.2. Performance Analysis

To demonstrate the speed and scalability of our approach, we simulated a number of scenes on a standard PC with a 2.13GHz Core2 Duo processor and GeForce 7900 GTX graphics card, recording performance data for each run. This data is shown in Table 6.1. Note that the sum of the times spent for each component of the simulation is greater than the total simulation time. This is because we used multi threading with two cores, so that some of these computations could be carried out in parallel.

Our *boat in tank* scene from Figure 6.5 has a height field resolution of 128×512, and

Table 6.1. Time results of our test scenes

	boat in tank	**offshore ocean**	**boat & boxes**	**boat armada**	**massive boxes**
Maximum Wave Particles	100,000	100,000	600,000	8,000,000	8,000,000
Height Field Resolution	128×512	256×256	256×512	256×512	256×512
Number of Objects	3	5	128	5,043	9,261
Number of Faces	176	130	6,176	295,856	444,528
Wave Particle Iteration	0.28 ms	0.12 ms	1.28 ms	57.54 ms	85.95 ms
Water to Object Coupling	0.29 ms	0.30 ms	1.74 ms	36.95 ms	129.23 ms
Rigid Body Simulation	0.04 ms	0.06 ms	0.49 ms	75.51 ms	1061.60 ms
Wave Particle Generation	1.15 ms	1.15 ms	2.59 ms	142.50 ms	115.15 ms
Height Field Generation	4.97 ms	6.27 ms	20.40 ms	132.93 ms	159.12 ms
Total Simulation Time	5.87 ms	7.83 ms	24.49 ms	206.78 ms	1073.38 ms
Frame Rendering Time	16.87 ms	8.43 ms	14.91 ms	126.04 ms	155.87 ms
Total Frames Per Second	44.1 fps	119 fps	27.3 fps	3.0 fps	0.8 fps
Simulation Frames Per Sec.	170 fps	128 fps	40.8 fps	4.84 fps	0.93 fps

includes a single propeller and rudder as user controllable elements, so the number of simulated objects is three and the total number of simulated object faces is 176. We allowed the simulation to use up to 100,000 active wave particles by keeping the amplitude threshold for killing wave particles very low. Nearly identical results can be achieved using far fewer (less than 10,000) wave particles. In Figure 6.5, the height field generation operation is by far the slowest part of the system, and takes about 5 ms. This is because the height field generation has a constant cost associated with the resolution of the generated height field. The rest of the simulation system works rather efficiently resulting in 5.87 ms total simulation time per frame, which corresponds to 170 fps.

Our *offshore ocean* scene shown in Figure 6.6 has a height field resolution of 256×256,

Figure 6.6. Offshore ocean scene with a single offshore boat that has two rudders and two propellers in the open ocean.

and included a single offshore boat with two rudders and two propellers. The number of simulated objects is five and the total number of simulated object faces is 130. We limited the number of wave particles to 100,000 and we could achieve a total simulation time of 7.83 ms, which corresponds to 128 fps. The computation times of this scene are very similar to the boat in tank scene. The only difference is that the height field generation takes a little more time, 6.27 ms, as compared to 4.97 ms in the boat in tank scene. This is mainly because generating the height field for the ocean is more costly than generating the height field for a pool.

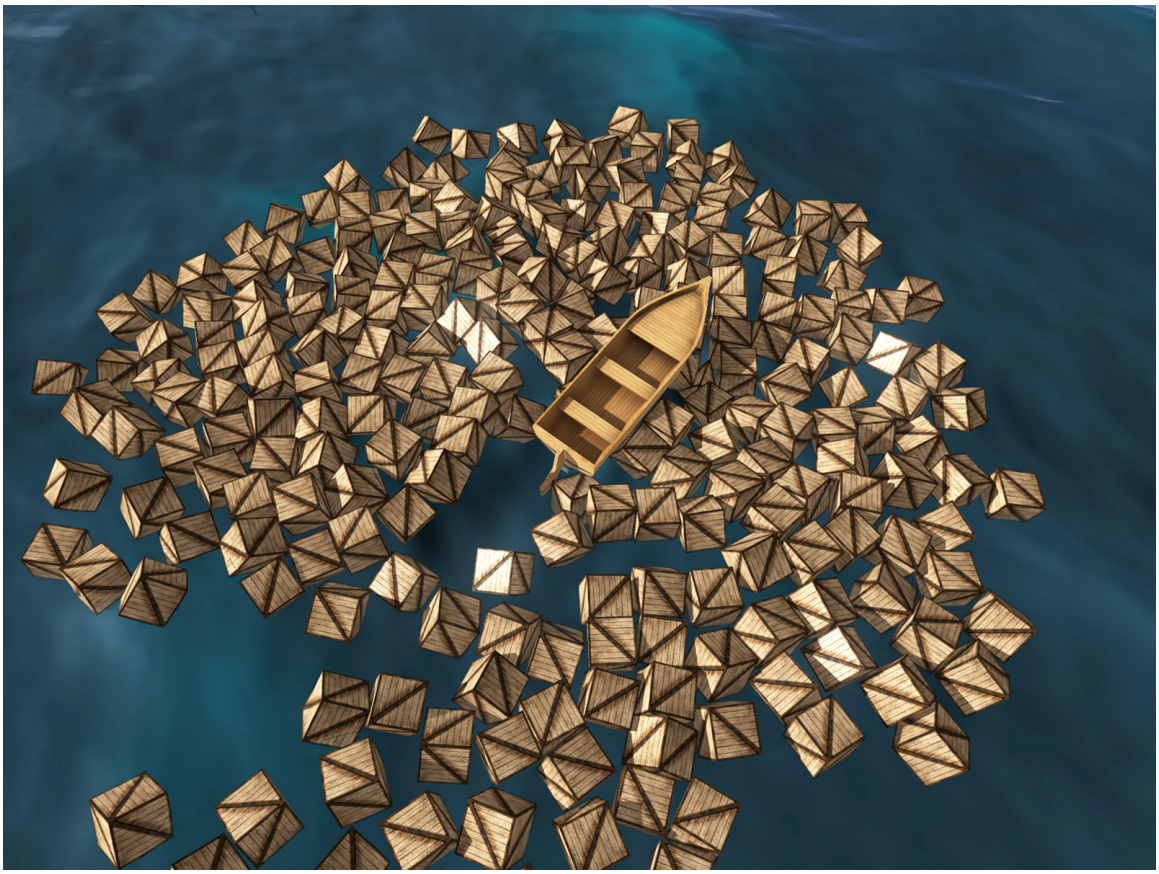


Figure 6.7. Boat and boxes scene with 125 boxes interacting with water.

Our third scene is the *boat & boxes* scene, shown in Figure 6.7, which used 600,000 active wave particles, a height field resolution of 256×512, and included an active boat driving through water populated with 125 boxes with a total of 6176 faces. Each box had eight triangular faces on each side, so that the lift and drag forces applied at the center of these faces could produce proper torque and rotate the boxes realistically. We achieved real-time performance with this scene as well. Each component of the simulation in this scene was significantly slower than the previous two scenes due to the increased number of interacting faces and objects. The number of objects not only increases the computation time of interaction computations, but also more

object interactions generate more wave particles, thereby increasing the computation of the wave particle system as well. The height field generation computation is the slowest part of the system in this scene as well, which takes about four times longer as compared to the previous two scenes. This is because in addition to the constant cost of height field generation, in this scene we have a lot more active wave particles; therefore, the point rendering step of the height field generation takes significantly longer. However, we still could achieve real-time performance with about 24.5 ms simulation time, which is 40.8 fps.

To stretch our method, we constructed two massive scenes shown in Figure 6.8 and 6.9, both using 8,000,000 active wave particles with height field resolutions of 256×512. The first, *boat armada*, had 1681 active boats with 295,856 faces, and runs at several frames per second. The second, *massive boxes*, had 9261 falling and floating boxes with 444,528 faces, and runs at nearly one frame per second. Comparing the computation times of different components in these two large scale simulations and in the other three scenes, we can see that the wave particle generation step took substantially longer and became very close to the height field generation, which was the slowest component in the other three scenes. The sheer number of interacting objects causes a large number of wave particles to be generated at every frame, which significantly increases the computation time of this component. In addition, the massive boxes scene spent quite a lot of time for rigid body simulation, since a large number of boxes were in collision throughout the simulation.

The computation times for each step of the simulation have different scene complexity dependencies:

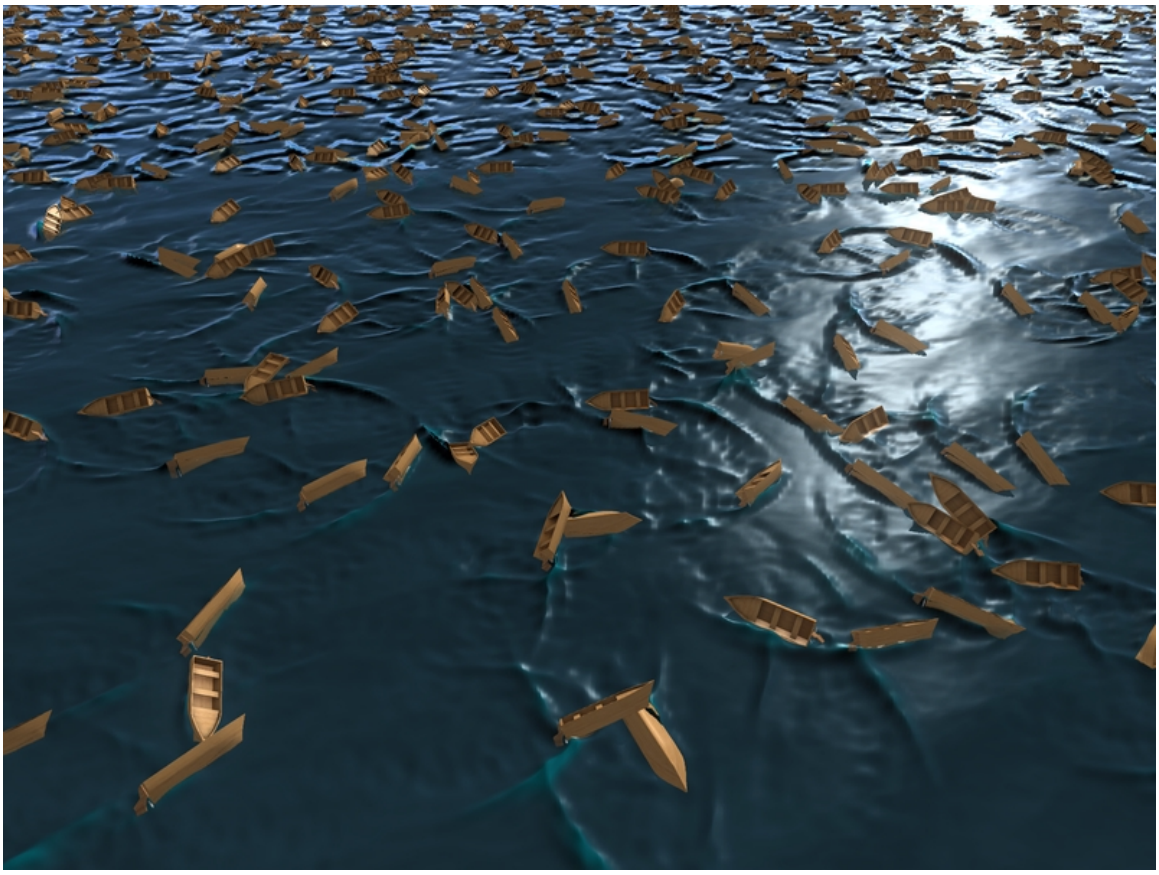


Figure 6.8. Boat armada scene with over 1600 boats, all interacting with water and generating wave particles.

- The cost of wave particle iteration is linearly dependent on the number of wave events within a time step, which is roughly proportional to the number of objects in the scene interacting with the water. Each object generates wave particles at every frame. When the object motion is slow, the generated wave particles die quickly and they do not add a major load onto wave particle iteration. Faster object motion generates wave particles with larger amplitudes that stay alive longer and each of them produces new wave particles as they subdivide.

- Object to water coupling computation depends on the total number of object

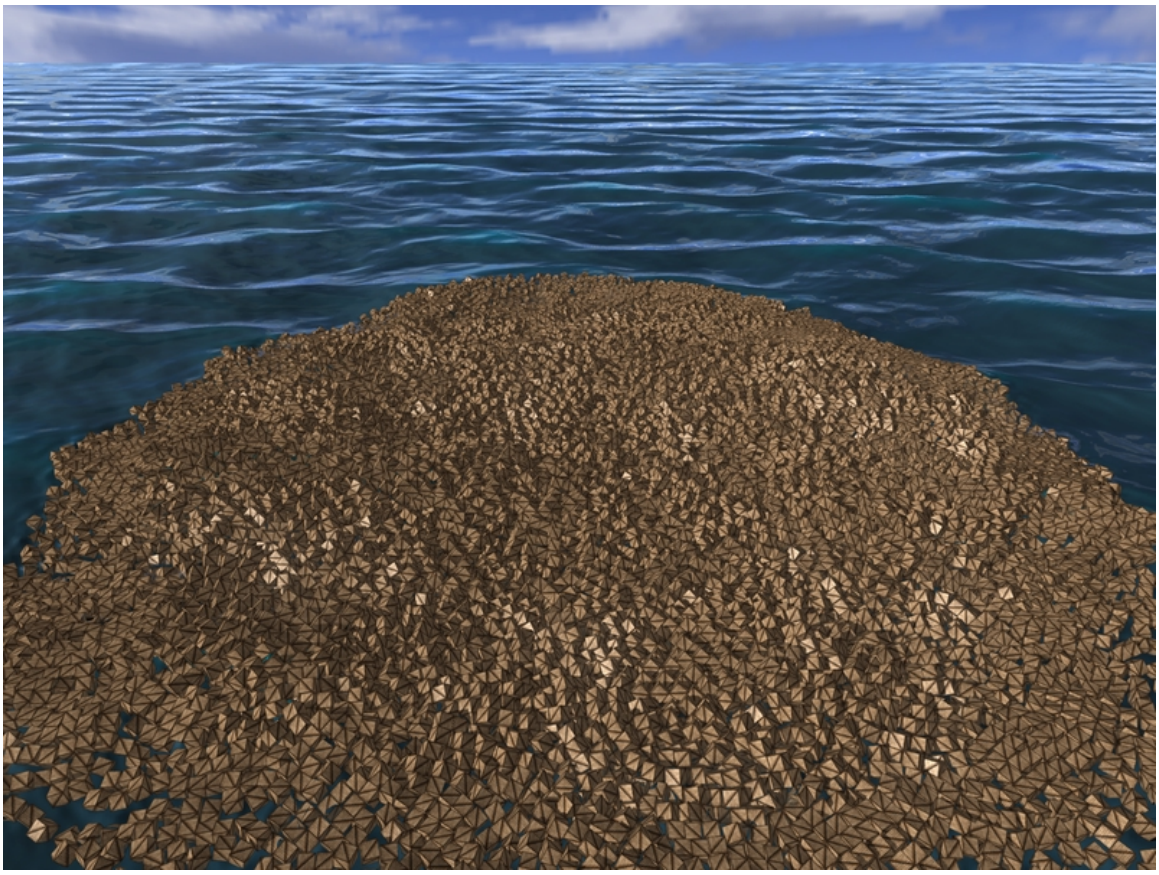


Figure 6.9. Massive boxes scene with over 9,000 boxes, all interacting with water and generating wave particles.

faces.

- The rigid body simulation time increases with the number of objects and the number of collisions between these objects.
- The wave particle generation computation also depends on the number of object faces.
- Height field generation has a cost due to the filtering operation that is related to the height field resolution and the wave particle radius. While this cost is fixed for a given simulation, there is also a cost based on the number of

active wave particles. As long as the number of active wave particles is small enough (less than 100,000 on our test hardware), the constant cost for the height field generation is the dominant factor. If many more particles are used, then rendering point primitives for these wave particles becomes the dominant factor.

Another interesting performance result we obtained is that in almost all tests the simulation time was comparable to the frame rendering time. One exception is the massive boxes scene, which spends quite a lot of time for rigid body simulation due to the larger number of collisions.

CHAPTER VII

DISCUSSION

The wave particles technique presented in this dissertation provides an approach to water simulation that differs significantly from all other current methods. Unlike water simulation techniques that discretize the water medium or the water body itself, wave particles only discretize the deformations on the water surface by representing them with a simple particle system. This approach to water simulation has various advantages along with some important limitations. In this chapter we discuss the advantages and limitations of the wave particles technique as well as possible future directions for improving or extending the wave particle simulation and its solution domain.

7.1. Advantages

The wave particles method is designed for real-time graphics applications, so it is very effective when it comes to the requirements of real-time graphics applications, such as computation speed, stability, scalability, and ability to handle various forms of user interaction. Furthermore, the wave particles method has implementation related advantages like its simplicity, its suitability for a GPU implementation, and the fact that it does not require any major precomputation. Moreover, unlike traditional fluid simulation techniques, the wave particles technique opens up new possibilities

for art-direction. In this section we discuss these advantages in some detail.

7.1.1. Computation Speed

For any method in computer graphics that is designed for real-time applications, the computation speed is perhaps the most important factor that determines the usability of the method in practice. An algorithm that fails to deliver high frame rates with reasonable quality has no place in a real-time application. On the other hand, there seems to be no upper bound for the desired computational efficiency of any algorithm that is intended to be used in a real-time application, because the extra computational resources that are not consumed by one algorithm, can be used by another algorithm that is supposed to run concurrently within the real-time application.

As we demonstrated in Chapter VI, the wave particles method provides high computational speed for moderately complex scenes. The implementation of wave particles presented in this dissertation produced very high frame rates on currently available computers with good gaming-level graphics hardware for all scenes we tested. Even for a highly complex scene with thousands of interacting objects, the wave particles method performs at near real-time frame rates.

Furthermore, it is possible to employ various techniques to achieve even higher performance than presented in Chapter VI. Some of these techniques are the following:

- One can place a relatively smaller upper bound on the number of active wave particles. This places an upper limit on the computation time devoted to the extended height field generation from wave particles. Note that extended height

field generation can easily be the bottle neck, when there are too many active wave particles.

- One can limit the number objects that can generate waves within a time step and force each object to generate waves only once in every few frames. This will not only reduce the wave generation computation time, but also the whole system will end up generating less wave particles, increasing the computation speed of the entire wave simulation system.

- Different level of detail approaches can be used both for wave generation and wave particle simulation itself. We discuss a few level of detail approaches in Section 7.3.2

7.1.2. Unconditional Stability

Another important advantage of the wave particles method is its stability. The wave propagation computation, including wave reflections and wave particle subdivisions, is unconditionally stable regardless of the time step size. One can literally take any long time step, and the result of the wave simulation can still be computed as accurately as it would be by taking multiple smaller time steps. This is because the wave particles provide an analytical solution to the wave equation, rather than a numerical solution that is employed by most traditional water simulation techniques. If the state of the wave particle system is known at any time, for any given time the state of the wave particle system can be directly and accurately computed.

Most traditional water simulation techniques use some sort of numerical integration method. While numerical integration is a very powerful tool in solving complicated

differential equations, it is also prone to numerical inaccuracies. Computational inaccuracies that arise at one step of the integration can quickly build up in subsequent steps. These inaccuracies are reduced by reducing the step size, which in turn increases the computation time. One popular fluid simulation technique in computer graphics is the stable fluids approach [Stam 1999], which provides an unconditionally stable integration method, but the numerical inaccuracies of this method results in excessive damping of the fluid motion.

The wave particles method, on the other hand, is not based on numerical integration. Wave particles provide an analytical approximation to the solution of the wave equation. Therefore, numerical inaccuracies do not build up and do not make the system unstable or introduce excessive damping. In fact, damping is optional with wave particles and one can implement a wave particle simulation system without any damping.

However, it is important to note that this unconditional stability does not extend to the water-object interaction computation, because at the heart of the water-object interaction we have a standard rigid body simulation system. First of all, the accuracy and the stability of rigid body simulation itself depends on the time step size.

The stability of the water to object coupling depends on the rigid body simulation, since it merely applies forces on interacting objects based on the object motion at the current time step. On the other hand, the magnitudes of the forces applied on the objects due to interaction with water do not depend on the time step size. Therefore, water to object coupling does not introduce instability based on the size of the time step. However, large time steps might significantly reduce the accuracy of water to

object coupling and may lead to undesired object motion.

The wave generation system itself is unconditionally stable. However, as the time step size increases the amplitudes of the wave particles generated within a time step increase as well, simply because as the time step gets larger each object face displaces a larger volume of water within a time step while making the same motion (i.e. moving with the same speed). Note that while the amplitudes of the waves generated within a time step increases with increasing time step size, the frequency of time steps (the number of time steps within a given time interval) decreases. Therefore, as long as the object makes the same motion, the total volume of generated wave particles stays the same.

However, if the time step size is too large, the amplitudes of the waves generated within a time step can be unrealistically large. Furthermore, at the next time step the object interacts with the deformed water surface that includes the deformation of these new wave particles generated in the previous time step. Therefore, if the amplitudes of the waves generated at a time step is too large, the change in the shape of the water surface between consecutive time steps would be large as well, which may introduce instability to the rigid body simulation.

7.1.3. Scalability

The wave particles method is designed for simulating water waves on large bodies of water. One important advantage of the wave particles approach is that it decouples the wave simulation from the water body and the water surface. As a result, extremely large volumes of water, such as an infinite ocean, can be simulated.

Traditional water simulation techniques use some sort of volumetric discretization of the water body, either by discretizing the space of the computation domain or by discretizing the water body itself by representing it with volumetric particles. In both of these cases, the volume of water that can be simulated is bounded, and increasing this volume either increases the memory use and the computation time or decreases the resolution of the simulation. On the other hand, the discretization introduced by the wave particles method discretizes the deformations on the water surface, not the water surface or volume itself. Therefore, the limits on the memory use and the computation time of the wave particles simulation bounds the number and/or the resolution of the deformations of the water surface. As a result, the wave particles simulation is not confined in a predefined space and it can be used for simulating an infinite ocean.

The wave particle simulation decouples the wave simulation from the extended height field that is used for displaying the water surface. The extended height field is needed where the water surface is visible. The wave particles themselves are simulated completely independently. However, parts of the water-object interaction computation described in Chapter V use the height field for achieving high computational efficiency.

7.1.4. User Interaction

As we discuss in the introduction of this dissertation, user interaction is an inherent part of almost all real-time graphics applications. The wave particles method as described in this dissertation permits two forms of user interaction: direct and indirect.

Direct user interaction refers to the direct manipulation of the wave particle system by the user. One simple form of direct user interaction is generating ripples on the water surface as the user drags the mouse cursor over the water surface. This form of interaction is supported by many ad-hoc wave simulation techniques as well. With wave particles, direct user interaction can also be in the form of removing waves or creating waves that are not in the shape of ripples.

Indirect user interaction provides a more interesting and often more useful user interaction. Water-object interaction described extensively in Chapter IV can be considered a form of indirect user interaction. As the user directly or indirectly moves various objects in a scene, the water-object interaction computation can generate waves on the water surface as well as produce physical responses on the objects interacting with water.

Notice that water object interaction described in Chapter IV is not tailored for certain types of objects or for certain types of motion, but it is designed to be as general as possible. In Chapter VI we show how this water-object interaction computation can be used, without modification, for simulating boats with propellers and rudders, thus demonstrating the generality of the water-object interaction computation.

7.1.5. Parallelization

In Chapter V we present how thread level parallelization can be used, together with the parallelization provided by implementing parts of the system on the GPU. The fact that each wave particle is handled independently from other wave particles makes the wave particle simulation very suitable for parallelization. Furthermore, different

parts of the simulation system can be executed concurrently on parallel threads.

The implementation of the wave particles method described in Chapter V heavily uses the GPU. However, this implementation does not use the GPU as a general purpose parallel computation unit, but only for graphics related tasks. For example, the extended height field generation from wave particles is simply a particle rendering task. For this task, we render point primitives and then apply image filtering to get the final result. Similarly, the computation of the buoyancy force as well as the silhouette pyramid method for wave generation use simple rendering and filtering operations. In that sense, the implementation of the wave particles method described in Chapter V uses the GPU largely for tasks that it was designed to be used for. For this reason, the wave particles method can be implemented on almost all GPUs, including those from earlier generations.

7.1.6. No Precomputation

Many algorithms designed for real-time graphics use extensive precomputation. Some of these algorithms require a very long computation time and they almost always require a large amount of memory at run-time to store the output of the precomputation. As the memory requirements for storing the precomputed data get larger, these methods become less desirable in practice as they reduce the available memory for other parts of the application that must work concurrently.

The implementation of the wave particles method described in Chapter V does not require any significant precomputation. Almost all components of the simulation are computed at run time.

7.1.7. Art-Directability

One important disadvantage of most fluid simulations is that they present various difficulties when it comes to art-direction. Even though there is a considerable amount of research in computer graphics about controlling fluids, modifying the result of a particular simulation to meet the requirements of art direction is still a challenging task for most simulation systems.

The main difficulty for traditional water simulations come from the fact that the connection between the data used for simulation and the shape of the final water surface is highly non-trivial. In the case of velocity fields, it is difficult to predict how the field should be modified to get the minor changes requested as a part of the art direction, while keeping the rest of the solution intact. Similar problems exist when using volumetric particles for simulation water, since any individual particle provides little information about the shape of the water surface around it.

Wave particles, however, directly define the deviations on the water surface. Therefore, it is almost trivial to predict what the outcome would be if a wave particle is modified. Furthermore, using wave superposition, one can modify the output of the wave particle simulation by generating new waves only and leaving all existing wave particles intact. This opens up new possibilities for art-directability. One possible related future direction that comes out of this property of wave particles is a wave choreography system that we discuss in Section 7.3.3.

7.2. Limitations

The wave particles method also has various limitations, which constrain the solution domain of the wave particles method as well as the physical accuracy of the simulation. As a matter of fact, most of the performance benefits discussed above are tightly connected to a limitation in the capabilities of the wave particles method. In this section we discuss these limitations and possible future directions for resolving these limitations.

7.2.1. Wave Simulation Only

One obvious limitation of the wave particles method is that it can only simulate wave behavior. Rather than formulating an ultimate technique to handle all fluid phenomena, the wave particles method concentrates on efficiently simulating surface waves on a large body of water. Therefore, the wave particles method cannot handle any other water behavior. Other characteristic behavior, such as splashes, bubbles, and foam are not an inherent part of the wave particles simulation. The 3D water motion, which might arise as a result of water-object interaction, is not simulated or even represented in the wave particle formulation. Furthermore, our formulation of the wave particles method does not include fluid flow, such as water currents.

On the other hand, surface waves form the majority of the water behavior for large bodies of water. Splashes, bubbles, and foam can be added on top of a wave particle simulation as a separate simulation and rendering layer as discussed in Section 7.3.1. The 3D water motion due to object interaction is not directly simulated, but its effect

on the water surface is handled by generating waves on the surface directly from the object motion.

7.2.2. Breaking Waves

In our formulation, when the amplitude of a wave is greater than half of its length, the final wave shape intersects with itself. A similar behavior is also observed by Gerstner [1802]. Bascom [1980] reports that waves become unstable and break when the amplitude is greater than one-seventh of the wave length.

Since the formulation of wave particles is based on the wave equation, breaking waves are not included in the solution of the wave particles method. Waves break when the wave motion cannot be sustained by the physical limitations of the water medium. Therefore, the breaking wave motion is not really a wave motion and it is not included in the wave equation. For this reason breaking waves cannot be simulated by a wave simulation that is based strictly on the wave equation.

On the other hand, the wave particle system might be useful in identifying and emulating the breaking wave motion. Unlike most other water simulation techniques, the wave particles method directly models the wave motion with particles. This might be useful in efficiently detecting when and where the waves should break and what the breaking wave motion should be. We believe that this would be a very useful future extension of the wave particles method.

When a wave particle with a large amplitude value is generated, it is safe to assume that the wave that this wave particle represents would not be stable in reality. There-

fore, this could be used as an indication for secondary effects like splash and foam, as well as breaking waves. On the other hand, the breaking wave condition may also be reached as a result of wave superposition. This is somewhat more difficult to detect, because a single wave particle alone may not indicate that the wave should begin breaking. Therefore, it might be necessary to analyze the extended height field generated by the wave particles, rather than testing each wave particle individually against a wave breaking condition.

7.2.3. Diffraction

Wave particles form an analytical solution to the wave equation, which is unconditionally stabile as discussed in Section 7.1.3. However, this solution has certain important limitations when it comes to some complicated wave behavior. Diffraction is one such wave behavior that the wave particles formulation presented in this dissertation cannot handle.

The diffraction behavior of waves is observed when part of a wavefront hits a stationary object and bounces off of it, while the rest of the wavefront continues to propagate without hitting the object. As a result the wavefront gets separated into multiple pieces and the parts that do not hit the object bend around the object. This behavior of waves is known as diffraction.

The diffraction behavior of waves is consistent with the wave equation. Remember that the wave equation in Equation 3.1 relates the spatial derivative to the time derivative. When a part of a wavefront gets reflected, this reflection effectively changes the shape of the water surface around the object as well as the spatial derivative.

Based on the wave equation, this changes the time derivative and the propagation direction of the wavefront near the object changes accordingly. In fact, this creates a chain reaction such that a larger portion of the non-reflected part of the wavefront gets bent as the wavefront propagates.

For representing a wavefront with wave particles, we place multiple identical wave particles side by side an equal distance apart. As we explain in Section 3.6, while discussing the diffraction limitation of wave particles, it is important that each wave particle on a wavefront has two identical neighbors; otherwise, the end result does not form a valid solution to the wave equation. Let us assume that we an ideal wavefront that satisfies the condition that each wave particle has two identical neighbors. This condition gets broken if the wavefront is somehow separated into two parts (when half of the wavefront is reflected). In this case, the wave particles on either side of the split point lose one of their neighbors. According to the wave equation, such wave particles must not continue their propagation without modification, but must spread towards the missing neighbor to make up for its absence.

In fact, for reaching a correct solution that includes diffraction, a wave particle must change its propagation properties not only when one of its neighbors is missing, but also when one of its neighbors is not identical to the wave particle. Otherwise, the resulting solution would be incorrect around those wave particles. When a wavefront gets split into two parts (via wave reflection or any other event), the wave particles right next to the split point must be modified due to the absence of a neighbor. This modification of the wave particles will cause their existing neighbors to change as well, since those neighbors will no longer have two identical neighbors. This change would propagate to all wave particles on the wavefront.

To be able to incorporate the diffraction behavior into the wave particles method, one needs to

- determine when a wavefront is split, and
- modify wave particles near the split point accordingly.

When detecting wavefront splits, it is important to be efficient, since we consider the wave particles method as a water simulation solution for real-time graphics. Detecting wavefront splits requires that each wave particle be aware of its two neighbors, so that when one of them gets reflected or modified the wave particle can react to it accordingly. This enforces a computational dependence between neighboring wave particles, which is likely to reduce the performance of the wave particle system. Note that our implementation of the wave particles method described in Chapter V makes use of the property that each wave particle is independent for highly efficient wave particle simulation. When this property no longer exists, parts of the wave particle propagation computation need to be replaced by potentially less efficient algorithms that do not rely on the independence of wave particles.

Another issue to be careful of is that when a neighbor of a wave particle reflects off of a boundary, this does not mean that the wavefront is split and that the wave particle must be modified immediately. Consider a wavefront hitting a linear boundary with some angle. In this case, wave particles get reflected one by one and no modification is needed to get a correct solution except for the reflection itself. However, if the neighbor of a wave particle gets reflected, but the wave particle does not get reflected soon afterwards, then the wave particle (and its neighbor) needs to be modified to properly handle diffraction. Therefore, for deciding whether or not a wave particle

should be modified due to diffraction (detecting whether a wavefront split occurred), it is probably more useful to track the distance between two neighbors on a wavefront than to analyze all wave particle reflections.

Modifying the wave particles near the split point to handle a detected diffraction event has its own difficulties. When a wave particle gets modified due to diffraction, we can no longer expect that the neighboring wave particle on the wavefront has two identical neighbors. Therefore, the neighboring wave particle needs to be modified as well, either immediately or later on as the wavefront propagates. As we mentioned above, this creates a chain reaction and multiple (perhaps eventually all) wave particles on the wavefront should be modified as their neighbors are modified.

Furthermore, it is not trivial to modify the dispersion angle of wave particles or their center positions due to a diffraction event. Handling wave particle modifications due to diffraction seems simple enough when one considers the classical diffraction example of a wavefront going through a narrow slit. However, in a more general case when a part of a wavefront gets reflected off of a curved boundary and the rest of the wavefront propagates without hitting the boundary, it is not as simple to find out how the wave particles of the wavefront should be modified.

A solution to most of these problems that arise with diffraction would constitute a complete reformulation of the wave particle system. Instead of thinking of wave particles as independent pieces of a wavefront, all wave particles of a wavefront could be stored in the same structure and modified together when the wavefront needs more resolution (i.e. more wave particles). While this approach might provide a solution to the diffraction related problems explained above, it would make the wave particle

system more complicated and its computation less efficient than what is presented in this dissertation.

7.2.4. Dispersion

Frequency dispersion of water waves refers to the dependency of the wave velocity on the wavelength of the wave. Water waves with larger wavelength propagate with larger speed than waves with smaller wavelength. Let h denote the depth of the water and λ be the wavelength. The wave speed v can be written as

$$v = \sqrt{\frac{g\lambda}{2\pi} \tanh\left(2\pi\frac{h}{\lambda}\right)}\,, \tag{7.1}$$

where g is the gravitational acceleration [Dean and Dalrymple 1984]. In this equation, the relationship between the water depth h and wavelength λ is important in determining the wavelength dependence of the wave speed. Figure 7.1 shows the $y = \tanh(x)$ curve, which we use for simplifying the equation above. In shallow waters, where $d \ll \lambda$, $\tanh(x)$ approaches x and the wave speed can be written as

$$v_{shallow} \approx \sqrt{gd}\,. \tag{7.2}$$

Therefore, in shallow waters the wave speed can be considered independent from the wavelength and it only depends on the depth of the water, so dispersion can be safely ignored in shallow waters. On the other hand, in deep waters, where $d \gg \lambda$, $\tanh(x)$ approaches 1 and the wave speed becomes

$$v_{deep} \approx \sqrt{\frac{g\lambda}{2\pi}}\,. \tag{7.3}$$

This simplified equation shows that in deep waters the wave speed depends on the wavelength and the effect of water depth can be ignored.

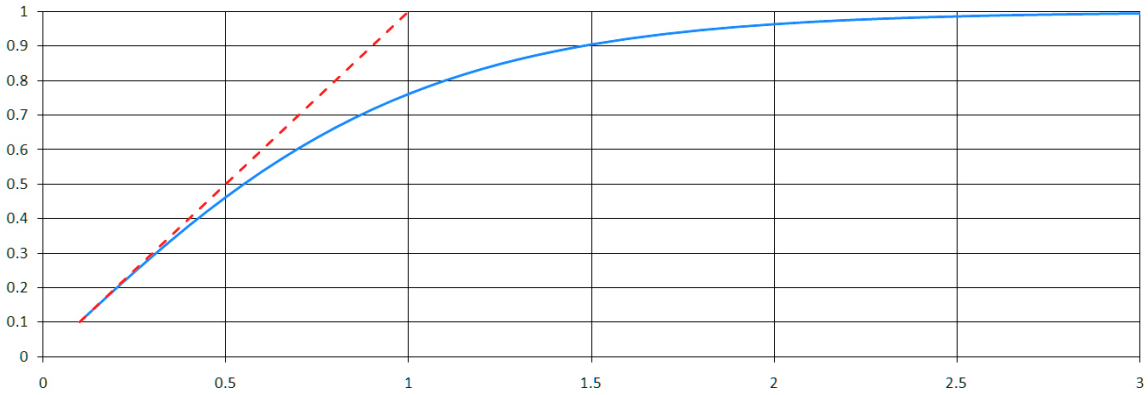


Figure 7.1. The behavior of $y = \tanh(x)$ (solid curve) as compared to $y = x$ (dashed line). Note that $\tanh(x)$ quickly approaches 1 as $x > 2$, and approaches to $y = x$ as $x < 0.5$.

Throughout this dissertation we assumed that the wave speed v is constant for all wave particles and that all all waves propagate with the same speed. However, this is not really a restriction on the wave particle formulation and each wave particle can be assigned a different propagation speed without having to modify the wave particle formulation or even the implementation. For example, we could simulate wave particles with different wave speeds, by simply applying a scale factor to a wave particle's direction vector.

However, deciding on what the speed of a wave particle should be is not as trivial. As we discussed in Section 4.1.4.3 as a part of wave size heuristics, the size of a generated wave particle is not necessarily the same as the wavelength of the wave that the wave particle is used for representing. Our wave generation system, discussed in Chapter IV, does not try to compute the wavelength of generated waves, so this

information does not exist and we cannot tell what the speed of a generated wave particle should be.

Therefore, the fact that wave dispersion is not included in the wave particle solution is mainly a limitation of the wave generation system rather than the wave particle formulation or implementation. A more sophisticated wave generation model that could compute the wavelength of the generated wave particles could easily include dispersion in the wave particles method.

7.2.5. Varying Depth and Wave Refraction

As we mentioned while discussing the dispersion behavior above, the depth of the water does not affect the wave speed in deep water, but it significantly affects the wave speed in shallow water. By making the wave speed v constant for all wave particles, in shallow water we inherently assume that water depth is constant everywhere. However, if the water depth is not constant for a shallow water simulation, the wave speed should be modified accordingly, as the waves propagate.

As simple as this varying wave speed problem sounds at first glance, it poses various problems in implementation as well as the theory of wave particles. First of all, our efficient implementation of wave particles relies on the fact that when the wave speed is constant we can easily calculate the current position of a wave particle using a known previous position and its propagation direction. However, if the wave speed is not constant everywhere in the water medium, we cannot use this simplification and the position of each wave particle must be computed with numerical integration.

Furthermore, the propagation direction is affected by changing water depth as well. Consider a pool with two sections separated by a straight line. Assume that both of these sections have constant water depth, but the depths of these two sections are different from each other. We know that when a wavefront that is generated on one section moves into the other section the propagation speed of the wavefront must be modified according to the depth change. If the wavefront generated on one section arrives at the line that separates the two sections at an angle, then not only must the speed of the wavefront change, but also its direction needs to be modified, due to refraction.

Modifying the wave direction due to refraction is quite simple, but computing the post-refraction dispersion angle is not as trivial. Moreover, the change in wave propagation speed, combined with refraction, changes the distribution of wave particles on a wavefront. Since, each wave particle requires two identical wave particles on either side of it to form a valid solution to the wave equation, if the modifications to a wave particle are different than its neighbors, the resulting simulation will no longer be a valid solution to the wave equation. For handling varying wave depths properly, the wave particle simulation should be able to handle the diffraction effect defined in Section 3.6. Therefore, simulating varying water depth and wave refraction becomes an easier task, if one can find a way to efficiently simulate diffraction using wave particles.

7.2.6. Physical Wave Generation

Having a good wave generation system is very important for the wave particles method. In the end, the wave particles method merely simulates the propagation of the generated waves. Therefore, if the wave generation system cannot produce plausible waves, the wave particles method cannot produce a plausible wave simulation.

As we discussed in Chapter IV, the wave generation method presented in this dissertation has various limitations. Even though the wave generation system in Chapter IV can produce plausible waves, it falls short when the results are compared to reality. We discuss the limitations of the wave generation system extensively in Section 4.1.6.

Unfortunately, building a physically accurate wave generation system is very complicated. First of all, as we discuss in Chapter IV, if physical accuracy is needed, waves should not be generated directly from water-object interaction. In general, water-object interaction induces a 3D fluid flow around the object, which eventually evolves into a wave motion. However, at the time that the water motion is induced by the object, the water motion does not correspond to a wave motion. If physical accuracy is important, this 3D motion of water around the object should be properly simulated using a 3D fluid simulation system.

Once the 3D motion of water induced by the object is properly simulated, the difficulty is to extract a wave motion from this 3D simulation. Unfortunately, to our knowledge there is no technique in computer graphics that can extract a wave motion from the solution of a 3D fluid simulation system. We discuss this in more detail as a part of a

future extension of the wave particles method that would integrate it with a 3D fluid simulation system in Section 7.3.4.

Furthermore, even if one can find a solution for integrating the wave particles method with a full 3D fluid simulation system to generate physically realistic waves, it is important that this physical wave generation system can be implemented efficiently and can be executed as a part of a real-time application. Otherwise, while trying to introduce physical wave generation into the wave particles method, we end up losing one of the most important properties of the wave particles method, its computational efficiency. This not only places additional computational restrictions on the 3D fluid simulation, but also means that the extraction of the wave information from the full 3D fluid simulation result should be implemented in a very efficient way.

Therefore, for building a real-time wave simulation system, assuming that waves are generated immediately from the object motion seems like the only acceptable practical solution. That said, the wave generation system we describe in Chapter IV has various other inaccuracies that can be improved:

- The current wave generation method is based on the volume conservation principle. This provides little information about how the volume of waves should be distributed around the object. The wave placement heuristics we use can be improved.

- The wave direction heuristics provide a simple methodology for determining the directions of the generated waves, which can be improved.

- The current wave generation system generates wave particles with the same size. This leads to limitations such as incorrect dispersion as discussed in Sec-

tion 7.2.4. A more sophisticated way of determining wave sizes based on the object motion might significantly improve the results of the wave generation system.

We used various real world experiments for developing the principles of the current wave generation system described in Chapter IV. More experimental analysis, including experiments with 3D fluid solvers, might be useful for improving the wave generation system.

7.2.7. Physical Forces on Objects

For handling water to object coupling, we apply buoyancy, drag, and lift forces on the interacting object. While the buoyancy force can be computed rather accurately and efficiently, the accuracy of drag and lift forces on an arbitrarily shaped object is questionable. This is mainly because we compute drag and lift forces completely independently on each face of the interacting object, while in reality different faces of the object might significantly affect the drag and lift forces on each other.

In fact, even a minor change in the shape of a certain face on the object can significantly affect the drag and lift forces on many faces along the surface of the object. Consider the wings of an airplane. In this case, the fluid medium is the air. The overall drag and lift forces acting on a wing of an airplane significantly depend on the shape of the wing.

On the other hand, for finding the total drag and lift force acting on an airplane, one might compute these forces independently for the wings and the body of the

airplane. While this may not be as accurate as computing the drag and lift forces considering the airplane as a whole, it might be significantly easier to compute these forces separately for different parts of the plane and the sum of the forces can be an accurate enough estimation to the total force acting on the airplane.

In our computation of drag and lift forces, we take this approach to its extreme by computing forces separately on each face. Therefore, we cannot expect that the drag and lift forces we compute are physically accurate. As we explain in Section 4.2.2, certain object shapes that have cavities or that are highly concave might be especially prone to this inaccuracy.

Computing physically accurate drag and lift forces is highly complicated. These forces are the direct result of the fluid flow around the object. Therefore, for physical accuracy, not only these forces need to be computed all at once for the whole object, but also the 3D fluid flow around the object should be taken into account. This requires a full blown 3D fluid simulation with moving boundaries at the object surface. In this case, the forces acting on the object surface can be computed from the fluid pressure around the object, and this would also include the buoyancy force in addition to the drag and lift forces. Such computations are very expensive even for offline graphics purposes.

That said, we believe that the accuracy of the drag and lift force computation we present in Section 4.2.2 can be improved. In our formulation, we assign drag and lift constants to the faces of the object that determine the magnitudes of these forces. In our implementation we use the same drag and lift constants for all faces of an object and they are considered to be user defined parameters. If one can find a physically

based way of computing these constants separately for each face of an object considering the whole shape of the object, the accuracy of the drag and lift force computation can be improved. Furthermore, it might be even more accurate to use different drag and lift constants based on the motion direction of a face. Unfortunately, we do not currently have a suggestion as to how these constants can be computed or how much they would improve the accuracy of the computed drag and lift forces. Also, it is questionable whether a more physically accurate computation of drag and lift forces would improve the plausibility of the resulting object motion.

7.3. Future Extensions

In the previous section, while discussing the limitations of the wave particles method, we outlined possible future extensions that could diminish these limitations or potentially eliminate them. In this section, we discuss other possible extensions of the wave particles method that are not necessarily tied to a limitation of the current formulation of wave particles. All these would be significant extensions of the wave particles method, and are worthy of future research.

7.3.1. Splashes, Bubbles, and Foam

Wave particles are designed for simulating larger bodies of water and they can only account for the wave behavior. Other common phenomena for large bodies of water are splashes, bubbles, and foam. We consider these as secondary effects that can be added on top of a wave particle simulation. In this section, we discuss how these

secondary effects can be handled and the challenges of handling them at real-time frame rates.

7.3.1.1. Splashes

Splashes are often produced with object interaction, especially when an interacting object is moving fast near the surface of the water, such as the splashes in front of a boat when it is moving, or scattered splashes when an object is thrown into the water that hits the water surface. Splashes can also be formed due to wind especially around wave crests on open ocean or when the waves break. In fact, a significant portion of breaking wave behavior can be simulated in the form of splashes.

To be able to include splashes in a real-time water simulation system, we need to handle the following operations:

- Generating splashes,
- Simulating the motion of splash droplets in the air,
- Rendering splashes, and
- Collisions of splashes with objects and the water body.

Physically based splash generation can be a rather complicated process. We first need to determine the positions where splashes should be generated, and then we need to assign initial velocities for each generated splash droplet. When splashes are generated, generally the properties of each individual droplet are rather unpredictable, but the overall behavior of a group of splashes can be more predictable. Therefore, a

reasonable splash generation approach would compute a general splash direction and some random variation for each individual droplet.

When splashes are generated as a result of water-object interaction, it would be good to incorporate splash generation with wave generation, such that the energy or the volume of water that is used for splashes is not used for wave generation. In that spirit, we experimented with embedding the splash generation into wave generation, such that when the wave volume for generating wave particles at any point was above a certain threshold, we used the excessive volume for generating splashes instead of waves. This approach did not produce high quality splashes for the following reasons:

- First, the resolution of the silhouette we used for wave generation turned out to be too low for splash generation. While the low resolution silhouette we used was good enough for waves, when splashes were generated around the interacting object, this low resolution silhouette failed to provide plausible realism. Unfortunately, increasing the resolution of the silhouette was not acceptable, since it would significantly increase the computational cost of wave generation as well as the number of generated wave particles; thereby reducing the performance of the whole simulation system.

- Another problem was in determining the directions of the generated splashes. We tried using the generated wave particle direction for determining the horizontal direction of splashes. The process of assigning a vertical direction was arbitrary. As a result, the final splash direction was not plausible especially when the shape and orientation of the object at the time of splash generation clearly suggested splashes with certain directions.

- For splash velocities, we tried increasing the average splash velocity with the

total splash volume to be generated. This could provide plausible results, but required carefully tuning some user defined parameters used for converting the splash volume to splash velocities.

- Finally, splashes were generated only at certain frames, such as when the object first hits the water surface. While this sounds reasonable, the end result was not plausible. Instead of using a strict threshold for splash generation, converting a portion of the wave volume to splashes even when the wave volume is not that high might provide better results. Alternatively, we could generate new droplets from existing droplets as they move in the air. This is a common trick that is used for simulating splashes in offline graphics.

For all these reasons, we believe that generating splashes directly from interacting object faces can provide a better solution. In this case, while computing the wave volume for a face at the surface of the water, a portion of this volume can be converted to splashes. These splashes can be directly generated on the object face near the water level. As for determining the splash direction, the orientation of the face can be used. The speed of the generated splashes can be formulated as a function of the total splash volume and the face velocity. While this is a rather ad hoc approach to splash generation, we believe that this approach could produce plausible splashes.

Once the splashes are generated, simulating the splash motion in the air can be straightforward if we ignore the interaction between individual splash droplets. Assuming that the splash droplets do not interact with each other (ignoring all inter droplet collisions), the only external force that contributes to the splash motion is gravity. Therefore, it is possible to derive a closed form formulation for the splash droplet position at any given time using the motion properties of the splash droplet

at the time it is generated. In this way, a large number of splash droplets can be efficiently simulated.

On the other hand, realistic rendering of splashes at real-time frame rates can be challenging. For realistic results, we would need to render a very large number of splashes. For high computational efficiency, we might consider simulating only a portion of the splashes, and use these to introduce new splashes used for rendering purposes only. A common way of achieving this is using camera facing sprites with transparent textures. This way, we can render a relatively small number of splash particles and create the illusion of more particles. For high visual realism, self shadows of splashes should be computed or approximated, but this might be both difficult to implement and computationally expensive.

Finally, when the splashes collide with objects, they can be discarded or converted into drops that stick on the object. When splashes collide with the water surface, they can be discarded or converted into foam particles or used for generating small ripples on the water surface.

7.3.1.2. Bubbles

In the case of large bodies of water, bubbles are often generated when an object or the water motion itself pushes air into the water volume. This happens when a large object hits the water surface or when splashes hit the water surface.

From a computational standpoint, bubbles are very similar to splashes. Their motion in water is often more complicated than the motion of splashes and they often continue

interacting with the object that generated them. In that sense, for high quality bubble motion, it would be necessary to numerically integrate their motion, rather than trying to find a closed form solution as we suggested for splashes.

7.3.1.3. Foam

Foam can be treated as a texture on the water surface. It may be generated directly from object interaction or when the splashes or bubbles collide with the water surface. We propose two ways to incorporate foam in our water simulation:

The first method of simulating foam would be as a dynamic texture on the water surface. This could be especially useful when simulating a closed volume of water like a pool. In this case, a foam texture can be updated as objects, splashes, and bubbles interact with the water surface. Rendering foam would be as simple as mapping this texture onto the water surface with a special foam shader. However, this dynamic texture approach can cause problems when simulating an open ocean, since in this case the portion of the water surface to be simulated and mapped with this dynamic texture can be infinitely large.

Another approach of simulating foam could be by representing foam using a particle system. This particle system could be converted to a texture right before rendering the water surface. In this way, foam on an open ocean might be handled.

7.3.2. Level of Detail Approaches

One of the important advantages of the wave particles method is that it can handle very large bodies of water. In such a system, level of detail approaches might significantly reduce the cost of the simulation for the parts of the water surface that are not visible in the current view or are far from the the viewpoint.

When wave particles are generated on an open ocean, they are free to continue moving along their propagation directions until they die. When a wave particle has a high enough amplitude and a small dispersion angle, the wave particle can travel a long distance before it finally dies out. Such a wave particle can easily go out of view or move too far from the view point to produce a visible deviation on the water surface. At the same time, such a wave particle is still visited when generating the extended height field for the water surface. Furthermore, these wave particles continue to produce new wave particles via subdivision. All these actions add extra computational cost into the system without any visual benefit.

This extra cost can be avoided using level of detail approaches. Wave particles that move out of the view or move far away from the view point can be discarded when generating the extended height field for the water surface. Furthermore, subdivision of these wave particles can be suspended. When the view point changes and some of these wave particles become important again, all the subdivisions of these wave particles that were previously suspended can be computed all at once. This does not introduce any inaccuracy, since wave particles provide an analytical solution to the wave equation and they permit arbitrarily large time steps.

Furthermore, level of detail approaches can be used with water-object interaction as well. Objects that are out of the view or far from the view point can be simulated using lower resolution silhouettes and their wave generation can be suspended.

Figure 7.2 shows a massive simulation scene with over 1600 boats. In this case, we have not used any level of detail approach to improve the efficiency of the simulation. Therefore, the boats that are close to the camera are given the same importance as the boats that are far away from the camera and they all generate wave particles in exactly the same way. The performance of such a simulation can be significantly improved using level of detail approaches.

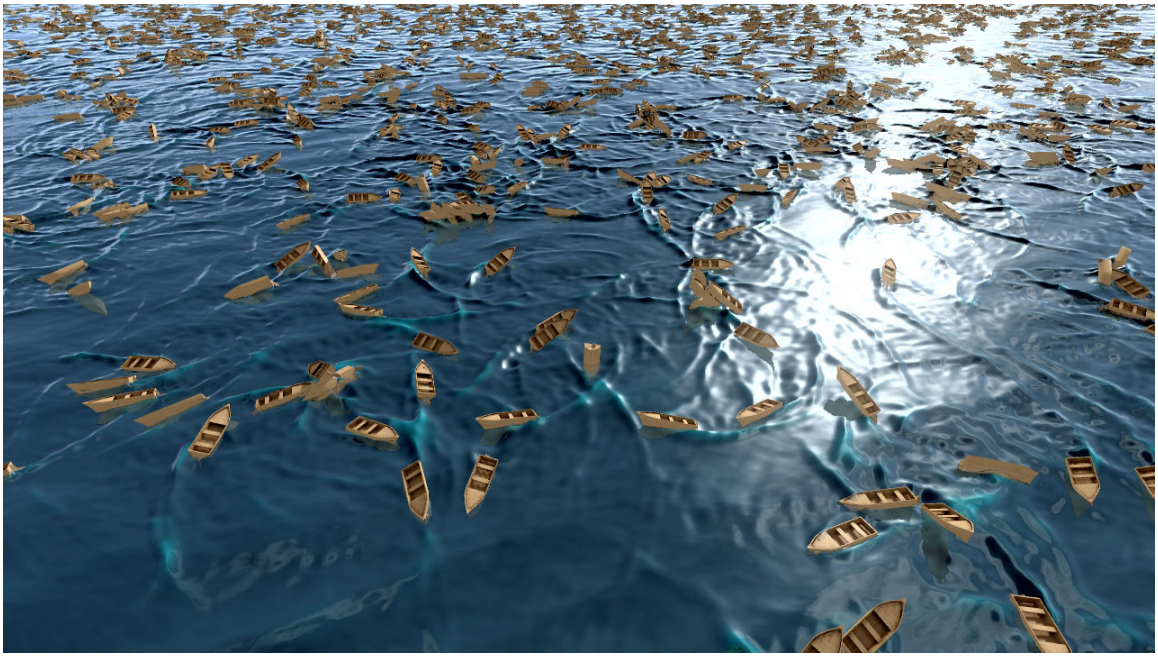


Figure 7.2. Over 1600 boats simulated without level of detail approaches.

7.3.3. Wave Choreography

Unlike traditional water simulation techniques, wave particles provide a very intuitive way of representing the deformation of the water surface. Since each wave particle directly modifies the water surface, it is very easy to predict the result of any modification of a wave particle. This opens up new fronts in controlling the water simulation.

Therefore, the wave particles method would be very useful for creating a wave choreography system. Using wave particles, a wave choreography system can easily generate waves with any desired shape and propagation direction. Also, existing waves can be easily modified to alter the shapes of the propagating waves as desired.

7.3.4. Integration with a 3D Fluid Solver

Consider an open ocean or a large pool and an object interacting with this water. The wave particles method can efficiently simulate waves on this large body of water, but it cannot handle any other water behavior. Moreover, its wave generation approach is not physically realistic, since this computation requires a full 3D fluid simulation. On the other hand, a 3D fluid simulation can properly and accurately handle almost all water behavior around the object, but it cannot efficiently simulate a large body of water. In that sense, integrating the wave particles method with a 3D fluid solver can combine the benefits of both methods. The water around the object can be simulated using the 3D fluid solver with high resolution handling all water phenomena. As the motion of the water induced by the object interaction turns into waves that propagate

away from the object, the wave particles method can simulate those waves on the rest of the water surface. This way, we can get both a high resolution 3D simulation around the object and simulate a large body of water with water waves. Furthermore, this approach can potentially overcome all limitations of the wave particles method we discussed in Section 7.2.

However, integrating a 3D fluid solver with the wave particles method is far from trivial. This is mainly because the internal simulation data used in traditional water simulation techniques does not directly tell us anything about the waves on the water surface. To be able to integrate a 3D fluid solver with the wave particles method, we need to determine a way to convert the simulation data of a traditional fluid solver into waves and eventually wave particles.

One possible approach for integrating wave particles with a 3D fluid solver is to separate the fluid domain into two distinct parts: one part that is fully simulated by the 3D fluid solver and the rest of the water medium that is simulated by the wave particles method. For integrating the two solutions, at the boundary of these two simulation domains we should convert the 3D fluid simulation result to wave particles so that the water motion induced in the 3D simulation domain can be carried out of that domain as water waves. Similarly, if dynamic water waves are coming back into the 3D simulation domain, they need to be converted back into the data structure of the 3D fluid simulation. Converting the water motion represented by wave particles can be easily converted to 3D fluid simulation data. However, converting the 3D fluid simulation data to wave particles is not as easy. First of all, there is no guarantee that the motion in the 3D fluid simulation domain near the boundary can be represented by wave motion at all. Even if it is a wave motion, how this motion can be translated

into wave particles is questionable.

We do not currently have a specific solution for this problem; however, we believe that this would be a valuable area for future research.

CHAPTER VIII

CONCLUSION

We presented wave particles, a method for simulating water surface waves at real-time frame rates. We showed that a broad class of water surface behavior can be modeled by this 2D particle system, which has an algorithmic formulation that is conceptually simple and computationally inexpensive. We developed a 2D particle system for efficiently simulating wave particles that provides an analytical solution to the wave equation. We showed how the interactions of objects with water can be efficiently handled applying fluid forces on the objects as well as by generating waves due to object motion in water. We provided details of our implementation of the wave particles method. The results we obtained show that the wave particles method can produce plausible wave simulation with two way object interaction at real-time frame rates.

We explained that the advantages of the wave particles method make it a very effective method for simulating water waves in real-time graphics applications. Along with the advantages of the wave particles method, we also discusses its limitations and how some of these limitations might be eliminated with future research. Moreover, we provided directions for future research on wave particles and our initial findings in our attempts in pursuing these future directions.

We believe that the wave particles approach makes two significant contributions to

computer graphics research and the graphics community:

First, the wave particles approach and its implementation explained in this dissertation provides an effective method for simulating water waves in real-time applications. Most current real-time applications avoid simulating large bodies of water due to their high computational requirements, which negatively impacts the realism of the virtual environments in these applications. The wave particles method would be an ideal solution for many such applications, as it is ready to be implemented in real-time applications without the need for further research and development. Although our method is tailored to the requirements of real-time graphics, it can easily be used in offline simulations to simulate thousands of objects interacting with water.

Also, the wave particles method provides an alternative approach to water simulation in computer graphics. Most traditional water simulations are based on a few fundamental approaches that use some form of numerical integration. The wave particles approach not only provides an alternative way of handling the wave simulation problem, but also shows that a seemingly complicated water behavior, like surface waves, can be modeled analytically, as opposed to numerically, and can be simulated with a conceptually simple formulation and high computational efficiency.

REFERENCES

Adams, B., Pauly, M., Keiser, R., and Guibas, L. J. 2007. Adaptively sampled particle fluids. In *SIGGRAPH '07: ACM SIGGRAPH 2007 Papers*, 48.

Bascom, W. 1980. *Waves and Beaches*. Anchor Books, Garden City, NY.

Batty, C., Bertails, F., and Bridson, R. 2007. A fast variational framework for accurate solid-fluid coupling. *ACM Transactions on Graphics 26*, 3, 100.

Baxter, W., Wendt, J., and Lin, M. C. 2004. Impasto: a realistic, interactive model for paint. In *NPAR '04: Non-photrealistic Animation and Rendering*, 45–148.

Carlson, M., Mucha, P. J., and Turk, G. 2004. Rigid fluid: animating the interplay between rigid bodies and fluid. *ACM Transactions on Graphics 23*, 3, 377–384.

Chen, J. X., and Lobo, N. d. V. 1995. Toward interactive-rate simulation of fluids with moving obstacles using Navier-Stokes equations. *Graphical Models and Image Processing 57*, 2, 107–116.

Chentanez, N., Goktekin, T. G., Feldman, B. E., and O'Brien, J. F. 2006. Simultaneous coupling of fluids and deformable bodies. In *SCA '06: Proceedings of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 83–89.

Chentanez, N., Feldman, B. E., Labelle, F., O'Brien, J. F., and Shewchuk, J. R. 2007. Liquid simulation on lattice-based tetrahedral meshes.

In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 219–228.

Clavet, S., Beaudoin, P., and Poulin, P. 2005. Particle-based viscoelastic fluid simulation. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ACM, New York, NY, USA, 219–228.

Cords, H. 2007. Mode-splitting for highly detailed, interactive liquid simulation. In *GRAPHITE '07: Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, 265–272.

Crane, K., Llamas, I., and Tariq, S. 2007. Real time simulation and rendering of 3d fluids. In *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley, ch. 30, 633–675.

Dean, R. G., and Dalrymple, R. A. 1984. *Water Wave Mechanics for Engineers and Scientists*. Prentice-Hall, Englewood Cliffs, NJ.

Enright, D., Marschner, S., and Fedkiw, R. 2002. Animation and rendering of complex water surfaces. In *Proceedings of SIGGRAPH '02*, 736–744.

Fedkiw, R., Stam, J., and Jensen, H. W. 2001. Visual simulation of smoke. In *Proceedings of SIGGRAPH '01*, 15–22.

Fedkiw, R. P. 2002. Coupling an Eulerian fluid calculation to a Lagrangian solid calculation with the ghost fluid method. *Journal of Computational Physics 175*, 1, 200–224.

Feldman, B. E., O'Brien, J. F., and Klingner, B. M. 2005. Animating gases with hybrid meshes. In *Proceedings of SIGGRAPH '05*, 904–909.

Foster, N., and Fedkiw, R. 2001. Practical animation of liquids. In *Proceedings of SIGGRAPH '01*, 23–30.

Foster, N., and Metaxas, D. 1996. Realistic animation of liquids. *Graphical Models and Image Processing 58*, 5, 471–483.

Foster, N., and Metaxas, D. 1997. Controlling fluid animation. In *CGI '97: Proceedings of the 1997 Conference on Computer Graphics International*, 178–188.

Fournier, A., and Reeves, W. T. 1986. A simple model of ocean waves. In *Proceedings of SIGGRAPH '86*, 75–84.

Génevaux, O., Habibi, A., and Dischler, J.-M. 2003. Simulating fluid-solid interaction. In *Graphics Interface*, A K Peters, CIPS, Canadian Human-Computer Commnication Society, 31–38.

Gerstner, F. V. 1802. Theory of waves. *Abhandlungen der Koenigl, boehmischen Gesellschaft der Wissenschaften zu Prag*.

Guendelman, E., Selle, A., Losasso, F., and Fedkiw, R. 2005. Coupling water and smoke to thin deformable and rigid shells. In *Proceedings of SIGGRAPH '05*, 973–981.

Hagen, T. R., Hjelmervik, J. M., Lie, K.-A., Natvig, J. R., and Henriksen, M. O. 2005. Visual simulation of shallow-water waves. *Simulation Modelling Practice and Theory 13*, 8, 716–726.

Harris, M. 2004. Fast fluid dynamics simulation on the GPU. In *GPU Gems*, R. Fernando, Ed. Addison-Wesley, ch. 38, 637–665.

Hong, W., House, D. H., and Keyser, J. 2008. Adaptive particles for incompressible fluid simulation. *The Visual Computer 24*, 7, 535–543.

Houston, B., Nielsen, M. B., Batty, C., Nilsson, O., and Museth, K. 2006. Hierarchical RLE level set: A compact and versatile deformable surface representation. *ACM Transactions on Graphics 25*, 1, 151–175.

Irving, G., Guendelman, E., Losasso, F., and Fedkiw, R. 2006. Efficient simulation of large bodies of water by coupling two and three dimensional techniques. In *Proceedings of SIGGRAPH '06*, 805–811.

Jensen, L. S., and Goliáš, R. 2001. Deep-water animation and rendering. In *Proceedings of the Game Developer's Conference*. (available at `http://www.gamasutra.com/gdce/2001/jensen/jensen_01.htm`).

Johanson, C. 2004. *Real-time Water Rendering*. M.S. thesis, Lund University.

Kass, M., and Miller, G. 1990. Rapid, stable fluid dynamics for computer graphics. In *Proceedings of SIGGRAPH '90*, 49–57.

Kim, J., Kim, S., Ko, H., and Terzopoulos, D. 2006. Fast GPU computation of the mass properties of a general shape and its application to buoyancy simulation. *The Visual Computer 22*, 856–864.

Klingner, B. M., Feldman, B. E., Chentanez, N., and O'Brien, J. F. 2006. Fluid animation with dynamic meshes. In *Proceedings of SIGGRAPH '06*, 820–825.

Koshizuka S., Tamako H., O. Y. 1996. particle method for incompressible viscous flow with fluid fragmentation. *Computational Fluid Dynamics Journal 29*, 4, 29–46.

Lenaerts, T., Adams, B., and Dutré, P. 2008. Porous flow in particle-based fluid simulations. *ACM Transactions on Graphics 27*, 3, 1–8.

Losasso, F., Gibou, F., and Fedkiw, R. 2004. Simulating water and smoke with an octree data structure. *ACM Transactions on Graphics 23*, 3, 457–462.

Losasso, F., Shinar, T., Selle, A., and Fedkiw, R. 2006. Multiple interacting liquids. In *Proceedings of SIGGRAPH '06*, ACM Press, 812–819.

Lucy, L. B. 1977. A numerical approach to the testing of the fission hypothesis. *Astronomical Journal 82*, 1013–1024.

Mastin, G. A., Watterberg, P. A., and Mareda, J. F. 1987. Fourier synthesis of ocean scenes. *IEEE Computer Graphics and Applications 7*, 3, 16–23.

Miller, G., and Pearce, A. 1989. Globular dynamics: A connected particle system for animating viscous fluids. *Computers and Graphics 13*, 3, 305–309.

Monaghan, J. J. 1977. Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics 30*, 543–574.

Müller, M., Charypar, D., and Gross, M. 2003. Particle-based fluid simulation for interactive applications. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 154–159.

Munson, B. R., Young, D. F., and Okiishi, T. H. 2006. *Fundamentals of Fluid Mechanics*. Wiley, New York, NY.

NVIDIA, 2010. Physx. http://www.nvidia.com/object/physx_new.html.

O'Brien, J. F., and Hodgins, J. K. 1995. Dynamic simulation of splashing fluids. In *CA '95: Proceedings of the Computer Animation*, 198–205.

Peachey, D. R. 1986. Modeling waves and surf. In *Proceedings of SIGGRAPH '86*, 65–74.

Perlin, K., and Hoffert, E. M. 1989. Hypertexture. In *Proceedings of SIGGRAPH '89*, 253–262.

Premoze, S., Tasdizen, T., Bigler, J., Lefohn, A., and Whitaker, R. 2003. Particle-based simulation of fluids. *Computer Graphics Forum (Eurographics Proceedings) 22*, 3, 401–410.

Reeves, W. T. 1983. Particle systems a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics 2*, 2, 91–108.

Robinson-Mosher, A., Shinar, T., Gretarsson, J., Su, J., and Fedkiw, R. 2008. Two-way coupling of fluids to rigid and deformable solids and shells. *ACM Transactions on Graphics 27*, 3, 1–9.

Robinson-Mosher, A., English, R. E., and Fedkiw, R. 2009. Accurate tangential velocities for solid fluid coupling. In *SCA '09: Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 227–236.

Schachter, B. 1980. Long crested wave models. *Computer Graphics and Image Processing 12*, 187–201.

Schneider, J., and Westermann, R. 2001. Towards real-time visual simulation of water surfaces. In *VMV '01: Proceedings of the Vision Modeling and Visualization Conference 2001*, 211–218.

Selle, A., Rasmussen, N., and Fedkiw, R. 2005. A vortex particle method for smoke, water and explosions. In *Proceedings of SIGGRAPH '05*, 910–914.

Shah, M. A., Konttinen, J., and Pattanaik, S. 2007. Caustics mapping: An image-space technique for real-time caustics. *IEEE Transactions on Visualization and Computer Graphics 13*, 2, 272–280.

Sin, F., Bargteil, A. W., and Hodgins, J. K. 2009. A point-based method for animating incompressible flow. In *SCA '09: Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 247–255.

Solenthaler, B., and Pajarola, R. 2009. Predictive-corrective incompressible sph. In *SIGGRAPH '09: ACM SIGGRAPH 2009 Papers*, 1–6.

Stam, J., and Fiume, E. 1995. Depicting fire and other gaseous phenomena using diffusion processes. In *SIGGRAPH '95: Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, 129–136.

Stam, J. 1999. Stable fluids. In *Proceedings of SIGGRAPH '99*, 121–128.

Takashi, T., Heihachi, U., and Kunimatsu, A. 2002. The simulation of fluid-rigid body interaction. In *SIGGRAPH '02: Sketches & Applications*, 226.

Takashi, T., Fujii, H., Kunimatsu, A., Hiwada, K., Saito, T., Tanaka, K., and Ueki, H. 2003. Realistic animation of fluid with splash and foam. *Computer Graphics Forum 22 3*, 391–401.

Terzopoulos, D., Platt, J., and Fleischer, K. 1989. Heating and melting deformable models (from goop to glop). *Graphics Interface 89*, 219–226.

Tessendorf, J. 2001. Simulating ocean water. In *Simulating Nature: Realistic and Interactive Techniques*, ACM SIGGRAPH '01 Course #47 Notes.

Tessendorf, J. 2004. Interactive water surfaces. In *Game Programming Gems 4*, A. Kirmse, Ed. Charles River Media.

Treuille, A., Lewis, A., and Popovic, Z. 2006. Model reduction for real-time fluids. *ACM Transactions on Graphics 25*, 3, 826–834.

Ts'o, P. Y., and Barsky, B. A. 1987. Modeling and rendering waves: wave-tracing using beta-splines and reflective and refractive texture mapping. *ACM Transactions on Graphics 6*, 3, 191–214.

Yuksel, C., House, D. H., and Keyser, J. 2007. Wave particles. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007) 26*, 3, 99.

Zhu, Y., and Bridson, R. 2005. Animating sand as a fluid. *ACM Transactions on Graphics 24*, 3, 965–972.

APPENDIX A

ERROR DUE TO THE RADIAL DEFINITION OF WAVE PARTICLES

Here we provide the details of our error analysis due to radial definition of wave particles that is presented in Section 3.7 as follows:

The maximum difference between the height of the wave crest for a linear wavefront and its representation with evenly spaced radial wave particles is less than 0.8% *of the wave amplitude, and the maximum difference between the shape of the wave and its wave particle representation is less than* 7.1% *of the wave amplitude, as long as the distance between two neighboring wave particles is less than or equal to half of the wave particle radius.*

Let us consider a linear wavefront represented by a number of wave particles placed side by side along the x axis with equal distance apart, and let d be the distance between two neighboring wave particles on this wavefront. The wave particle system keeps $d \leq \frac{1}{2}r$, where r is the radius of a wave particle. Therefore, we only need to look at the possible d values in the range $0 < d < \frac{1}{2}r$.

Figure A.1 shows the placement of wave particles on the wavefront for different values of d. Due to symmetry, we only need to examine the shaded region, where $0 \leq x \leq \frac{d}{2}$ and $x = 0$ is in the middle of two wave particles. When $d = \frac{1}{2}r$, only four wave particles affect this region. For d values in the range $\frac{2}{5}r \leq d < \frac{1}{2}r$, five wave particles affect this region, and when $\frac{1}{3}r \leq d < \frac{2}{5}r$, the shaded region is affected by six wave particles, as seen in Figure A.1. The number of wave particles n affecting the shaded

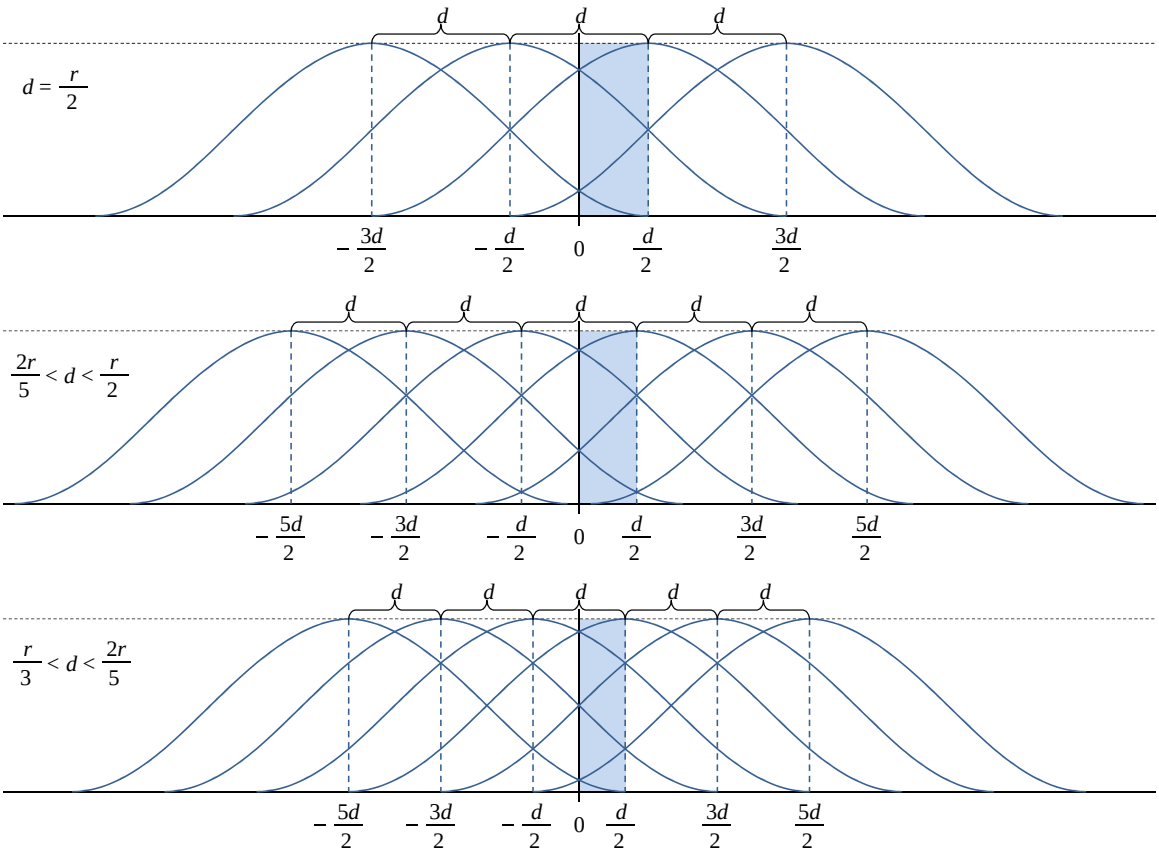


Figure A.1. The placement of wave particles on a wavefront for different values of d in the range $\frac{r}{3} < d \leq \frac{r}{2}$.

region is $n = \lceil \frac{2r}{d} \rceil$. The total vertical deviation in the shaded region $0 \leq x \leq \frac{1}{2}d$ is

$$D_z = \frac{a}{2} \sum_{i=0}^{\lceil \frac{r}{d} \rceil} \left(\left(\cos(X_i^+) + 1 \right) \Pi \left(\frac{X_i^+}{2} \right) + \left(\cos(X_i^-) + 1 \right) \Pi \left(\frac{X_i^-}{2} \right) \right) , \tag{A.1}$$

where a is the wave particle amplitude and X^+ and X^- are

$$X_i^+ = \frac{\pi}{r} \left(x + \frac{(2i+1)d}{2} \right) , \text{ and} \tag{A.2}$$

$$X_i^- = \frac{\pi}{r} \left(x - \frac{(2i+1)d}{2} \right) . \tag{A.3}$$

Equation A.1 sums pairs of wave particles in the positive and negative x directions. To find the local maximum and minimum of this equation, we look at its derivative with respect to x

$$D'_z = -\frac{a}{2} \sum_{i=0}^{\lceil \frac{r}{d} \rceil} \left(\sin(X_i^+) \, \Pi \left(\frac{X_i^+}{2} \right) + \sin(X_i^-) \, \Pi \left(\frac{X_i^-}{2} \right) \right) . \tag{A.4}$$

For the region $0 \leq x \leq \frac{1}{2}d$, the derivative D'_z is zero only at $x = 0$. This means that the curve D_z in this range has a local minimum or maximum at $x = 0$, thus the maximum and minimum values of D_z are at the end points of this region. Therefore, the maximum absolute difference between D_z and the ideal vertical deviation $a\frac{r}{d}$ at $x = 0$ and $x = \frac{1}{2}d$ gives us the maximum error for the given d value. Using this procedure, we found that

- over the range $\frac{2}{5}r \leq d < \frac{1}{2}r$ the error is less than 0.8%,
- over the range $\frac{1}{3}r \leq d < \frac{2}{5}r$ the error is less than 0.5%, and
- over the range $\frac{2}{7}r \leq d < \frac{1}{3}r$ the error is less than 0.3%

of the wavefront peak amplitude. As one would expect, the error decreases as more and more wave particle influence the shaded region with decreasing d value. Hence, the maximum error appears in the region $\frac{2}{5}r < d < \frac{1}{2}r$, where at most five wave particles affect the shaded region. Therefore, we can conclude that the maximum error on the wave crest is always less than 0.8% of the wave amplitude.

As for the maximum error of the overall wave shape, it appears when $d = \frac{1}{2}r$, i.e. the largest permitted distance between neighboring wave particles. To find the maximum error of the overall wave shape, we should compare the D_z values of the radial definition of wave particles to the target wave shape for $0 \leq y \leq r$. In this range for

y, the target vertical deviation can be written as

$$D_z^{target} = \frac{a\,r}{2d}\left(\cos\left(\frac{\pi y}{r}\right)+1\right)\ . \tag{A.5}$$

On the other hand, the vertical deviation provided by the radial definition of the wave particles is

$$D_z = \frac{a}{2}\sum_{i=0}^{\lceil \frac{r}{d} \rceil} 2\left(\cos(R_i)+1\right)\ \Pi\left(\frac{R_i}{2}\right)\ , \tag{A.6}$$

where

$$R_i = \frac{\pi}{r}\sqrt{\left(\frac{(2i+1)d}{2}\right)^2 + y^2}\ . \tag{A.7}$$

Note that, unlike Equation A.1, Equation A.6 does not have two separate components for each pair of wave particles, because at $x = 0$ both wave particles of each pair evaluate to the same value. For $d = \frac{1}{2}r$, the maximum difference between the value of D_z in Equation A.6 and D_z^{target} happens at $y \approx 0.55r$, and the error at this point is less that 7.1% of the main amplitude.

APPENDIX B

FRACTION OF A FACE INSIDE THE WATER

Here we provide a simple GPU shader function for computing the fraction of a face inside the water. Figure B.2 shows an example triangle that is partially in water.

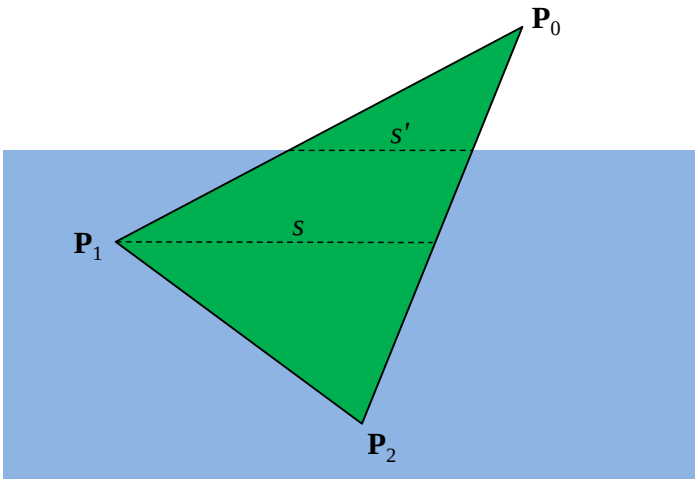


Figure B.2. A triangle inside the water.

Without loss of generality, let us assume that the surface normal of this triangle is in the y-direction. In this figure, s denotes the horizontal line that connects the point $\mathbf{P}_1$ to the edge that connects $\mathbf{P}_0$ and $\mathbf{P}_2$. The area of this triangle can be written as

$$A = \frac{1}{2}\, s\, (z_0 - z_2)\ , \tag{B.1}$$

where z_0, z_1, and z_2 are the z-coordinates of the points $\mathbf{P}_0$, $\mathbf{P}_1$, and $\mathbf{P}_2$ respectively.

Similarly, the area outside the water is

$$A_{out} = \frac{1}{2}\, s' \left(z_0 - z_w\right) , \tag{B.2}$$

where z_w is the height of the water surface. For simplicity we assume that the water height is constant near the face and it is computed at the centroid of the face. Furthermore,

$$\frac{s'}{s} = \frac{z_0 - z_w}{z0 - z1} . \tag{B.3}$$

Combining these three equations, we find the ratio of the triangle area outside the water as

$$\frac{A_{out}}{A} = \frac{\left(z_0 - z_w\right)^2}{\left(z0 - z1\right)\left(z_0 - z_2\right)} . \tag{B.4}$$

The following GPU shader function computes the fraction of the triangle inside the water from the vertical positions of the three vertices of a triangular face. The input to this function is the water height and the vertical coordinates of the three vertices of the triangle `z0`, `z1`, and `z2`, which are sorted such that `z0` is the smallest and `z2` is the largest.

```
//-------------------------------------------------------------------------------------------

float FractionInWater( float waterLevel, float z0, float z1, float z2 )
{
    float fraction = 1;                                 // assume fully in water
    if ( waterlevel < z0 ) ratio = 0;                   // not in water
    else {                                              // in water
        float h01 = z0 - z1;
        float h02 = z0 - z2;
        float h12 = z1 - z2;
        if ( waterlevel < z1 ) {                        // lower part of the triangle
            float h0w = z0 - waterlevel;
            fraction = ( h0w * h0w ) / ( h01 * h02 );
        } else if ( waterlevel < z2 ) {                 // higher part of the triangle
            float hw2 = waterlevel - z2;
            fraction = 1 - ( hw2 * hw2 ) / ( h12 * h02 );
        }                                               // otherwise, fully in water
    }
    return fraction;
}

//-------------------------------------------------------------------------------------------
```

APPENDIX C

HEIGHT FIELD GENERATION SHADERS

Here we include the shader source code of our implementation of the height field generation. It has three passes:

- Rendering wave particle points,
- Filtering in the x direction, and
- Filtering in the y direction.

For more information on these passes, please see Section 5.1.5.

The implementation presented here outputs three separate textures: 3D surface deviation, 2D gradient, and 3D water surface velocity. In our original implementation, we output two textures only by packing these three outputs into eight channels of two textures. We use the separable filter approximation to compute these values.

The separable filters for vertical and horizontal deviations are provided in equations 5.6 through 5.11. The filter equations for surface gradients are

$$g_x^X(x) = -\frac{1}{2}\sin\left(\frac{\pi x}{r}\right)\left(\frac{\pi}{r}\right), \tag{C.1}$$

$$g_x^Y(y) = \frac{1}{2}\left(\cos\left(\frac{\pi y}{r}\right)+1\right), \tag{C.2}$$

$$g_y^X(x) = \frac{1}{2}\left(\cos\left(\frac{\pi x}{r}\right)+1\right), \text{ and} \tag{C.3}$$

$$g_y^Y(y) = -\frac{1}{2}\sin\left(\frac{\pi y}{r}\right)\left(\frac{\pi}{r}\right), \tag{C.4}$$

where g_x and g_y are the x and y components of the surface gradient. Note that functions g_x^X and g_y^Y correspond to the spatial derivative of the vertical deviation function (equations 5.6 and 5.7). Since they are derived from the vertical deviation function, these filters approximate the surface gradient in the absence of horizontal deviation.

To include the effect of the horizontal deviation on the surface gradient, we need to divide these gradient values by one plus the spatial derivative of the horizontal deviation. We approximate the spatial derivative of the horizontal deviation using the filter functions

$$h_x^X(x) = -\frac{1}{2}\left(\cos\left(\frac{2\pi x}{r}\right)+\cos\left(\frac{\pi x}{r}\right)\right)\left(\frac{\pi}{r}\right), \tag{C.5}$$

$$h_x^Y(y) = \frac{1}{4}\left(\cos\left(\frac{\pi y}{r}\right)+1\right)^2, \tag{C.6}$$

$$h_y^X(x) = \frac{1}{4}\left(\cos\left(\frac{\pi x}{r}\right)+1\right)^2, \text{ and} \tag{C.7}$$

$$h_y^Y(y) = -\frac{1}{2}\left(\cos\left(\frac{2\pi y}{r}\right)+\cos\left(\frac{\pi y}{r}\right)\right)\left(\frac{\pi}{r}\right). \tag{C.8}$$

In this form, the gradient in the x direction is calculated as $g_x/(1+h_x)$ and, similarly, the gradient in the y direction is $g_y/(1+h_y)$. This way, we take the horizontal deviation into account while computing the surface gradient. Notice that the functions h_x^X and h_y^Y are the spatial derivatives of the horizontal deviation functions (equations 5.8 and 5.11).

The horizontal velocity is computed similarly using the filter functions that correspond to the time derivative of the horizontal deviation functions (equations 5.8 and 5.11),

such that

$$v_x^X(x) = -\frac{1}{2}\left(\cos\left(\frac{2\pi x}{r}\right) + \cos\left(\frac{\pi x}{r}\right)\right)\left(\frac{\pi \upsilon}{r}\right) , \tag{C.9}$$

$$v_x^Y(y) = \frac{1}{4}\left(\cos\left(\frac{\pi y}{r}\right) + 1\right)^2 , \tag{C.10}$$

$$v_y^X(x) = \frac{1}{4}\left(\cos\left(\frac{\pi x}{r}\right) + 1\right)^2 , \text{ and} \tag{C.11}$$

$$v_y^Y(y) = -\frac{1}{2}\left(\cos\left(\frac{2\pi y}{r}\right) + \cos\left(\frac{\pi y}{r}\right)\right)\left(\frac{\pi \upsilon}{r}\right) , \tag{C.12}$$

where υ is the wave speed. For computing the vertical velocity, we first find the average wave propagation direction. Then, we calculate the surface gradient in this average wave propagation direction by taking a dot product of it with the computed surface gradient. Multiplying this value with the wave speed υ gives us the vertical component of the water surface velocity.

Looking at all the filter functions we use, one can see that there are only five types of filter functions, such that

$$f_1(v) = \frac{1}{2}\left(\cos\left(\frac{\pi v}{r}\right) + 1\right) , \tag{C.13}$$

$$f_2(v) = -\frac{1}{2}\sin\left(\frac{\pi v}{r}\right) , \tag{C.14}$$

$$f_3(v) = -\frac{1}{2}\left(\cos\left(\frac{2\pi v}{r}\right) + \cos\left(\frac{\pi v}{r}\right)\right) , \tag{C.15}$$

$$f_4(v) = 2\, f_1(v)\, f_2(v) , \text{ and} \tag{C.16}$$

$$f_5(v) = [f_1(v)]^2 . \tag{C.17}$$

In the code below we use the `GetFilter` function to compute the values of the first three functions. In our implementation we precompute these values and use a small 1D texture to pass them to the shader.

The implementation presented here is for a pool simulation. In an open ocean simu-

lation with projected height field, extra lines should be added to adjust the filter size at each fragment of the filtering passes.

```
//-----------------------------------------------------------------------------------------------

// The vertex program for rendering wave particles as point
// primitives on the extended height field texture.

void HF_Points_VP ( in  float3 posTime   : POSITION  // x:posx y:posy z:time
                  , in  float3 velAmp    : NORMAL    // x:velx y:vely z:amplitude
                  , out float4 clipPos   : POSITION  // output position
                  , out float3 waveInfo1 : TEXCOORD0 // x:screenPosX y:screenPosY z:amplitude
                  , out float2 waveInfo2 : TEXCOORD1 // x:velX y:velY z:posX w:posY
                  , uniform float time               // current time
                  )
{
   if ( velAmp.z == 0 ) {          // if wave amplitude is zero
      clipPos = float4(2,2,2,1);   // move the point outside the view
   } else {
      // set position
      float2 pos = posTime.xy + (time-posTime.z) * velAmp.xy;
      float2 spos = WAVE_TEXCOORD(pos.xy);
      waveInfo1.x = spos.x * WAVE_TEXTURE_WIDTH;
      waveInfo1.y = spos.y * WAVE_TEXTURE_HEIGHT;
      clipPos = float4( spos * 2.0f - 1.0f, 0, 1 );

      // set amplitude
      waveInfo1.z = velAmp.z * exp( (posTime.z-time) * WAVE_DAMPING );

      // set velocity
      waveInfo2.xy = velAmp.xy;
   }
}


//-----------------------------------------------------------------------------------------------


// The fragment program for rendering wave particles as point
// primitives on the extended height field texture.

void HF_Points_FP ( in  float3 waveInfo : TEXCOORD0 // x:posX y:posY z:amplitude
                  , in  float2 waveVel  : TEXCOORD1 // wave velocity
                  , in  float2 wpos     : WPOS      // fragment position
                  , out float4 velAmp   : COLOR     // x:velX y:velY z:amplitude
                  )
{
   // pool edges
   waveInfo.x = (waveInfo.x < 0.5) ? 0.5 :
       ( waveInfo.x > (WAVE_TEXTURE_WIDTH -0.5) ) ? (WAVE_TEXTURE_WIDTH -0.5) : waveInfo.x;
   waveInfo.y = (waveInfo.y < 0.5) ? 0.5 :
       ( waveInfo.y > (WAVE_TEXTURE_HEIGHT-0.5) ) ? (WAVE_TEXTURE_HEIGHT-0.5) : waveInfo.y;

   // antialiasing
   float ax = 1.0f - abs(waveInfo.x - wpos.x);
   float ay = 1.0f - abs(waveInfo.y - wpos.y);
   float a  = ax * ay;

   velAmp.xy = waveInfo.z * a * waveVel.xy;  // wave particle velocity
   velAmp.z  = waveInfo.z * a;               // wave particle amplitude
   velAmp.a  = 1.0f;                         // full alpha, not used
}
```

```
//-------------------------------------------------------------------------------------------------


// returns filters f_1, f_2, and f_3

float3 GetFilter( in float v )
{
   float s, c;
   sincos( PI*v, s, c );
   return float3(
       0.5f  * (c + 1.0f),      // 0.5 ( cos(v) + 1 )
      -0.5f  * s,               // -0.5 sin(v)
      -0.25f * (c*c - s*s + c)  // cos(2v) + cos(v)
      );
}


//-------------------------------------------------------------------------------------------------

// f_1:  0.5  * ( cos(v) + 1 )
// f_2: -0.5  * sin(v)
// f_3: -0.25 * ( cos(2v) + cos(v) )
// f_4: -0.5  * sin(v) * ( cos(v) + 1 )
// f_5:  0.25 * ( cos(v) + 1 )^2

// The fragment program for filtering the height field texture
// in the horizontal direction.

void HF_HFilter_FP ( in float2 texcoord        : TEXCOORD0 // texture coordinate
                   , out float4 f123           : COLOR0    // x:f_1 y:f_2 z:f_3  (times amplitude)
                   , out float4 f45v           : COLOR1    // x:f_4 y:f_5 z:velX w:velY
                   , uniform sampler2D texture : TEXUNIT0  // x:velX y:velY z:amplitude
                   )
{
   // current pixel
   float3 velAmp = tex2D( texture, texcoord ).xyz;          // x:velX y:velY z:amplitude
   f123 = float4( velAmp.z, 0, 0.5f*velAmp.z, 1 );          // x:f_1 y:f_2 z:f_3  (times amplitude)
   f45v = float4( 0, velAmp.z, sign(velAmp.z)*velAmp.xy );  // x:f_4 y:f_5 z:velX w:velY
   float texOffset = 0;                                     // loop variable

   // neighboring pixels
   for ( float i=WAVE_TEXEL_WIDTH_IN_WORLD; i<WAVE_PARTICLE_RADIUS; i+=WAVE_TEXEL_WIDTH_IN_WORLD )
   {
      texOffset += TEXEL_WIDTH;
      float3 velAmpL = tex2D( texture, float2( texcoord.x + texOffset, texcoord.y ) ).xyz; // right
      float3 velAmpR = tex2D( texture, float2( texcoord.x - texOffset, texcoord.y ) ).xyz; // left
      float  ampSum = velAmpL.z + velAmpR.z;
      float  ampDif = velAmpL.z - velAmpR.z;

      float3 f = GetFilter( i/float(WAVE_PARTICLE_RADIUS) );
      f123.x  += ampSum * f.x;       // f_1
      f123.y  += ampDif * f.y;       // f_2
      f123.z  += ampSum * f.z;       // f_3
      f45v.x  += ampDif * f.x*f.y;   // f_4
      f45v.x  += ampSum * f.x*f.x;   // f_5

      // average velocity
      f45v.z  += ( sign(velAmpL.z)*velAmpL.x + sign(velAmpR.z)*velAmpR.x ) * f.x;
      f45v.w  += ( sign(velAmpL.z)*velAmpL.y + sign(velAmpR.z)*velAmpR.y ) * f.x;
   }
}
```

```
//-------------------------------------------------------------------------------------------------


// The fragment program for filtering the height field texture in the vertical direction.

void HF_VFilter_FP ( in float2 texcoord      : TEXCOORD0 // texture coordinate
                   , out float4 deviation    : COLOR0    // x:devX  y:devY  z:devZ
                   , out float4 gradient     : COLOR1    // x:gradX y:gradY
                   , out float4 velocity     : COLOR2    // x:velX  y:velY  z:velZ
                   , uniform sampler2D texX0 : TEXUNIT0  // x:f_1 y:f_2 z:f_3  (times amplitude)
                   , uniform sampler2D texX1 : TEXUNIT1  // x:f_4 y:f_5 z:velX w:velY
                   )
{
   // current pixel
   float3 f123 = tex2D( texX0, texcoord ).xyz;           // x:f_1 y:f_2 z:f_3  (times amplitude)
   float4 f45v = tex2D( texX1, texcoord ).xyzw;          // x:f_4 y:f_5 z:velX w:velY
   deviation   = float4( f45v.x, 0, f123.x, 1 );         // initialize deviation at this pixel
   gradient    = float4( f123.y, 0, 0, 1 );              // initialize gradient at this pixel
   velocity    = float4( f123.z, -0.5f*f45.y, 0, 1 );    // initialize velocity at this pixel
   float3 gradCorr  = float2( f123.z, f45v.y );          // initialize gradient correction
   float2 dir       = val1.zw;                           // average direction
   float  texOffset = 0;                                 // loop variable

   // neighboring pixels
   for ( float i=WAVE_TEXEL_WIDTH_IN_WORLD; i<WAVE_PARTICLE_RADIUS; i+=WAVE_TEXEL_WIDTH_IN_WORLD )
   {
      texOffset += TEXEL_HEIGHT;
      float3 f123B = tex2D( texX0, float2( texcoord.x, texcoord.y + texOffset ) ).xyz;  // bottom
      float4 f45vB = tex2D( texX1, float2( texcoord.x, texcoord.y + texOffset ) ).xyzw; // bottom
      float3 f123T = tex2D( texX0, float2( texcoord.x, texcoord.y - texOffset ) ).xyz;  // top
      float4 f45vT = tex2D( texX1, float2( texcoord.x, texcoord.y - texOffset ) ).xyzw; // top

      float3 f = GetFilter( i/float(WAVE_PARTICLE_RADIUS) );

      deviation.x += (f45vB.x + f45vT.x) * f.x*f.x;      // deviation X
      deviation.y += (f45vB.y - f45vT.y) * 2*f.x*f.y;    // deviation Y
      deviation.z += (f123B.x + f123T.x) * f.x;          // deviation Z

      gradient.x  += (f123B.y + f123T.y) * f.x;          // gradient X
      gradient.y  += (f123B.x - f123T.x) * f.y;          // gradient Y
      gradCorr.x  += (f123B.z + f123T.z) * f.x*f.x;      // gradient X horizontal deviation
      gradCorr.y  += (f45vB.y + f45vT.y) * f.z;          // gradient Y horizontal deviation

      velocity.x  += (f123B.z + f123T.z) * f.x*f.x;      // velocity X
      velocity.y  += (f45vB.y + f45vT.y) * f.z;          // velocity Y
      dir += ( f45vB.zw + f45vT.zw ) * f.x;              // average direction

   }
   // fix gradient considering horizontal deviation
   gradCorr *= PI / WAVE_PARTICLE_RADIUS;
   gradient.xy *= ( PI / WAVE_PARTICLE_RADIUS ) / (1 + gradCorr);

   velocity.xy *= PI * WAVE_SPEED / WAVE_PARTICLE_RADIUS;   // fix velocity magnitude
   dir = normalize(dir);                                    // average propagation direction
   velocity.z = dot( dir, gradient.xy ) * WAVE_SPEED;       // vertical velocity

   AddAmbientWaves( deviation, gradient, velocity );        // precomputed ambient waves
}

//-------------------------------------------------------------------------------------------------
```

VITA

Name:	Cem Yuksel
Email Address:	cem@cemyuksel.com
Web-site:	`www.cemyuksel.com`
Address:	Department of Computer Science and Engineering Texas A&M University TAMU 3112 College Station, TX 77843-3112
Education:	B.S., Physics, Boğaziçi University at Istanbul, Turkey 2000 M.S., Computer Engineering, Boğaziçi University at Istanbul, Turkey 2003