An Interactive System for Real-time Wavefront-based Water Waves Simulation

by

Sean Dillon, B.Sc.

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

August 2017
Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Sean Dillon

August 29, 2017
Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Sean Dillon

August 29, 2017
Acknowledgments

I would like to thank my thesis supervisor, Carol O’Sullivan, for her invaluable feedback and encouragement from the beginning to the end of this project.

I would also like to thank my family for their understanding and support throughout this year.

Sean Dillon

University of Dublin, Trinity College

August 2017
An Interactive System for Real-time Wavefront-based Water Waves Simulation

Sean Dillon
University of Dublin, Trinity College, 2017

Supervisor: Carol O’Sullivan

The work of this thesis aims to bring a state of the art research by S. Jeschke and C. Wojtan [Jeschke and Wojtan(2015)], for water waves simulation through wavefront propagation into the real time domain. The original work stands out for two major qualities: firstly, a very accurate height calculation scheme is used for each rendered point of the surface. Additionally, their system is the first to unify the 4 desired wave behaviours, which enables rich environmental interactions. Our work consists of three parts. Firstly, we manage to perform the simulation and the rendering simultaneously by substituting the original interpolation scheme with a heuristic propagation model. This novelty allows us to propose some methods for enabling interactions between dynamic bodies and the water surface, which is our secondary contribution. Lastly, we perform a technical evaluation on the implementation with the hopes of providing a better understanding of both the technical and design limitations of such a system and we provide our insights for the direction of the future work.
Contents

Acknowledgments iv

Abstract v

List of Figures ix

Chapter 1 Introduction 1

1.1 Wavefronts Versus Other Approaches ................................. 1
1.2 Motivation and Goals ....................................................... 2
1.3 Thesis Outline ............................................................... 4

Chapter 2 Background 5

2.1 Desired Wave Behaviours .................................................. 5
2.2 Literature Review ........................................................... 7
2.3 Previous Work ............................................................... 8

Chapter 3 Wavefront Propagation 12

3.1 Elements of Linear Wave Theory ........................................ 13
  3.1.1 Angular Frequency and Phase Speed ............................... 13
  3.1.2 Amplitude, Energy Density and Group Speed ...................... 15
  3.1.3 Phase Function ......................................................... 16
3.2 Wavefront Propagation .................................................... 17
3.3 Resolution Refinement of the Wavefront Curve ....................... 19
Appendix A  Abbreviations  61

Appendix B  Appendix: GLSL shading  62

Bibliography  72
List of Figures

1.1 Visual comparison between two systems. On the left we have a 2-Dimensional grid system that is based on simplifications on the Navier Stokes equation. On the right we have the system described in this thesis. ........................................... 2

2.1 Examples of Refraction (blue) and Dispersion (yellow). The blue lines show the direction of the waves and the yellow underline the crests to show the difference of wavelength. Photos taken at Howth, Ireland (from our personal collection). .................................................. 6

2.2 Examples of Reflection (green) and Diffraction (red). The lines show the direction of the waves. Photos taken at the Grand Canal and the Liffey riven in Dublin, Ireland. (from our personal collection). ....................... 6

3.1 Depth and wavenum plots against angular frequency, $g=9.8, \sigma = 1, \rho = 1$. ............................................. 14

3.2 Depth and wavenum plots against phase speed, $g=9.8, \sigma = 1, \rho = 1$ .................................................. 15

3.3 Propagation of a circular wavefront. The yellow map represents amplitude, the blue map angular frequency, the red map arrival time and the green map depth. Values are mapped to color intensities. ............................ 19

3.4 The length of the segment $v_a, v_b$ surpasses the maximum limit and thus the midpoint vertex $v_c$ is introduced .................................................. 20

3.5 Wavefront partition: A single wavefront is propagated (top left), and is steadily subdivided (chronological order: top left, top right, bottom left, bottom right) .................................................. 21
3.6 Collision detection for two neighbouring segments

3.7 Three cases of the Left predicate. In the degenerate cases, the directed perpendicular normal is calculated for the projection $q'$ of $q$, and the decision is based on the location of $q'$.  

3.8 Search spaces for two neighbouring segments (magenta). The axis-aligned bounding box (red) is also depicted.

3.9 A visual representation of how values are stored on a wavefront curve for interpolating when a collision is detected. $a_{i-j}$ denotes the average between $a_i$ and $a_j$. In addition to the values shown, some values that do not vary over the curve, such as the wavefront ID, are stored at the covered grid point.

4.1 Refraction of a wave front: The original direction is the crossing diagonal, but the vertices of the curve (cyan) change direction due to change of depth and come closer to the shoreline.

4.2 Refraction normal vectors of an ocean bed: Green is the up vector, red is the right vector and blue is any vector with positive coordinates. A threshold is being used to cutoff the higher depths.

4.3 Wavefronts interacting with static obstacles. Notice that the leftmost obstacle caused a reflection and the furthermost a diffraction.

4.4 A degenerate case of the reflection and diffraction collision detection. Note that the size of the obstacle (cyan) is close to the minimum length of the wavefront edges (red, dashed).

4.5 The pre-processing in stages. The oriented curve with the appropriate normals is generated from a 3D convex hull.

4.6 Wavefronts generated from vertical and horizontal moving bodies. The big arrow corresponds to the direction of the linear momentum of the body.
4.7. Example of sampling the flow field. Each vertex position of the object’s curve is used as a sampling point: $v_a = IDW(v_{-1,-1}, v_{0,-1}, v_{-1,0}, v_{0,0})$, $v_b = IDW(v_{0,-1}, v_{1,-1}, v_{1,0}, v_{0,0})$, $v_c = IDW(v_{-1,0}, v_{0,0}, v_{0,1}, v_{-1,1})$, $v_d = IDW(v_{0,0}, v_{1,0}, v_{1,1}, v_{0,1})$.

5.1. Different grid resolutions representing the same wave. From Top to Bottom, Left to Right: 200x200, 100x100, 50x50, 25x25.

5.2. First approach example. The sampling grid has a resolution of 25 by 25 and the heights grid is 400 by 400.

5.3. Examples of “block” artifacts in the first approach.

5.4. Two cases of Algorithm 8. Red represents the pivots, green the positions and blue are the positions that are missed. The size of the dots relates to the weight of the point.

5.5. Example of traversal that handles missed cases. The direction vector is (1,0.5). Red arrows show the pivot point traversal, green ones the position points traversal and blue ones the special case points traversal. Note that each point of the grid is covered exactly once.

5.6. Propagation directions: the first two cases will form perfect waves while the third case will have a more chaotic effect.


6.2. Rendering example that demonstrates the Fresnel effect and the smaller waves.

7.1. Snapshots from the demo scene.

7.2. Time performance when dynamic bodies do not generate wavefronts.

7.3. Time performance when dynamic bodies generate wavefronts.

7.4. Memory performance for wavefronts generated from moving bodies.
7.5 Memory performance for no wavefronts generated from moving bodies...
Chapter 1

Introduction

In this chapter we give some introductory context and motivation about this work. We also provide an outline about the organization of this text.

1.1 Wavefronts Versus Other Approaches

The first thing that we would like to clarify about our motivation is why we chose wavefronts for water simulation. The initial aim of this study was to animate ocean waters. Unfortunately, the number of studies around this topic go beyond the scope of this thesis (and possibly any thesis) since it is approached by many sciences. One could divide these studies into Accurate Simulations for Science and Engineering and Simulations for Entertainment.

Since we are interest in entertainment applications, we explore only the second category. The two most common approaches are the Eulerian and the Lagrangian, which work with 3-dimensional grids and they differ on their frame of reference. While these are great for simulating behaviours of fluids in closed spaces, they are unfeasible for rendering large scale scenes in real time. A more feasible option is using 2-dimensional grid methods that produce surfaces with the trade-off of not being able to model the separation of water into large bodies. Despite of their limitations, they are usually a good fit for large scale
Figure 1.1: Visual comparison between two systems. On the left we have a 2-Dimensional grid system that is based on simplifications on the Navier Stokes equation. On the right we have the system described in this thesis.

The above reasoning directs us to 2-dimensional techniques. Alas, there is no standard approach in this category since researchers over the years have devised different models that make assumptions or simplifications to the Navier-Stokes equation. Fortunately, during the background reading of this thesis, we discovered a more cohesive group of studies that model ocean water surfaces a combination of wave functions.

While following the work on this area, we found that the state of the art applies these wave functions on the surface by modeling wavefronts and propagating them in space, similarly to light waves in the theory of optics. This is where the wavefront term is derived from in the title of this work. As shown in the following chapters, these approaches have the potential to bring far more rich water animation into real-time entertainment than models that are usually applied. We provide two images from two implementations, one is the wavefront based one described in this thesis and the other is based on O’Brien and Hodgins(1995). See Figure 1.1 for a visual comparison.

1.2 Motivation and Goals

Our motivation derives mainly from the great quality seen in the results of the research by Stefan Jeschke and Chris Wojtan Jeschke and Wojtan(2015). A high detail approach to
calculate the heights of the water surface is used and the waves react with the surrounding environment. The only caveat of their work is that, while an interactive editing is possible for manipulating the final animation result, changes in the simulation create abrupt changes in the rendering. In an interactive entertainment environment, i.e. a videogame, where the user should be able to interact with the surface without re-initializing the simulation but rather influencing its current state, this system would not be appropriate.

One could propose the usage of a secondary system for handling this kind of interactions and then combining the results into a final surface by adding together the two height maps. The main issue with this approach is that each system has its own interpretation of water physics and thus the results are expected to be odd. For example, in the work of James F. O’Brien and Jessica K. Hodgins [O’Brien and Hodgins(1995)], the simulation is expressed in terms of pipes, pressure, pipe flow and volumes, quantities that resemble hydraulics and make characteristics such as the waves amplitude and phase speed very hard to control. A combination of the two techniques is essentially a fusion of two different visual interpretations of a natural phenomenon.

While researching combinations of existing systems and evaluating their plausibility is a very interesting topic, we have decided to solve the problem by creating a novel system that encapsulates the aforementioned interactivity while featuring some of the qualities of the state of the art. More specifically, our work aims for:

1. Interactivity for entertainment purposes.

2. High level height map approximation (as in the state of the art).

3. Environment interactions (as in the state of the art).

4. Scalability of the resolution of the water surface.

5. A reasonable level of rendering quality.

6. Real-time frame rates.
It is important at this point that we disambiguate what we mean by *real-time*. In the context of rendering, the term is used to describe content that is not pre-processed but rather calculated during the display of the results. In our case, it is used to describe two different things; one is the nature of the application as described above, and the second is the nature of the simulation: in contrast with the state of the art, the parameters for rendering the results are not found before rendering the scene. On the contrary, the simulation integration and the rendering routines run in the same loop. As such, goal number six refers to optimizing the modeling and implementation of the system for running in acceptable frame rates, while goal number one refers to running the simulation integration and the scene rendering simultaneously.

### 1.3 Thesis Outline

The text is organized in 7 chapters. Chapter 1 serves as an introduction to the topic by explaining the motivation and goals of this work. In Chapter 2, we provide background information about the 4 desired wave behaviours: Dispersion, Refraction, Reflection and Diffraction and we give our literature review. In Chapter 3, we explain how the wavefront propagation and recording of parameters is performed, together with some elements of linear wave theory, as they are presented in the state of the art. In Chapter 4, we present the already found ideas we used for modeling the environment interactions through wave behaviours and we propose methods for interactions with floating and self-driven bodies. In Chapter 5, we explain how interactivity for entertainment is enabled through a heuristic propagation of a set of wavefront parameters. Chapter 6 is dedicated to the Real-time Rendering component of our implementation and the evaluation and discussion is featured in Chapter 7.
Chapter 2

Background

In this chapter, we provide some background knowledge for the reader and we give both an extensive description of the previous work as well as a literature review that groups the examined papers by the type of technique that they propose.

2.1 Desired Wave Behaviours

A good part of the research on water wave simulation focuses on modeling some desired behaviours. We explain them by giving some intuitive definitions and some annotated real-life images for the novice readers. It is important that these phenomenons are understood since a major part of our interest and the work done so far by other authors is their replication in computer graphics environments.

Dispersion: Dispersion refers to frequency dispersion: many waves travel with different phase speeds. Dispersion in nature can be observed by noticing a variety of wavelengths in waters waves, especially when they break at a the coast line. See Figure 2.2 (right) for an example.

Refraction: Similarly to optics, refraction refers to the change of direction of the wave when moving in different mediums. The mediums here are waters with different depths. Refraction is more effective in shallow waters and less effective in deep waters.
Figure 2.1: Examples of Refraction (blue) and Dispersion (yellow). The blue lines show the direction of the waves and the yellow underline the crests to show the difference of wavelength. Photos taken at Howth, Ireland (from our personal collection).

Figure 2.2: Examples of Reflection (green) and Diffraction (red). The lines show the direction of the waves. Photos taken at the Grand Canal and the Liffey riven in Dublin, Ireland. (from our personal collection).
See Figure 2.1 (left) for an example.

**Reflection**: When a wave hits an obstacle from a sharp angle, it is reflected from that obstacle similarly to how a ray reflects from a flat surface. See the example at Figure 2.2 (left).

**Diffraction**: When a wave hits an obstacle from an oblique angle, it avoids it by traveling around it. See Figure 2.2 for two examples of diffraction.

### 2.2 Literature Review

Most of the literature of interest tries to achieve one or more of the 4 desired wave behaviours. Some researchers approach the challenge by simulating waves as geometrical entities (i.e. rays or piecewise-linear curves) that travel in space, and they integrate the simulation over time or space in discrete steps ([Fournier and Reeves(1986)], [Ts’o and Barsky(1987)], [Gamito and Musgrave(2002)], [Jeschke and Wojtan(2015)]). A second group of authors try to approach these behaviours through spectrum based methods and they operate on the frequency domain by the means of the Fast Fourier Transform ([Mastin et al.(1987)Mastin, Watterberg, and Mareda], [Thon et al.(2000)Thon, Dischler, and Ghazanfarpour], [Loviscach(2002)], [Canabal et al.(2016)Canabal, Miraut, Thuerey, Kim, Portilla, and Otaduy]). Finally, some authors complement their spectrum based approaches by capturing an image of the scene from specific positions and then work in image space ([Loviscach(2002)], [Canabal et al.(2016)Canabal, Miraut, Thuerey, Kim, Portilla, and Otaduy]).

With only a single exception ([Jeschke and Wojtan(2015)]), none of these publications have managed to combine the 4 desired wave behaviours into a single model. Most of them focus on a subset, and some works can be combined together. During our reading, we found that most spectrum-based techniques try to approximate Dispersion. A lot of effort was also put into modeling Refraction in geometrical techniques. Diffraction and Reflection seem to be the least explored.
In addition to these methods, we have also reviewed some 2-dimensional approaches that are of interest, not for their simulation model but for their rendering techniques ([Schneider and Westermann(2001)]) and some nice ideas about additional effects. ([O’brien and Hodgins(1995)]).

2.3 Previous Work

M. Kass and G. Miller proposed a fluid dynamics computational model that simplifies the Navier Stokes Equations on the basis of three assumptions in [Kass and Miller(1990)]. While the results cannot be used for fluid studies, they were convincing for animating shallow water. The most constraining assumption is that of the water surface being a height field; the wave behaviours are limited and effects such as breaking waves cannot be represented. Their work provides a great insight on their derivation of a simplified wave equation, as well as a way to stabilize the fluids using a normalized damping coefficient.

Splashing fluids have been studied in [O’Brien and Hodgins(1995)], by O’Brien and Hodgins. The model they propose divides the spatial area in water tanks that hold volume information and these tanks are 8-way connected via virtual pipes that hold flow information. A surface is derived from this volume model and external forces applied to the surface are mapped to the volume model. Their work also includes a spray model.

In [Schneider and Westermann(2001)], Schneider and Westermann calculate the values of a height function as the superposition of the well known turbulence function and a NURBS surface, created from the points of the grid. The normal vectors are calculated via the approximation of the partial derivatives.

In [Mastin et al.(1987)Mastin, Watterberg, and Mareda], Mastin et al. render and animate waves via image synthesis. Starting from a white noise image, they produce its frequency domain through the Fast Fourier Transform (or FFT for short). Then, the magnitude of the frequency domain is filtered with a spreading factor and the result is converted back to the spatial domain through the inverse FFT. The spreading factor
depends on the input parameters which are the wind velocity and wind angle. The surface
is animated by shifting the phase on the spatial domain, in two ways; one that is mentioned
as ad hoc and a second that calculates a phase velocity. The rendering is achieved with
a ray-tracing algorithm of their own. The algorithm is accelerated by using axis aligned
bounding boxes in an octree hierarchy to approximate the surface.

In [Fournier and Reeves(1986)], Fournier et al. create the so-called wave trains where
they model the motion of each particle as a trochoid. They calculate the phase with the
depth as a parameter, capturing the transition between deep waters to shallow floors and
ultimately the break of the wave at the shore. Their method employs a grid per wave
train. The fidelity of the grid is such that captures the details of the ocean floor and
the final height on the grid is computed as the summation of the disturbance caused by
each wave train. The data for the rest of the points are bi-linearly interpolated using the
nearest neighbours. They render the surface using a ray tracer.

In [Ts’o and Barsky(1987)], Ts’o and Barsky also model wave trains, with the signifi-
cant difference that they perform the so-called wave tracing. Given the direction vector
of each wave train, which they call wave orthogonal, they approximate the path of the
wave train by calculating the direction of refraction, using Snell’s law. The obstacles that
they check against for refraction are the height contours of the ocean bed. Each such ray,
contributes to the height of the nearest grid point, in a linear manner since they base
their work on Airy wave theory [Airy(1841)]. The heights are treated as control points
for a 3-dimensional Beta spline and they use the tension shape parameter to adjust the
sharpness of the crests. They render the surface using a ray tracing renderer in combina-
tion with textures and the calculation of the Frensel coefficient. Their solution seems to
suffer from appropriate resolution values for modeling the ocean bed height contours as
well as for the appropriate number of wave trains.

In [Thon et al.(2000)Thon, Dischler, and Ghazanfarpour], Thon et al combine the
principle of superposition and Airy’s wave theory [Airy(1841)] with a spectrum based
approach. The change the wave profile from a sinusoidal function to a trochoid one and
they pose the question of how to select the appropriate amplitude, frequencies and phase. Similarly to Mastin et al. in [Mastin et al. (1987) Mastin, Watterberg, and Mareda], they use the Pierson-Moskowitz filter ([Pierson and Moskowitz (1964)]), but instead of smoothing a noise image and taking the IFFT, they use it for sampling the ocean spectrum. From the constructed spectrum, they keep the most representative values, based on amplitude, and use Perlin’s Turbulance function [Perlin (1985)], as additive noise to the final height estimation. Their approach has the advantage against the work of Mastin et al. [Mastin et al. (1987) Mastin, Watterberg, and Mareda], in that it is not bound by the texture resolution; instead of using the IFFT to transform from the frequency spectrum to a height field, they use the frequency spectrum to sample the inputs to the trochoid functions. Thus their technique can be used for an infinite surface. They also provide an interesting solution to aliasing effects that applies to all methods that are based on the superposition principle; for surface points away from the screen, they neglect or reduce the amplitude of trochoid functions that have high frequencies for that distance.

In [Loviscach (2002)], J. Loviscach presents a spectrum model that uses a convolution kernel to reproduce the rings that are formulated from raindrops as well as Kelvin wakes from swimming bodies. While his work can handle dynamic events such as a body penetrating the surface, it is unable to demonstrate reflections, refraction or diffraction. Additionally, he proposes to disturb the height field close to the boundaries so that reflection effects would be avoided. For Kelvin wakes, he captures binary images of the swimming objects from a top-down view and uses consecutive frames to calculate them.

In [Gamito and Musgrave (2002)], M. N. Gamito and F. Kenton Musgrave provide a more accurate model for wave refraction that the one proposed by Ts’o and Barsky in [Ts’o and Barsky (1987)]. Instead of the Snell’s law, they base their computation on Fermat’s Principle of the Shortest Optical Path [Born and Wolf (1980)]. Their analysis leads to two Ordinary Differential Equations, where the independent variable is space and time respectively. Similarly to Ts’o and Barsky, they model wave trains as wave rays with the their addition of wavefronts, linear segments formed by two consecutive rays with the
same phase. They spawn a new ray whenever a wavefront surpasses a threshold that is
dependent of the size of the integration step. They chose to integrate over the spatial
independent variable due to scheme for collision detection with the water surface grid;
they triangulate the endpoints of the rays during the current and previous integration
step and check for grid points inside the triangles. Then, they interpolate phase and
amplitude values via barycentric coordinates. They update the amplitude at each time
step proportionally to the depth at the position of the endpoint of the wave ray and
inversely to the expansion of the wavefront arc. An interesting part of their work is the
usage of a trochoid geometrical model for the wave function which they improve with
convolutional noise for more realistic effects.

In [Jeschke and Wojtan(2015)], Jeschke and Wojtan manage to get the 4 desired
wave behaviours dispersion, refraction, reflection and diffraction in one model. They
model wavefronts as piece-wise linear curves that propagate through the water surface
and physically interact with obstacles. The heights are calculated with the linear wave
theory equations. Additionally, they interpolate the wavefront parameters (phase function
and amplitude) instead of the final heights, as a way to improve the sampling for the final
result. Their simulation and their rendering are done in two separate steps.

In [Canabal et al.(2016)Canabal, Miraut, Thuerey, Kim, Portilla, and Otaduy], Can-
abal et. al. use a dispersion convolution kernel, defined from the equations of linear wave
theory, to calculate the vertical acceleration of each surface point. Their main contri-
bution is the refinement of this convolution technique for usage in the spatial instead of
the frequency domain, as normally done in some spectrum based techniques. They also
handle reflection of waves by modulating the kernel; they compute the shadow of static
and dynamic obstacles, that is projected on the surface, and neglect the points that are
covered by the shadow in the acceleration computation.
Chapter 3

Wavefront Propagation

In this chapter, the wavefront propagation, based on the works of S. Jeschke and C. Wotjan [Jeschke and Wojtan(2015)], Gamito and Musgrave [Gamito and Musgrave(2002)], Ts’o and Barsky [Ts’o and Barsky(1987)] and Fournier et al [Fournier and Reeves(1986)] as well as A. Fournier and W. T. Reeves [Fournier and Reeves(1986)] is described.

We follow the geometrical, time integration approach of Ts’o and Barsky [Ts’o and Barsky(1987)], Gamito and Musgrave [Gamito and Musgrave(2002)] and S. Jesche and C. Wotjan [Jeschke and Wojtan(2015)], approximating a wavefront as a piece-wise linear curve (as in [Jeschke and Wojtan(2015)]) whose vertices are traveling on the surface. The main idea is to record the trail of a piece-wise linear curve over time by dividing the available space into a regular grid and storing the accumulated simulation time and the state of the curve, on a point of the grid, when the curve piece passes over it.

Section 3.1 describes the mathematical model that is used for calculating physical quantities of the system. In Section 3.2, the integration method for acquiring solutions from the aforementioned model is presented. Section 3.3, 3.4 and 3.5 are technical and they explain how we record the information produced in each integration step in an efficient manner.
3.1 Elements of Linear Wave Theory

The fundamental concepts of the linear wave theory are presented here. The symbols and some simplifications are inherited from [Jeschke and Wojtan(2015)] since it is the starting point of this work.

Linear wave theory models the vertical motion of a point on the water surface as the superposition of multiple sinusoidal functions

$$\eta(\vec{x}, t) = \eta_0 + \sum_{i=1}^{N} a_i \sin(\omega_i(\phi_i(\vec{x}) - t))$$  \hspace{1cm} (3.1)

where $\vec{x}$ is the position of the particle on the surface, $t$ is time, $\eta_0$ is an initial height and $a_i$, $\omega_i$ and $\phi_i$ are the amplitude, angular frequency and phase function of the $i$th wave that has reached this position of the surface. $N$ indicates the total number of waves.

It is obvious that each wave is assumed to be propagating as a simple sinusoidal function without damping on the amplitude or the frequency. The value of $\eta_0$ represents the initial state of the surface and for initially stable surfaces it can be set to zero. The main interest is how the amplitude, angular frequency and phase function of each wave are derived.

3.1.1 Angular Frequency and Phase Speed

The angular frequency dictates how fast the point will move vertically over time and it is calculated as

$$\omega = \sqrt{(g k + \frac{\sigma}{\rho} k^3) \tanh(kh)}$$  \hspace{1cm} (3.2)

where $g$ is the gravitational constant, $k$ is the wave number, $\sigma$ is the surface tension, $\rho$ is the water density and $h$ is the depth at that point.

By observing the plots in Figure 3.1 it is evident that for a constant wave number, the angular frequency decreases as it reaches the coast. Also, for a big depth, the angular frequency is proportional to the wave number.
Figure 3.1: Depth and wavenum plots against angular frequency, $g=9.8$, $\sigma = 1$, $\rho = 1$
The phase speed is the speed by which the wave propagates through the medium

\[ c = \frac{\omega}{k} = \sqrt{\left(\frac{g}{k} + \frac{\sigma k}{\rho}\right) \tanh(kh)} \]  

(3.3)

Figure 3.2: Depth and wavenum plots against phase speed, \( g=9.8 \), \( \sigma = 1 \), \( \rho = 1 \)

and we can see that it is directly related to the angular frequency. Thus, the previous observations also hold here: waves slow down as they reach the coast.

### 3.1.2 Amplitude, Energy Density and Group Speed

The amplitude is related to the energy of the wavefront. The energy density is calculated as

\[ D = \frac{1}{2}(\rho g + \sigma k^2)a^2 \]  

(3.4)
where $D$ is the energy density, $a$ the amplitude and $\rho$, $g$, $\sigma$ and $k$ as in equations 3.2 and 3.3.

The energy is calculated as

$$E = \int Ddl$$

where $l$ is the length of the wavefront.

Group speed is the speed by which the energy of the wave propagates and it can be different than the phase speed

$$c_g = \frac{d\omega}{dk}$$

The conservation of energy and energy transportation is expressed by the following equation

$$\frac{d}{ds} c_g E = 0$$

which implies that the energy flux $c_g E$ must be the same between two consecutive points in the $s$ direction.

### 3.1.3 Phase Function

The phase function $\phi$ ensures that the initial conditions are met and that the wave travels with the phase speed. This is expressed with the *eikonal equation*

$$|\nabla \phi| = \frac{1}{c}$$

where $\nabla \phi$ is the direction of wave travel.
3.2 Wavefront Propagation

A full time-variant integration solution for the above equations is presented in [Jeschke and Wojtan(2015)]. We follow their derivation for the propagation of the wavefronts.

The initial condition for each wavefront is an ordered set of \( N \) vertices who carry information such as position and velocity. Each vertex is advanced as

\[
\vec{x}_i = \vec{u}_i \Delta t
\]  

(3.9)

and the velocity is calculated as

\[
\vec{u}_{i+1} = \frac{\vec{u}_i}{|\vec{u}_i|} c_{\vec{x}_i}
\]  

(3.10)

The quantity \( c_{\vec{x}_i} \) is the phase speed of the wavefront at the position \( \vec{x}_i \). Since the wavefront is assumed to have the same wave number throughout the simulation, Equation 3.3 varies only with depth values.

In addition to the vertices, \( N - 1 \) edges are created to store the energy density, group speed, wave number, and amplitude of the edge. The first three quantities make little sense but they are useful for the calculation of the amplitude. To be more specific, the amplitude is calculated by solving Equation 3.4 for \( a \)

\[
a = \sqrt{\frac{2D}{\rho g + \sigma k^2}}
\]  

(3.11)

and \( D \) is calculated by discretizing Equation 3.7

\[
E_{c_{q}}^{old} \cdot c_{q}^{old} - E_{c_{q}}^{new} \cdot c_{q}^{new} = 0
\]  

(3.12)

and by approximating energy as
\[ E = DL \] (3.13)

where \( L \) is the length of the wavefront segment, Equation 3.12 becomes

\[ D^\text{old} L^\text{old} c^\text{old}_q - D^\text{new} L^\text{new} c^\text{new}_q = 0 \] (3.14)

By setting \( D = D^\text{new} \), the amplitude can be calculated. The values of \( L^\text{old} \) and \( L^\text{new} \) are the length of the edge on the previous and current step and they are calculated from distance of the vertices that form the edge. Since \( D^\text{old} \) and \( c^\text{old}_q \) can be saved as the previous values of \( D^\text{new} \) and \( c^\text{new}_q \) and assuming that an initial value for energy, energy density or amplitude is given, the only thing that is left to be calculated is the current group speed. By using finite differences to approximate Equation 3.6 we have

\[ c_g(k, h) \approx \frac{\omega(k + \Delta k, h) - \omega(k, h)}{\Delta k} \] (3.15)

where \( \Delta k \) is a very small number, \( 10^{-6} \) for instance. Please note that \( k \) here is not the same wave number as the one used for the wavefront propagation. Linear wave theory mentions that the angular frequency must be the same for the whole wavefront, as thus when calculating the group speed, the wave number must vary with depth. Jeschke and Wojtan in [Jeschke and Wojtan(2015)] approximate the value of \( k \) with an iterative scheme of the form

\[ k := \frac{\omega}{c(k, h)} \] (3.16)

where \( \omega \) is the constant wavefront wave number and the phase speed is calculated by Equation 3.3.

Figure 3.3 showcases the amplitude, angular frequency and arrival time values of the sampling points of the grid, after a circular wavefront has propagated. The depth values are also given for reference. The amplitude decreases over time in this case since the length of the curve increases. If the curve length was shrinking, the amplitude would
Figure 3.3: Propagation of a circular wavefront. The yellow map represents amplitude, the blue map angular frequency, the red map arrival time and the green map depth. Values are mapped to color intensities.

grow over time. Angular frequency is dependent of the depth for shallow cases and it is independent otherwise.

3.3 Resolution Refinement of the Wavefront Curve

Since the curve vertices can travel in non-parallel directions, the curve segments can grow in length which is a bad representation of the curve. Analogously, a curve segment may become very small, causing unnecessary collision detection checks. For this reason, a segment is divided when its length goes over a maximum length and it is omitted when it goes under a minimum length, as expressed in Jeschke and Wojtan(2015). The position and velocity of the new vertex is the average of its two neighbours and the values of the newly inserted edge are copied from the original edge. Figure 3.4 illustrates a division operation.
Figure 3.4: The length of the segment $v_a, v_b$ surpasses the maximum limit and thus the midpoint vertex $v_c$ is introduced.

### 3.3.1 Partitioning of the Wavefront Curve

The wavefront vertices may have variant directions due to refraction as well as reflection and diffraction. This characteristic may cause self-overlapping curves which feel uncontrollable and may cause visual artifacts. Additionally, it is easier to control a scene by having a notion of direction for each wavefront. For this reason, each wavefront is subdivided into smaller wavefronts when the vertices do not agree on a general direction.

The criterion for our operation is the difference in direction between the vertices and the group direction of the wavefront; after the calculation of the group direction as the average of the first and last vertices of the curve, each inner vertex’s direction is compared against it. If the in-between angle is over a threshold, then the vertex is marked as negative, otherwise as positive. When all vertices have been marked, the algorithm searches for segments with more than 3 consecutive vertices of the same marking. Each of these segments forms a separate wavefront curve. Segments that have less than 3 vertices are erased from the curve, creating a gap. An example is shown in Figure 3.5.
The algorithm for partitioning the wavefront is the following:

**Result:** Wavefronts set $W_{n+1}$ by partitioning the wavefronts set $W_n$

```plaintext
foreach Wavefront $w$ in $W_n$ do
    $direction_w = \frac{\vec{w}_{\text{first}} + \vec{w}_{\text{last}}}{2}$;
    foreach Vertex $v$ in $w$ do
        if $\vec{v}_{\text{direction}} \cdot direction_w >$ threshold then
            $v_{\text{sign}} =$ positive
        else
            $v_{\text{sign}} =$ negative
        end
    end
    foreach Sign-consecutive subset $w_{cons}$ of $w$ do
        $W_{n+1} = W_{n+1} \cup w_{cons}$
    end
end
```

**Algorithm 1:** partitioning algorithm for wavefront curve
3.4 Detection of Grid Points Coverage by a Wave-front Curve

In [Ts’o and Barsky(1987)], Ts’o and Barsky detect collisions between wave rays and a set of sample points. This cannot be applied in our case since we propagate a curve. In [Gamito and Musgrave(2002)], Gamito and Musgrave create a triangulation between past and current positions of the vertices of each segment of the curve. This is possible because the parameters for amplitude estimation are stored in a per vertex basis. Since we have followed the model by Jeschke and Wojtan [Jeschke and Wojtan(2015)], the energy parameters are directly related to the edges instead of the vertices and thus a triangulation is not a good option. A trapezoid is formulated instead, from the 4 (2 current and 2 past) vertices of the curve segment.

The coverage of a point by a curve is described in detail. For simplicity, we refer to this action as collision detection. The design of the system aims for a robust approach that operates on the minimum possible set of grid points.

When the wavefront is propagating, we need to find the grid points that the wavefront passes over during a time step. To achieve this, a double buffer system for the vertices of the piecewise-linear curve is used. By breaking down the procedure into finding the collisions between each individual linear segment and the nearest grid points, the collision
Collision detection for segment 1

Collision detection for segment 2

Figure 3.6: Collision detection for two neighbouring segments

\[ q_2 \cdot e - s \cdot q_1 \]

trivial case

\[ q' \cdot q \]

degenerate case 1

\[ q' \cdot q \]

degenerate case 2

Figure 3.7: Three cases of the Left predicate. In the degenerate cases, the directed perpendicular normal is calculated for the projection \( q' \) of \( q \), and the decision is based on the location of \( q' \).

detection predicate is formed as follows:

Result: decision if a propagating linear segment \( v_a, v_b \) covers a grid point \( q \)

if \( \text{Left}(q, v_{a,\text{old}}, v_{b,\text{old}}) \) and \( \text{Left}(q, v_{b,\text{old}}, v_{b,\text{new}}) \) and \( \text{Left}(q, v_{b,\text{new}}, v_{a,\text{new}}) \) and \( \text{Left}(q, v_{a,\text{new}}, v_{a,\text{old}}) \)

or

\( \text{Right}(q, v_{a,\text{old}}, v_{b,\text{old}}) \) and \( \text{Right}(q, v_{b,\text{old}}, v_{b,\text{new}}) \) and \( \text{Right}(q, v_{b,\text{new}}, v_{a,\text{new}}) \) and

\( \text{Right}(q, v_{a,\text{new}}, v_{a,\text{old}}) \) then

| point \( q \) is covered by \( v_a, v_b \)

else

| point \( q \) is not covered by \( v_a, v_b \)

end

Algorithm 2: collision detection between propagating piecewise-linear segment and a grid point

An illustration of the method can be seen in Figure 3.6. The predicate Left and Right are defined in Algorithms 3 and 5 respectively.

If \( \text{sign} \) is positive, the point is on the right side of the line, and if it’s negative, it’s on the left side. If it’s zero, it resides on the line and a special handling takes place. The handling
Result: decision if a point \( q \) is on the left side of a linear segment \( s, e \)
if \( e==s \) then
  \( q \) is both on the left and the right side of \( s, e \)
else
  \( \text{sign} = (e_x - s_x)(q_y - s_y) - (e_y - s_y)(q_x - s_x); \)
  if \( \text{sign} < 0 \) then
    \( q \) is on the left side of by \( s, e \)
  else
    if \( \text{sign} == 0 \) then
      \( \text{normal} = \text{DirectedPerpendicular}(s, e); \)
      \( q' = q + \text{normal}; \)
      \( \text{sign} = (e_x - s_x)(q'_y - s_y) - (e_y - s_y)(q'_x - s_x); \)
      if \( \text{sign} < 0 \) then
        \( q \) is on the left side of \( s, e \)
      else
        \( q \) is not on the left side of \( s, e \)
    end
    else
      \( q \) is not on the left side of \( s, e \)
  end
end

Algorithm 3: Left predicate

routine creates a unit vector that is perpendicular to the line and projects the query point towards that direction. This is necessary to disambiguate between neighbouring segments that would otherwise both classify the query point as either an inside or outside point.

The prefix Directed of the vector calculation routine is derived from the fact that the direction of the perpendicular vector must be the same when the order of parameters is changed. In any other case, the handling would fail since the normal would project the query point always towards the inner side or outer side of the line. This kind of handling
is shown in Figure 3.7. The routine is defined as follows:

**Result:** a directed perpendicular vector $\text{perp}$ of the line defined by $s, e$

\[
\begin{align*}
\text{if } s_x &= e_x \text{ then} & & \\
& & & switch = s_y > e_y \\
\text{else} & & & switch = s_x > e_x \\
\end{align*}
\]

\[
\begin{align*}
\text{if } switch \text{ then} & & & \\
& & & normal = \frac{(e-s)}{|e-s|} \\
\text{else} & & & normal = \frac{(s-e)}{|s-e|} \\
\end{align*}
\]

\[
\begin{align*}
\text{perp}_x &= normal_y; \\
\text{perp}_y &= -normal_x;
\end{align*}
\]

**Algorithm 4:** calculation of directed perpendicular vector of a line

The Right predicate is defined as the Left predicate by swapping the start and end vectors:

**Result:** decision if a point $q$ is on the right side of a linear segment $s, e$

\[
\begin{align*}
\text{if } e==s \text{ then} & & \\
& & q \text{ is both on the left and the right side of } s, e \\
\text{else} & & \text{return } Left(e,s,q); \quad \text{//the first two arguments are swapped}
\end{align*}
\]

**Algorithm 5:** Right predicate

Now that the grid point to wavefront curve collision detection method is defined, we need to decide the candidate grid points for each collision check. It is always valid to check all the grid points but this adds an unnecessary, linear to the number of sampling points, overhead on the system. By agreeing on the usage of a regular quadrilateral grid,
the calculation of the search range becomes trivial:

**Result:** integer range vectors $s$ and $e$ for propagating curve segment $v_a, v_b$

\[
\begin{align*}
AABB_{x}^{\text{max}} &= \max_x(v_a^{\text{old}}, v_a^{\text{new}}, v_b^{\text{old}}, v_b^{\text{new}}); \\
AABB_{y}^{\text{max}} &= \max_y(v_a^{\text{old}}, v_a^{\text{new}}, v_b^{\text{old}}, v_b^{\text{new}}); \\
AABB_{x}^{\text{min}} &= \min_x(v_a^{\text{old}}, v_a^{\text{new}}, v_b^{\text{old}}, v_b^{\text{new}}); \\
AABB_{y}^{\text{min}} &= \min_y(v_a^{\text{old}}, v_a^{\text{new}}, v_b^{\text{old}}, v_b^{\text{new}}); \\
s &= \text{floor}((AABB_{x}^{\text{min}}_{\text{offset}}) - (1, 1)); \\
e &= \text{ceil}((AABB_{y}^{\text{max}}_{\text{offset}}) + (1, 1));
\end{align*}
\]

**Algorithm 6:** calculation of search space limits for a propagating piecewise-linear curve segment

An example of Algorithm 6 is shown in Figure 3.8.

### 3.5 Recording the Wavefront Parameters on the Grid

Points

The final step of the simulation is to create the appropriate record for each collision

detection event, and store it on the covered grid point. Since the collision detection

is performed on a linear segment level, we can interpolate the values found on the edges of

the segment.
Firstly, the values that need to be interpolated must be specified. From Equation 3.1, it is evident that we need to store an amplitude, an angular frequency, and a phase function. The angular frequency can be derived from the wave number and the angular phase speed according to Equation 3.3. Recall that in Equation 3.10, the current velocity is calculated as a normalized vector, scaled by the phase speed, and thus the latter can be derived as the length of the first. Therefore, we can interpolate the velocity and derive the phase speed and the angular frequency. Interpolating the velocity also offers the advantage of storing a direction vector for each wave so a flow field for the surface is possible to be constructed.

By assuming that the initial state of the surface is flat, we can simplify the phase function \( \phi \) to the arrival time of the wavefront at that point. We assign the arrival times for the previous state vertices as the accumulated simulation time minus the magnitude of the previous time step and the values for the current state vertices as the accumulated simulation time, and we interpolate the arrival time, assuming that the phase speed of the nodes remains the same during the integration steps.

The amplitude \( a \) can also be interpolated similarly to the above. Since it is stored on the edges of the piecewise-linear curve and not on the vertices, we can either assign directly the value found on the edge to the vertex or interpolate between the two neighbouring edges. See Figure 3.9 for an illustration of how the interpolated values are organised on the wavefront curve.

Another consideration is the interpolation scheme that should be used. Some generic solutions include the Inverse Distance Weighted and Nearest Neighbour Interpolation. The latter requires the construction of a Voronoi Diagram, so IDW is preferred for simplicity. IDW is defined as follows:

\[
   u(x) = \begin{cases} 
   \frac{\sum_{i=1}^{N} \frac{w(x)u_i}{w_i(x)}}{\sum_{i=1}^{N} w_i(x)}, & \text{if } d(x, x_i) \neq 0 \text{ for all } i \\ 
   u_i, & \text{if } d(x, x_i) = 0 \text{ for some } i 
   \end{cases} 
\]

(3.17)
Figure 3.9: A visual representation of how values are stored on a wavefront curve for interpolating when a collision is detected. $a_{i-j}$ denotes the average between $a_i$ and $a_j$. In addition to the values shown, some values that do not vary over the curve, such as the wavefront ID, are stored at the covered grid point.

$$w_i(x) = \frac{1}{d(x, x_i)^p} \tag{3.18}$$

where $d(x, x_i)$ denotes the distance between the points $x$ and $x_i$, and $p$ is an integer.

With this information, a new record can be formed and stored in a record list of the grid point. Due to the resolution refinement for the curve, a grid point can be unintentionally covered by a wavefront multiple times. For this reason, we also store an ID of the wavefront and perform a look-up to avoid duplicate entries.
The wavefront curve propagation scheme was explained in the previous chapter and we can now move on to how the wavefront curves interact with their surrounding environment. We can divide the interactions into two categories, the interactions with static bodies and the interaction with dynamic bodies.

With the term static we emphasize the fact that the body properties that we care about do not change over time. More specifically, we refer to the ocean bed and bodies that cannot be moved in the scene, such as walls or large rocks. The interactions with the ocean bed are modeled through Refraction, as explained in Section 4.1, while the interactions with the rest is done via Reflection and Diffraction, as described in Section 4.2.

Dynamic bodies are bodies who can be moved in the scene, either by energy sources that do not belong in the system or by the waves. These interactions are described in Sections 4.4 and 4.5.
Figure 4.1: Refraction of a wave front: The original direction is the crossing diagonal, but the vertices of the curve (cyan) change direction due to change of depth and come closer to the shoreline

4.1 Refraction of Waves

In [Gamito and Musgrave(2002)], Gamito and Musgrave find the direction of the vertices when a refraction happens, using Fermat’s Principle of the Shortest Path. While this method is accurate, it is computationally and implementation-wise expensive since it requires the solution of an ODE. We prefer to use the Snell’s law as proposed by Ts’o and Barsky [Ts’o and Barsky(1987)] for calculating the new direction

\[
\frac{\sin i}{\sin r} = \frac{c_i}{c_r} \tag{4.1}
\]

where \(\sin i\) and \(\sin r\) are the sines of the angles between the ocean bed normal and the incident and refracted direction vectors of the vertex, respectively. Similarly, \(c_i\) is the phase speed of the incident vertex and \(c_r\) the phase speed of the refracted vertex. An example is shown in Figure 4.1. The ocean bed normal has to be calculated which is a new problem by itself.

4.1.1 Estimating the Water Bed Normals

For the needs of refraction, a set of normal vectors has to be calculated. Each of these directional vectors points from an origin position towards a neighbouring position of higher depth. We calculate a normal vector for each point of the height map that represents the
ocean bed.

Using the original heights for calculating the normal vectors was our first approach but this created odd re-directions of the curve. We turned into contouring the height map to achieve less unexpected behaviours. The method for calculating the contoured heights is given in Algorithm 7.

**Result:** Each height of the set $H$ is marked with a contour index $\in [0, N]$  

$$H_s = \text{sort}(H);$$

for $i = 0; i < \text{size}(H); i+ = \frac{\text{size}(H)}{N}$ do

$\text{Limits.add}(H_s^i)$

end

foreach height $h$ in $H$ do

$\text{find entry index } k \text{ where } h < \text{Limits}_k;$

$\text{store } k \text{ as contour index of } h;$

end

Algorithm 7: Calculating the contour indices

Ts’o and Barksy [Ts’o and Barsky(1987)] create contour lines from the height information and check for collisions against them. We have tried a similar approach where the refraction takes place only when the previous and current positions of a vertex are located in areas with different contour index. The results were discouraging since small cavities on the contour lines can change the direction of the vertex until the next height level is met.

The solution that worked in our case was to check for redirection of the vertices in each time step, independently of the difference in the contour indices. To calculate the normal vectors, we find the closest point with a smaller contour index and direct the normal towards that direction. Since refraction happens only in shallow cases, we refine the contours to shape only in areas of low depth by adding a height threshold to the working set. An example of the final solution can be seen in Figure 4.2.

The above approach assumes that the ocean bed height map has no noise. For arbitrary input, we can pre-process the data by using a Gaussian or a Median or a Mean or any other kind of filter for smoothing the surface.
Reflection and Diffraction of Waves

Reflection and Diffraction are two wave behaviours that can be modeled together. According to [Jeschke and Wojtan (2015)], this is possible by handling differently a collision of the wavefront with an obstacle, depending on the incident angle of the wavefront curve and the normal vector of the obstacle’s surface. We followed the same approach, since after some experimentation with a more accurate method of our own, it was obvious that the first was more elegant and had a better speed performance.

Each model is supplied by an oriented curve together with the appropriate normals, tangents and bitangents, that represents a convex projection of the mesh on the water surface plane (see Subsection 4.2.1). For each vertex of the wavefront curve, we check if it has crossed an edge of the object’s curve. The intersection point is calculated and the angle between the direction of the vertex and the normal of the edge is used to determine if the collision response will be a reflection or a diffraction. When the angle is acute, reflection is selected and the vertex is positioned according to the laws of reflection. When the angle is oblique, diffraction takes place and the vertex is positioned very close to the intersection point, while its moving direction changes to either the tangent or bitangent vector of the
edge. Deciding which depends on the angle between the incident and the tangent vector. If it’s less than 90 degrees, the bitangent is preferred and vice versa. An algorithmic description is given in Algorithm 8. A result of using this method is shown in Figure 4.3.

The above approach has the drawback that collisions are checked only on a per vertex base. This assumes that the wavefront resolution is high enough to not allow degenerate cases such as the one illustrated in Figure 4.4. We have attempted a more accurate approach that compares edges against edges but we found that the implementation had multiple corner cases and the performance dropped dramatically.

Additionally, for practical reasons, we refine the wavefront curve resolution according to the resolution of the grid, since we check for collisions between them frequently. This creates an undesired connection between the robustness of the collision detection for Reflection and Diffraction and the time performance of the collision detection for recording the wavefront parameters over space. It is worth mentioning that the authors in Jeschke and Wojtan (2015) avoid this problem by using a triangulation over space that is more detailed in points of interest such as the object boundaries.
Figure 4.4: A degenerate case of the reflection and diffraction collision detection. Note that the size of the obstacle (cyan) is close to the minimum length of the wavefront edges (red, dashed).

**Result:** A collision between object curve edge $c$ and wavefront curve vertex $v$ is detected and handled

```plaintext
if $\text{Left}(c_s,c_e,v^{pos})$ and $\text{Right}(c_s,c_e,v^{pos})$ and $\text{Left}(v^{pos},v_{old}^{pos},c_s)$ and $\text{Right}(v^{pos},v_{old}^{pos},c_e)$
then
    incident = $-v$ direction;
    angle = dot(incident, normal);
    intersection = $\text{CalculateIntersectionPoint}(c_s,c_e,v_{old}^{pos},v^{pos})$;
    if angle > threshold then
        $v^{dir}$ = reflect(normal, incident);
    else
        if dot(incident, tangent) > cos(90) then
            $v^{dir}$ = tangent
        else
            $v^{dir}$ = bitangent
        end
    end
    $v_{pos}$ = intersection;
else
    No collision
end
```

**Algorithm 8:** Wavefront vertex to obstacle curve collision detection and handling
4.2.1 Producing the Body Curves

In [Lazo et al. (2013) Lazo, Bauza, Boroni, and Clausse], the authors classify the triangular faces of a model that floats on a water surface, according to their immersion level: wet, totally dry, and semi-wet. They produce the normal vectors for the semi-wet triangles. We follow the same principle and produce a closed curve that is essentially the border between the object and the plane that represents the height of the water surface. To differentiate it from the wavefront curve, we name it *body curve*.

Given the mesh of the model representing an obstacle, we produce the 3D convex hull. For each edge of the convex hull, we calculate the intersection point with the surface plane. By assuming that the water surface plane is the x-z plane, we can remove the second coordinate, and store the projection to a new set. From the produced set, we remove duplicates and an oriented path is calculated by the convex hull algorithm described in [Dawson-Howe (2014)] (Algorithm 8.3). The constructed curve will be used for the collision detection check against a wavefront. Since the collision detection time complexity is linear to the number of edges, we can always improve the performance by reducing the curve’s level of detail.

4.3 Wavefronts Generated from Self-driven Bodies

By allowing the simulation to run together with the rendering, we are able to create waves from objects that flow on the surface but are driven by energy sources other than the water waves. For convenience we call them *self-driven bodies*. The basic idea is to
create wavefront curves that their shape and motion is derived from the geometry and
dynamics of the body.

The first problem that needs to be solved is deciding on the shape of the produced
wavefront curves. A sensible approximation is to copy the shape of the body curve. If
the body is dropped on the surface, we can propagate the curve using the normals as
direction vectors. If the motion is not vertical, we need to select a subset of the curve
since we will have only waves traveling in the same horizontal direction as the body. See
Figure 4.6 for the two aforementioned examples, and Algorithm 10 for how we obtain the
curve in the second case.

The model for rendering this special case of wavefronts is different. We want to
render a big wave that moves along a direction, followed by waves with gradually smaller
amplitude. For this purpose we use a linear curve that defines the magnitude of a wave’s
amplitude over time. The height equation (Equation 3.1) now becomes

\[
\eta(\vec{x}, t) = \eta_0 + \sum_{i=1}^{N} a_i w_i(t) \sin(\omega_i(\phi_i(\vec{x}) - t))
\]  

(4.2)

where

\[
w(t) = \begin{cases} 
\text{lerp}(a_{max}, a_{min}, (t - t_{start})/(t_{end} - t_{start})), & \text{if } t \in [t_{start}, t_{end}] \\
0, & \text{otherwise}
\end{cases}
\]

(4.3)

and \text{lerp} being the linear interpolation function for scalar values.
An important aspect of this behaviour is the follow-through waves: when the direction of the linear momentum changes, or its magnitude decreases, the movement of the wavefront is not controlled by the body anymore. Instead, it propagates with the last updated information and a new wavefront is generated from the new linear momentum if its magnitude is not decreasing. See Algorithm 9 for a more detailed description.

**Result:** Wavefronts related to the body are handled according to its linear momentum $P$ at the current step $t$ and at the previous step $t-1$

if $|P_t| < |P_{t-1}|$ or $\frac{P_t}{|P_t|} \neq \frac{P_{t-1}}{|P_{t-1}|}$ then

if $|P_t| < |P_{t-1}|$ then
- release current wavefronts to the system
else
- release current wavefronts, generate new ones and attach them to the body

end

else

if $|P_t| > |P_{t-1}|$ and there are no wavefronts attached to the body then
- generate new wavefronts and attach them to the body
else
- update wavefronts velocity and position according to the velocity of the body

end

end

**Algorithm 9:** Handling of wavefronts from self-driven bodies

**Result:** Generate a wavefront $W$ from the linear momentum $P$ and the curve $c$ of the body

for vertex $v = c_{\text{start}}; v \neq c_{\text{end}}; v = v_{\text{next}}$ do

if $v_{\text{normal}} \cdot \frac{P}{|P|} > 0$ then
- add vertex $v$ to the vertices of $W$
- increase the position of $v$ in $W$ by some small length in the direction of $v_{\text{normal}}$

end

end

**Algorithm 10:** Generation of wavefronts from self-driven bodies

### 4.4 Flow of Bodies

During the wavefront propagation, we store velocity vectors on the grid points. These velocities can be used to form a simple 2-dimensional force field that acts on a body curve:
By using the following formulation, proposed by the authors in [Lazo et al.(2013)Lazo, Bauza, Boroni, and Clausse]

\[ \vec{F} = k(\vec{v} \cdot \vec{n})\vec{n} \]  

(4.4)

where \( \vec{F} \) is the force, \( k \) an arbitrary constant, \( \vec{v} \) velocity of the flow at the coordinates of the vertex and \( \vec{n} \) the normal of the vertex, we can calculate a force for each vertex of the oriented curve. By applying these forces on a rigid body dynamics system, and modeling the object as a rigid body, the object is pushed on a path that is defined by the wavefronts.

The application of the force field on the bodies motion is based on the work of David Baraff [Baraff(2001)]. We present the most interesting (for our purpose) part of his publication: our goal is to calculate two vectors, the linear momentum \( P \) and the angular momentum \( L \). The linear momentum is expressed as

\[ \frac{dP}{dt} = F \]  

(4.5)

and can be also expressed as

\[ \Delta P = F \Delta t \]  

(4.6)
where \( F \) is the net force calculated as the summation of the \( N \) forces that act on the body

\[
F = \sum_{i=0}^{N} f_i
\]  

(4.7)

We obtain these forces by sampling the force field at every vertex position of the body curve (see Figure 4.7).

The angular momentum is expressed similarly to the linear momentum

\[
\Delta L = T \Delta t
\]  

(4.8)

Where \( T \) is the net torque calculated also as the summation of the \( N \) torques that act on the body

\[
T = \sum_{i=0}^{N} t_i
\]  

(4.9)

while each torque can be derived by the corresponding force and the center of mass \( x \) of the body

\[
t_i = (p_i - x) \times f_i
\]  

(4.10)

where \( p_i \) is the point where the force acts on. In our case there correspond to the vertex positions of the body curve.

We update the linear and angular momentum at each frame by increasing them by the quantities calculated in Equations 4.6 and 4.8 respectively. It’s worth mentioning that the size of the time step \( \Delta t \) can have an impact on the accuracy of our calculation when it becomes very large.

From the linear momentum we can derive the linear velocity of the body as

\[
\Delta v = P/m
\]  

(4.11)
where $m$ is the mass of the body and similarly we can derive the angular velocity from the angular momentum as

$$\Delta \omega = I^{-1}T \Delta t$$  \hspace{1cm} (4.12)

where $I$ is the body’s inertial tensor.

At each time-step the linear and angular velocities are updated by adding the quantities found in Equations 4.9 and 4.10 respectively.

The position vector of the body is finally calculated as

$$x_{t+\Delta t} = x_t + v_t$$  \hspace{1cm} (4.13)

and its rotation matrix as

$$R_{t+\Delta t} = R_t + \omega^* R_t \Delta t$$  \hspace{1cm} (4.14)

where

$$\omega^* = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}$$  \hspace{1cm} (4.15)

We constraint the orientation of the body on its local $y$ axis by setting $\omega_x = 0$ and $\omega_z = 0$ and we handle vertical motion separately.

For vertical movements, we sample the height at the center of mass and by caching the previous value, we apply a vertical force if their difference surpasses a threshold. Additionally, we apply a gravity force when the center of mass is above sea level and the aforementioned force is not applied.
Chapter 5

Level of Detail

In the previous chapters, we assumed a regular grid that divides the space where the wavefront propagation takes place. So far, it was implied that the resolution of the grid is such that the simulation is recorded in good detail. Since the complexity of the collision detection between the grid and wavefront curves is linear to the resolution of the grid, it would be desirable to be able to reduce the resolution without affecting the level of detail of the final rendering. In other words, we would prefer to decouple the grid used for the simulation from the heights that are used for rendering the final surface.

In [Jeschke and Wojtan(2015)], Jeschke and Wojtan achieve this by recording the simulation parameters on the vertices of a triangulation, and then interpolating them for the intermediate points when they need to render the surface. Following their example, we introduce a secondary data structure that holds the points that are rendered. This structure is also a regular grid but it has a higher resolution. In the rest of this chapter we explain how this is done in detail.

5.1 The Interpolation Issue

At this point, we would like to bring your attention to a limitation of using an interpolation scheme for the rendered points; assume that a triangle or a rectangle is going to be covered
Figure 5.1: Different grid resolutions representing the same wave. From Top to Bottom, Left to Right: 200x200, 100x100, 50x50, 25x25

by a curve, that is each of its vertices will be covered by the wavefront over time. To be able to interpolate the intermediate values, we need to have a record related to the mentioned wavefront curve on each of the vertices. This would allow us to use some interpolation method on each of the parameters and recalculate the final height from these values. In the case when the simulation is a pre-processing step, as in Jeschke and Wojtan (2015), an interpolation scheme is ideal since the simulation will finish before we start rendering the surface and the information is final. For our purposes, interpolating the records is an issue since updating a vertex by adding a new record would result in a sudden animation of some part of the surface. This creates a popping artifact which is obviously not acceptable so we have to resort to a different approach.
5.2 Estimation of Wavefront Propagation

In Chapter 3, the wavefront collides with and stores records on the sampling points. By triangulating these points, we can represent the surface and animate each point by the superposition of its records. In practice, the density of these points can influence the time performance so for real-time targets, a less dense set of sampling points is preferred. Unfortunately, reducing the sampling resolution leads to loss of detail. An example is show in Figure 5.1: The original wave has a small wavelength and the representation in a different sampling resolution gives the impression that the wave has a small wave number.

We overcome this problem similarly to [Jeschke and Wojtan(2015)] by using an underlying grid with a higher resolution: When a record is added to a point of the top level grid, the neighboring points of the underlying grid are also updated according to the information of the record. This concept is based on the modeling of linear waves as

\[ \eta(x, y, t) = \sum_{i=1}^{n} a_i \sin(p_i x + q_i y - \omega_i t) \]  

(5.1)
presented in \cite{Ts'o and Barsky(1987)}. Here, $p_i$ and $q_i$ are the coordinates of a direction vector, that corresponds to the direction of the wavefront. In Section 3.5 we explain that each record holds a direction vector and the magnitude corresponds to the phase speed of the wavefront at the moment of collision. We use that information to approximate the time of arrival of the wavefront at each point of the underlying grid, as if the wavefront was propagating with the exact same properties. For clarity, we refer to the top-level grid the *sampling grid* and the underlying grid as the *height grid*.

Our first approach was to select one of the neighbouring cells of the collided point in the sampling grid according to the direction vector and apply Equation 5.1 to each of the point of the heights grid that the cell contained. By ensuring that the resolution and offsets of the two grids are such that the points of the sampling grid fall on points of the heights grid, and that the sampling grid covers all of the heights grid uniformly, these points can be found fast by taking the ratio

\[
\text{limit} = \frac{\text{grid}_{\text{heights}}}{\text{grid}_{\text{sampling}}} \tag{5.2}
\]

and the set of points can be described as

\[
Cell_{m,n} = \{p_{i,j}|i \in [m, m + \text{sign}(p)\text{limit}_x), j \in [n, +\text{sign}(q)\text{limit}_y)}\} \tag{5.3}
\]

The initial phase can be calculated as

\[
\phi_{i,j} = t_{m,n} + (i|p_{m,n}| + j|q_{m,n}|)c_{m,n} \tag{5.4}
\]

where $t_{m,n}$ is the arrival time of the wavefront at sampling point $p_{m,n}$, and $c_{m,n}$ is the corresponding phase speed. In the same way as in \cite{Jeschke and Wojtan(2015)}, when the total phase is negative, we do not take the record entry into account for the height calculation. This allows us to render the estimated motion of the wavefront as it approaches the next grid points.
The amplitude in this height calculation approach can also be scaled as in Equation 4.2. More specifically, Equation 5.1 can be also calculated as

$$\eta(x, y, t) = \sum_{i=1}^{n} a_i w_i(t) \sin(p_i x + q_i y - \omega_i t)$$

(5.5)

where $w(t)$ is the same as in Equation 4.3.

The immediate results showed that we can successfully estimate the phase when the direction does not change (Figure 5.2) but we have discontinuity artifacts (Figure 5.2 and 5.3), in neighbouring cells with different direction vectors.

These artifacts discouraged us from estimating the initial phase for a cell of the grid and work with something that is independent of how the space is segmented. Following the wavefront modeling in Ts'o and Barsky(1987), we temporarily model the wavefronts as rays instead of curves and we propagate the appropriate information along a ray: When a sampling grid point is covered, we cast a line starting at the point in the recorded direction. The line extends into a cell and ends when the border of the cell is met (see Figure 5.6). For each point along the line, we extend two new lines that start at the said point and extend in the two, perpendicular to the wavefront, directions (Figure 5.5). The length of these lines is the same as the original. We calculate the phase of each of the traversed points by calculating their projection on the original line and linearly interpolating between the phase values of its extreme points. Additionally, we calculate a normalized weight according to their distance from the original line which is used as a
Figure 5.4: Two cases of Algorithm 8. Red represents the pivots, green the positions and blue are the positions that are missed. The size of the dots relates to the weight of the point.

linear filter on the amplitude. For simplicity, we assume that each grid has dimensions of the same length and thus Equation 5.2 becomes

$$\text{limit} = \frac{\text{grid}_\text{size}_\text{heights}}{\text{grid}_\text{size}_\text{samples}}$$  

An example of the method is illustrated in Figure 5.4. It is evident that some points are missed. We detect such cases by observing that these points appear when we perform a diagonal traversal step both in the perpendicular and the wavefront direction (see Figure 5.5).

The above modeling has a side effect that can be desired or not, depending on the type of scene that is represented. When the direction of the lines is either orthogonal ($x = 0$ or $y = 0$) or diagonal ($|x| = |y|$), the lines align, forming a group of bigger lines (Figure 5.6). In any other case, this does not hold and the result looks more chaotic which is closer to open water waves, so perfect round waves cannot be described. A possible solution is to mark the end of each line on the sampling grid, and when a new collision is detected, cast a new line from one of the nearest markings. For the moment, this is left as an idea for future work.
Figure 5.5: Example of traversal that handles missed cases. The direction vector is (1,0.5). Red arrows show the pivot point traversal, green ones the position points traversal and blue ones the special case points traversal. Note that each point of the grid is covered exactly once.

Figure 5.6: Propagation directions: the first two cases will form perfect waves while the third case will have a more chaotic effect.
Chapter 6

Real-time Rendering

The final part of this work is rendering the water surface in real-time. This is essential both for presenting the results in nice aesthetics but also for having a complete pipeline of the method for measuring the overall performance.

The rendering approach can be divided into the CPU and the GPU processing. In the CPU processing we prepare some data that are passed to the GPU, and the GPU produces the final results.

6.1 CPU Processing

The CPU part of the implementation is responsible for supplying the GLSL shaders with the appropriate data. In general, we try to reduce the CPU workload by assigning tasks that can be executed in parallel, to the GPU. Some of these tasks, for example, the normal vectors calculation, require that the height values of the water surface to be stored in shared memory. For this reason, we calculate the final heights and store them on a texture map that is send to the GPU in every frame. Together with the heightmap, we calculate two more textures that hold the information about the dominant direction and speed of each height grid point at each frame. This information will be used for a secondary effect during the GPU processing. Calculating the directions for each point is
done as

\[ \text{direction}_{i,j} = \frac{\sum_{k \in \omega_{i,j}} \frac{v_k}{|v_k|}}{|\omega_{i,j}|} \]  

(6.1)

where \( \omega_{i,j} \) is the set of records on sample point \((i, j)\) whose initial phase is less than the accumulated time of the simulation at the current frame. \( v_k \) is the velocity stored in record \( k \). In a similar manner we calculate the speed at each point as

\[ \text{speed}_{i,j}(t) = \frac{|\sum_{k \in \omega_{i,j}} \frac{v_k}{|v_k|} \cdot w_k(t) \cdot a_k|}{|\omega_{i,j}|} \]  

(6.2)

where \( \omega_{i,j} \) and \( v_k \) are the same as in Equation 6.1. \( w_k(t) \) and \( a_k \) are the weight function and the amplitude of record \( k \), respectively.

Finally the height map is calculated as in Equation 5.1 where only records with non-negative phase are taken into account.

### 6.2 GPU Processing

The GPU rendering is broken down into three shading stages, the vertex shader, the geometry shader and the fragment shader. We provide the GLSL code at the Appendix and here we give an abstract description of our GPU pipeline.

#### 6.2.1 Vertex and Geometry Shaders

The vertex shader is simply setting the vertex global position using the model, view and projection matrices while storing the local space \( x - z \) coordinates of the vertex for texture look-ups in the next stages.

The geometry shader is responsible for producing a triangulation of the surface from the passed vertices and for calculating the normal vectors for each vertex. The normal vectors are calculated by taking the average normal

\[ \text{normal}_{i,j} = \frac{\text{normal}^{N,E} + \text{normal}^{E,S} + \text{normal}^{S,W} + \text{normal}^{W,N}}{4} \]  

(6.3)
normal\_N,E = (\eta_{i,j} + 1 - \eta_{i,j}) \times (\eta_{i+1,j} - \eta_{i,j}) \quad (6.4)

normal\_E,S = (\eta_{i+1,j} - \eta_{i,j}) \times (\eta_{i,j+1} - \eta_{i,j}) \quad (6.5)

normal\_S,W = (\eta_{i,j} - \eta_{i,j+1}) \times (\eta_{i,j} - \eta_{i,j}) \quad (6.6)

normal\_W,N = (\eta_{i,j} - \eta_{i,j}) \times (\eta_{i,j} + 1 - \eta_{i,j}) \quad (6.7)

and \eta_{i,j} is the height at position (i, j).

The mentioned information is interpolated and passed to the fragment shader.

### 6.2.2 Fragment Shader

In the fragment shader we perform two tasks. The first is to calculate the Fresnel ratio for shading the water surface. This value is used to determine how we will blend between the reflection of the skybox and the a blurred image of the ocean bed for the colour of that fragment. Normally, the ocean bed colour should be formulated by taking a refraction
ray and look up on the skybox the appropriate color value. Since our skybox is not updated with the ocean bed image we simply substitute this color information with a semi-transparent cyan color so that the ocean bed is visible through a colored filter. After calculating the reflection vector, we obtain the reflection pixel values from the skybox and we blend by using the Fresnel ratio that is calculated as

\[
\text{ratio} = F + (1 - F)(1 - (\mathbf{\text{view}} \cdot \mathbf{\text{normal}}))^p;
\]  

(6.8)

where \( F \) is an arbitrary normalized constant, \( \mathbf{\text{view}} \) is the view vector and \( \mathbf{\text{normal}} \) is the normal vector. The final colour is calculated by linearly interpolating between the two colour values where \( \text{ratio} \) is used as the interpolant.

A second task performed at the fragment shader is the rendering of waves with very small amplitudes. This is achieved through normal mapping; by having some direction and speed information at each fragment we can advocate texture coordinates and use them to look up a normal map. We use the normal map shown in Figure 6.1. To be able to capture all different orientations, we create texture coordinates by converting to cartesian the polar coordinates produced by the following approach: we set the azimuth according to the direction information, map the pole at the center of the map and increase the radius over time according to the speed information. The direction and speed direction information is calculated on the CPU and passed as a texture to the GPU. An example of both this technique and the final shading is shown in Figure 6.2.
Figure 6.2: Rendering example that demonstrates the Fresnel effect and the smaller waves.
Chapter 7

Evaluation and Discussion

We have managed to convert the work of Jeschke and Wojtan \cite{Jeschke and Wojtan(2015)} into a system suitable for interactive entertainment and we have proposed some dynamic body interactions. Since we push the boundaries towards interactive applications, it is essential that we measure the performance in terms of resource consumption and speed and that we detect bottlenecks that could lead the future work of this research.

7.1 Evaluation

For the purposes of demonstration and performance measurement, we have created a small demo scene with 10 floating bodies and a self driven body. The resolution of the sampling grid is 100 by 100 and the resolution of the height grid is 400 by 400. The scene features an ocean bed that is modeled as a 400 by 400 map and we have registered a static obstacle of small geometrical complexity. The scene is initialized by the release of two big wavefront curves in opposing directions. A video of the demo scene can be found at https://www.youtube.com/watch?v=2hIaybwhuMw.
7.1.1 Performance Measurement

We evaluate the performance using some metrics directly related to the time a task needs to be completed per frame (in milliseconds) and the memory consumption for the whole system. There are four time measurements. The first is Simulation time, which corresponds to the time needed by the simulation to run per frame. The time to render the water surface and the rest of the scene is captured in Water Rendering and Other Rendering times respectively. The total time for each frame is given by the Total Time metric. In terms of memory, we measure the Virtual and the Physical memory consumption by the application process, in bytes.

We have run the system in two different scenarios. In the first, we do not move the boat for the purpose of avoiding the generation of wavefronts from moving bodies. In the second, we accelerate and steer the boat to produce multiple wavefronts. The results for the first case are shown in Figure 7.2 and the results for the latter in Figure 7.3. The graphs of memory consumption over time are shown in Figure 7.4 and 7.5.


7.1.2 Performance Analysis

**Memory consumption** Figure 7.4 and 7.5 show the growth of memory consumption over time in the two aforementioned cases. It seems that they follow a similar trend which resembles a logarithmic curve. Also there seems to be a convergence after 2K frames and the maximum consumption during the 6K and 9K frames is about 215 Mega bytes.

**Scenario 1** Figure 7.2 shows stable per frame times for rendering the rest of the scene and a less stable graph for rendering the water surface. The latter seems to take most of the per frame time as it varies mostly between 7 and 17 milliseconds with many abrupt spikes. The times for the simulation have an intensive start but they stabilize before the first 3K frames and the stay bellow 5 milliseconds per frame for the most part. The total time graph is very similar to the one for water rendering, which implies that the frametime is dominated by the time needed to render the surface.

**Scenario 2** In Figure 7.3, we can see that the times for rendering the rest of the scene remain stable while the times to render the water surface have higher spikes but the graph values and patterns are similar to the corresponding one in Scenario 1. The simulation times have grown dramatically in a range between 3 and 25 milliseconds and there does not seem to be any stabilization. The total times graph is also different as it seems to be dominated by the simulation times instead of times for water rendering.

**Scenario 1 vs Scenario 2** It is evident that when no wavefronts are generated from the moving bodies, the main bottleneck is the rendering of the water surface. When we have moving bodies, the simulation seems to have the biggest influence on the total time. In both scenarios, the rendering of the rest of the scene has no significant influence.

7.2 Final Discussion

While the main goal of this thesis was achieved successfully there are many considerations that one should take into account when extending the results of this work. Technical and modeling limitations exist but we believe that one could improve upon these issues and
bring this system one step closer to be applied in real-time entertainment software.

7.2.1 Limitations

The most significant limitations are those caused by our modeling. We give a brief description for each of them and mention some technical ones as well.

**Perfect waves:** When we allowed the simulation to be done dynamically by propagating the wave parameters (Chapter 5), we have created a side effect that the produced waves look rough when the wavefront orientation is not orthogonal or diagonal (See right-most example in Figure 5.6). While this is a desired feature since it is rare that we find perfect waves in nature, it constrains the target scenes to open sea waves or any other scenario where the initial phase is not distorted.

**Accuracy of the heuristic approach:** In the idea described in Chapter 5, we use a heuristic approach to estimate the position that the wave will have reached in a time

Figure 7.2: Time performance when dynamic bodies do not generate wavefronts.
interval, derived by the velocity of the wave at the time of collision. This operation works well for the tested grid resolutions of the top level grid, but if we reduce the resolution to improve time performance, it is expected that we will suffer loss of accuracy since the wave direction and speed might change very early after the collision.

**Collision detection versus scale:** As shown in Chapter 4 (see Figure 4.4), we have an issue when the wavefront curve resolution and the scale of the obstacles are such that some collisions may be missed. Since the wavefront curve resolution is directly related to the resolution of the top level grid, we have means of verifying if the scale of an obstacle is appropriate (i.e. comparing the area of the projected convex hull of the obstacle against the area of the grid cell), but resolving the issue would require to either change the scale of our models or increase the top level grid resolution.

There are also some technical limitations that were discovered by profiling the demo scene.

Figure 7.3: Time performance when dynamic bodies generate wavefronts.
Figure 7.4: Memory performance for wavefronts generated from moving bodies.

Figure 7.5: Memory performance for no wavefronts generated from moving bodies.
Number of Wavefronts generated from self-driven bodies: From the performance graphs, it is evident that generated wavefronts from self-driven bodies has an impact on the overall performance. A rational explanation is that we release wavefronts when the linear momentum of the body changes either in magnitude or direction, which can happen in any frequency since these bodies are expected to be controlled by human players or artificial intelligence agents.

Workload imbalance on the sampling points: Another technical limitation is the distribution of workload for calculating the final heights on the sampling points. Since each point has a list of records that may vary significantly in length, distributing the workload is not a trivial task. In our implementation, we have used the OpenMP for loop parallelism scheme [Dagum and Menon(1998)] to improve the performance, but this is not the right answer to the general case of the problem.

7.2.2 Conclusions

We have managed to create an accurate and intelligent water wave system that is suitable for interactive entertainment. The first benefit of such a system is accuracy; we use a set of parameters to recalculate the heights at each rendered point rather than interpolating the height values from the set of sampling points. The second benefit is the awareness of the system about its environment: manual pre-processing for environment information is eliminated from the work flow when one is using this system.

Our major contribution, which can be described as exchanging the original interpolation scheme for a heuristic propagation scheme not only unifies the simulation and the rendering steps but also opens the way for real-time interactivity. As a secondary contribution, we have proposed some basic interactions between the surface and floating bodies, that use existing features and do not require the combination with an additional system.

Throughout the research and the technical evaluation, we have discovered limitations that may be discouraging but should not stop one from improving the results of this
7.2.3 Future work

Our advice for readers who would like to extend this work is to focus on technical limitations, and specifically on the issue regarding the number of generated wavefronts. From the modeling limitations, we believe that the collision detection versus scale is the most significant.

There is also a great number of rendering effects used by the games industry that one could add to the system. These include particle effects, wave surf, rain drops and better shading according to ocean depth to name a few.

Finally, an interesting problem would be the regulation of Levels of Detail according to the distance of the rendering points from the camera. There are many approaches that one could take, from changing the density of the grid points to changing the number of effective records on each point. The latter has the challenge that there is not a definitive way to prioritize a record over another: choosing the one with the maximum amplitude may sound like a reasonable approach, but preferring the one with the minimum angular frequency reduces aliasing effects if the point is away from the point of view.
# Appendix A

## Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>IFFT</td>
<td>Inverse Fast Fourier Transform</td>
</tr>
<tr>
<td>ODE</td>
<td>Ordinary Differential Equation</td>
</tr>
<tr>
<td>IDW</td>
<td>Inverse Distance Weighting</td>
</tr>
<tr>
<td>NURBS</td>
<td>Non-Uniform Rational B-Spline</td>
</tr>
</tbody>
</table>
Appendix B

Appendix: GLSL shading

Vertex shader

#version 450

uniform mat4 MV; //ModelView Matrix
uniform mat4 MVP; //ModelViewProjection Matrix

layout(location = 0) in vec3 pos;

out vec2 xzcoords;

void main(void){
    xzcoords = vec2(pos.x,pos.z);
    gl_Position = MVP * vec4(pos,1);
}

Geometry shader

#version 450

layout(points) in;
layout(triangle_strip, max_vertices = 6) out;

uniform mat4 MVP;
uniform vec2 offset;
uniform ivec2 size;
uniform sampler2D diffuseMap;

int kernelSize=2;
float gridSize=1.0;

float gaussianHeight(vec2 xzcoords);

vec3 calculateNormal(vec2 xzcoords){
    float heightC = gaussianHeight(xzcoords);
    float heightR = gaussianHeight(xzcoords+vec2(offset.x,0.0f));
    float heightL = gaussianHeight(xzcoords+vec2(-offset.x,0.0f));
    float heightN = gaussianHeight(xzcoords+vec2(0.0f,offset.x));
    float heightS = gaussianHeight(xzcoords+vec2(0.0f,-offset.x));
    vec3 vecN = vec3(0,heightN-heightC,offset.x);
    vec3 vecS = vec3(0,heightS-heightC,-offset.x);
    vec3 vecL = vec3(-offset.x,heightL-heightC,0);
    vec3 vecR = vec3(offset.x,heightR-heightC,0);
    vec3 crosss=

63
(\text{cross}(\text{vecN}, \text{vecR}) + \text{cross}(\text{vecR}, \text{vecS}) + \text{cross}(\text{vecS}, \text{vecL}) + \text{cross}(\text{vecL}, \text{vecN}))

\text{return normalize(crosss);}

\}

\text{float gaussianHeight(vec2 xzcoords)\{}

\text{float height=0.0f;}

\text{for(int i=-kernelSize;i<=kernelSize;i++){}

\text{for(int j=-kernelSize;j<=kernelSize;j++){}

\text{height+=}

\text{texture(diffuseMap,(xzcoords+vec2(i,j))/gridSize).r;}

\}

\}

\text{return height/((\text{kernelSize+1})*(\text{kernelSize+1}));}
\}

\text{in vec2 xzcoords[];}

\text{out vec2 uvCoords;}

\text{out vec3 normal;}

\text{out vec3 pos;}

\text{void main() \{}

\text{//set global vars}

\text{gridSize=offset.x*size.x;}

\text{//calc positions}

\text{vec4 point1 = vec4(xzcoords[0].x, gaussianHeight(xzcoords[0]), xzcoords[0].y, 1.0f);}

\text{vec4 point1xz = vec4(point1.x, 0, point1.z, 1.0f);}
vec4 point2 =
point1xz+vec4(offset.x,gaussianHeight((xzcoords[0]+vec2(offset.x,0))),0,0);
vec4 point3 =
point1xz+vec4(0,gaussianHeight((xzcoords[0]+vec2(0,offset.x))),offset.x,0);
vec4 point4 = point1xz+
vec4(offset.x,gaussianHeight(xzcoords[0]+vec2(offset.x,offset.x)),offset.x,0);

//calc normals
vec3 normal1 = calculateNormal(xzcoords[0]);
vec3 normal2 = calculateNormal(xzcoords[0]+vec2(offset.x,0));
vec3 normal3 = calculateNormal(xzcoords[0]+vec2(0,offset.x));
vec3 normal4 = calculateNormal(xzcoords[0]+vec2(offset.x,offset.x));

gl_Position = MVP*point2;
uvCoords = vec2(1.0f,0.0f);
pos = point2.xyz;
normal = normal2;
EmitVertex();

gl_Position = MVP*point1;
pos= point1.xyz;
uvCoords=vec2(0.0f,0.0f);
normal = normal1;
EmitVertex();

gl_Position = MVP*point3;

uvCoords = vec2(0.0f,1.0f);
pos = point3.xyz;
normal = normal3;
EmitVertex();

gl_Position = MVP*point3;
pos = point3.xyz;
  uvCoords = vec2(0.0f,1.0f);
normal = normal3;
EmitVertex();

gl_Position = MVP*point2;
pos = point2.xyz;
  uvCoords = vec2(1.0f,0.0f);
normal = normal2;
EmitVertex();

gl_Position = MVP*point4;
pos = point4.xyz;
  uvCoords = vec2(1.0f,1.0f);
normal = normal4;
EmitVertex();

EndPrimitive();
}
Fragment Shader

#version 450

uniform samplerCube skyboxMap;
uniform sampler2D diffuseMap;
uniform sampler2D normalMap;
uniform sampler2D specularMap;
uniform sampler2D extraMap1;
uniform vec4 color;
uniform float alphaValue;
uniform float uvOffset;

uniform vec2 offset;
uniform ivec2 size;

uniform mat4 MV;
uniform mat4 MVP;
uniform mat4 NM; //Normal Matrix (Transpose of inverse of view)
uniform mat4 VI; //View Inverse
uniform mat4 V; //View

layout(location = 0) out vec4 fragment_color;

in vec2 uvCoords;
in vec3 normal;
in vec3 pos;
const float EtaR = 0.97;
const float EtaG = 0.98; // Ratio of indices of refraction
const float EtaB = 0.99;
const float fresnelPower = 5.0;
const float F = ((1.0-EtaG) * (1.0-EtaG)) / ((1.0+EtaG) * (1.0+EtaG));

float fresnelRatio = 0.0f;
vec2 uvDirection = vec2(1,1);

int kernelSize = 0;

int isSmall(vec2 inp){
    return int(length(inp)<0.4f);
}

int isNotBig(vec2 inp){
    return int(length(inp)<0.99f);
}

vec2 gaussianDirection(vec2 xzcoords){
    vec2 speed = vec2(0.0f, 0.0f);
    float gridSize = offset.x * size.x;
    for(int i=-kernelSize;i<=kernelSize;i++){
        for(int j=-kernelSize;j<=kernelSize;j++){
            speed += -
                (((texture(specularMap, (xzcoords+vec2(i,j))/gridSize).rg) -
            normalize(vec2(1.0f,1.0f)));
        }
    }
}
```cpp
vec2 ssign = sign(vec2(0,0)*isSmall(speed)+(1-isSmall(speed))*speed);
float slen = clamp(length(ssign),0.0f,1.0f);
speed = normalize(speed)*slen+speed*(1-slen);
speed = speed*(1-isNotBig(speed));
return speed/((kernelSize+1)*(kernelSize+1));
}

vec3 calculateReflection(vec3 pos, vec3 norm){
    vec4 world_pos = (MV*vec4(pos,1));
    vec4 world_view = world_pos-vec4(0.0f,0.0f,0.0f,1.0f);
    vec3 viewDirection = vec3(VI*world_view);
    vec3 world_normal = (NM*vec4(norm,0.0f)).xyz;
    vec3 normalDirection = normalize(vec3(vec4(world_normal,0.0f)*V));
    fresnelRatio= F+(1.0-F)*pow(1.0-dot(normalize(-viewDirection),
                             normalize((NM*vec4(norm,0)).xyz)),fresnelPower);
    return normalize(reflect(viewDirection,normalize(normalDirection)));
}

vec2 directionMapCoordinates; // for direction
vec2 normalMapCoordinates; // for normal mapping
float tilingFactor=1.0f;
float uvSpeed=1.0f;

float gridSize= offset.x*size.x;

float sum(vec2 v){
    return v.x+v.y;
}
```
void main (void){

    float gridSize=offset.x*size.x;

directionMapCoordinates = mod(pos.xz/gridSize,vec2(1.0,1.0));

vec2 directionTexel =gaussianDirection(pos.xz);

uvSpeed = texture(extraMap1,directionMapCoordinates).r;

    float directedDistance =
length(pos)*dot((MV*vec4(pos,1.0f)).xyz,vec3(0,0,-1));

vec2 directionSign = sign(directionTexel);

vec2 uvDistance =
(directionTexel*(uvOffset*uvSpeed+ sum(directionTexel*pos.xz*0.5f)*tilingFactor));

normalMapCoordinates=
mod(uvDistance,vec2(1.0f,1.0f));

vec2 newUVCoords
=(vec2(0.5,0.5)+directionSign*0.5f*normalMapCoordinates);

vec3 newNormal =
normalize(0.15f*normal+clamp(1.0f/directedDistance,0.0,0.05f)*
(texture(normalMap,newUVCoords).rgb*2.0-1.0));

vec3 newRelfection = calculateReflection(pos,newNormal);

float newRatio = fresnelRatio;

vec3 reflectColor = vec3(texture(skyboxMap, newRelfection));

if(newRatio>0.5){
    newRelfection = calculateReflection(pos,normal);
}

vec4 color = mix(vec4(0,0,0.2,0.4f), vec4(reflectColor,1.0f),

clamp(newRatio, 0.0f, 1.0f);

fragment_color = color;

}
Bibliography


[Airy(1841)] George Biddell Airy. Tides and waves. 1841.


