Fast, interactive water wave simulation in gpu for games

Chen, Ming

2010

Chen, M. (2010). Fast, interactive water wave simulation in gpu for games. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/47488>

<https://doi.org/10.32657/10356/47488>

Nanyang Technological University

Downloaded on 26 Apr 2023 19:14:21 SGT

287
2X
D/3/100

FAST, INTERACTIVE WATER WAVE SIMULATION IN GPU FOR GAMES

CHEN MING

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Master of Engineering

2010

Abstract

This thesis report presents the concept of an algorithm and the development of a framework for interactive water wave simulation mainly based on the height field approach. Dynamic user inputs can interact with the simulated water to create and propagate waves accordingly. This interaction can provide immersive feeling to video game players.

The most important feature of the algorithm is the very short computational time it takes. The algorithm is targeting the modern video games with a high-definition (HD) standard resolution. It can be run at 720p or even 1080p resolution to achieve a frame rate suitable for the video games. The algorithm is well implemented on the GPU architecture for acceleration purpose.

A water simulation platform is also designed in this research which is able to combine various types of water wave effects such as ocean wave, ripples and even breaking waves together. This architecture is good for making the result more realistic.

Acknowledgements

The author would like to express his gratitude to Assistant Professor Zhou Suiping from the School of Computer Engineering, who has helped him with his support for the project.

He would also like to thank the staffs and colleagues from TQ Global who had given him ideas and constructive comments on his work and helped him dealing with various problems with the project as well as providing him with the resources he need.

The last but not least, thanks to all the friends who have supported him throughout the project.

Table of Contents

Abstract	i
Acknowledgements	ii
Table of Contents	iii
List of Figures	v
List of Tables	vii
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Objectives and Measurements	2
1.3 Contribution	3
1.4 Overview	4
Chapter 2 Background	5
2.1 Overview of Water Waves	5
2.1.1 Physics Principle behind Water Waves	5
2.1.2 The Shape of Water Waves	8
2.1.3 Wave Parameters	10
2.1.4 Characteristics of Water Waves	11
2.1.5 Classification of Water Waves	12
2.2 GPU and GPU Programming	14
2.2.1 GPU	14
2.2.2 Fix-function Graphics Pipeline	16
2.2.3 Programmable GPU Pipeline	18
Chapter 3 Related Work	21
3.1 Particle System Approach	21
3.1.1 Lagrangian Solution	22
3.1.2 Eulerian Solution	22
3.1.3 Non-Navier Stokes Solver Solution	23
3.1.4 Summary of Particle System Approach	25
3.2 Spectral Approach	26
3.2.1 Basic Ideas of Spectral Approach	26
3.2.2 Multi-Band Spectral Approach	27
3.2.3 Summary of Spectral Approach	28
3.3 Procedural Approach	29
3.3.1 Basic Ideas of Procedural Approach	29
3.3.2 Summary of Procedural Approach	32
3.4 Height Field Approach	33

- 3.4.1 Basic Ideas of Height Field Approach 34
 - 3.4.2 Implementation using Vertex Texture Fetching in GPU 34
 - 3.4.3 Summary of Height Field Approach..... 41
 - 3.5 Summary on the Related Work 42
- Chapter 4 Design Fast Interactive Water Simulation for Games 44**
 - 4.1 Interactive Water Wave Simulation 45
 - 4.1.1 Analysis on the nVIDIA's Height Field based Solution 45
 - 4.1.2 Assumption of using Height Fields on Large Scale Water..... 45
 - 4.1.3 Adhesive Type of Height Field..... 47
 - 4.1.4 Optimization on Wave Simulation 51
 - 4.2 Interactive Water Wave Rendering 57
 - 4.2.1 Water Wave Mesh 58
 - 4.2.2 Hybrid Framework Design 58
 - 4.2.3 Water Wave Shading 64
 - 4.3 Cater for Interaction with Multiple Objects 66
- Chapter 5 Results 70**
 - 5.1 Performance Measurement Criteria 70
 - 5.2 Demo Program Setup 73
 - 5.3 Demo Program Performance..... 75
 - 5.3.1 Timing Performance with One Boat 75
 - 5.3.2 Timing Performance with Many Boats..... 76
 - 5.3.3 Interaction Performance 78
 - 5.3.4 Visual Performance 79
- Chapter 6 Conclusion & Future Work..... 82**
 - 6.1 Conclusion..... 82
 - 6.2 Future Work 83
 - 6.2.1 Wave Simulation Timing Control..... 83
 - 6.2.2 Higher Wave Details and More Complex Wave Effect 83
 - 6.2.3 Adapt to more Physical Behaviours..... 84
- References..... 85**

List of Figures

Figure 2.1	Circular Motion of Water Particles along a Wave [5]	6
Figure 2.2	Motion of Water Particles, A = Deep Water, B = Shallow Water [3]	7
Figure 2.3	The Difference of Deep Water Waveform and Shallow Water Waveform due to Change in Motion of Water Particles	8
Figure 2.4	Comparison of a Trochoidal Curve and a Sine Curve [8]	9
Figure 2.5	Example of actual Waveform recorded in Water [4]	10
Figure 2.6	Wave Characteristics[4]	10
Figure 2.7	Superposition of two Waves	12
Figure 2.8	Relationship between Wave Speed and Wavelength[5]	13
Figure 2.9	The Basic Architecture of SIMD	16
Figure 2.10	Fix-function Graphics Pipeline [11]	17
Figure 2.11	Fix GPU Pipeline (in white) and Programmable GPU Pipeline (in blue) .	19
Figure 3.1	Illustration on the Steps to Process the Density in Stam's Solution [21] ..	23
Figure 3.2	Illustration of D2Q9 and D3Q19 Lattice Node Model [21]	24
Figure 3.3	Flowchart for Mastin's Spectral Approach	26
Figure 3.4	Concept for Multi-Band Synthesis of Spectral Approach	28
Figure 3.5	Different Combination of κ and r which affecting Shape of Wave [31]	30
Figure 3.6	Result of Shoreline Waves on a gently Slopes shown in Fournier and Reeves' Solution [31]	31
Figure 3.7	Illustration on how the up and down Acceleration of Water Wave is calculated from the Pixels on the Height Map Texture	36
Figure 3.8	Illustration on nVIDIA's Solution, how the 3 Height Map Textures are updated for Water Simulation	39
Figure 3.9	Illustration on nVIDIA's Solution, how the 3 Height Map Textures are updated for Water Simulation (Continued)	40
Figure 3.10	Result of nVIDIA's Solution [40]	41
Figure 4.1	Screenshot from video game Surf's Up (2007)	47
Figure 4.2	Dampening Texture	48
Figure 4.3	Illustration of the Translation Step for Adhesive Type of Height Fields ..	49
Figure 4.4	Illustration of the Nearest-Point Sampling and Bi-Linear Sampling	51
Figure 4.5	Illustration on how 3 Copies of Height Field stored and used in one Texture	56

Figure 4.6 Integrating Three Different Wave Types into the Framework 59

Figure 4.7 Integrating Ocean Waves into the Framework..... 60

Figure 4.8 Integrating Interactive Waves into the Framework..... 60

Figure 4.9 Capillary Wave Normal Map generated by Spectral Approach 62

Figure 4.10 Integrating Capillary Waves into the Framework..... 63

Figure 4.11 Final Result of the Framework after integrating all Three Wave Types ... 63

Figure 4.12 Concept of Cube Environment Mapping 65

Figure 4.13 Falloff texture for Fresnel Effect 66

Figure 4.14 Illustration on the Shape of a Sample Screen-based Height Map 67

Figure 4.15 Illustration on how Screen-based Texture is updated 68

Figure 5.1 Result of Traditional Timing Measurement 71

Figure 5.2 Result of Timing Measurement with WaitGPUIidle() Function 72

Figure 5.3 Timing Measurement Strategy used in the Demo 73

Figure 5.4 Timing Performance Trend with increasing Number of Boats 78

Figure 5.5 A Single Boat interacting with Water, Adhesive Grid Version 80

Figure 5.6 10 Boats interacting with Water, Ocean Grid with Many Boats Version .. 80

Figure 5.7 Highlight the Details of Interactive Waves and Capillary Waves with
Breaking Waves..... 81

Figure 5.8 Effects of Ocean Wave Simulation 81

List of Tables

Table 5.1	Timing Performance for all 4 Version of Water Wave Simulation at 720p	75
Table 5.2	Timing Performance for all 4 Version of Water Wave Simulation at 1080p	75
Table 5.3	Ocean Grid with Many Boat Version: Timing Performance vs. Number of Boat at 720p	77
Table 5.4	Ocean Grid with Many Boat Version: Timing Performance vs. Number of Boat at 1080p	77

Chapter 1 Introduction

This chapter provides a short introduction, the motivation and the objective to this research. It also gives an outline of the structure of this report.

1.1 Motivation

Water wave simulation, as one of the important categories of natural phenomenon simulation in computer graphics, is now popularly used in modern video games. Game players now expect to see realistic water environment varying from a small water puddle to a gigantic ocean in games they play. In order to provide more vivid and immersive feeling to players, the simulated water waves are expected to be interactive as well. It requires good physics simulation and high quality rendering for the water wave simulation algorithm.

Challenges come in with the modern game era. Nowadays the high-definition or HD video is becoming a common standard for video games in the market. HD video requires higher resolution than the standard-definition (SD) video. The HD video standard defines a minimal display resolution of 1280x720 pixels (720p) and with a future tendency of 1920x1080 pixels (1080i/1080p) [1]. This means in each frame the game program must update more than 900k pixels onto the screen. Today, video game system such as PlayStation 3 and Xbox 360 can output an HD signal. These game consoles can output display resolution up to 1080p. Both Xbox 360 and Playstation 3 games on the market are required to label their output resolution in HD standard on the back of

their package [2]. In addition to it, all video games are expected to have a minimal frame rate requirement between 30 and 60 frame / second [3]. This means in each second at least 30 frames are expected to have a glitches-free display result.

Both resolution and frame rate requirements for HD video games set the constraints to the researchers and game developers. It becomes a challenge to design a fast interactive water wave simulation algorithm to be practically used in the modern video games.

1.2 Objectives and Measurements

The objective of the research described in this thesis report is to propose a fast, interactive water wave simulation algorithm and framework to be practically used in the video game.

To demonstrate the effectiveness of the algorithm, a demo of boats moving in the water is developed with the algorithm.

There are three goals for this algorithm. It should be extreme fast, interactive and realistic.

The first two goals are easy to measure.

To demonstrate the algorithm is very fast, the demo is to be run on 720p or even 1080p resolution. A frame rate of at least 60 fps is expected. To set an even more accurate measurement by excluding non-relevant activities of the demo program, the exact amount of time taken for water wave simulation and

rendering are measured. The author of the report targets the total amount of time spent for water wave simulation and rendering in each frame to be less than 10ms under 720p resolution setting.

The interactive feature is easy to demonstrate as well. The boats in the demo will be user controllable and response to dynamic user inputs naturally.

Lastly comes to the realistic aspect of the algorithm, which is however not that easy to measure. Traditionally, the term realistic means the algorithm must be as close to the real water wave physics as possible. However, that will involve very complicated maths and physics calculation, making the algorithm very slow. As the speed and simulation accuracy of an algorithm are always conflicting requirements, the author of the report decides not to use a highly accurate physics model to simulation water waves. Instead only a basic physics model for the water wave simulation algorithm is chosen, to the extent that the result looks like water waves.

1.3 Contribution

The key contribution of the research is to provide a very fast simulation algorithm for interactive water waves of any size. The algorithm is able to be used in modern video game applications where HD standard resolution (720p or higher) and a fast frame rate are required.

The research also adapts a framework which is able to integrate various types of waves of different simulation approaches together for realism purpose. Such

framework can combine the advantage of each approach and produce a realistic simulation result with the minimal computational cost.

1.4 Overview

The thesis report is organized into the following chapters. Chapter 1 presents the motivation and objectives of the research. Chapter 2 provides some background knowledge of the relevant fields, which is essential to understand the thesis. Chapter 3 provides a literature review of the related research works done by other researches to date. Chapter 4 will be focused on presenting the author's algorithm and framework for interactive water wave simulation in both wave simulation and rendering aspects. The result and performance of the algorithm will be analyzed in Chapter 5. Lastly, Chapter 6 is going to conclude the research and provide some future work suggestion.

Chapter 2 Background

The solution described in this thesis requires some basic background knowledge in relevant fields. This chapter gives a brief overview of the relevant knowledge.

2.1 Overview of Water Waves

Water waves are surface waves that can be found in almost all kind of water. The water waves are usually the result of the manifestation of external forces acting on the fluid [4]. Such external forces could be a gust of wind, any solid objects that contracts with water, or even some far away apart but gigantic objects, such as the sun and the moon. Waves occur in all sizes and forms, ranging from ripples all the way towards swells, or even tsunamis. The sizes and forms of water waves depend on the magnitude of the forces acting on the water.

2.1.1 Physics Principle behind Water Waves

The external forces to interface with water tend to deform the water surface against the action of gravity and surface tension. Once the initial deformations are created, the gravitational and surface tension forces are activated to allow the waves to propagate. Eventually, the waves are stopped due to internal frictions (viscosity), external source of frictions such as seafloor, or the surface tension. The creation, propagation and break of water waves are actually a process of energy transfer. The energy is brought from other media into water

Chapter 2**Background**

via the external forces into the form of kinetic energy, and in the end consumed through internal or external frictions.

In the micro view of water waves, the propagation of waves is formed by the circular motion of water particles [5]. The motion of water particles is forward at the peak of the wave passes, but backward at the trough of the wave passes, and arriving again at the same position when the next peak arrives [6], under ideal condition where there is no frictions.

Figure 2.1 Circular Motion of Water Particles along a Wave [5]

In general, the larger the external force is, the more energy it brings to make water waves propagate faster.

The motion of water particles is also depth dependent. [7] When the water wave propagating in deep water, such as ocean, the water particles near the surface moves along a perfect circle. The orbital motion of fluid particles decreases rapidly with increasing depth below the surface. When the water wave propagates in shallow water, the particle trajectories however are compressed into ellipses. The elliptical movement of a fluid particle flattens when the depth increases.

Figure 2.2 Motion of Water Particles, A = Deep Water, B = Shallow Water [3]

The change of circular motion of water particles will affect the waveform shape of water wave. In shallow water, the elliptical movement of the wave will decrease the wave speed. The wavelength of the wave then becomes shorter. The height of the wave peak also increases. At certain level, the peak of the wave becomes unstable. It moves faster than the water below, which eventually collapses and creates breaking waves.[6]

Figure 2.3 The Difference of Deep Water Waveform and Shallow Water Waveform due to Change in Motion of Water Particles

2.1.2 The Shape of Water Waves

The shape of a single water wave is often thought as a sine wave. However, to be more accurate, the experimental shape of a wave is described as a trochoid [6]. A trochoid can be defined as the curve traced out by a point on a circle as the circle is rolled along a line.

Figure 2.4 Comparison of a Trochoidal Curve and a Sine Curve [8]

When the amplitude of wave is small, the shape of trochoid curve looks almost the same as sine curve. When the amplitude increases, the peaks of the trochoid will become steeper and narrower compared to sine wave.

From the Bascom's experiment[8], it is suggested that the ratio of peak height to wavelength of a deep water wave is at a ratio of 1:7 for the maximum and an angle of 120° is the minimum angle for a peak.

The trochoid curve can be formulated by parametric equations:

$$\begin{aligned} x &= r_b \theta - r_a \sin(\theta) \\ y &= r_b - r_a \cos(\theta) \end{aligned}$$

, where θ is the variable angle through which the circle

rolls, r_a and r_b are the diameters of two circle A and B. [9] Usually $r_a < r_b$ is used.

However in practical, there are always too many single waves with different amplitude and wavelength inside water. What is actually observed is a

superposition of a large number of waves with various of frequency and wavelength propagating towards different direction. A possible waveform been recorded in actual water could be like the following diagram.

Figure 2.5 Example of actual Waveform recorded in Water [4]

2.1.3 Wave Parameters

Figure 2.6 Wave Characteristics[4]

A two-dimensional schematic of a water wave propagating in the x direction illustrated above can be characterized by the following basic parameters:

- Wave height, or amplitude, the distance from trough to crest, denoted as H
- Wavelength, the distance between two consecutive crest, denoted as L or λ
- Period, the time interval between arrival of consecutive crests at a stationary point, denoted as t
- The direction of wave propagation
- The water depth, over which they are propagating, denoted as h

2.1.4 Characteristics of Water Waves

Like other kind of waves, water waves preserve a number of wave characteristics:

- Periodic: ideal wave perform periodic up and down motion in addition to water particles' circular motion. It is characterized by crests and troughs.
- Dispersion: wave splitting up by frequency. The water waves of different wavelength travel at different phase speeds.
- Reflection: the wave propagation direction changes when hitting a reflective surface. The reflecting direction is symmetric to the propagating direction with the reference of boundary normal.
- Interference: superposition of two waves that come into contact with each other. The composite waveform is simply the summation of two individual waveforms of the two waves. After they pass through each

other, the two waves still carry their own amplitude, frequency and velocity.

Figure 2.7 Superposition of two Waves

2.1.5 Classification of Water Waves

Von Arx [5] classified two major types of water waves based on the dominating dynamic forces. The two types are gravity waves and capillary waves.

Gravity waves, as can be guessed by its name, are mainly controlled by gravity and inertia force. The wave speed of gravity waves increases with wavelength.

For waves with wavelength shorter than 1.73cm, von Arx described them as capillary waves. These waves are dominated by the surface tension. They propagate faster at shorter wavelength.

Figure 2.8 Relationship between Wave Speed and Wavelength[5]

The gravity waves receive more concern because they are more significant in amplitude and speed. They are further classified into deep water, intermediate water and shallow water according to the depth of the wave.

Deep water is defined when the depth-wavelength ratio $\frac{h}{\lambda} > \frac{1}{2}$.

Shallow water is defined when the depth-wavelength ratio $\frac{h}{\lambda} < \frac{1}{20}$.

Intermediate water sits in the gap in-between.

The wave speed, or called celerity, of an idealized gravity water wave is wavelength and depth dependent. [6] The wave speed relationship with these two parameters is:

$$v = \sqrt{\frac{g\lambda}{2\pi} \tanh\left(2\pi \frac{h}{\lambda}\right)}, \text{ where } \lambda \text{ is the wave length, } d \text{ is the depth of water}$$

and g is the acceleration of gravity.

For deep water, the wave speed is approximated as $v \approx \sqrt{\frac{g\lambda}{2\pi}}$, for $d > \frac{\lambda}{2}$

For shallow water, the wave speed is approximated as $v \approx \sqrt{gh}$, for $d < \frac{\lambda}{20}$

There are further classification of water waves based on the wavelength, such as chops, wind waves, ocean waves and swells. There are also many special waves in shallow water, such as different breaking waves (spilling, plunging and surging) and surf. These details are not closely relevant to this research, therefore not discussed in this background section.

2.2 GPU and GPU Programming

Today, every computer has a GPU (Graphics Processing Unit) dedicated to provide high performance and visual rich graphic content. With the continuously demand drift of video games and the technology advance of GPU manufacturers, nowadays the raw computational power of a GPU is better than that of the most powerful CPU for handling graphics related tasks [10]. Modern GPUs also replace the traditional fixed-function graphics pipeline with a flexible computational engine, such that users can use GPU programming tools to achieve tremendous visual effects.

2.2.1 GPU

While CPU (Central Processing Unit) is a generalized processor to handle all kind of tasks, GPU (Graphics Processing Unit) is a specialized processor that offloads graphics related tasks from CPU.

There are two major aspects that makes GPU a faster device than CPU for graphics related tasks.

Firstly, GPUs are specially designed for floating points operations. It contains many floating point based registers than CPU dedicated for arithmetic operations. A GPU also incorporates custom microchips which contain specially optimized mathematical operations and graphics related operations in a way that they run much faster than similar operations run in CPU.

Secondly, GPUs have a parallel architecture, which adapts the SIMD technique. SIMD stands for Single Instruction, Multiple Data. It is a technique to process a batch of data from the data pool simultaneously with the same instruction from instruction pool. The basic unit of SIMD technique contains four processing units. The basic SIMD arrangement is called “vector” instruction, which process a vector of four data at a time. The SIMD technique perfectly matches the graphic related tasks. Most of the 3D graphics related parameters, such as 3D position, direction (3 values for x, y & z axis and the 4th value is set to 1.0 in most cases) or color of RGBA format, are in a vector format of four floating points. For graphics related activities such as coordinate transformation, interpolation, texture fetching, shading or alpha blending, each element in the vector will be processed with exactly the same arithmetic operations. In actual cases, GPU will have many SIMD units cascade together to have more powerful batch data processing ability.

Figure 2.9 The Basic Architecture of SIMD

2.2.2 Fix-function Graphics Pipeline

Graphics pipeline refers to the method GPU uses to perform any rendering related activities. It is a series of steps from inputs of 3D vertex data all the way towards the output of a frame buffer to be drawn onto the display screen.

Old GPU technology adopted a fix-function graphics pipeline, in which the pipeline flow was fixed and not changeable. Only a limited number of render states can be controlled by users.

The fix-function graphics pipeline flow is illustrated in Figure 2.10. A brief introduction of each step along the pipeline can be helpful to understand the function of GPU.

Figure 2.10 Fix-function Graphics Pipeline [11]

When the vertex data are input to the GPU, the first step is the Transformation & Lighting (T&L). In T&L, two tasks are delivered. Firstly, vertices are transformed from local coordinate system into projection coordinate system related to camera view. Secondly, per-vertex lighting is done to provide illumination for each vertex.

Next come to Primitive Assembly, where individual vertices are assembled into primitives such as triangles according to their relationship. Primitive Assembly can output a collection of triangles representing surfaces of models.

Usually most of the triangles are not visible in the current viewport of the camera. It is therefore very necessary to discard those invisible triangles to increase rendering speed. The Viewport Culling & Clipping is to handle this task. Culling and clipping are different processes. Culling is to remove the

triangles totally not visible. Clipping is to handle the triangles partially visible, where it divides those triangles into parts and keeps only those visible parts.

In the next step, the Rasterizer Setup, vector based triangles are converted into raster format with the parameters for each pixel interpolated.

In the following steps, Texture Blending is to do the texture content fetch, while Per Fragment Operations is to handle some other pixel related task such as alpha blending. After these two steps, the final colors of all pixels are determined.

Once all the pixel values are determined, the Frame Buffer Operations will transfer these output color onto the frame buffer, and eventually displayed on the screen.

2.2.3 Programmable GPU Pipeline

Though simple and easy to use, the fixed-function graphic pipeline can only produce very simple graphic effects due to the limited number of configurable render states. It is no more capable nowadays especially with the increasing demanding on dazzling graphic effects from criticized video game players. Luckily, modern GPU provides programmable GPU pipeline feature, which allows programmers to alter certain stages of graphics pipeline with customized methods. The revolutionary new feature makes many complex graphics effects been realized in the video games.

Two major programmable processors available to replace the units of fixed-function pipeline are Vertex Processor and Fragment Processor[11], as shown

in the below figure. Vertex Processor allows customized T&L method implemented, providing a way to modify the attributes of each vertices. Fragment Processor alters Texture Blending and Per Fragment Operations, to replace them with a customized fragment processing pipeline. [12]

Figure 2.11 Fix GPU Pipeline (in white) and Programmable GPU Pipeline (in blue)

Vertex Processor and Fragment Processor are programmed through a set of software instructions called Shader. There are two major types of Shader to correspond to the two processors. Vertex Shader is used for Vertex Processor. Fragment Shader, which also commonly called as Pixel Shader, is used for Fragment Processor.

Both Vertex Shader and Pixel Shader are actually a set of software instructions, which is used to calculate rendering effects based on the GPU architecture with high degree of flexibility. Shaders are well adapted into the GPU architecture,

Chapter 2

Background

making good use of optimized instructions available in GPU microchips through intrinsic functions to achieve fast throughput for graphics related tasks. Shader is also a parallel processing programming language with the use of SIMD concept.

Chapter 3 Related Work

Throughout the years, researchers have contributed their ideas with various of solutions for water waves simulation. This chapter provides a general review of these related works.

In general, there are four board categories of approaches for water wave simulation. They are particle system approach, procedural approach, spectral approach and height-field approach.

3.1 Particle System Approach

The particle system approach is a classical method theoretically suitable for simulating not only water waves, or fluids, but also all other kind of fuzzy objects. The method was first introduced by Reeves [13]. He defined the particle system as a collection of many minute particles that together represent a fuzzy object. In such a system over a period of time, particles were generated into the system with a set of attributes, moved and evolved within the system, and died from the system. Typical attributes for particles are position, color, velocity, size, transparency, shape and lifetime, etc.

There are two major solutions based on particle system approach for water simulation, the particle-based Lagrangian solution and the Eulerian grid-based solution. Both of the methods are trying to solve the Navier-Stokes equations [14], which describes accurately the motion of fluid, to update the fluid physics.

3.1.1 Lagrangian Solution

Lagrangian solution, or called gridless particle solution, considers the physics interaction at individual particle level. The position of particles can change. Different particles could occupy the same position as well. Some of the methods based on Lagrangian solution were smoothed particle hydrodynamics [15], use a special type of particle, surfels to represent surface [16, 17], and moving particle semi-implicit method [18]. These applications could provide very realistic water simulation result of complex phenomenon in high details. However, they required a large number of particles in order to achieve high quality results. Lagrangian solutions suffered extreme expensive computational cost, which prevented them from being used for real-time applications.

3.1.2 Eulerian Solution

Eulerian solution considers particles in a different way. It divides the simulation space into many 3D or 2D grids. Each cell or voxel in the grid-based space then become a particle. These particles in Eulerian solution cannot change their position, but to exchange parameters with each other. [19-21] In Eulerian solution, the number of particles is fixed, and usually much less than Lagrangian solution. With the reduction of the number of particles, the Eulerian solution is faster than Lagrangian solution, however, with degradation in simulation quality.

One of the important improvements in grid-based Eulerian solution is Stam's solution[18]. He integrated the feature of Lagrangian solution into the

Eulerian grid space, by creating a parameter called density. In his implementation, inside each voxel there is a density, representing the number of particles available in that voxel. All particles within a voxel share the same value for parameters such as velocity, heat, etc. Stam also defined the special steps on these densities, such as initial density, update force, density self-diffusion and move density. This density parameter can reduce the grid resolution, thus reduces the number of particles in the system, making the solution possible to work in real-time applications.

Figure 3.1 Illustration on the Steps to Process the Density in Stam's Solution [21]

Some other improvements based on grid-based Eulerian solution tried to further reduce the computational cost, such as using an octree structure for grid-space management [22] , or applying tetrahedral meshes to adapt the level of detail [23].

3.1.3 Non-Navier Stokes Solver Solution

No matter using Lagrangian or Eulerian solutions, the non-linear Navier-Stokes equations are still expensive to solve, and not easy to be implemented

on GPU architecture. As a GPU adaptable solution requires a linear equation system, some researchers provided approximation methods from Navier-Stokes equation. One of those methods is Lattice Boltzmann method. [24]

Similar to grid-based Eulerian solution, Lattice Boltzmann method (LBM) also separates the simulation space into 3D grids, named as Lattice nodes. Each Lattice node has a few packet distributions, denoted as f_{qi} , each associates with a velocity vector, denoted as e_{qi} . Among all velocity vectors in each node, one of the velocity vectors is connected to the node itself, and the others are connected to the neighboring nodes. In 2D space, there are 9 packet distributions and velocity vectors in total (D2Q9). In 3D space, there are 19 packet distributions and velocity vectors (D3Q19).

Figure 3.2 Illustration of D2Q9 and D3Q19 Lattice Node Model [21]

The density of a Lattice node, ρ and the velocity of a Lattice node u are calculated with formula:

$$\rho = \sum_{qi} f_{qi}$$

$$u = \frac{1}{\rho} \sum_{qi} f_{qi} e_{qi}$$

The LBM updates the packet distribution values at each node based on two rules. Collision rule describes the redistribution of packets at each node. Streaming rule describes the packet distributions move to the nearest neighbour along the velocity direction.

Collision Rule: $f_{qi}^{new}(x, t) - f_{qi}(x, t) + F_i = \Omega_{qi}$

Streaming Rule: $f_{qi}(x + e_{qi}, t + 1) = f_{qi}^{new}(x, t)$

where Ω_{qi} is a general collision operator and F_i is the external force.

The linearity feature of LBM made it suitable to be fit into GPU programming, making the processing speed potentially faster than the grid-based Eulerian solutions of Navier Stokes equations solver. However, this method could use a lot of GPU memory storage, as a 3D simulation requires 1 copy of memory volume for packet distribution and 19 copies of memory volume for velocity vector. It is not suitable to simulate large scaled water area.

3.1.4 Summary of Particle System Approach

Although particle system approach can simulate water in a very realistic and complex way, it is a very expensive method. Even the optimized LBM method can merely simulate a middle scaled water area in real-time at around 15 fps with the acceleration of a GPU [24]. The approach is more suitable for offline simulation.

3.2 Spectral Approach

The spectral approach is also referred as Fourier synthesis. It reconstructs water surface waves from a series of frequency function defined in frequency domain.

3.2.1 Basic Ideas of Spectral Approach

Mastin, together with other researchers, first introduced the spectral approach to water wave simulation in 1987 [25]. They used data of pre-measured spectral properties of a real ocean wave to synthesize ocean wave.

In their solution, the information of ocean wave in frequency domain, was first added by some white noise, and then filtered according to certain wave model with Pierson-Moskowitz spectrum, and lastly went through IFFT (Inverse Fast Fourier Transform) to convert it back to spatial domain. The result was a height map, which then tiled up seamlessly to form the entire water surface area.

Figure 3.3 Flowchart for Mastin's Spectral Approach

With the contribution of Perlin noise function [26], it is then possible to use a software generated data instead of real data to perform real-time water wave simulation. [27]

Tessendorf described a similar solution [28] to Mastin's, but used Phillips spectrum rather than the Pierson-Moskowitz spectrum. He also proposed a few enhancements to tune the spectrum through dispersion relation to influence the direction and speed of water wave field, reflecting the change of wind.

3.2.2 Multi-Band Spectral Approach

In 2005, Mitchell [29] presented a multi-band spectral approach to simulate ocean water based on Tessendorf's method. The special feature in Mitchell's solution was that it supported multiple frequency domain wave profiles as inputs.

In their solution, two copies of profiles are taken as inputs. The low band contained the low frequency portion of the ocean wave information, representing the gravity waves. A low resolution height map was generated for vertex displacement on mesh. The broad band contained the entire frequency band of ocean wave information, representing the superposition of gravity waves together with capillary waves. A high resolution height map was generated for pixel shading.

Figure 3.4 Concept for Multi-Band Synthesis of Spectral Approach

The advantage of the multi-band synthesis solution is to make it possible to have a low resolution polygonal mesh to represent the surface waves. It is actually a level-of-detail scheme to get high quality result with minimal computation cost and memory consumption.

3.2.3 Summary of Spectral Approach

Spectral approach is a simple and very fast water wave simulation approach suitable for real-time application. The cyclic feature (referred by [30]) of FFT makes it possible to generate a small sized simulation texture, and to be used for texture tiling. It is very suitable to represent large area of water surface with very low computational cost and memory consumption.

However, the approach is not capable of simulating waves caused by water-object interaction, due to the nature of the approach that uses pre-sampled data used as inputs. As a result, the approach is not applicable for interactive water wave simulation. As the simulation result is a height map, it is also not possible to simulate complex water wave effects where horizontal displacement of wave exists, such as shallow water waves, or even breaking waves.

3.3 Procedural Approach

Procedural approach refers to the process to compute a particular simulation function of time. The result of the simulation is a collection of 3D positions, which is to be used for vertex displacement on the 3D simulation grid.

3.3.1 Basic Ideas of Procedural Approach

In 1986, Both Fournier and Reeves [31] and Peachey [32] first suggested the procedural approach to simulate a large range of wave shapes and phenomena for ocean waves.

In Fournier and Reeves' solution, they simulated the ocean waves as a series of trochoid based on Gerstner Wave Model where particles of water described circular or elliptical stationary orbits.

The basis equation for Gerstner Wave Model used in their solution was simply

$$x = x_0 + r \sin(\kappa x_0 - \omega t)$$

$$z = z_0 - r \cos(\kappa x_0 - \omega t), \text{ with the assumption that wave only propagated}$$

from one direction (along x-axis).

The good aspect for using the Gerstner model is that the characteristics of the water waves can be easily controlled by controlling the parameters in the equation. The height of the wave is $H = 2r$, the wavelength is $L = 2\pi/\kappa$, the wave period is $T = 2\pi/\omega$ and phase angle is $\varphi = \kappa x_0 - \omega t$. The factor κ can be adjusted according to user requirement. Fournier and Reeves actually suggested a value of κ times r to result in bumpier wave shape. The steepness of the wave is then described by $\delta = H / L = \kappa r / \pi$.

Figure 3.5 Different Combination of κ and r which affecting Shape of Wave [31]

Fournier and Reeves even made some modifications to the Gerstner wave equation to simulate the crest of wave, as well as to associate the ocean depth with the height of the waves. In this way, their solution could also simulate shallow water waves with realistic wave shapes.

Figure 3.6 Result of Shoreline Waves on a gently Slopes shown in Fournier and Reeves' Solution [31]

The procedural approach introduced by Fournier and Reeves was simple and straightforward. Later a lot of researchers had developed improvements based on their solution.

Tessendorf [28] suggested that instead of using Gerstner wave model, which is a single sine wave profile, using the summation of a set of sine waves would be more appropriate to simulate the ocean wave. He suggested to use the following equations to produce a simulation result to combine waves from a broadband wavelength.

$$x = x_0 + \sum_{i=1}^N r_i \sin(\kappa_i x_0 - \omega_i t)$$

$$z = z_0 - \sum_{i=1}^N r_i \cos(\kappa_i x_0 - \omega_i t)$$

Ts'o and Barsky [33] used a Beta-spline function to construct contour lines with tension shape parameter to simulate the water wave. Their method could

easily add more complexity to the wave surface which appears to be more realistic.

A lot of research has been done to get wave equations suitable for simulating different wave effects as well. For examples, Gamito et al. [34] proposed an equation suitable for rendering wave refraction in shallow water. Hinsinger et al. also implemented a real-time simulation for deep ocean[30]. With sophisticated simulation functions designed, researchers such as Szécsi et Arman could even simulate a entire wave series along the shoreline with different type of waveforms including waves from deep water, intermediate water and shallow water [35].

3.3.2 Summary of Procedural Approach

Procedural approach is suitable for real-time simulation of large area of unbounded water surface, such as open sea. Theoretically, the approach can best simulate any kinds of wave type with high quality, provided a realistic wave equation is used. The disadvantage of the method is that it relies on the vertex displacement technique. To have a wave simulation with high details including low wavelength capillary waves, a simulation grid with very high resolution is required. In this case, even the fastest GPU cannot process the vertex processing to meet game applications requirement. In other words the method is still not fast enough if we want to simulate wave in high details.

To help solving the problem, some researchers worked out a solution to combine the feature of procedural approach and spectral approach[36]. In their solution, the Gerstner wave model is used to simulate gravity waves,

which were added with a Perlin turbulence for the capillary waves simulation. The solution would only require a relatively low resolution grid to simulate waves in high detail. Other researchers had suggested an adaptive surface mesh scheme [30]. Their idea was to generate mesh dynamically in which the resolution of the grid was based on the size appeared on the viewing screen. The solution was actually a level-of-detail structure which could minimize the number of vertex to be manipulated in GPU for any particular time frame. The procedural approach could even be combined with particle system approach, which was amazingly introduced by Yuksel et al. [37], where the particles are created, propagated and removed on a 2D surface, and Gerstner wave model is used for procedural method. This solution could perform very good results of water interaction in real-time.

The other thing not so good about procedural approach is that it is difficult to model fluid-object interaction.

3.4 Height Field Approach

In fact, the height field approach should belong to the procedural approach as a simplified version. Both of these approaches are based on some time domain wave functions. The difference, however, is that in procedural method, each vertex on the polygonal mesh can move along x-, y-, and z-axis. In height field method, the vertex can only move along one direction, the z-axis, with its position on x- and y-axis to be fixed. The method can use a two-dimensional height map to represent surface waveform information, which is so called a 2.5D simulation.

3.4.1 Basic Ideas of Height Field Approach

In 1990, Kass et Miller [38] derived a simplified 2D water wave equation on a dynamic height field, as shown below:

$$\frac{\partial^2 h}{\partial t^2} = gd \left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} \right) \quad (1)$$

The parameter g was the gravitational acceleration, h is the height of wave, and d is the depth of the wave.

The equation simply told that the wave up and down acceleration was proportional to how quickly the steepness of the water surface was changing. The equation was an approximation from the Navier Stokes equations. It was a second order linear partial differential equation, which was easier to solve and compute. From the equation, for every 2D position (x, y) , we could compute the wave height at a particular time t . This led to a height field updated every time frame to represent the entire simulated water surface. The height field presented in a height map texture was used for vertex displacement and pixel rendering techniques.

Chen et de Vitoria Lobo [39] also provided similar method by defining a height field derived from 2D solution of Navier Stokes equation, which was used to simulate interactive wave.

3.4.2 Implementation using Vertex Texture Fetching in GPU

The publishing of Shader model 3.0 integrated in DirectX 9.0c by Microsoft in 2004 made the method even more popular. Shader model 3.0 supported

vertex texture fetching technique. Vertex texture fetching allowed a vertex shader to use the pixel information of an input texture to manipulate vertices. With the support of vertex texture fetching, height field method could cooperate with GPU programming, greatly increased the processing speed from traditional CPU programming.

The nVIDIA technical support notes 2004 implemented the height field approach with vertex texture fetching technique for water wave simulation [40]. The solution was based on a simplified version of Kass and Miller's equation:

$$\frac{\partial^2 h}{\partial t^2} = c^2 \left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} \right) \quad (2)$$

Firstly, the right hand side of the equation (2) is to be solved. In discrete domain, the second order linear partial differential equation can be solved by performing the following operations of a selected pixel with its four neighbours.

$$c^2 \left(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} \right) = c^2 (y_{-1,0} + y_{1,0} + y_{0,-1} + y_{0,1} - 4y_{0,0}) \quad (3)$$

In equation (3), $y_{0,0}$ represents the wave height value for the selected pixel, while $y_{-1,0}$, $y_{1,0}$, $y_{0,-1}$ and $y_{0,1}$ represents the wave height value of the neighbouring pixels to the left, right, top and bottom.

Figure 3.7 Illustration on how the up and down Acceleration of Water Wave is calculated from the Pixels on the Height Map Texture

Next, we solve the Newton's Law which computes the position of a point $p(t_2)$ at a time frame t_2 from the velocity $v(t_1)$ and acceleration $a(t_1)$ from the previous time frame t_1 . The equation is:

$$p(t_2) = p(t_1) + v(t_1) \cdot (t_2 - t_1) + \frac{1}{2} a(t_1) \cdot (t_2 - t_1)^2 \quad (4)$$

The velocity $v(t_1)$ in equation (4) is still unknown, which can be further expressed by equation (5), where t_0 is the previous time frame from t_1 :

$$v(t_1) = \frac{p(t_1) - p(t_0)}{t_1 - t_0} \quad (5)$$

Both equation (4) and (5) make assumption that the time interval between two time frames is short enough, i.e. $\Delta t \approx 0$. Let's put equation (5) into equation (4), we get the following derivation:

Chapter 3**Related Work**

$$p(t_2) = p(t_1) + \frac{p(t_1) - p(t_0)}{t_1 - t_0} \cdot (t_2 - t_1) + \frac{1}{2} a(t_1) \cdot (t_2 - t_1)^2$$

$$p(t_2) = p(t_1) + \frac{p(t_1) - p(t_0)}{\Delta t} \cdot \Delta t + \frac{1}{2} a(t_1) \cdot \Delta t^2$$

$$p(t_2) = 2 \cdot p(t_1) - p(t_0) + \frac{1}{2} a(t_1) \cdot \Delta t^2 \quad (6)$$

The resultant equation (6) suggested that the new wave height at time frame t_2 is related to its current height at t_1 , its previous height at t_0 , and the acceleration at t_1 .

The most unrealistic part of the equation (6) is that there is no internal resistance force considered. The wave simulated using equation (6) will never settle down. To force the wave settle down, nVIDIA's solution uses a so called dampening factor to simulate the water internal resistance, which gradually reduces a portion of the height of wave, and eventual makes the wave stable. They suggested a dampening factor set of 1.99 and 0.99, which changes the equation (6) to:

$$p(t_2) = 1.99 \cdot p(t_1) - 0.99 \cdot p(t_0) + \frac{1}{2} a(t_1) \cdot \Delta t^2 \quad (7)$$

nVIDIA's solution is based on equation (3) and (7) to simulate a boat interacting in a small pool of water, with assumption of no other external force interacting with the water.

In their implementation, three height map textures reflecting height status of the current, the previous and the next time frame are used. In each simulation iteration, the heights from height map representing current and previous time

frame are fetched, and to generate the height map texture for next time frame using equation (3) and (7) . An additional step is applied to update the new interaction between water and boat, which is simply drawing a so called perturb brush texture on the height map. This process is not physically accurate, but is an approximation to get fast but nice looking results. At the end of the iteration, there is a time frame update step. With the step, the next time frame height map becomes the current time frame height map, the current one becomes the previous one, while the previous one retires, and is going to be used to hold the new height status in the next iteration.

The below figures provide a graphical illustration on how the method works. The white square represents the update of water-boat interaction. The yellow circles represent the resultant water waves generated by the method.

Figure 3.8 Illustration on nVIDIA's Solution, how the 3 Height Map Textures are updated for Water Simulation

Figure 3.9 Illustration on nVIDIA's Solution, how the 3 Height Map Textures are updated for Water Simulation (Continued)

nVIDIA implemented their solution in a demo with a boat interacting in a small water pool, which could get a very fast result of interactive water wave simulation.

Figure 3.10 Result of nVIDIA's Solution [40]

3.4.3 Summary of Height Field Approach

Height field approach is also an approach suitable for real-time water simulation. The most important advantage of this approach is that it is good at simulating water-object interaction. Although it is not a physically accurate model, the simulation result looks quite realistic with a high resolution height field and well designed rendering algorithm.

However, there are two disadvantages of this approach.

Firstly, the approach is not suitable to simulate very large water surface area. With the increase of simulation size, resolution of height map texture is increased as well. The computational time for height field approach will increase significantly. It is very expensive to maintain three copies of height map with high resolution in both memory concern and speed concern. A pure height field approach is only able to simulate small water surface area, such as ponds, small rivers and water puddles.

Researchers came out some hybrid solutions to combine height field approach with procedural approach to simulate a large area of water surface [41]. However, their solution targeted close physics simulation of water, which was not able to run in real-time applications.

In addition, the height field approach, similar to spectral method, cannot be used to simulate complex wave phenomenon such as breaking wave. There is no way to perform horizontal vertex displacement using the height field alone. In order to simulate those effects, researchers have adopted some hybrid solutions based on the height field approach. One of such examples is to use a steep wave detection algorithm [42] to detect the crest of the waves. After that, some connected particles are added to modify the structure of wave meshes near the crest, which forms the shapes of breaking waves.

3.5 Summary on the Related Work

With a general review of related work done on the water wave simulation, we can get a clear picture of the advantage and disadvantage for each approach. To meet the requirement of the research presented by the report, in which the speed and visually realistic water-object interaction are the main concerns, the height field approach is the most appropriate. The most significant drawback of height field approach is that it requires very high resolution height map for large scaled water.

We also observe that many researchers prefer hybrid solutions which combine two or more basic approaches. Those hybrid solutions are able to use the advantage of one approach to remove the side effect from the disadvantage of

Chapter 3

Related Work

the other approaches. Under such a concern, a hybrid solution mainly based on height field approach is preferred to fulfil all the requirements in this research.

Chapter 4 Design Fast Interactive Water Simulation for Games

This research aims to create a very fast water simulation solution featuring water-object interaction. The water simulation solution provided by this report is a hybrid solution mainly based on the height field approach. To be more particular, the solution is based on the nVIDIA's solution [40], which has been discussed in Section 3.4.2. The reason to use height field approach is that it is the fastest and the best approach among all we have reviewed, to simulate water-object interaction.

Of course, the height field approach is not a physically accurate approach, especially compared to procedural approach and particle system approach. However, the accuracy of physics model is not the primal concern in video game applications. Game players only expect a water wave simulation looks realistic, while whether to use the true physics or not has little matter to their game play experience. Based on this concept, this research does not consider a physically correct model water-object physics interaction. The major concerns are the simulation speed and water-object interaction, followed by the visual appearance of simulation.

Chapter 4 Design Fast Interactive Water Simulation for Games

4.1 Interactive Water Wave Simulation

The interactive water wave simulation only considers how to generate water waves from object-water interaction into a height map.

4.1.1 Analysis on the nVIDIA's Height Field based Solution

nVIDIA's solution keeps on updating three height map textures. Each texture contains the height (in one channel) and the normal (in three channels) of the simulation grid. The height is used in vertex shader for vertex displacement. The normal is used in pixel shader for rendering.

Two major problems prevent nVIDIA's solution from being practically used in video games, where the size of water can be large.

Firstly, three 128x128 height map textures used in nVIDIA's solution is only able to simulate a very small scaled water pool. It must be improved so that interactive simulation on large scaled water, but still with high speed, is possible.

Secondly, the solution is not efficient in memory usage, which can still be optimized.

4.1.2 Assumption of using Height Fields on Large Scale Water

In order to make the height field approach work on large scaled water, the research is based on an important assumption that the resultant water waves from object-water interaction only propagate within a limited area with reference of the object.

Chapter 4 Design Fast Interactive Water Simulation for Games

With this assumption, no matter whatever scale the water has, only the area surrounding to the object requires height field for simulation. In this way, we can use small sized but high resolution height map textures to simulate waves of a large scaled water area.

The assumption is correct for a slow motion object inside water, as the waves caused by the object will be soon stopped by the water internal friction (viscosity).

However, it seems the assumption does not work well when the object inside water is in fast motion. In that case, the object will be further away from the waves it creates, who just propagate a little distance apart from its initial position. Typical phenomenon of such case is a fast moving boat in the water which produces a very long tail of waves.

To settle this case, a trade-off must be made in the research. In one hand, we try to make the height field texture large enough (in terms of resolution and actual dimension in the game scenes) to cover as many surrounding area as possible. This is to give enough space for the above mentioned long tail effect. In the other hand, those areas not able to be covered by the height field texture should be discarded by all means. We do not show them even though they should appear accordingly to the wave physics.

This trade-off setting is actually quite suitable for video game applications. For example, in cases when a player is controlling the big penguin shown in Figure 4.1, he can only see a certain range of surrounding water area from the object he controls. The long tail waves are not observable. For those areas do not

Chapter 4**Design Fast Interactive Water Simulation for Games**

appear on the screen, no simulation is required. The other case is for AI controlled objects, such as the small penguin in Figure 4.1 who is ahead of player. Unrealistic long tail waves may be observed if we discard those parts which are out of the height field range. However, a player might pay more attention to those moving objects, thinking of strategy on how to over take or get ride of them, especially when these objects are moving very fast. Very little players would rather notice the waves caused by those objects. In other word, those not so realistic waves would still be tolerable if they are not very important in the game.

Figure 4.1 Screenshot from video game Surf's Up (2007)

4.1.3 Adhesive Type of Height Field

Based on all these concepts and assumptions, a set of medium sized height map textures containing height fields is constructed for every object able to interact with water. These textures are attached with the object and always move together with it. The author of the report called this type of height fields

Adhesive Type, in comparison of those traditional Static Type height fields that never move.

The solution does not apply the dampening factor as shown in Equation (7) of Section 3.4. To have a more flexible control, a dampening texture is multiply to the resultant height field, the dampening factor of which only increases significantly near the circular boundary.

Figure 4.2 Dampening Texture

During simulation with Adhesive Type height fields, the original two steps in nVIDIA's solution, the creation of initial waves by drawing perturb brush texture, and the propagation of wave through wave equations we discussed in Section 3.4.2 are performed as usual, with one more translation step added.

The simulation height field textures always move together with the objects, while in nature, the waves do not move together with the object who creates them. So the translation step is to move the height fields back to the correct position in the world. The distance of translation equals to the distance that object moves in the same time frame, while the direction of translation is along the reverse direction of translation vector of the object, according to the relative movement law.

Chapter 4 **Design Fast Interactive Water Simulation for Games**

Figure 4.3 Illustration of the Translation Step for Adhesive Type of Height Fields

The translation step of height fields is done in pixel shader, by translating the texture coordinates to a new value reflecting the change of object position. Some special concerns should be taken.

- Discrete Domain Data Shift Issue

The first concern is that as the height fields are stored in discrete domain of height map textures. During the translation, it is important to make sure that the height fields are shifted exactly an integer number of pixels.

It can be easily done by rounding the x- direction and y-direction components of the displacement vector, based on the dimensions appear in the world for a pixel. For example, if the x-direction component of displacement is 4.5 pixels, it will be rounded to 4 pixels. However, it is not a good solution. With this rounding strategy, the cases of displacing 4.1 pixels, 4.3 pixels, 4.5 pixels, etc will all be rounded to 4 pixels. When the velocity is increased to the level such that the displacement is 5.0 pixels, the

Chapter 4 Design Fast Interactive Water Simulation for Games

resultant wave will suddenly be displaced by 5 pixels. There is no smooth transition in-between and the simulation result looks abruptly.

A much better rounding strategy used in the solution provided by this research is to use an accumulative parameter. Every time during the rounding, the remainder will be accumulated. If the accumulated value exceeds 1.0, then one more pixel will be shifted. For example, if the previous accumulated value is 0.8, and we get 4.3 pixels to displace. The actual number of pixels to displace is 5 pixels, with $0.8 + 0.3 - 1.0 = 0.1$ pixels accumulated for future use. This rounding strategy can get a smooth wave simulation.

- **Texture Sampling Issue**

The second concern is the texture sampling. The translation step is a simple one-to-one mapping, as the input and output height map textures are of the same resolution. However, during the time when data are processed from vertex shader into pixel shader, the values of texture coordinates might be changed in rasterization process. Most probably they are no longer at the center of a pixel, but in somewhere near its four neighbouring pixels. Texture sampling, the strategy that pixel shader fetches the most suitable pixels from input texture and maps them onto the output texture, is required then.

Pixel shader has two basic sampling approaches to decide the pixel value. Nearest-point sampling takes the value from the nearest pixel, while Bi-

Chapter 4 **Design Fast Interactive Water Simulation for Games**

linear sampling performs a bi-linear interpolation from all four neighbouring pixels to get the new interpolated value.

Figure 4.4 Illustration of the Nearest-Point Sampling and Bi-Linear Sampling

As we need to get the exact height field information in the translation step, nearest-point sampling must be used. Using bi-linear sampling however will cause the simulated waves shift to the right, which is definitely not desirable.

4.1.4 Optimization on Wave Simulation

nVIDIA’s solution itself is not an optimized solution in terms of memory usage and processing time. When the translation step is added to the solution, it further increases the complexity of the algorithm. An optimization on both memory and time consumption is required.

Chapter 4**Design Fast Interactive Water Simulation for Games**

nVIDIA's static height field solution requires 3 copies of height map textures, to represent the height field status for previous, current and next time frame. Each texture contains 1 channel of height data and 3 channels of normal data.

The pseudo code for nVIDIA's solution is:

Procedure: StaticWaveSimulation()

SetRenderTarget(texture[nextTimeFrame])

ClearRenderTarget(texture[nextTimeFrame])

**PerformWaveSimulation(texture[currentTimeFrame],
texture[previousTimeFrame])**

DrawBrushTexture (brushTexture)

Update time frame together with texture ID

End of Procedure

A little explanation is need for the above pseudo code. SetRenderTarget() sets the render target, which is basically a texture to be updated by shader, ClearRenderTarget() will erase the content in a texture who is selected as render target. PerformWaveSimuation() calculates the wave equation with the two height map textures as inputs, and outputs the height field for next time frame in the render target. DrawBrushTexture() will draw the brush texture onto the render target.

After adding the translation step for height map textures of current and previous time frame, the pseudo code becomes:

Procedure: AdhesiveWaveSimulation()

Calculate the number of pixel to shift, and update the accumulate value

Chapter 4**Design Fast Interactive Water Simulation for Games**

```

SetRenderTarget( texture[temp] )

ClearRenderTarget( texture[temp] )

PerformTranslation( texture[currentTimeFrame] )

SetRenderTarget( texture[currentTimeFrame] )

ClearRenderTarget( texture[currentTimeFrame] )

PerformTextureCopyBack( texture[temp] )

SetRenderTarget( texture[temp] )

ClearRenderTarget( texture[temp] )

PerformTransilation( texture[previousTimeFrame] )

SetRenderTarget( texture[previousTimeFrame] )

ClearRenderTarget( texture[previousTimeFrame] )

PerformTextureCopyBack( texture[temp] )

SetRenderTarget( texture[nextTimeFrame] )

ClearRenderTarget( texture[nextTimeFrame] )

PerformWaveSimulation( texture[currentTimeFrame],
texture[previousTimeFrame] )

DrawBrushTexture ( brushTexture )

Update time frame together with texture ID

```

End of Procedure

In this pseudo code, the height map textures for current and previous time frame will be translated one by one. A temporary texture together with a PerformTextureCopyBack() procedure is also used. This is because shader does not allow its input and output to be the same memory location. A

Chapter 4 Design Fast Interactive Water Simulation for Games

temporary texture will hold the result from translation step, and then copied back to the original height map texture.

It is obvious that the `AdhesiveWaveSimulation()` procedure involves much more computation steps. It is about 3 times slower than nVIDIA's solution.

Luckily, some optimization steps can be done through the following steps to improve its performance.

- Remove all `ClearRenderTarget()`.

For all the procedures such as `PerformTranslation()`, `DrawBrushTexture()`, `PerformTextureCopyBack()` and `PerformWaveSimulation()`, the shader is written with a manner to guarantee all the pixels of the texture are updated to overwrite the old content.

- Combine height field textures.

In nVIDIA's solution, each height map uses one channel for height field, and three channels for normal. The normal is also calculated from the height field stored in the 1st channel. The normal information however, is not needed during simulation, which can be calculated later in water wave rendering. In this way, only 1 channel of height field data per texture is required.

Now we have three height map textures to maintain based on nVIDIA's solution, each contains only one channel of height field data. It is wise to combine them into one single texture.

Chapter 4 Design Fast Interactive Water Simulation for Games

The solution presented in the report combines three height fields into one single height map texture. Red channel of the texture stores the height field of current time frame. Green channel stores height field of next time frame. Blue channel stores height field of previous time frame. The advantage of this arrangement is that, only one `PerformTranslation()` call is required instead of two. `PerformWaveSimulation()` call will also take in only one height map texture input, instead of two. Furthermore, now only two height map textures, one of which is temporary texture, are required, while previous non-optimized solution will need four height map textures.

When the time frame advances, the name of each time frame will be changed. In order to maintain the current channel definition consistency, the height field data in green channel should be moved to red channel, and the height field data in red channel moved to blue channel. The height field data in blue channel is expired, so that new green channel will be empty.

Chapter 4 Design Fast Interactive Water Simulation for Games

Figure 4.5 Illustration on how 3 Copies of Height Field stored and used in one Texture

Switch data content for each channel can be done very fast within shader. Therefore the PerformWaveSimulation() procedure is combined with the step to Update time frame together with texture ID. The new procedure is named as PerformFastWaveSimulation().

- Eliminate PerformTextureCopyBack()

PerformTextureCopyBack() actually is nothing extra than providing a temporary data buffer. It is a quite expensive step. The step can be eliminated if we can directly use the output from the translation step for future calculation. This is done by creating two pointers pointing at the height field texture and temporary texture. The PerformTextureCopyBack()

Chapter 4 Design Fast Interactive Water Simulation for Games

procedure can be replaced by simply switching the contents of two pointers, which is much faster.

With all these optimizations, the pseudo code for the wave simulation with Adhesive grid type now become:

Procedure: FastAdhesiveWaveSimulation()

Calculate the number of pixel to shift, and update the accumulate value

SetRenderTarget(texture[temp])

PerformTranslation(texture[data])

Switch pointers between data and temp

SetRenderTarget(texture[temp])

PerformFastWaveSimulation(texture[data])

DrawBrushTexture (brushTexture)

Switch pointers between data and temp

End of Procedure

It is a lot simpler and much faster. The time consumption is even almost the same as nVIDIA's solution, though new feature and steps have been added. The memory consumption is only 2/3 of nVIDIA's solution.

4.2 Interactive Water Wave Rendering

The interactive water wave rendering is to use the height map texture generated in simulation to draw realistic water waves on the display screen.

4.2.1 Water Wave Mesh

The solution presented in this thesis report uses regular grid meshes to model the surface of water. Each mesh has 8 x 8 vertices. Many instance of meshes can tile up together to form a very large water surface area.

There is no culling on meshes implemented for this solution. Instead the stream instancing [43] is used. Stream instancing is to use two vertex buffer, one normal vertex buffer and another instance stream buffer holding the position and size of every instance. The stream instancing technique changes the offset in the instance stream buffer to render one instance at a time. This technique reduces the amount of data transferred to GPU, which can eventually achieve high throughput for rendering.

4.2.2 Hybrid Framework Design

The solution presented in this thesis report is mainly based on the height field method. However, as to make the results more realistic by combining many different types of water waves in the same scene, a hybrid framework is designed.

The author has designed a framework to integrate three different types of water waves: long wavelength ocean waves, medium wavelength interactive waves and short wavelength capillary waves caused by blowing wind. Figure 4.6 shows how the three wave types are combined by getting the superposition of them. However the actual methods used in the framework is much more complicated.

Figure 4.6 Integrating Three Different Wave Types into the Framework

The ocean waves are first added to the framework. The author uses procedural approach to displace the position of every vertex on the water wave meshes. The Gerstner Wave Model seems to be the most appropriate simulation function for ocean waves. However a much simpler sine wave function is used in this framework. The reason is quite straightforward. To make the solution extremely fast, high resolution wave meshes are not affordable. If low resolution meshes are used instead, the feature of horizontal vertex displacement provided by Gerstner Wave Model is hardly observable. It is pretty difficult to distinguish between a trochoid curve and a sine curve in this case. Therefore the simpler and faster simulation function is used. The result

Chapter 4 **Design Fast Interactive Water Simulation for Games**

after integrating the ocean waves is illustrated in Figure 4.7. At the output of this stage, two batches of important data are preserved. Firstly, the positions of each vertex are calculated as the output of vertex shader. Secondly, the heights of each pixel are calculated and passed down to the pixel shader.

Figure 4.7 Integrating Ocean Waves into the Framework

Next, interactive waves caused by object-water interaction are integrated with ocean waves. The deliberated designed height field approach as discussed in Section 4.1 is used. To get the superposition of interactive water waves and ocean waves, simply sum up their height fields, as illustrated in Figure 4.8.

Figure 4.8 Integrating Interactive Waves into the Framework

Chapter 4 Design Fast Interactive Water Simulation for Games

Notice that the field fields are only used in pixel shader stage in the framework for the speed concern, as the framework only has low polygonal water meshes. In order to render with high detailed height fields, a technique called bump mapping [44] is used. Bump mapping is a rendering technique uses a perturbation to surface normal of an object to simulate high surface details, even though the object is in low polygon. With the combined height field, bump mapping can generate the normal for each pixel in pixel shader. The normal data is the output in this stage, to be further integrated with the capillary waves.

Of course, there are other mapping techniques such as parallax mapping [45] and relief mapping [46]. But they either have visual defects, or are too slow to be used in this framework.

Lastly, it's time to integrate the capillary waves caused by the blowing wind with the rest. The framework uses a wave normal map pre-calculated from a frequency domain noise function. This means the spectral approach used to generate the normal map is done offline. It does not introducing any extra processing time in the real-time wave simulation. The normal map used is shown in Figure 4.9.

Figure 4.9 Capillary Wave Normal Map generated by Spectral Approach

Normal mapping technique [47, 48] is used to render using normal map. However, simply using this normal map on the mesh could only produce a static capillary wave. To mimic the moving capillary wave, a few copies of the normal map are fetched with different tiling size and make them move along certain directions based on the time. Combining all these copies can produce realistic moving capillary waves.

The normal of capillary waves are combined with the normal data from the previous stage. The framework simply combines them by taking the sum of two normal vectors can be one solution and then normalize the vector.

Figure 4.10 Integrating Capillary Waves into the Framework

After integrating all three different wave types, the framework holds two batches of data as the final result: the vertex position representing ocean waves only, and the pixel normal representing the combination of all three waves. The normal data are then used in water wave shading stage. The framework is an approximation method but can provide a fast simulation result.

Figure 4.11 Final Result of the Framework after integrating all Three Wave Types

Chapter 4 Design Fast Interactive Water Simulation for Games

In addition to these three wave types, breaking waves are also simulated. In the framework, it is assumed that the breaking waves only caused by the interactive waves. Similar to the solution described in [42], a certain sharp edge of the interactive wave is detected from height field. These edges are defined at which the wave normal direction exceeds certain degree. When this happens, a foam texture will be mapped onto the position of the wave mesh.

4.2.3 Water Wave Shading

With the normal calculated by combination of different types of waves, Phong shading technique [49] is used to render the water waves. The equation of Phong shading, as described below, consists of three lighting components, which are ambient, diffuse and specular.

$$I_{Phong} = k_{ambient}I_{ambient} + \sum_{light} (k_{diffuse}(L \cdot N)I_{diffuse} + k_{specular}(R \cdot V)^a I_{specular})$$

I_{phong} : illumination of Phong shading model

$k_{ambient}$, $k_{diffuse}$, $k_{specular}$: intensity factor for ambient, diffuse and specular lighting,

$i_{ambient}$, $i_{diffuse}$, $i_{specular}$: luminance of the ambient, diffuse and specular light

L : incident light direction vector

N : normal vector

R : reflecting light direction vector

In addition to the three components, the solution also includes a fourth component, which is the reflection from a skybox using cube environment mapping [50].

The cube environment map is actually a composition of six textures forming a sky box. As shown in below figure, each of the six textures are to the +x, -x, +y,

Chapter 4 **Design Fast Interactive Water Simulation for Games**

-y, +z and -z direction of the object. To render reflection, we assume that the object is at the center of the skybox as the sky is located at infinity. Shader can simply fetch the pixel value on the environment map with the eye reflection vector which is calculated from surface normal and viewing direction vector. Although cube environment mapping cannot get the object image on water reflection, it is the fastest reflection rendering technique, which is very suitable for video game applications.

Figure 4.12 Concept of Cube Environment Mapping

The Fresnel effect is also included in the water wave rendering. Fresnel effect is related to water refraction and total internal reflection effect. It can be simply described in the following. When we observe water from the top we see no reflection of sky but image below water surface. If we observe from an angle almost parallel to the water surface, we can see the reflection of sky but hardly anything below water surface.

A falloff texture which is a gradient texture from white to black is used to simulate Fresnel effect. The angle between view direction and water surface normal is measured. The larger the angle is, the smaller the coordinate it is to fetch pixel from falloff texture. In this case pixel shader will fetch the left part of the falloff texture, which is a factor close to 1.0. If the angle is small, the right part of falloff texture is fetched, which is a value almost 0.0. The fetched pixel is multiplied with the reflection result from cube environment mapping, to produce the Fresnel effect.

Figure 4.13 Falloff texture for Fresnel Effect

4.3 Cater for Interaction with Multiple Objects

It would be more practical if the interactive water simulation can cater for multiple objects.

The wave simulation part is straightforward. Each object in the water must have its own height map texture to be attached with. The total number of texture required for N objects is $N+1$, with the extra one as temporary texture.

It is a bit tricky for wave rendering part. When the waves caused by different objects meeting with each other, the resultant waveform is the superposition of all individual waves. In order to achieve this, we need to update every simulated height map textures onto the final water surface.

However, it is not feasible to provide all height map textures as inputs to the rendering shader. The reason is because the maximum number of texture

Chapter 4 Design Fast Interactive Water Simulation for Games

inputs supported by Shader is 16, while the number of interacting objects can easily go beyond this figure. To overcome this technical limitation, a main height field texture is used to collect the height fields contributed by every single object. Again the consideration is that a high resolution of this main height field is not affordable. Instead of using the rational method that use a height field texture to cover every part of water surface mesh in the game word, the solution provided in the report uses a relatively small screen-based texture.

The screen-based texture shows the wave meshes in the world with reference of screen coordinate system. In short, it is actually the view of what we see from the display screen. The resolution of this screen based texture is set to be the same as the resolution of the game application screen, which is 720p or higher.

Figure 4.14 Illustration on the Shape of a Sample Screen-based Height Map

A special shader is used to update all height fields of objects. The screen-based texture is initialized to zero height at the beginning for each time frame. The height fields of objects are then updated onto it one by one using `RenderScreenBasedHeightMap()` procedure. The pipeline settings of shader

can automatically perform culling to remove those back-faced height field as well as height field outside screen space. It can also perform water wave superposition by adding the height of two collided height field.

Figure 4.15 Illustration on how Screen-based Texture is updated

Of course, in game applications where at least a 720p resolution is required, a RGBA texture with floating point data width may take up about 5MB which is quite high memory consumption. To save memory and increase the processing speed, the screen-base texture is set to contain only one channel of 32bit floating point format, which makes the memory consumption reduced to 1.25MB at 720p.

The pseudo code of this implementation is:

```

Procedure: RenderWaveMultipleObjects()
    SetRenderTarget( screenBasedTexture )
    ClearRenderTarget(screenBasedTexture )
    For each object that can interact with water

```

Chapter 4 Design Fast Interactive Water Simulation for Games

RenderScreenBasedHeightMap(texture[id])

SetRenderTarget(BackBufferRenderTarget)

PerformWaveRendering(screenBasedTexture)

End of Procedure

Chapter 5 Results

To demonstrate the result, a demo program with many boats running in the open ocean is developed. A few versions of this demo is made, to show the feature improvement and optimization process, as well as to compare the performance difference with single boat or multiple boat. A performance chart is presented in this chapter to provide the comparison with a few other solutions.

5.1 Performance Measurement Criteria

The most popular parameter used for algorithm performance measurement is the frame rate. Frame rate, or expressed in frame per second (fps), is the metrics to show how fast a graphic device can produce drawing of a specific scene. It is a good measurement for the overall performance of a graphical interfaced program, such as a video game.

However from the author's point of view, frame rate is not a very accurate measurement for an algorithm solution, which is only part of the overall system. A program usually has a list of structured and systematic activities to follow. Taking video game as an example, in each frame the game program performs user data updating, physics calculation, UI updating, AI logic calculation, sound decoding etc, in addition to graphics related activities. Even within graphics related activities, there are many other works to be done besides a specific algorithm solution. For example, we need to simulate and render boats in this demo program, not to mention many other objects in an

actual video game. An algorithm which can run at 60 fps in the demo will not likely get 60 fps in actual game. The author therefore takes the actual amount of time required for the algorithm solution as a much more accurate measurement.

Traditionally, we take the system time before and after the algorithm. The actual time taken for the algorithm can be calculated by subtracting these two timing values.

But the traditional way is still not accurate in program involving GPU activities. As shown in the below figure, after transferring all necessary data as inputs to GPU for handling the algorithm task, CPU continue to process other activities when GPU handles the rendering job. As the timing measurement is done in CPU, it will not count in the timing taken by GPU to complete the algorithm.

Figure 5.1 Result of Traditional Timing Measurement

To solve the problem, a special function `WaitGPUIidle()` is called just at the beginning and end of the algorithm in CPU program code. The function will

halt CPU from continuing processing the rest of program code until GPU finishes all the tasks. As shown in below figure, the timing measurement is now accurate.

Figure 5.2 Result of Timing Measurement with WaitGPUIdle() Function

In the solution presented in this report, the author of the report has divided the algorithm into two parts as the water wave simulation and water wave rendering, as shown in below figure. Timing measurements are taken for each of the two parts. Though in wave simulation and rendering a few functions have utilized GPU for calculation, WaitGPUIdle() will only be called at beginning and end of each part. It is not necessary to measure the actual timing taken for each functions.

Figure 5.3 Timing Measurement Strategy used in the Demo

5.2 Demo Program Setup

The demo program is tested on the PC with a configuration of Intel Core2 2.66GHz, 2GB RAM and an nVIDIA GeForce 8600 GTS graphic card with 512MB GPU memory.

Four different versions of water wave simulation are presented in the demo.

The first version is called Static Grid, which is actually the nVIDIA’s solution [40] discussed in Section 3.4.2, but with two major differences. Firstly, the wave simulation grid used is 256x256 instead of 128x128. This is to provide higher resolution of wave details for the simulated waves. Secondly, a different wave rendering method is used. The rendering method used is discussed in Section 4.2 with some exceptions. The ocean waves are not simulated in this version. Only one water surface mesh of resolution 2x2 is used.

The second version is called Adhesive Grid, as discussed in Section 4.1.3. Besides using the adhesive type of height field instead of static type, the other settings are exactly the same as in Static Grid. The simulation step used is

AdhesiveWaveSimulation(), which is not optimized, as described in Section 4.1.4.

The third version, called Ocean Grid with One Boat, is also using adhesive type of height field, but with the optimized simulation method FastAdhesiveWaveSimulation(). The ocean waves are simulated and rendered in addition to interactive waves and capillary waves, as discussed in Section 4.2. To simulate a huge ocean water surface, we use 10x10 water surface meshes, each in the resolution of 8x8.

The last version is called Ocean Grid with Many Boats, uses the idea discussed in Section 4.3. Besides the use of a screen based height map, all the other settings are exactly the same as Ocean Grid with One Boat. It is the final version of the solution described in this thesis report.

All the four versions are tested under 720p and 1080p resolution. Frame rate, the timing for water wave simulation and rendering of each version are recorded for performance comparison.

Both simulation and rendering time are measured with the help of WaitGPUIidle() function described in Section 5.1. The rendering time is measured with the simulated water waves cover the entire screen. Under this condition, pixel shader has to process every pixel defined in the output resolution setting, which is the worst case. Both frame rate values with and without WaitGPUIidle() function in use are measured as well.

5.3 Demo Program Performance

5.3.1 Timing Performance with One Boat

Firstly, the timing performance of all 4 versions in the demo measured at both 720p and 1080p are summarized in the following two tables. At this stage, only one boat is presented in each version.

Version	Simulation Time	Rendering Time	FPS with WaitGPUIdle()	FPS without WaitGPUIdle()
Static Grid	0.32ms	3.73ms	162.0	173.0
Adhesive Grid	0.94ms	3.75ms	145.0	155.0
Ocean Grid with One Boat	0.31ms	4.80ms	133.0	165.0
Ocean Grid with Many Boats (1 boat)	0.32ms	5.20ms	127.0	153.0

Table 5.1 Timing Performance for all 4 Version of Water Wave Simulation at 720p

Version	Simulation Time	Rendering Time	FPS with WaitGPUIdle()	FPS without WaitGPUIdle()
Static Grid	0.32ms	8.25ms	88.4	94.0
Adhesive Grid	0.96ms	8.24ms	83.5	88.2
Ocean Grid with One Boat	0.32ms	10.17ms	74.4	82.0
Ocean Grid with Many Boats (1 boat)	0.33ms	11.18ms	69.0	77.0

Table 5.2 Timing Performance for all 4 Version of Water Wave Simulation at 1080p

A few observations can be obtained from these data.

The simulation time is independent to output resolution. The timing obtained for 720p and 1080p are quite close to each other. The data show a significantly increase of time from Static Grid to Adhesive Grid. It also shows the effectiveness of the optimization process described in Section 4.1.4 in Ocean Grid with One Boat and Ocean Grid with Many Boats.

The rendering time depends on output resolution. The major factor is the number of pixels processed in pixel shader. 1080p resolution has twice the

number of pixels than 720p resolution. The ratio of pixel number is reflected in the timing measured. In all versions it takes more than twice the time duration to render at 1080p, compared to 720p. There is a significant increase of time from Static / Adhesive Grid to Ocean Grid with One Boat. This is due to the increase of number of instances (from 1 to 100) and number of vertices (from 2x2 to 8x8). Ocean Grid with Many Boats also takes more time than Ocean Grid with One Boat, due to the extra step to render the screen based height map.

The result proves the solution of this report, Ocean Grid with Many Boats, is a very fast solution. The solution takes about 5.5ms (simulation time + rendering time) at 720p, which meets the author's 10ms target timing. This provides a frame rate around 153 fps, with a main program platform specially developed to minimize the timing taken for all non-relevant activities. When it moves to 1080p, the solution takes about 11.5ms (simulation time + rendering time), still hitting 77 fps which is higher than the 60 fps under the same program platform.

5.3.2 Timing Performance with Many Boats

At this stage, the timing performance of various numbers of boats interacting with water is measured on the last version, Ocean Grid with Many Boats only. The following two tables show such results measured at 720p and 1080p respectively.

Number of Boats	Simulation Time	Rendering Time	FPS with WaitGPUIdle()	FPS without WaitGPUIdle()
1 boat	0.32ms	5.20ms	127.0	153.0
2 boats	0.58ms	5.36ms	121.0	153.0
3 boats	0.83ms	5.44ms	117.0	145.0
5 boats	1.27ms	5.55ms	109.0	132.0
10 boats	2.46ms	5.73ms	94.0	115.2
20 boats	4.75ms	5.75ms	76.7	90.2
50 boats	11.32ms	5.81ms	49.4	54.7

Table 5.3 Ocean Grid with Many Boat Version: Timing Performance vs. Number of Boat
at 720p

Number of Boats	Simulation Time	Rendering Time	FPS with WaitGPUIdle()	FPS without WaitGPUIdle()
1 boat	0.33ms	11.18ms	69.0	77.0
2 boats	0.61ms	11.35ms	67.3	76.0
3 boats	0.85ms	11.51ms	65.2	74.7
5 boats	1.28ms	11.61ms	62.0	70.4
10 boats	2.44ms	11.65ms	58.6	65.0
20 boats	4.75ms	11.68ms	51.2	56.6
50 boats	11.42ms	11.80ms	37.0	40.2

Table 5.4 Ocean Grid with Many Boat Version: Timing Performance vs. Number of Boat
at 1080p

The most obvious finding is that, the simulation time increases proportionally to the number of boats. The rendering time only increases a little bit, even when the boat count increases to 50. The data shows that there is a tendency the rendering time will not exceed an upper bound no matter how many boats are there in the scene. This proves the use of screen based height map an effective design to handle any number of objects in the water.

Figure 5.4 Timing Performance Trend with increasing Number of Boats

Under 720p, the time taken for 20 boats is about 10.5ms (simulation time + rendering time). This means the system can support up to 20 boats interacting with water at 720p with the 10ms target set by the author of report met. When under 1080p, 20 boats require about 16.3ms and a frame rate around 56 fps is to be achieved.

5.3.3 Interaction Performance

The demo program provides interaction between boat and water in the following ways.

Initially, the boat does not move around. It only floats on the water surface and perform up and down motion along with the crest of ocean wave.

Player can perform three types of control with the boat. Firstly, player can accelerate the boat and make it start to move on water surface. Secondly,

player can make the boat turn left or right, allowing it to rotate to any direction. Lastly, player can decelerate the boat and to reduce the speed of boat, and eventually make it stop on water surface.

No matter what type of control player uses, he will change the movement of the boat and create interactive waves in real-time. The height of the initial wave is proportional to the speed of the boat. After being created, these waves will start to spread around the water surface. The amplitude of interactive waves will be gradually reduced to simulate the effect of water internal friction. Finally interactive waves break after travelling over certain distance.

5.3.4 Visual Performance

A few screenshots of the demo program are taken to show that the solution given by the thesis can provide interactive and realistic water wave simulation with good visual performance.

From these screenshots, we can see many different types of waves have been simulated. The great amplitude of ocean waves, ripples of interactive wave, details of capillary waves and foams created by breaking waves are all combined together to provide a vivid water scene.

Due to simulation timing concern, I try to avoid expensive operations such as displace the position of every vertices on water surface mesh, with the exception of ocean waves who really require high amplitude. However, with the support of bump mapping, I can still fake the flat surface to make it looks bumpy.

Chapter 5

Results

The other visual features, such as cube sky-box reflection, Phong Shading as well as Fresnel Effect, are all playing their parts well to make the simulated water waves feel more realistic.

Figure 5.5 A Single Boat interacting with Water, Adhesive Grid Version

Figure 5.6 10 Boats interacting with Water, Ocean Grid with Many Boats Version

Figure 5.7 Highlight the Details of Interactive Waves and Capillary Waves with Breaking Waves

Figure 5.8 Effects of Ocean Wave Simulation

Chapter 6 Conclusion & Future Work

This chapter presents the conclusion of the research presented in this thesis report. It also discusses some aspects that can be made for future improvements.

6.1 Conclusion

The thesis report has presented the design and development of a fast, interactive water wave simulation utilizing GPU architecture suitable for video game applications.

The solution can provide visually realistic waves of different types, ranging from long wavelength ocean waves, medium wavelength interactive waves, to short wavelength capillary waves. Even the breaking waves with foams can be simulated. With the height field approach used as the core of the solution, object-water interaction can be simulated fast and effectively.

The most important contribution of this water wave simulation solution is that it is extremely fast. Although the model does not use a precise physics simulation model, it only takes 0.30ms wave simulation time per boat. The rendering part of the solution is also optimized such that it can handle fast drawing to give satisfying frame rate for video games at resolution of 720p or even 1080p.

The framework of the solution also has well adaptation of water wave interactions with multiple objects. The wave rendering function is kept a good

manner such that no matter how many objects are there, its time consumption will be well bounded and tend towards a constant value.

6.2 Future Work

The research presented in the report still can be improved in a few ways.

6.2.1 Wave Simulation Timing Control

The testing result shows that the simulation time is proportionally increased with the number of objects interact with water surface. It is an obvious performance bottleneck. In fact, if the resultant wave of the object is outside the viewing screen, or faraway from the camera, it does not need to be simulated. A good culling or level-of-detail scheme to arrange wave simulation is a potential solution to solve the bottleneck.

6.2.2 Higher Wave Details and More Complex Wave Effect

The wave simulation solution of this thesis is based on height field approach and tends to use a low resolution water surface mesh. The solution uses only limited number of vertical vertex displacement for high amplitude waves, and most of the other details are performed using bump mapping. The resultant wave looks quite well from top down viewing direction, but still not good enough with a close look at a viewing angle almost parallel to the water surface. In addition, lacking of horizontal vertex displacement prevents it from simulating of certain wave phenomenon such as plunging break wave, and surging waves. Of course high resolution mesh is not a feasible solution for video games. But there is a potential to integrate some other approaches such

as particle system approach to add these dramatic wave effects to the game scene.

6.2.3 Adapt to more Physical Behaviours

With the framework of the algorithm settled to provide fast water simulation result, it is time to replant more physics behaviours into the solution to make it more realistic. We still do not aim at 100% accuracy to follow the physics law of fluids, but try to visually simulate as many physics behaviours as possible. One of the directions is to simulate how the depth of water will affect the waves. Another one may be to treat water as incompressible media, and to simulate how the amplitude and speed of the waves will reflect to the simulation results.

References

1. J. Ive. *Image formats for HDTV*, in *Sony Technical Review*. 2004, Sony Europe.
2. Wikipedia. *High-definition video, HD in video gaming*. [cited; Available from: http://en.wikipedia.org/wiki/High-definition_video.
3. Wikipedia. *Wind wave*. [cited; Available from: http://en.wikipedia.org/wiki/Water_wave.
4. R.G. Dean, and R.A. Dalrymple. *Water wave mechanics for engineers and scientists*. 2 ed. Advanced Series on Ocean Engineering. Vol. 2. 1991: World Scientific.
5. W.S. von Arx. *An introduction to physical oceanography*. 1962: Addison-Wesley.
6. C.R. Nave. *Hyperphysics: ocean waves*. 2006 [cited; Available from: <http://hyperphysics.phy-astr.gsu.edu/hbase/waves/watwav2.html>.
7. H. Lamb. *Hydrodynamics*. 6th ed. 1994: Cambridge University Press.
8. W. Bascom. *Waves and beaches*. 1964: Doubleday.
9. E.W. Weisstein. *Trochoid*. [cited; Available from: <http://mathworld.wolfram.com/Trochoid.html>.
10. L. David, and H. Greg. *How GPUs Work*. Computer, 2007. **40**(2): p. 96-100.
11. H. Tan-Chi. *Introduction to programmable graphics hardware*. 2005 [cited; Available from: http://cggmwww.csie.nctu.edu.tw/courses/cgu/2005/gpu_intro.pdf.
12. M. Ecker. *Programmable graphics pipeline architectures*. 2003 [cited; Available from:

References

- <http://xengine.sourceforge.net/pdf/Programmable%20Graphics%20Pipeline%20Architectures.pdf>.
13. W.T. Reeves. *Particle systems: a technique for modeling a class of fuzzy objects*, in *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*. 1983, ACM: Detroit, Michigan, United States.
 14. G.K. Batchelor. *An introduction to fluid dynamics*. 1967: Cambridge University Press.
 15. M. Müller, D. Charypar, and M. Gross. *Particle-based fluid simulation for interactive applications*, in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 2003, Eurographics Association: San Diego, California.
 16. M. Müller, et al. *Point based animation of elastic, plastic and melting objects*, in *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 2004, Eurographics Association: Grenoble, France.
 17. M. Pauly, et al. *Meshless animation of fracturing solids*, in *ACM SIGGRAPH 2005 Papers*. 2005, ACM: Los Angeles, California.
 18. S. Premoze, et al. *Particle-based simulation of fluids*, in *Comp. Graph. Forum (Eurographics Proc.)*. 2003. p. 401-410.
 19. N. Foster, and R. Fedkiw. *Practical animation of liquids*, in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 2001, ACM: New York, NY.
 20. N. Foster, and D. Metaxas. *Modeling the motion of a hot, turbulent gas*, in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. 1997, ACM Press/Addison-Wesley Publishing Co.
 21. J. Stam. *Stable fluids*, in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. 1999, ACM Press/Addison-Wesley Publishing Co.

References

22. F. Losasso, F. Gibou, and R. Fedkiw. *Simulating water and smoke with an octree data structure*, in *ACM SIGGRAPH 2004 Papers*. 2004, ACM: Los Angeles, California.
23. B.E. Feldman, J.F. O'Brien, and B.M. Klingner. *Animating gases with hybrid meshes*, in *ACM SIGGRAPH 2005 Papers*. 2005, ACM: Los Angeles, California.
24. W. Li, et al. *Flow simulation with complex boundaries*, in *GPU gems 2*. 2005, Addison Wesley. p. 747-764.
25. G.A. Mastin, P.A. Watterberg, and J.F. Mareda. *Fourier synthesis of ocean scenes*. IEEE Comput. Graphi. Appl., 1987. 7(3): p. 16-23.
26. K. Perlin, and E.M. Hoffert. *Hypertexture*, in *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*. 1989, ACM: New York, NY.
27. J. Schneider, and R. Westermann. *Towards real-time visual simulation of water surfaces*. in *Vision Modeling and Visualization Conference*. 2001.
28. J. Tessendorf. *Simulating ocean water*, in *SIGGRAPH course notes*. 2004.
29. J.L. Mitchell. *Real-time synthesis and rendering of ocean water*. ATI Research Technical Report, 2005
30. D. Hinsinger, F. Neyret, and M.-P. Cani. *Interactive animation of ocean waves*, in *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 2002, ACM: San Antonio, Texas.
31. A. Fournier, and W.T. Reeves. *A simple model of ocean waves*, in *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. 1986, ACM: New York, NY.
32. D.R. Peachey. *Modeling waves and surf*, in *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. 1986, ACM: New York, NY.

References

33. P.Y. Ts'o, and B.A. Barsky. *Modeling and rendering waves: wave-tracing using beta-splines and reflective and refractive texture mapping*. ACM Transaction on Graphics, 1987. **6**(5(July)): p. 191-214.
34. M.N. Gamito, and F.K. Musgrave. *An accurate model of wave refraction over shallow water*. Computer and Graphics, 2002. **26**(2): p. 291-307.
35. L. Szécsi, and K. Arman. *Procedural ocean effects*, in *Shader X6, Advanced Rendering Techniques*. 2008, Charles River Media. p. 331-349.
36. S. Thon, J.-M. Dischler, and D. Ghazanfarpour. *Ocean waves synthesis using a spectrum-based turbulence function*. in *Computer Graphics International*. 2000.
37. C. Yuksel, D.H. House, and J. Keyser. *Wave particles*, in *ACM SIGGRAPH 2007 papers*. 2007, ACM: San Diego, California.
38. M. Kass, and G. Miller, *Rapid, stable fluid dynamics for computer graphics*, in *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*. 1990, ACM: Dallas, TX, USA.
39. J.X. Chen, and N.D.V. Lobo. *Toward interactive-rate simulation of fluids with moving obstacles using navier-stokes equation*. Graph. Models Images Process, 1995. **57**(2): p. 107-116.
40. J. Zelnack. *nVIDIA SDK White Paper: Vertex texture fetch water*. nVIDIA Technical Support Notes, 2004.
41. N. Thürey, U. Rüdé, and M. Stamminger. *Animation of open water phenomena with coupled shallow water and free surface simulations*, in *Proceedings of the 2006 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 2006, Eurographics Association: Vienna, Austria.
42. N. Thürey, et al. *Real-time breaking waves for shallow water simulations*, in *Pacific Graphics*. 2007.
43. MSDN. *Instancing sample*. 2009 [cited; Available from: [http://msdn.microsoft.com/en-us/library/bb174602\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/bb174602(VS.85).aspx)].

References

44. J.F. Blinn. *Simulation of wrinkled surfaces*, in *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*. 1978, ACM: New York, NY.
45. T. Kaneko, et al. *Detailed shape representation with parallax mapping*. In *Proceeding of ICAT*, 2001: p. 205-208.
46. V. Pamplona, M.M. Oliveira, and L.P. Nedel. *Animating relief impostors using radial basis functions textures*, in *Game programming gems 7*, S. Jacobs, Editor. 2008, Charles River Media, Inc. p. 401-412.
47. J. Cohen, M. Olano, and D. Manocha. *Appearance-preserving simplification*, in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. 1998, ACM: New York, NY.
48. P. Cignoni, et al. *A general method for preserving attribute values on simplified meshes*, in *Proceedings of the conference on Visualization '98*. 1998, IEEE Computer Society Press: Research Triangle Park, North Carolina, United States.
49. B.T. Phong. *Illumination of computer-generated pictures*. *Comm. ACM*, 1975. **18**(6): p. 311-317.
50. N. Greene. *Environment mapping and other applications of world projections*. *IEEE Comput. Graphi. Appl.*, 1986. **6**(11).