Ocean Wave Simulation in Real-time using GPU

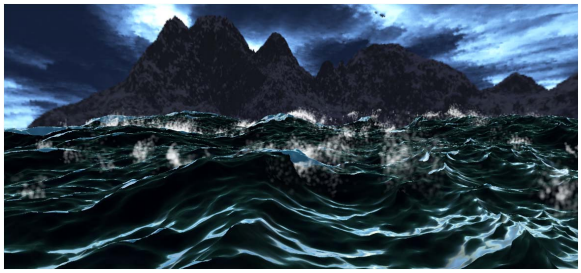
Chin-Chih Wang
Department of Computer Science and Information Engineering
National Taiwan University
Taipei,Taiwan
Email: r98922125@ntu.edu.tw

Jia-Xiang Wu
Graduate Institute of Networking and Multimedia
National Taiwan University
Taipei,Taiwan
Email: taco.wu@gmail.com

Chao-En Yen
Graduate Institute of Networking and Multimedia
National Taiwan University
Taipei,Taiwan
Email: twelalic@gmail.com

Pangfeng Liu
Department of Computer Science and Information Engineering
National Taiwan University
Taipei,Taiwan
Email: pangfeng@csie.ntu.edu.tw

Chuen-Liang Chen
Department of Computer Science and Information Engineering
National Taiwan University
Taipei,Taiwan
Email: clchen@csie.ntu.edu.tw



***Abstract*—Ocean wave simulation is a popular topic in computer graphics. We often use Fast Fourier Transforms to simulate ocean wave in a statistical model. In order to make ocean wave more realistic we use choppy wave to generate sharp wave, a particle system to simulate wave breaking, and a cube map to shade the ocean wave. We have implemented our simulation program on a GPU, and the implementation can simulate ocean wave in real time.**

***Keywords*-Wave simulation; FFT; GPU; CUDA;**

I. Introduction

This paper reports our implementation of ocean wave and spraying simulating using various computer graphic methods. In the last ten years we have seen more and more computer graphics water scenes in movies and games. As the performance of graphics hardware becomes increasingly powerful, and GPU programming becomes much easier than before, we can compute mass information and reach real-time rendering performance.

Our implementation uses CUDA[1] to reach ideal frame rate. In our implementation we do not concern about the interaction of light between water surface and clouds or air, so we simply use cube map [5] shading to reduce the computation load. We do not concern about the water volume under the waver surface either. Our implementation focuses on spray position detection and production, because it is crucial to show the spray produced by wave so that the wave will look realistic.

The rest of the paper is organized as follows. Section II introduces related work about ocean simulation. Section III introduces the algorithms we use to generate the basic wave surface. Section IV describes choppy effect which makes basic wave move horizontally to generate sharp wave. Section V introduces how to detect and produce spray (breaking wave). Section VI describes system flow chart in our implementation. Section VII describes results from profiling our implementation and analyzes possible performance bottleneck, and Section VIII concludes with observation and future works.

II. Related Works

There have been many efforts of ocean simulation in the literature. A. Fournier, W. T. Reeves [6] presented a simple ocean wave model. J. Tessendorf [10] introduced a off-line FFT-based ocean wave simulation which used choppy effect to form the dramatic wave shape and particle system to perform breaking wave. FFT-ocean in [2] which implemented basic FFT-based ocean wave in CUDA, and it demonstrate GPU parallel computing power on FFT. T. Nishita and E. Nakamae [11] introduced a way to render under-water optical effects but it takes minutes to generate a frame. L. S. Jensen and R. Goliáš [7] rendered foam by texture-based method which provided a deep water animation. S. Premoze

and M. Ashikhm [9] used physical-based method and a non-real-time light transport to render water surface.

III. Basic Ocean Wave

In this section we focus on the main algorithms used in our implementation. J. tessendorf [10] described that Oceanographic literature tend to used statistical ocean model for realistic purpose. Statistical models are also based on the ability to decompose the wave height field as a sum of sine and cosine waves. Therefore we can use inverse FFT to composite many sine and cosine waves to form the wave surface. The following is a description of FFT-based ocean wave.

A. FFT ocean wave

J. Tessendorf [10] provides Equation 1 to composite sin and cosine wave by inverse-FFT.

$$h(\mathbf{x},t) = \sum_{\mathbf{k}} \tilde{h}(\mathbf{k},t)\exp(i\mathbf{k}\cdot\mathbf{x}) \tag{1}$$

Equation 1 is explained as follows. $h(\mathbf{x},t)$ and $\tilde{h}(\mathbf{k},t)$ represent wave amplitude at time t in spacial and frequency domain respectively. $\mathbf{x} = (x,z)$ stands for wave position and $\mathbf{k} = (k_x,k_z)$ stands for wave direction, $k_x = \frac{2\pi n}{L_x}, k_z = \frac{2\pi m}{L_z}$, and m, n are both integers with bound $-\frac{N}{2} \leq n < \frac{N}{2}$ and $-\frac{M}{2} \leq m < \frac{M}{2}$. We can use inverse FFT to generate the height field at each discrete point $\mathbf{x} = (\frac{nL_x}{N}, \frac{mL_z}{M})$ for corresponding n, m.

J. Tessendorf provides an efficient method to get $\tilde{h}(\mathbf{k},t)$ [10]. First, we consider this model for spatial spectrum, called Phillips Spectrum [8]:

$$P_h(\mathbf{k}) = A\frac{\exp(\frac{-1}{(kL)^2})}{k^4}\left|\hat{\mathbf{k}}\cdot\hat{w}\right|^2 \tag{2}$$

Where L is the maximum waves height which is related to speed of wind and constant of gravity, and $\hat{w}$ is the wind direction, and A is a constant. The term $\left|\hat{\mathbf{k}}\cdot\hat{w}\right|^2$ removes those waves that move vertically to the direction of wind. Then, we compute the Fourier amplitudes of a wave height field with Phillips Spectrum:

$$\tilde{h}_0(\mathbf{k}) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(\mathbf{k})} \tag{3}$$

Where ξ_r and ξ_i are 2 Gaussian random variables with mean 0 and standard deviation 1. Finally, after given a dispersion relation $\omega(k)$, we can compute the wave amplitudes at time t as follows.

$$\tilde{h}(\mathbf{k},t) = \tilde{h}_0(\mathbf{k})\exp(i\omega(k)t) + \tilde{h}_0^*(-\mathbf{k})\exp(-i\omega(k)t) \tag{4}$$

We now describe our algorithm that generates FFT ocean wave. At the beginning of simulation we compute $\tilde{h}_0(\mathbf{k})$ on CPU because it just needs to be calculated once per simulation. In each frame, we use $N \times M$ CUDA threads to compute $\tilde{h}(\mathbf{k},t)$ for each $\mathbf{k}$ in parallel, and use inverse-FFT library provided by CUDA to transform $\tilde{h}(\mathbf{k},t)$ to $h(\mathbf{k},t)$.

IV. Choppy Wave

Figure 1. Choppy effect

Now we have a wave simulation without storm, but the shape is too regular to be realistic. Therefore we add choppy effect into our simulation. J. Tessendorf [10] used choppy effect to make the wave peak sharp and the wave bottom flat, as shown in Figure 1. We move $\mathbf{x}$ to $\mathbf{x} + \lambda D(\mathbf{x},t)$ at time t in order to obtain sharper wave instead of sine wave, where λ is a coefficient to control choppy effect. The movement function $D(\mathbf{x},t)$ is as follows.

$$D(\mathbf{x},t) = \sum_{k} -\frac{\mathbf{k}}{k}\tilde{h}(\mathbf{k},t)\exp(i\mathbf{k}\cdot\mathbf{x})) \tag{5}$$

From Equation 5 the equation of the horizontal movement is similar to FFT but shifts phase forward $\frac{3\pi}{2}$. Consequently the horizontal movement become smaller when the original FFT wave moves near the peak or trough, but becomes larger when the wave moves near the base, which makes our simulating waves much sharper. We can still use CUDA FFT library to compute choppy intensity. Figure 2 and Figure 3 show simulation results with and without choppy effect.

Choppy waves do have problems. When we increase λ, choppy wave causes "overlapping" on the wave simulation. That is, the external force is so huge that waves cannot keep their shapes, so we can see the effect in Figure 4. When wave breaks, it generates spray in real situation, so we will introduce how to detect the overlapping and produce the spray.

V. Spray

We describe the problem when the peaks of waves overlap. Now we can use this property of breaking waves to generate spray.

A. Detecting Spray Positions

According to [10], J. Tessendorf provides a simple method to detect overlapping by using the Jacobian, which has the following form.

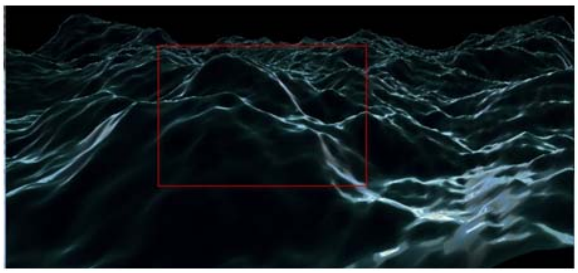


Figure 2. original fft ocean wave

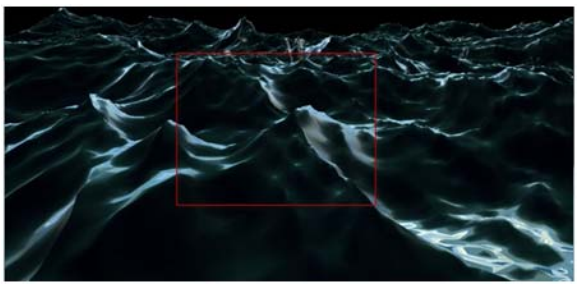


Figure 3. choppy wave

$$J(\mathbf{x}) = J_{xx}J_{yy} - J_{xy}J_{yx} \tag{6}$$

where

$$J_{xx}(\mathbf{x}) = 1 + \lambda\frac{\partial D_x(\mathbf{x})}{\partial x} \tag{7}$$

$$J_{yy}(\mathbf{x}) = 1 + \lambda\frac{\partial D_y(\mathbf{x})}{\partial y} \tag{8}$$

$$J_{xy}(\mathbf{x}) = \lambda\frac{\partial D_x(\mathbf{x})}{\partial y} \tag{9}$$

$$J_{yx}(\mathbf{x}) = \lambda\frac{\partial D_y(\mathbf{x})}{\partial x} = J_{xy}(\mathbf{x}) \tag{10}$$

$D = (D_x, D_y)$ is a function of the coordinate (D_x, D_y) on the horizontal plane. The Jacobian is less than zero if the $\mathbf{x}$ is in the overlapping region. The Jacobian is zero when there are points that will transform to the same position (the transform is not invertible), so when Jacobian is less than zero then region of overlapping appears.

As a result we can now find positions of overlapping region by the Jacobian. We can also use the Jacobian to compute the initial velocity of the spray. To determine the velocity, we have to compute the eigenvalues and the eigenvectors from the Jacobian matrix. According to J. Tessendorf in [10] eigenvalues and eigenvectors can be computed as follows.

$$J_{\pm} = \frac{J_{xx} + J_{yy}}{2} \pm \frac{((J_{xx} - J_{yy})^2 + 4J_{xy}^2)^{\frac{1}{2}}}{2} \tag{11}$$

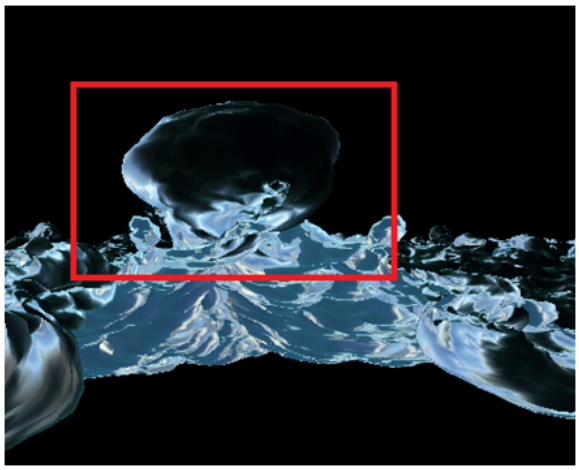


Figure 4. overlapping

$$\hat{e}^{\pm} = \frac{(1, q_{\pm}))}{\sqrt{1 + q_{\pm}^2}} \tag{12}$$

and

$$q_{\pm} = \frac{J_{\pm} - J_{xx}}{J_{xy}} \tag{13}$$

J_+ and J_- are the larger and smaller eigenvalues and $J_+ \times J_- = J(\mathbf{x})$. $J(\mathbf{x})$ is less than zero if and only if J_- is less than zero and J_+ is larger than zero. As a result we can check $J_- < 0$, instead of $J(\mathbf{x}) < 0$, and the eigenvector corresponding to J_- is the direction of spray velocity.

B. Generating Particle Spray

In our implementation we use a particle system to represent spray from breaking waves because CUDA programing model is very suitable for particle system computation. In each frame we generate new particles on the overlapping wave surface. The direction of the velocity of the particle is determined by the eigenvector corresponding to the smaller eigenvalue, and the magnitude of the spray velocity is a random number between 0 and $(J_T - J_-)$, where J_T is a fixed threshold. After we initialize new particles, we update the information of all *living* particles, including velocity, position, and age. We just consider the effect of gravity because it is much more significant than other forces generated by the interaction between particle and wave surface. When a particle dies, we stop updating its information and make it invisible until we recycle it as a new spray particle. After these optimization we can achieve a frame rate 30 frames per second using 131072 particles at the same time.

VI. Rendering

Our rendering has seven steps.

1) We use inverse FFT to generate height map at time t.

2) We calculate choppy effect to form wave on the (256×256) original grid mesh.
3) We detect overlapping regions to determine spray positions
4) We generate new particles on these positions as sprays, assign eigenvector of Jocobian as their velocities.
5) We update all the information of living particles, including age, velocity, and position.
6) We draw six pictures on the sky box that surrounds the scene.
7) We sample colors from the cube map and use them as the reflection on the water surface. We also render the particles of spray with a point sprite, which simplifies water particle simulation without complicated calculation. All the rendering work is done by OpenGL[4] shader.

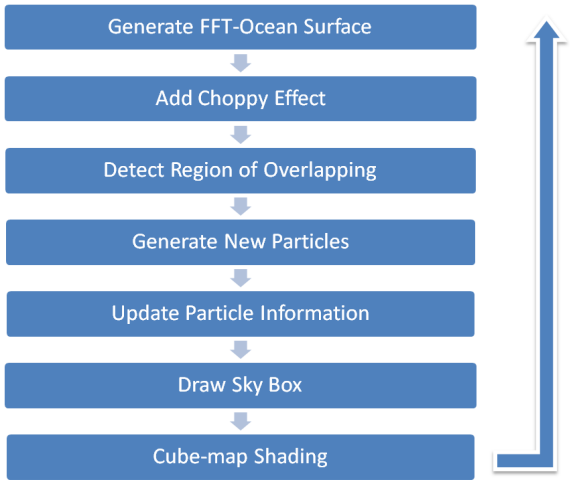


Figure 5. flow chart

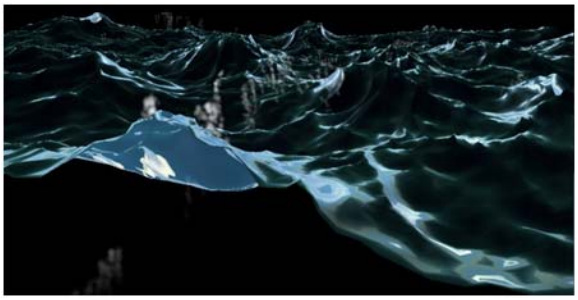


Figure 6. choppy wave with spray

VII. Experiment

Our experiment platform consists of one Intel Duo core 2.0GHz CPU, one NVIDIA Geforce 9800GT video card that contains 112 CUDA cores and 512 MB device memory, and 4 GB host memory.

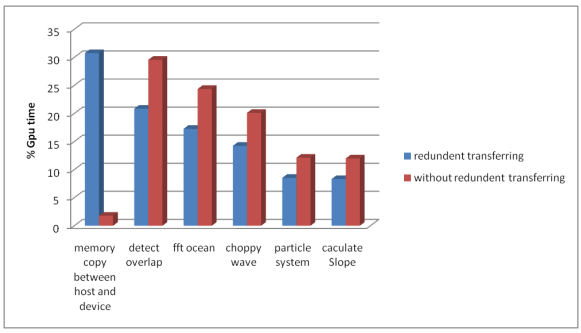


Figure 7. percentage of GPU time

We use CUDA Visual Profiler [3] to profile our implementation in order to determine possible performance bottlenecks.

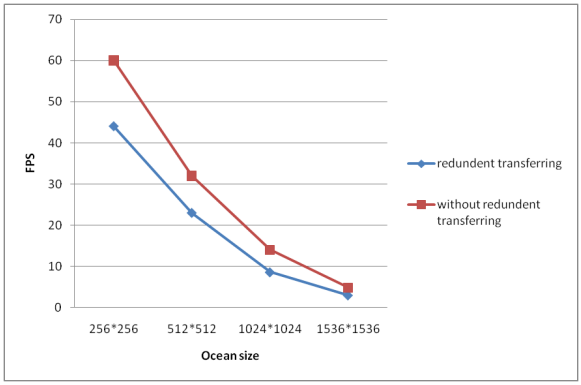


Figure 8. system performance

Blue bars in Figure 7 indicate the breakdown of GPU execution time before optimization, and red bars indicate those after optimization.

Before the optimization we observe memory transferring between host and device consumes most of the GPU execution time.

There are two reasons for this high data transfer time. First we transfer data from CUDA memory to OpenGL shader memory through the host memory. Second we generate new particles in CPU but the computation requires J_{-} to determine whether to generate the particles or not. The particle generation also requires $h(\mathbf{x}, t)$ and $\mathbf{D}(\mathbf{x}, t)$ to decide the locations of generated particles. All these data are in GPU memory so the data transfer demands tremendous communication time between host and device, for each frame we want to render.

CUDA provides a memory mapping mechanism that maps CUDA device memory to OpenGL[4] shader memory. Therefore OpenGL can directly accesses the data to be rendered on GPU memory without transferring through the host.

We tried to generate new particle on GPU to avoid redun-

dant data transferring, but we encountered a synchronization problem. When GPU generates new particles in parallel, they will compete for particle buffer, therefore they do not know which particle buffer segment are safe to use.

To address this synchronization problem we design a lock mechanism to control access to the particle buffer. We use A CUDA function `atomicInc` that guarantees to add 1 to a memory location without interference from other threads. We use this lock to protect the index of particle buffer so that threads can access the particle buffer safely and concurrently.

Recall that red bars in Figure 7 indicate the breakdown of GPU execution time *after* optimization. After we remove the redundant data transferring we observe that the memory transferring in Figure 7 reduces from 30 percent (blue bar, before optimization) to 3 percent (red bar, after optimization). The system performance improves about 30 percent in Figure 8.

VIII. Conclusion

We implement an FFT-based ocean wave simulator that has choppy effect and detect regions that need to spray particles, and spray particles at those regions. We implement this large scale (512×512 grids) ocean wave simulator with GPU, which achieves real-time (more than 30 fps) particles spraying performance for 131072 particles.

We conclude that GPU does provide significant computing power that enables complicated Nature phenomenon simulations in real-time. However, the memory bandwidth between host and device could become a performance bottleneck. As a result how to avoid communication between host and device is a crucial issue while using CUDA [1] together with OpenGL shader.

References

[1] Cuda. http://developer.nvidia.com/object/gpucomputing.html.

[2] Cuda sdk. http://www.nvidia.com/object/cuda_get_samples.html.

[3] Nvidia compute visual profiler. http://developer.nvidia.com/object/cuda_3_1_downloads.html.

[4] Opengl. http://www.opengl.org.

[5] Opengl cube map texturing. http://developer.nvidia.com/object/cube_map_ogl_tutorial.html.

[6] A. Fournier and W. T. Reeves. A simple model of ocean waves. In *Computer Graphics, Vol.20, No.4*, 1986.

[7] L. S. Jensen and R. Golias. Deep-water animation and rendering. In *Gamasutra article on real-time water*, September 2001.

[8] O.M.Phillips. The equilibrium range in the spectrum of wind-generated waves. February 1958.

[9] S. Premoze and M. Ashikhmin. Rendering natural waters. In *PG '00: Proceedings of the 8th Pacific Conference on Computer Graphics and Applications*, pages 23–30. IEEE Computer Society, 2000.

[10] J. Tessendorf. Simulating ocean water. In *SIGGRAPH course notes*, 1999.

[11] N. Tomoyuki and N. Eihachiro. Method of displaying optical effects within water using accumulation-buffer. In *Proc. of ACM SIGGRAPH*, August 1994.