2012 International Conference on Image, Vision and Computing (ICIVC 2012)
IPCSIT vol. 50 (2012) © (2012) IACSIT Press, Singapore
DOI: 10.7763/IPCSIT.2012.V50.34

GPU-Accelerated Large-Scale Water Surface Simulation

Xiaohua Ren [+] and Xin Zhou

College of Computer Science

National Key Laboratory of Fundamental Science on Synthetic Vision, Sichuan University

Chengdu, China

Abstract. General-purpose computation on graphics processors (GPGPU) has become a powerful tool for computationally demanding tasks in a wide variety of application domains. In this paper, efficient algorithms, coupled with data structures used for GPU computation, are presented for computer simulation of water surface. To date no other hardware accelerated method has supported for creating triangle mesh for water surface rendering. Our method is based on triangle strips supported by mainstream graphics library. To compute water surface normal, we propose a GPU-based search algorithm according to our triangle mesh generation method. Further, to render large-scale water surface, a GPU-based projected grid algorithm is presented.

Keywords: GPGPU computing; water surface simulation; triangle strips; projected grid

1. Introduction

With the increasing demand for more complex animations, interactive rendering of natural phenomena becomes a very important issue. As an indispensable part of nature, water surface simulation is one of most daunting challenges for its complex real-time computation of mesh generation, height field and normal calculation etc. On the other hand, with the rapid increase in the performance and computation ability of graphics hardware, GPGPU (for "General Purpose GPU") computing has motivated researchers and developers to harness its power to hack problems needed complex computation [12].

In this work, algorithms and data structures are proposed to move those computations from CPU to GPU. The main contributions of this work are:

1) Using projected grid method to create view-dependent four bound corner points,
2) An efficient triangle-strip-based method and GPU-friendly data structures to generate triangle mesh on the GPU,
3) A GPU-Based search algorithm to compute normal.

2. Previous Work

Kryachko [8] used a static radial grid, centered at the camera position, to tessellate the water surface. However the number of tessellated vertices that end up within the view frustum is about 25%.

In [3, 14, 16], standard continuous LOD (level-of-detail) techniques ROAM and adaptive quad-tree were presented, however, they are too complex and performance consuming to be used as method of water surface tessellation in real-time.

[+] Corresponding author. Tel.: + (15881130455).
E-mail address: (xiaohuasuper@163.com).

Demers [2] tessellated in eye space and mapped a regular grid to the ocean plane within the view frustum. This allows users to render only the visible geometry and tessellate more finely in the foreground than the background. However, further implementation detail is no currently available.

Johanson [6] introduces projected grid algorithm to tessellate the visible region of the water surface according to the current viewpoint. Contrast to the LOD scheme, projected grid deliver a polygonal representation that provides spatial scalability along with high relative resolution without resorting to multiple levels of detail. However, Johanson gives a CPU-based implementation to generate triangle mesh which burdens CPU heavily. In this paper, we use projected grid concept to calculate four bound corner points which change with the current viewpoint. Using generated corner points, we use our GPU-accelerated method to generate triangle strips mesh.

3. Water Surface Simulation

3.1. Projected Grid

In this section, we briefly describe the projected grid concept [6] and subsequently show how this approach can calculate four bound corners changing with the viewpoint.

Vertex data transformed from its original state into the screen coordination will pass several spaces via using 4x4 matrices. Vertex transformations process is shown in Figure 1. The matrix notations from OpenGL [4] will be used.

Fig 1: Vertex transformations process and Matrix notation used in this paper

A point is transformed from world space to projected space by transforming it through the view and perspective matrices ($M_{view} \cdot M_{perspective}$) of the camera. If that transform is inverted, we obtain the following:

$$M_{projector} = [M_{view} \cdot M_{perspective}]^{-1} \quad (1)$$

$$P_{world} = M_{projector} \cdot P_{projector} \quad (2)$$

Equations (1) and (2) show how a point is transformed from projected space to world space. The camera frustum (view frustum) is defined as the volume in which geometry has the potential to be visible by the camera. In projected space, the camera frustum is a cube which is defined as follow:

$$P_{projector} = \{(x, y, z) \mid x, y, z \in [-1,1]\} \quad (3)$$

From definition (3), equations (1) and (2), we can obtain world-space coordinates $p_i (i = 0...7)$ of its eight corner points ($\pm 1, \pm 1, \pm 1$). Using these eight corners, we will calculate four bound corner points. The relationship between camera frustum and water surface is shown in Figure 2:

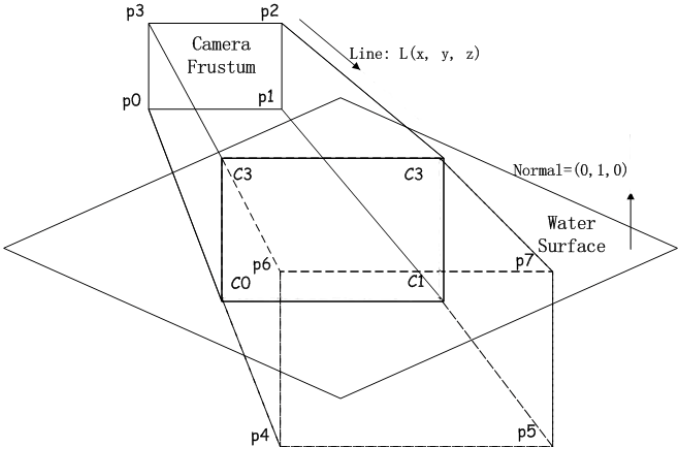

Fig 2: world-space points $p_i (i = 0...7)$ and bound corner points $c_i (i = 0,1,2,3)$

Water surface and lines of camera frustum equations is defined as follow:

$$Plane: y / w = h \text{ (for simplicity's sake)} \quad (4)$$

$$Line: L(x, y, z) = (x + \Delta x \cdot t, y + \Delta y \cdot t, z + \Delta z \cdot t, w + \Delta w \cdot t) \quad (5)$$

From equations (4) and (5), we can obtain the line-plane intersection with homogeneous coordination by the following equality:

$$\frac{y + \Delta y \cdot t}{w + \Delta w \cdot t} = h \Leftrightarrow \frac{w \cdot h - y}{\Delta y - \Delta w \cdot h} \quad (6)$$

Using t given by (6), the point of intersection in homogeneous coordination is $(x + \Delta x \cdot t, y + \Delta y \cdot t, z + \Delta z \cdot t, w + \Delta w \cdot t)$. Through illustration above, we can calculate corner points $c_i (i = 0,1,2,3)$.

3.2. Mesh Triangle Strip Generation

In this section, we will describe our triangle strip mesh generation approach. First we will introduce triangle strip.

Triangle strips [5, 7] provide a compact representation of triangular meshes and are supported by mainstream graphics library. Considering the triangle strip in Figure 3, the set of triangles can be compactly represented by a triangle strip $^{(0,1,2,3,4,5,6,7,8)}$, where the i^{th} triangle is described by the i^{th}, $(i+1)^{st}$ and $(i+2)^{nd}$ vertices in this sequence. Such a sequential strip can reduce the cost to transmit n triangles from $3n$ to $n+2$ vertices.

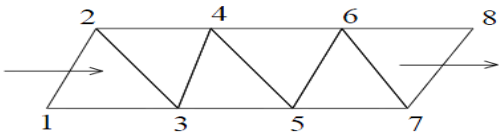

Fig. 3: A triangle strip

So our task is to use triangle strips to represent water surface. In [6], the author specifies one triangle with three vertices. Considering a $m \cdot n$ triangular mesh with $(m-1) \cdot (n-1) \cdot 2$ triangles, we can represent the mesh using only $(m) \cdot (n-1) \cdot 2$ vertices, while $(m-1) \cdot (n-1) \cdot 2 \cdot 3$ needed in [6]. The pseudocode of our triangle strip sequence generation algorithm is given in Figure 4:

```
1  Allocate a stack buffer p and assign an integer variable i = 0
   m = the mesh width and n = the mesh height
2  foreach v < n − 1
3    foreach u < m
4      push back i, i + m into p
5      if u < m − 1
6        if u is an even
7          i++
8        else
9          i−−
10       endif
11     else
12       i = i + m
13     endif
14   endforeach
15 endforeach
```

Fig. 4: Our triangle strip sequence generation algorithm

Using the triangle strip generated by the above algorithm, we can give a compact representation of water surface triangular mesh.

3.3. Wave Generation

In our simulation, we use a noise synthesis approach called Perlin noise [9] to simulate the appearance of the water surface. More in-depth understanding of Perlin noise is presented in [9, 10, and 13]. In order to simulate water waver, we need to layer multiple noise-functions at different frequencies and amplitudes to create a fractal noise. The frequency of each layer is usually the double of the previous layer which is so called an "octave".

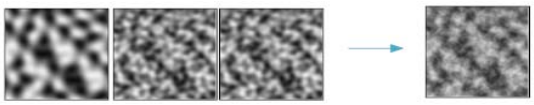

Fig. 5: Multiple octaves of Perlin noise sums up to a fractal noise.

In figure 5, three octaves of Perlin noise are layered to form a fractal noise. The following equation shows how a fractal noise can be created from multiple octaves of Perlin noise.

$$fnoise\ (x) = \sum_{i=0}^{octaves-1} \alpha^{i} \cdot noise\ (2^{i} \cdot x) \qquad (7)$$

where the parameter α is the amplitude. Higher values of α will form a rougher noise whereas lower values will make the noise smoother.

For the consideration of efficiency, we will make the precomputed fractal noise as a texture map, which will be used as our height map for water surface.

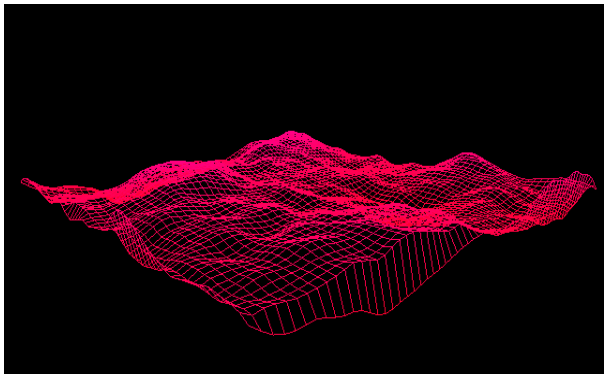

Fig. 6: Wave generation from a noise height map.

The figure 6 shows wave generation from a noise height map on a regular size water surface mesh.

4. Gpu-Implementation

Through illustration of section 3, preparations for the simulation have been made, which are four bound corner points, a mesh triangle strip sequence, and a Perlin noise height map. In this part, we will use those to implement water surface simulation on the GPU.

Contrast with the traditional CPU sequential programming model, the GPU has a different one called "Stream Programming Model" [1, 11] which needs us to design data structures fitting that model. In the stream model, the element in a stream (that is, an ordered array of data) is processed by the instructions in a kernel (that is, a small program). The Figure 7 shows the kernels and streams in our simulation.

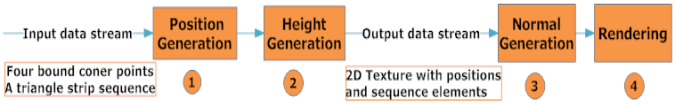

Fig. 7: The nodes are kernels and edges are streams.

The following sections will introduce parts respectively in Figure 7.

4.1. Data Structures

The most common GPGPU data structure is a contiguous multidimensional array. These arrays are often implemented by first mapping from N-D to 1D, then from 1D to 2D [1].

In our simulation, the input stream is a 1-D triangle strip sequence. We map this 1-D data to 2D and store in texture coordination.

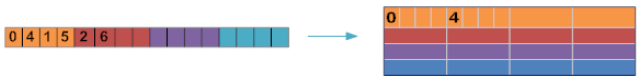

Fig. 7: Map 1D sequence to 2D texture coordination

The output data structure is illustrated by the following figure.

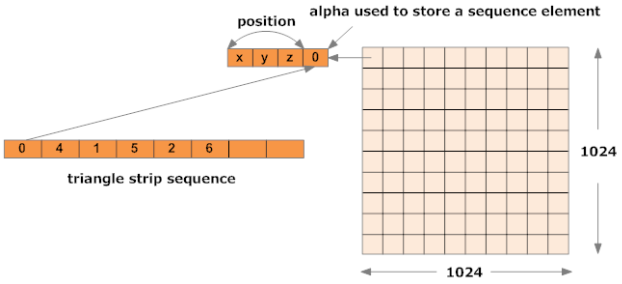

Fig. 8: Output data structure is a 2D texture.

The (r, g, b) channels store the mesh point position and alpha channel stores a sequence element which will be used to compute normal.

4.2. Mesh Generation on the GPU

In this section, we briefly describe the methods used in the kernel ① and ② in Figure 7.

In the kernel①, the bilinear interpolation method is used to generate point positions. The equation is shown below:

$$\begin{aligned} P_{01} &= (1-dx)\cdot c_0 + dx\cdot c_1 \\ P_{23} &= (1-dx)\cdot c_2 + dx\cdot c_3 \\ P &= (1-dy)\cdot P_{01} + dy\cdot P_{23} \end{aligned} \tag{8}$$

where $c_i (i = 0,1,2,3)$ is the corner points computed in part A of section 3, dx and dy is computed by the triangle strip sequence element.

In the kernel②, the height value is produced by sampling the noise map generated in the part C of section 3.

4.3. Computing Normal on the GPU

A search algorithm for computing normal is implemented in the kernel ③in Figure 7. Assuming the water surface S is given by the following function:

$$S: z = f(x, y) \tag{9}$$

Given a point P(x, y, z) on the surface S, the normal is computed by:

$$normal = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, -1) = -T_x \times T_y \tag{10}$$

where $T_x = (1,0,\frac{\partial f}{\partial x}), T_y = (0,1,\frac{\partial f}{\partial y})$. To discrete T_x and T_y, we use the centered first difference approximation:

$$T_x \approx (1,0,\frac{f(x+h,y)-f(x-h,y)}{2h}) \tag{11}$$

$$T_y \approx (0,1,\frac{f(x,y+h)-f(x,y-h)}{2h}) \tag{12}$$

From equations (11), (12) and (13), we can obtain point P's normal in Figure 9 (a):

$$normal_p = (P_0 - P_2)\times(P_3 - P_1) \tag{13}$$

So our problem becomes to search points $P_i(i=0,1,2,3)$ surrounding P. We search the sequence number $pos_i(i=0,1,2,3)$ of $P_i(i=0,1,2,3)$ instead of searching $P_i(i=0,1,2,3)$ directly and use them to sample the 2D texture marked in Figure 7 to get $P_i(i=0,1,2,3)$. We divide the search into four types showed in Figure 9.

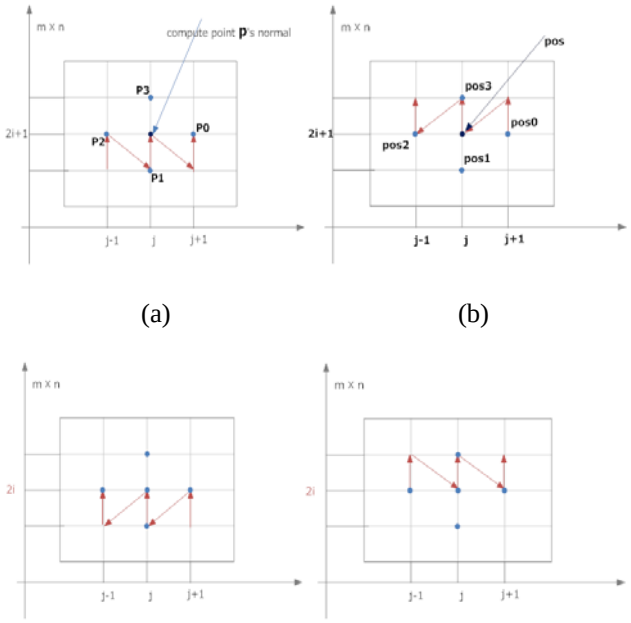

Fig. 9: Four types of point order sequence when processed by the kernel①. The red arrow lines are the triangle strip sequence flowing into the kernel. The center point is the element being processed by the kernel (that is, computing normal).

The core of the search algorithm is listed in Figure 10:

$$[pos_0, pos_1, pos_2, pos_3] =$$
$$(a):[pos+2, pos-1, pos-2, pos+(m-1-j)*4+1]$$
$$(b):[pos-2, pos-(m-1-j)*4-2, pos-(m-1-j)*4-2, pos+1]$$
$$(c):[pos-2, pos-1, pos+2, pos+j*4+2]$$
$$(d):[pos-j*4-2-1, pos-j*4-2, pos-2, pos+1]$$

Figure 10: Overview the core of the search algorithm. m is the width of the mesh. pos is the sequence number of the current element.

The complexity of our search algorithm is linear O (4*n).

5. Results and Discussions

We have implemented our algorithms on a PC with NVIDIA Quadro FX 1700 GPU.

Fig. 11: A screenshot for a mesh (32x32) generated by the kernel①.

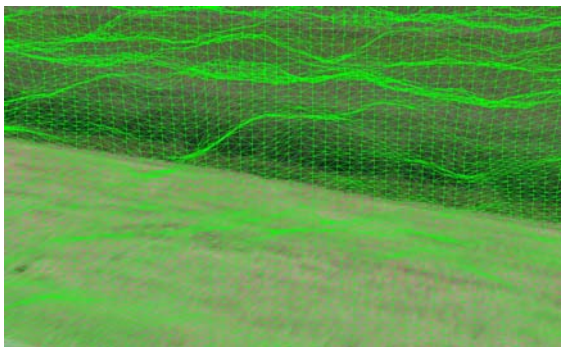

Fig. 12: A screenshot for a mesh (64x128) with height generated by the kernel②.

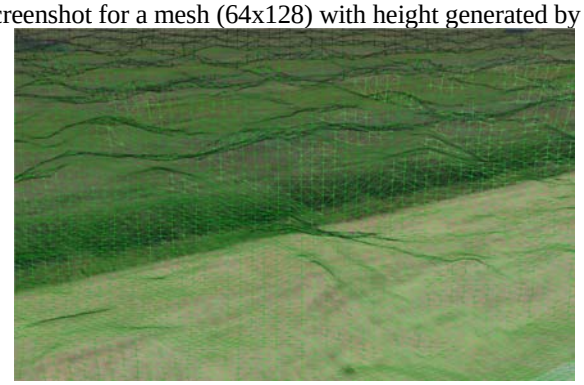

Fig. 13: A screenshot for a mesh (64x128) with normal generated by the kernel③.

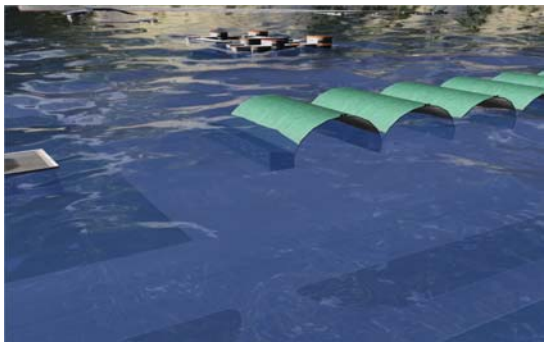

Fig. 14: A screenshot for water surface (128x256) rendered by the "Rendering" kernel in the Figure 7.

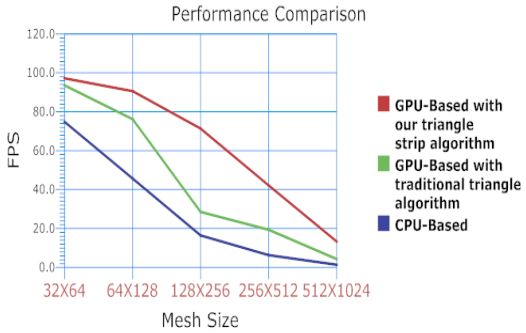

Fig. 15: Performance of three different implementations.

Figure 11 ~ 14 show the results generated by the kernels marked in the Figure 7. The difference between Figure 12 and Figure 13 is the optic effect calculated through normal computed in the "Normal Generation" kernel.

Figure 15 is performance comparison among three implementations. We can see that the red line FPS (frame per second) is rather higher than other two. The difference between the red line and the green line demonstrates that our triangle strip algorithm is quite more efficient than the traditional triangle algorithm which has been analyzed in the part B of section 3. Our GPU-Accelerated mesh generation method and GPU-based normal computation make the great difference between the red line and the blue line.

6. Conclusion and Future Work

In this paper, we present an efficient triangle-strip-based method to generate triangle mesh on the GPU, along with GPU-friendly data structures, and a novel GPU-Based search algorithm to compute normal. It is worth notice that our work is concentrating on enhancing the efficiency of the simulation, while ignoring the nature effects of the water surface. However, it is quite flexible to extend in our work. All needed to do is to improve the "Height Generation" kernel and "Rendering" kernel in Figure 7. To achieve more natural water wave, we can improve the “Height Generation” kernel with fast Fourier transformation (FFT) method [15]. All optic effects can be added and rendered in the "Rendering" kernel. Our future work is to achieve more natural water surface based on the current work.

7. Acknowledgment

We would like to thank Gang Li and Yanci Zhang for their support and enthusiasm. This work is supported by National “863” Plan Projects of China (No. 2009AA01Z332).

8. References

[1] BUCK I., FOLEY T., HORN D., SUGERMAN J., FATAHALIAN K., HOUSTON M., HANRAHAN P.: Brook for GPUs: Stream computing on graphics hardware. ACM Transactions on Graphics 23, 3 (Aug. 2004), 777–786.

[2] Demers, J., “The Making of ‘Clear Sailing,” Secrets of the NVIDIA Demo Team, CEDEC 2004.

[3] Duchaineau, M., Wolinsky, M., Sigetti, D.E, Miller, M.C., Aldrich, C., Mineev-Wienstein, M.B. ROAMing terrain: Real-Time Optimally Adapting Meshes. Proceedings of the 8th IEEE Visualization '97 Conference.

[4] Dave Shreiner etc. OpenGL Programming Guide, Seventh Edition. Addison-Wesley Publishing Company, 2009.

[5] F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In IEEE Visualization ’96 Proceedings, pages 319 – 326. ACM/SIGGRAPH Press,October 1996.

[6] Johanson, C., “Real-time water rendering,” Master of science thesis, Lund University, March, 2004.

[7] J. El-Sana, E. Azanli, and A. Varshney. Skip strips:Maintaining triangle strips for view-dependent rendering.In IEEE Visualization, pages 131–138,1999.

[8] Kryachko, Y., “Using vertex texture displacement for realistic water rendering,” GPU Gems 2, Chapter 18, 2005.

[9] K. Perlin. An image synthesizer. In B. A. Barsky (ed), Computer Graphics (SIGGRAPH ’85 Proceedings).

[10] K. Perlin. Improving noise. ACM. 2002: 681--682. ISBN 1-58113-521-1

[11] Lefohn. Implementing efficient parallel data structures On GPUs. In GPU Gems 2

[12] Owens, John, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron Lefohn, and Timothy Purcell. 2005. "A Survey of General-Purpose Computation on Graphics Hardware." In Proceedings of Eurographics 2005, State of the Art Reports, pp. 21–51.

[13] Stefan Gustavson. Simplex noise demystified. 2005-03-22 [2008-09-17]

[14] Turner, B. Real-Time Dynamic Level of Detail Terrain Rendering with ROAM. Gamasutra, April 03, 2000, http://www.gamasutra.com/features/200000403/turner_01.htm

[15] Tessendorf, J. Simulating Ocean Water. SIGGRAPH 2001 Course notes. http://home1.gte.net/tssndrf/index.html.

[16] Ulrich, T. Continuous LOD Terrain Meshing Using Adaptive Quadtrees. Gamasutra, February 28, 2000, http://www.gamasutra.com/features/200000228/ulrich_01.htm