ICIC Express Letters
Volume **10**, Number **6**, June **2016**

ICIC International ©2016 ISSN 1881-803X
pp. **1239–1245**

GPU-BASED SIMULATION OF OCEAN WATER USING FLUID DYNAMICS MODEL AND DISPLACEMENT MAPPING

Jie Xu[1], Jinhua Fu[2,3] and Hongtao Zhang[4,*]

[1]School of Software
[3]School of Computer and Communication Engineering
Zhengzhou University of Light Industry
No. 5, Dongfeng Road, Zhengzhou 450002, P. R. China

[2]State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou 450002, P. R. China

[4]College of Information Engineering
Zhengzhou University
No. 100, Kexue Ave., Zhengzhou 450000, P. R. China
*Corresponding author: htzhangzz@qq.com

Received November 2015; accepted February 2016

Abstract. *Real-time realistic simulation of liquids like ocean water is an important task nowadays in the field of computer graphics. In this paper, we present a novel method to achieve plausible visual results of ocean water at higher rendering rates. Based on evolution of the density and the velocity, we establish the fluid dynamics model to express properties of the fluid and simulate ocean water efficiently. Also we implement the displacement mapping by utilizing the heightmap and design the shading shader on the GPU (Graphics Processing Unit), which gives a realistic appearance in higher rendering speed. The results show that we can acquire realistic rendering effects such as water color, light reflection, caustics and soft shadow, while achieving a relatively faster speed of rendering.*

Keywords: Real-time realistic simulation, Ocean water, Fluid dynamics model, Displacement mapping

1. **Introduction.** The occurrences of fluids are very common in everyday life, and are witnessed in many different forms outside of the realm of just liquids such as smoke, fire, hurricanes, water, and other related phenomena. Ocean water simulation has become a hot topic in computer graphics in recent years, due to its popularity and frequent use in special effects for feature films, virtual reality and 3D games.

So far, many methods for water simulation have been proposed in the past. Fluid simulation in computer graphics begins with lower dimensional techniques such as the particle system [1, 2]. These lower quality techniques are used to create 2D shallow water models, and semi-random turbulent noise fields. Then Clavet et al. [3] attempt to improve the particle-based simulation by using viscosity. This simulation takes into account surface tension and tries to implement non-linear plastic behavior, which is achieved through the use of two techniques: the Eulerian grids and Lagrangian particles. However, they only achieve realistic small-scale behavior of substances such as paint or water as they splash on moving objects. For large-scale dynamic river motion phenomenon, Shi et al. [4] describe a combination of modeling methods for flood routing process simulation: FFT-based (Fast Fourier Transform) large-scale water surface modeling and dynamic flood peak generation on water surface. Especially considering optical properties of water, Algan et al. [5] design and implement a model for rendering and animating water drops moving on a surface under physical effects such as gravity, surface affinity and wind. Even though the physics of water droplets is rendered, it cannot be implemented in highly interactive

environments such as 3D games. Shi et al. [6] estimate the optical properties of polluted water regardless of the kinds and the concentrations of pollutants in water, which can generate polluted water effects unachievable by standard rendering methods. Recently, using the GPU-based method, Liu and Xiong [7] propose a novel rapid simulation of water interacting with objects; however, they have to improve the rendering speed in the future work. To integrate with the multi-core technique and improve the collaboration of CPU-GPU, Oh [8] introduces a novel single-phase particle simulation for trapped air bubbles in a turbulent water flow. His method can be easily implemented by extending an existing rigid body interaction of fluid solver. In order to simulate the transition between different types of flows. Lim et al. [9] model these transitions by constructing a very smooth fluid surface and a much rougher, splashy surface separately, and then blending them together in proportions that depend on the flow speed. Nowadays, with the increasing requirement of modern 3D technology, the main challenge of rendering is not only to approximate the realistic effect of ocean water to give naturally looking appearance, but also develop sufficiently fast method to allow for feasible computation based on GPU rendering pipelines.

Motivated by meeting the requirement, our research focuses on faster GPU-based rendering of ocean water while keeping visually plausible appearance via fluid dynamics model. We have to keep in mind that the look of the water depends mostly on how it interacts with the environment lighting. So for this reason, we have also implemented the water rendering via the displacement mapping, which is fit for 3D games. Displacement mapping is an effective technique that is becoming more common in the video game industry, especially with current generation PC's and the next generation of console development [10].

The rest of this paper is organized as follows. The fluid dynamics model for ocean water is established in Section 2. Realistic rendering via displacement mapping on the GPU is given in Section 3. The results are discussed in Section 4, and conclusions are drawn in Section 5.

2. Fluid Dynamics Model for Ocean Water.

2.1. Fluid simulation.

The fluid simulator is based on the Navier-Stokes equations, given as follows:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u}\cdot\nabla)\mathbf{u} + \nu\nabla^2\mathbf{u} + f \tag{1}$$

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u}\cdot\nabla)\rho + \kappa\nabla^2\rho + S \tag{2}$$

where $\mathbf{u}$ is the velocity, ρ is the density, ν is the viscosity, κ is the diffusion coefficient, f is the external force, and S is the mass source.

In this paper, we consider ν and κ as constants, which describe the properties of the fluid. Fluid where ν can be considered as a constant is called as Newtonian fluid. Here, $\mathbf{u}$ and ρ fully describe the state of the fluid. The terms f and S can be used as inputs to control the fluid. The simulation basically consists of two steps: evolution of the density, and evolution of the velocity. We consider a $(N+2)\times(N+2)\times(N+2)$-divided mesh, and calculate the time evolution of the density and velocity within the three dimensional field.

2.2. Evolution of density.

The time evolution of fluid density is described by Equation (2). We will first observe the second term, which is the diffusion term. Naively discretizing Equation (2) with the forward difference operator yields the difference equation

$$\begin{aligned}\rho^{t+1}(i,j,k) = \rho^t(i,j,k) + \frac{\Delta t}{\Delta x^2}\kappa(&\rho^t(i+1,j,k) + \rho^t(i-1,j,k) + \rho^t(i,j+1,k)\\ &+\rho^t(i,j-1,k) + \rho^t(i,j,k+1) + \rho^t(i,j,k-1) - 6\rho^t(i,j,k))\end{aligned} \tag{3}$$

where Δt and Δx are the time steps and mesh widths, respectively. Equation (3) provides an iteration for evolving ρ over time, where the density of the next step is an explicit function of the previous steps.

However, this method is an unstable algorithm, i.e., ρ can diverge depending on the simulation settings, such as values of N and κ. We avoid this by using a backward difference operator for the discretization, yielding the difference equation

$$\rho^t(i,j,k) = \rho^{t+1}(i,j,k) - \frac{\Delta t}{\Delta x^2}\kappa\big(\rho^{t+1}(i+1,j,k) + \rho^{t+1}(i-1,j,k) + \rho^{t+1}(i,j+1,k) + \rho^{t+1}(i,j-1,k) + \rho^{t+1}(i,j,k+1) + \rho^{t+1}(i,j,k-1) - 6\rho^{t+1}(i,j,k)\big) \quad (4)$$

This equation can be interpreted as a system of linear equations

$$\rho^t(i,j,k) = \frac{\Delta t}{\Delta x^2}\left(\frac{\Delta x^2}{\Delta t} + 6\ 1\ 1\ 1\ 1\ 1\ 1\right)\begin{pmatrix} \rho^{t+1}(i,j,k) \\ \rho^{t+1}(i+1,j,k) \\ \rho^{t+1}(i-1,j,k) \\ \rho^{t+1}(i,j+1,k) \\ \rho^{t+1}(i,j-1,k) \\ \rho^{t+1}(i,j,k+1) \\ \rho^{t+1}(i,j,k-1) \end{pmatrix} \quad (5)$$

By collecting this equation for all i, j and k, we obtain the linear equation in the simple form

$$\mathbf{A}\rho^{t+1} = \rho^t \quad (6)$$

where ρ^{t+1} and ρ^t are vectors constructed by straightening out the i, j, k components of $\rho^{t+1}(i,j,k)$ and $\rho^t(i,j,k)$, respectively, into column vectors. Since $\mathbf{A}$ and ρ^t are known, calculating the time evolution of diffusion reduces to solving this linear system of equations for ρ^{t+1}.

The sparse structure of $\mathbf{A}$ can be used for efficient calculation of the solution. In this paper, an iterative method known as the Gauss-Seidel method is used. The third term of Equation (2), S, can be implemented trivially by simply incrementing ρ by $S\Delta t$ in each time step.

2.3. **Evolution of velocity.** Due to the similarity of Equation (1) and Equation (2), evolution of velocity can be done using the exact same formulations as the evolution of density. The difference in the velocity step is that some transformations are applied to satisfy mass conservation, for more realistic results. This appears as a preprocessing step for $\mathbf{u}$ in the algorithm.

This step can be explained as follows. We have the assumption that the flow of the fluid is incompressible, i.e.,

$$\nabla \cdot \mathbf{u} = 0 \quad (7)$$

It is known that combined with the Navier-Stokes equations, Equation (1) and Equation (2), this is equivalent to the equation

$$\frac{\partial \rho}{\partial t} + \mathbf{u} \cdot \nabla \rho = 0 \quad (8)$$

which is also known as the continuity equation, an equation that describes mass conservation in a differential form. Therefore, we have that incompressibility is equivalent to mass conservation.

However, the computed values of the velocity field do not necessarily satisfy Equation (7), due to computational errors and discretization errors. The violation of mass conservation hinders the quality of the simulation to a visually recognizable level. Therefore, we must somehow force the simulation results to satisfy mass conservation.

In this paper, this is accomplished by decomposing the computed velocity field to a mass-conserving part and non-mass-conserving part, and subtracting the non-mass-conserving part away from the computed velocity field, which is a process called Hodge decomposition [11].

Let $\mathbf{u}'$ be the computed velocity field, where $\nabla \cdot \mathbf{u}' := g \neq 0$, it is known that every three dimensional vector field can be decomposed using a vector potential and a scalar potential. Therefore, there always exists some vector potential χ and scalar potential ϕ, such that $\mathbf{u}$ can be decomposed as

$$\mathbf{u}' = \nabla \times \chi + \nabla\phi \tag{9}$$

By taking the divergence of Equation (9), we have

$$\nabla \cdot \mathbf{u}' = \nabla \cdot \nabla \times \chi + \nabla^2\phi \tag{10}$$

therefore

$$g = \nabla^2\phi \tag{11}$$

Equation (11) is known as the Poisson equation. Suppose that we have found solved Equation (11) and found ϕ. We can then construct a new velocity field

$$\mathbf{u}'' := \mathbf{u}' - \nabla\phi \tag{12}$$

which must satisfy

$$\nabla \cdot \mathbf{u}'' := \nabla \cdot (\mathbf{u}' - \nabla\phi) = \nabla \cdot (\nabla \times \chi) = 0 \tag{13}$$

Therefore, the new velocity field $\mathbf{u}''$ satisfies Equation (7), ultimately meaning that it satisfies mass conservation. Therefore, we can use this transformation from $\mathbf{u}'$ to $\mathbf{u}''$ to force the computed velocity field to satisfy the mass conservation law.

In this paper, this transformation, i.e., solving Equation (11) and using Equation (13) is done before computing the diffusion and advection terms for the velocity field. The Poisson equation, Equation (11), is solved by discretization, which yields a linear system of equations. The Gauss-Seidel method is used for solving the obtained linear system of equations, where the sparse structure is used in this case as well.

3. **Realistic Rendering via Displacement Mapping on the GPU.** Reflections, refractions and appearance details contribute the most to the perceived realism of the simulation of water surfaces. In order to express these realistic effects, we design and implement the displacement mapping which is shown in Figure 1. We take that one step further by utilizing an additional map called a heightmap, which describes the bumps and crevices of a surface. A heightmap is essentially just a gray scale image where each pixel is interpreted as a height value. This is used when we tessellate a mesh, where the heightmap is sampled in the domain shader to offset vertices in the normal vector direction, which in turn, adds geometric detail.

FIGURE 1. Displacement mapping effects

The following formula is used to displace a vertex position P', where the outward surface normal vector N is used as the direction of displacement:

$$P' = P + (h - 1) \times N \tag{14}$$

where this equation "*pops*" the geometry inward by using the h value that was obtained from the heightmap.

When rendering on the GPU, each triangle has to be tessellated differently depending on how close or far it is to the eye. The closer it is, the more tessellation it receives, and vice versa. The vertex shader helps compute a distance to determine this amount, which is then passed onto the hull shader. Add the following shader code to $DisplacementMap.fx$ which creates a linear function of distance that determines how much to tessellate based on distance of triangle.

```
// Transform to world space.
vout.PosW=mul(float4(vin.PosL, 1.0f), gWorld).xyz;
vout.NormalW=mul(vin.NormalL, (float3x3)gWorldInvTranspose);
vout.TangentW=mul(vin.TangentL, (float3x3)gWorld);
// Output vertex attributes for interpolation across triangle.
vout.Tex=mul(float4(vin.Tex, 0.0f, 1.0f), gTexTransform).xy;
float d=distance(vout.PosW, gEyePosW);
// Normalized tessellation factor.
float tess=saturate((gMinTessDist-d)/(gMinTessDist-gMaxTessDist));
//Rescale [0,1] to [gMinTessFactor, gMaxTessFactor].
vout.TessFactor=gMinTessFactor+tess*(gMaxTessFactor-gMinTessFactor);
```

The domain shader takes inputs of the hull shader outputs, patch data, and tessellation factors, and it outputs the position of a vertex. It samples the heightmap and offsets the vertices in the normal direction, and is called for every vertex created by the tessellator. Add the following shader code to support the domain shader within the $DisplacementMap.fx$ file:

```
// Interpolate patch attributes to generated vertices.
dout.PosW=x*tri[0].PosW+y*tri[1].PosW+z*tri[2].PosW;
dout.NormalW=x*tri[0].NormalW+y*tri[1].NormalW+z*tri[2].NormalW;
dout.TangentW=x*tri[0].TangentW+y*tri[1].TangentW+z*tri[2].TangentW;
dout.Tex=x*tri[0].Tex+y*tri[1].Tex+z*tri[2].Tex;
// Interpolating normal can unnormalize it, so normalize it.
dout.NormalW=normalize(dout.NormalW);
// Begin displacement mapping.
const float MipI=20.0f;
float mipLevel=clamp((distance(dout.PosW, gEyePosW)-MipI)/MipI, 0.0f, 0.6f);
// Sample height map (stored in alpha channel).
float h=gNormalMap.SampleLevel(samLinear, dout.Tex, mipLevel).a;
// Offset vertex along normal.
dout.PosW+=(gHeightScale*(h-1.0))*dout.NormalW;
// Project to homogeneous clip space.
dout.PosH=mul(float4(dout.PosW, 1.0f), gViewProj);
```

4. **Rendering Results.** We have implemented the fast simulation of ocean water using fluid dynamics model and displacement mapping on the GPU. All experiments are run on a PC with AMD Athlon II X4 Four Cores, and NVIDIA GeForce GT430. The software platform is based on MS Visual Studio 2012 and OpenGL as the programming language. We also employ GLSL (OpenGL Shading Language) as vertex and fragment shader language on the GPU. We have given rendering effects of ocean water and compare the rendering speed of our method with the previous method. Our final rendering clearly shows the effective implementation of the above mentioned technique.

We firstly implement realistic rendering of ocean water in different views, and show different scenes to express different rendering effects. The images from our water simulation via our method are shown in following Figure 2.

From Figure 2, we can observe that the rendering appearance of ocean water using our method is realistic in such as water color, light reflection, caustics and soft shadow, which

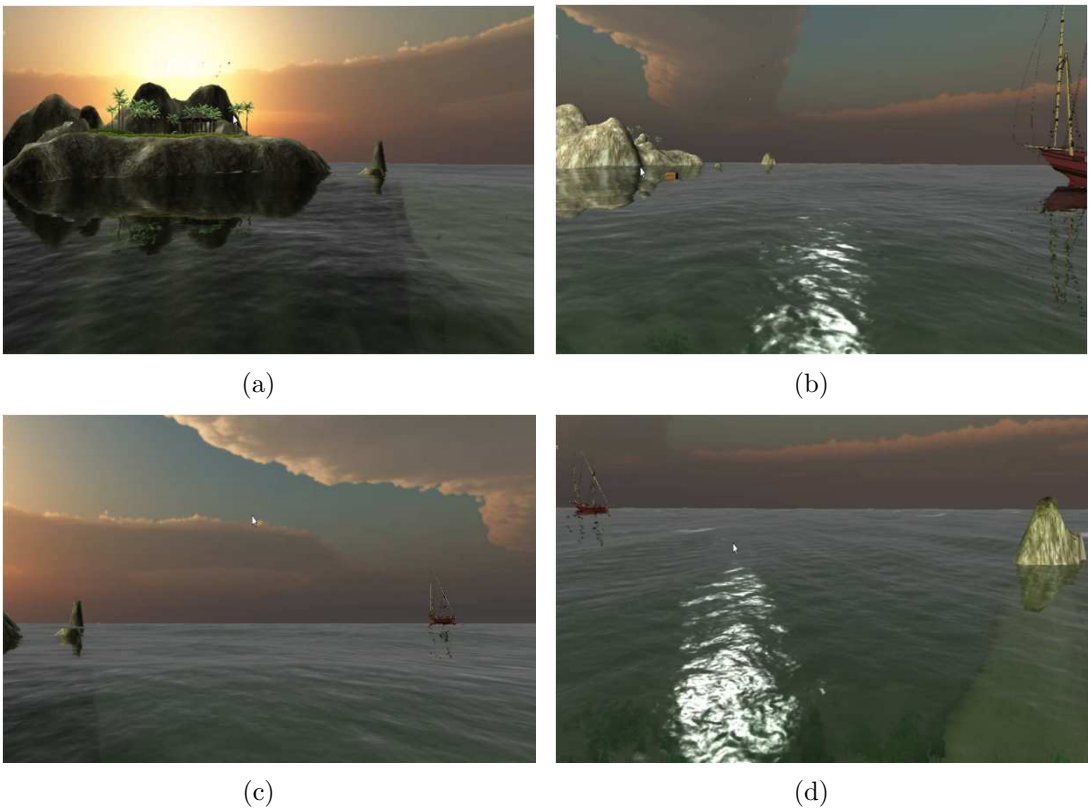

FIGURE 2. Rendering results of ocean water in different views

FIGURE 3. Foam effect via reference method [7]

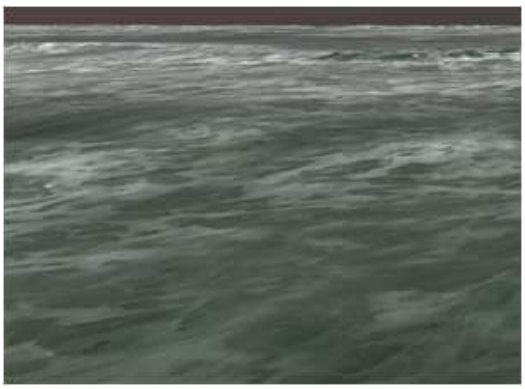

FIGURE 4. Foam effect via our method

is accord with physical phenomena. Part of it is reflected back in the upward direction and part of it is refracted inside the water volume. The reflected ray can further hit other objects causing reflective caustics. So we can simulate the physical phenomena when a ray hits the water surface.

Under different rendering methods, we show different scenes to express more realistic effects such as details in foam using our method as shown in Figure 3 and Figure 4. The foam effect in detail via our method is more obvious and clear, which shows our technique is very effective.

In the performance of rendering speed, using displacement mapping on the GPU we have obtained about 66 frames per second (FPS) when rendering 800 points; however, the real-time rendering method such as Liu and Xiong's method [7] only acquires 45 FPS. The difference of rendering time between our method and the method [7] is more obvious when increasing rendering points greatly. The difference of rendering speed is between

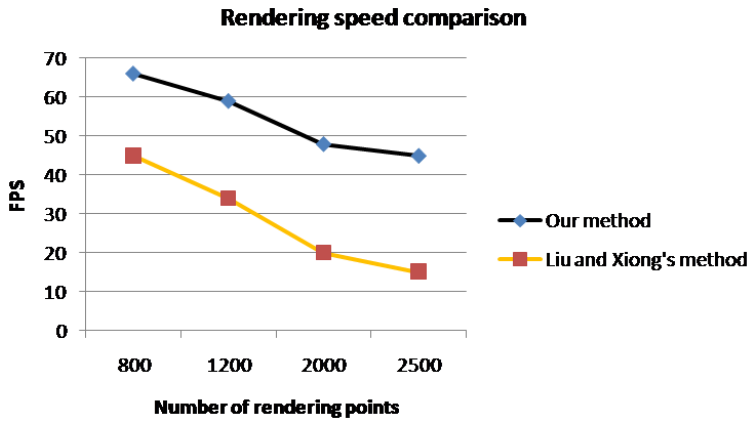

FIGURE 5. Rendering speed comparison between our method and the literature method [7]

two methods as shown in Figure 5. From Figure 5, our method is about 1.47-3 times faster than Liu and Xiong's method [7].

5. **Conclusions.** We have implemented the fast simulation of ocean water using fluid dynamics model and displacement mapping on the GPU. The rendering results show that our rendering method can obtain realistic effects of ocean water in such as water color, light reflection, caustics and soft shadow, which is accord with physical phenomena. In the performance of rendering speed, we are able to achieve faster GPU-based rendering while keeping visually plausible appearance of water. The difference of rendering speed is more obvious when increasing rendering points greatly. So our method has a great practical value in the industry of 3D games.

There are still some ways to improve the rendering program. One useful skill would be to implement particle-solid collision detection. This would open a lot of potential to create realistic scenes such as a waterfall or a fountain.

REFERENCES

[1] S. Premžoe, T. Tasdizen, J. Bigler, A. Lefohn and R. T. Whitaker, Particle-based simulation of fluids, *Computer Graphics Forum*, no.10, pp.401-410, 2003.

[2] M. Müller, D. Charypar and M. Gross, Particle-based fluid simulation for interactive applications, *Proc. of the ACM Siggraph/Eurographics Symposium on Computer Animation*, pp.154-159, 2003.

[3] S. Clavet, P. Beaudoin and P. Poulin, Particle-based viscoelastic fluid simulation, *Proc. of the ACM Siggraph/Eurographics Symposium on Computer Animation*, pp.219-228, 2005.

[4] S. Shi, X. Ye, Z. Dong and Y. Zhang, Real-time simulation of large-scale dynamic river water, *Simulation Modeling Practice & Theory*, vol.15, no.6, pp.635-646, 2007.

[5] E. Algan, M. Kabak, B. Ozguc and T. Capin, Simulation of water drops on a surface, *3DTV Conference: The True Vision-Capture, Transmission and Display of 3D Video*, pp.361-364, 2008.

[6] J. Shi, D. Zhu, Y. Zhang and Z. Wang, Realistically rendering polluted water, *Visual Computer*, vol.28, nos.6-8, pp.647-656, 2012.

[7] S. Liu and Y. Xiong, Fast and stable simulation of virtual water scenes with interactions, *Virtual Reality*, vol.17, no.1, pp.77-88, 2013.

[8] S. Oh, Single-phase trapped air simulation in water flow, *Proc. of WSCG*, 2014.

[9] J. G. Lim, B. J. Kim and J. M. Hong, Water simulation using a responsive surface tracking for flow-type changes, *Visual Computer*, pp.1-11, 2015.

[10] L. Szirmay-Kalos and T. Umenhoffer, Displacement mapping on the GPU-state of the art, *Computer Graphics Forum*, vol.27, no.6, pp.1567-1592, 2008.

[11] G. Schwarz, Hodge decomposition – A method for solving boundary value problems, *Lecture Notes in Mathematics*, vol.1607, 1995.