Procedural Water Shading in Real-time

Michael Novén*
M.Sc. Computer Science, Linköping University
TNM0084 - Procedural Methods for Images
January 12, 2017

Figure 1: *Procedurally genereated water and terrain*

Abstract

Water remains one of the most challenging things to simulate in computer graphics and is still a popular topic for research. This report aims to discuss some well-known methods for simulating water with their respective advantages and disadvantages and how procedural methods can be applied to get rid of the need for textures. Also, the process of implementing a real-time water shader in WebGL with Simplex noise and dynamic terrain is explained in detail. This implementation is meant to run in real-time applications in a fast frame rate entirely on the GPU on a mediocre laptop.

Keywords: water simulation, real-time, procedural methods

1 Introduction

In computer graphics, water can be really difficult to simulate due to its extremely complex nature and many attributes. A lot of research have been made through the years to describe the natural movement of water, where the Navier-Stokes equations have been dominating in this field how to describe water mathematically. However in movies and games, everything does not always need to be physically correct and shortcuts can be made to achieve a pleasant result that does not even have to look very realistic as long as it is satisfying in its context.

*e-mail:micno751@student.liu.se

2 Previous Work

A numerical solution to Navier-Stokes equations was early invented by Chorin [3] which made it possible to discretize them and run them on computers to simulate water. When speaking of physically based water simulation using mathematical approaches these days, two different methods are usually dominating which are the Lagrangian and Eulerian methods. These are both built on top of the Navier-Stokes equations and the difference is that Lagrangian is particle based while the Eulerian is grid based. The particle based uses discrete data objects that hold different attributes such as density, velocity, pressure etc. while the grid based divides the scene into a grid and keeps a state of the attributes of the particles as a position in the grid. These methods are often discussed and compared against each other where Braley and Sandu [2] claim that the grid based approach is often more accurate but slower than the particle based. Furthermore the particle based can be better suitable to simulate splashes of water while the grid based may have smoother transitions between the edge of the water and the borders.

Even though physically based water simulations give very accurate and realistic results they are often very expensive to compute. Therefore it is not always desirable to simulate water using the particle or grid based approaches, especially if interactive frame rates are desired. Methods that can be run entirely on the GPU have been developed in later years with the purpose to give pleasant results in real-time, often in games. Tesseldorf [9] made use of the Fast Fourier Transfoprm (FFT) to simulate realistic wave motions in real-time by computing a 2D wave height field on the CPU. Finch [5] proposed method that can be run entirely on the GPU using a dynamic normal map to achieve wave motion. Instead of

using FFT he proposed to use sums of sine functions because FFT was at that time difficult to perform on the GPU.

Most of the current methods use normal maps for lightning and derivative maps to for ripples or waves. Another way is to instead use a procedural approach with mathematical noise functions such as Simplex noise [6], this will get rid of the need for textures and scale to infinite sizes without having to consider texture resolutions and tiling. DS Ebert et al. [4] explains that one of the most important of procedural techniques is abstraction and storage saving. This practically means that it is possible to create inherent multi resolution textures and models that can evaluate to the resolution needed. Another big advantage is parametric control, which means that a few parameters yield large amounts of detail.

The aim of this report is to implement a water shader in WebGL with procedural generated ripples and terrain. The application should give a pleasant result that is efficient enough to be run in real-time applications such as games. The application should also make use of the parametric control that procedural techniques offers to be able to create different looks of the terrain and water by changing a single number.

3 Method

The application is developed in WebGL with the framework Three.JS and GLSL for shader programming. A scene is created with two planes, one plane represents the water and another plane represents the terrain and bottom. A directional light source is created over the plane in order to create diffuse and specular lightning on the water surface. The planes are shown as wireframes in Figure 2, where the number of triangles on the water plane has been increased for illustration purposes where it should only consists of a pair of two triangles.

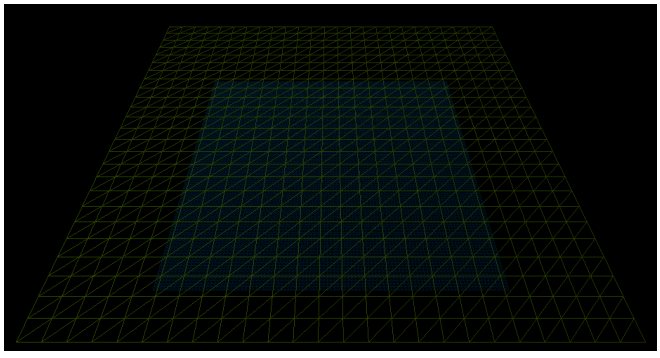

Figure 2: *Water and terrain planes*

4 WebGL Pipeline

WebGL is equivalent to OpenGL ES 2.0 for the web, which is a subset of OpenGL 3.0 and above. OpenGL ES 2.0 does not have any support for the old deprecated fixed function pipeline or any old OpenGL calls such as `glBegin()` or `glEnd()`, these are instead replaced by shaders. Also it does not exist any support for newer types of shaders such as compute shaders, geometry shaders or tessellation shaders. The only types of shaders that can be programmed by the developer are the vertex and the fragment shaders. No double-precision floating points are supported and the only data type allowed in the shaders is a single-precision float which can be problematic if working with a very large scene that requires accurate estimations of for example positions. No kind of 3D textures are supported either, even though these can be created using 2D textures this is is a lot more cumbersome to implement. An overview of the WebGL pipeline is shown in Figure 3

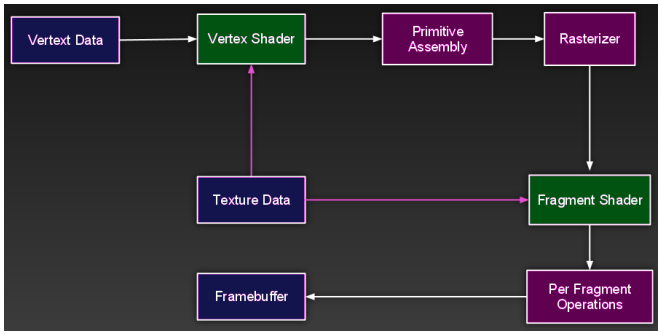

Figure 3: *WebGL Pipeline*

It should be noted that at the time of writing WebGL 2.0 is almost out and exists in the nightly versions of Chromium web browsers with the proper flags enabled. By using WebGL 2.0 almost all specifications in OpenGL ES 3.0 can be used.

5 Reflection and Refraction

To be able to simulate reflection and refraction projective texturing is used, which in the scope of water surfaces is described in detail by Belyaev [1]. The method he describes is to use multi pass rendering to render the scene to a reflection and a refraction texture which are passed to the fragment shader using clipping planes. The equation of a plane is described as:

$$Ax + By + Cz + D = 0 \tag{1}$$

Where A, B and C is the normal of the plane and D is the distance from the origin. When implementing clipping planes in WebGL, all points with a dot product of zero with the normal of the plane is cut away from the scene. This means that a clipping plane parallel to the water surface with a normal in positive y-axis can be defined when rendering to the reflection texture, and the same plane with a negative y-axis for the refraction texture. By using clipping planes

when rendering to the textures the performance is also increased by only rendering objects in the scene that will be sampled in respective texture. When rendering to the reflection texture the camera has to be flipped upside down by the y-axis, an illustration of this is shown in Figure 4.

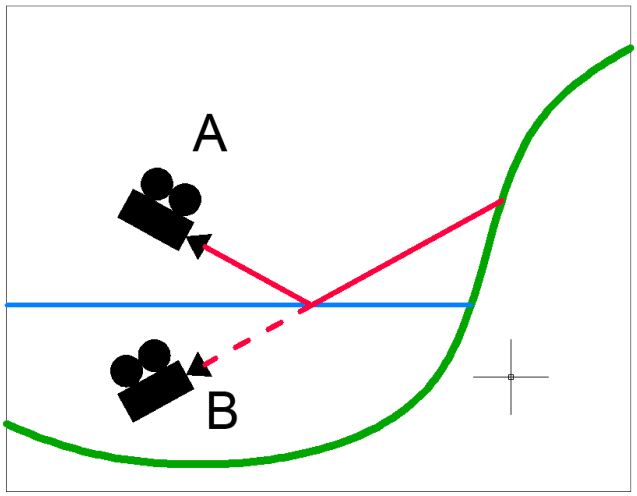

Figure 4: *The reflection camera is flipped by the y-axis when rendering to the reflection texture*

5.1 Fresnel

When simulating reflection and refraction of any surface, an important physical phenomena to take account is the Fresnel effect. The Fresnel effect describes how light behaves when passed through two different mediums and thus how much light should be reflected or refracted. The equations that describes this phenomena are given below:

$$R_s = \left| \frac{n_1 \cos\theta_i - n_2 \sqrt{1 - \left(\frac{n_1}{n_2}\sin\theta_i\right)^2}}{n_1 \cos\theta_i + n_2 \sqrt{1 - \left(\frac{n_1}{n_2}\sin\theta_i\right)^2}} \right|^2 \tag{2}$$

$$R_p = \left| \frac{n_1 \sqrt{1 - \left(\frac{n_1}{n_2}\sin\theta_i\right)^2} - n_2 \cos\theta_i}{n_1 \sqrt{1 - \left(\frac{n_1}{n_2}\sin\theta_i\right)^2} + n_2 \cos\theta_i} \right|^2 \tag{3}$$

$$R = \frac{R_s + R_p}{2} \tag{4}$$

Where n_1 and n_2 are the transmission indices between each medium and θ is the angle between the normal of the surface and the view direction. When applying this to a water surface this means that the water will be more reflective when looking parallel to the surface and more refractive when looking straight above the surface. The bottleneck of the Fresnel equations is that they are quite heavy to compute, since this is done for every single pixel in the scene every frame. To avoid the computational overhead Schlick's approximation is used instead, which can be seen as a simplified version of Fresnel more suitable for real-time applictions. These equations are given below:

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos\theta)^5 \tag{5}$$

$$R_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2 \tag{6}$$

Here, the indices are constant if air and water are the only types of medium in the scene and R_0 can simply be set as a constant value and does not need to be recalculated every frame.

6 Ripples

When sampling from the refraction and reflection texture, perfect samples are not desired since water does not behave as a perfect mirror or glass. To address this, some kind of distortion is needed and in terms of procedural approaches Simplex noise [6] is a popular way to accomplish this. Simplex noise is derived from the classic Perlin noise by Ken Perlin [8] but comes with a some advantages listed below:

- Lower computational complexity
- Scales to higher dimensions with significant less computational cost
- No noticeable directional artifacts
- continuous gradient everywhere that can be computed cheaply
- Easy to implement in hardware

Since the scene in this application is in three dimensions, Simplex noise is well suitable for this purpose. The idea is to sample the reflection and refraction texture using texture coordinates with an offset returned by the noise function in the vertex- and fragment shaders:

$$n_x = snoise(uv_x, uv_y, time) \tag{7}$$

$$n_y = snoise(uv_x, uv_y + time, c) \tag{8}$$

Where a time variable is passed to the shaders as a uniform variable to get a moving effect of the ripples since water is rarely completely still in reality.

7 Terrain Generation

To create the terrain and bottom around the water surface a Gaussian function in two dimension is used in the vertex shader:

$$f(x,y) = A\exp\left(-\left(\frac{(x-x_0)^2}{2\sigma_x^2} + \frac{(y-y_0)^2}{2\sigma_y^2}\right)\right) \quad (9)$$

Where x_0 and y_0 are the center point, σ_x and σ_y are the spreads in the x and y direction and A is the amplitude. Simplex noise is also applied to the amplitude and the spreads in the xy-plane to distort the edges and make the terrain less perfect in order to achieve a more realistic result.

The height of each vertex is passed to the fragment shader and interpolated to determine if the output color should be grass or mud for each pixel. Some additional noise is also applied to mix each output color with sand.

8 Lightning

In order to apply lightning to objects in computer graphics, the normals of the objects are used. Since the normals of the planes are pointing straight up in positive y-direction the lightning becomes pretty unrealistic. The perturbed vector from the noise function needs to be calculated to give a more realistic result.

As mentioned in section 6, Simplex noise has a continuous gradient that can be calculated easily, using the library for Simplex noise the gradient is obtained and accumulated with every noise seed. This gradient can be used to calculate the perturbed normal in 3D object space which is described in detail by Gustavson [7]. The method is simply to project the gradient onto the normal and subtract the part which is parallel to the normal from the gradient. This gives the part that is orthogonal to the normal and can be added directly to the normal. The equations are displayed below, where the gradient $\boldsymbol{g}$ is obtained by the Simplex noise function:

$$\boldsymbol{g}_{\parallel} = (\boldsymbol{g}\cdot\hat{\boldsymbol{N}}_{\mathbf{0}})\hat{\boldsymbol{N}}_{\mathbf{0}} \quad (10)$$

$$\boldsymbol{g}_{\perp} = \boldsymbol{g} - \boldsymbol{g}_{\parallel} \quad (11)$$

$$\boldsymbol{N} = \hat{\boldsymbol{N}}_{\mathbf{0}} - \boldsymbol{g}_{\perp} \quad (12)$$

$$\hat{\boldsymbol{N}} = \frac{\boldsymbol{N}}{|\boldsymbol{N}|} \quad (13)$$

With the perturbed normals, the Phong reflection model can be applied to both the terrain and water:

$$I_p = k_a i_{m,a} + \sum_{m\,\in\,lights}\left(k_d(\hat{L}_m\cdot\hat{N})i_{m,d} + k_s(\hat{R}_m\cdot\hat{V})^{\alpha}i_{m,s}\right) \quad (14)$$

Where I_p is the outgoing light at a point p. The constants k_a, k_d and k_s are the attribute of the ambient, diffuse and specular material where $i_{m,a}$, $i_{m,d}$ and $i_{m,s}$ are the intensities at each light source. $\hat{L}_m$ is the view direction, $\hat{R_m}$ is the perfect reflection vector and $\hat{V}$ is the view direction, all of these vectors are normalized. The specular highlight depends on α. An illustration over this model i displayed in Figure 5,

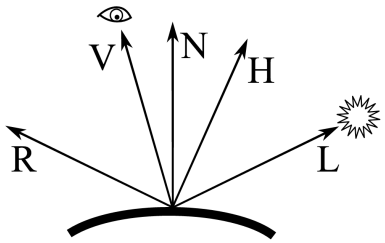

Figure 5: *Illustration over Phong reflection model*

from this figure it is easy to see the perfect reflection vector that can be calculated as:

$$\hat{R}_m = 2(\hat{L}_m\cdot\hat{N})\hat{N} - \hat{L}_m \quad (15)$$

9 Results

After implementing the parts in the previous secitons a result is achieved which is displayed in Figure 6. The application is run in Chrome version 54 on a Macbook Air OSX 10.11.6 with a 1.3 GHz Intel Core i5 CPU and Intel HD Graphics 5000 GPU in almost constant 60 fps.

Figure 6: *Final result*

If the camera is placed almost parallel to the plane the reflection can be seen clearly which is a result from the Fresnel term. This is shown in Figure 7.

The parameters of the terrain can be tweaked live through a GUI to change the seed and strength of the noise applied to the Gaussian function. An alternative more extreme terrain is shown in Figure 8 where the noise plays a more significant role to the final result.

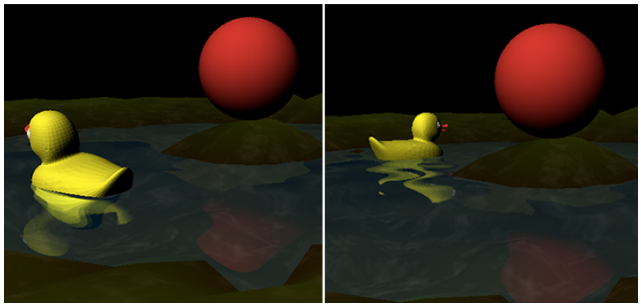

Figure 7: *The result of the Fresnel term and reflection texture*

Figure 8: *Terrain with increased strength of Simplex noise*

10 Conclusion

Creating realistic water in real-time entirely on the GPU without having to solve complex mathematical equations is entirely possible and gives a realistic result that can easily be applied to real-time applications such as games or other demos. However the approach described in this paper is meant to take place in a scene that does not require any user interaction, if any interaction with the water is needed some other approach might me more suitable.

In this application the water is still and there are no real waves. If one was about to implement waves in the vertex shader with for example sums of sine functions, the normals when sampling the reflection and refraction texture need to be handles differently depending on how the water moves. This technique is discussed in detail by Belyaev [1] in his approach of simulating water.

Using procedural approaches with mathematical functions to generate noise instead of the traditional derivative and normal maps has many advantages. Firstly, procedural noise is scalable in infinite number of dimensions where textures lack a lot of flexibility. Secondly, since the gradient of the noise is easily obtained the normal is easily calculated. By using normal textures, one has to match the normal map with the derivative map to give a realistic result and tiling the textures can be a problem. One of the major advantage of using normal textures is that artists are so used their software to create these, while the equivalent of creating the same texture procedurally would require programming. Also, today's GPU hardware are optimized for texture lookups and it might in a lot of cases be faster than procedural approaches. However, since the number of kernels of the GPU's increase rapidly these days while the amount of memory almost remains the same, procedural methods are likely to become even better in the future.

References

[1] V. Belyaev. Real-time simulation of water surface. 2003.

[2] C. Braley and A. Sandu. Fluid simulation for computer graphics: A tutorial in grid based an particle based methods. 2010.

[3] A. J. Chorin. Numerical solution of the navier-stokes equations. 1968.

[4] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texture and Modeling, Third Edition: A Procedural Approach*. Morgan Kaufmann, 2003.

[5] M. Finch. Effective water simulation from physical models. *GPU Gems*, pages 5–29, 2004.

[6] S. Gustavson. Simplex noise demystified. 2005.

[7] S. Gustavson. Recomputing normals for displacement and bump mapping, procedural style. 2016.

[8] K. Perlin. An image synthesizer. 1985.

[9] J. Tessendorf. Simulating ocean water. 2001.