Pseudo-Realistic Ocean Waves Simulation

Chaitanya Shah chaitans@usc.edu
David Mohrhardt mohrhard@usc.edu
Stuti Rastogi srrastog@usc.edu
Vritvij Kadam vkadam@usc.edu

***Abstract*—Achieving realistic simulation of water and specifically ocean waves has been an important research topic in computer graphics and rendering. In this project, we implement the Gerstner Wave function to simulate ocean waves. We have used DirectX for this project. The idea is to create waves with some input parameters to control the behavior. Along with this, we have a wire frame shader to see the movement of the waves in detail.**

I. Introduction

In fluid dynamics, a trochoidal wave or Gerstner wave is an exact solution of the Euler equations for periodic surface gravity waves. The wave solution discovered by Gerstner in 1802 has sharper crests and flatter troughs. This makes the wave simulations more realistic as compared to water having a periodic circular look. The rounded water waves work better for ponds or lakes, but ocean waves need to have a higher level of steepness.

Apart from modelling the structure of waves to be as realistic as possible, it is also vital to have provisions to simulate accurate behaviour of ocean waves. A good model would make it easy to give a wide variety of waves from very low ripples to high storm water. It is also helpful to have a way to parametrise wind speed and direction that affects motion of waves in the sea.

The Gerstner wave function was originally developed long before computer graphics to model ocean water on a physical basis. As such, Gerstner waves contribute some subtleties of surface motion that are quite convincing without being overt. In Computer Graphics, usually a multi -directional extension of the traditional physical Gerstner wave is implemented, combined with Fast Fourier Transforms (FFT) for feasibility of animation of the wave motion.

The last part of a pseudo-realistic ocean wave simulation would be to have appropriate shading and lighting for the scene. It is common to use textures for water rendering, and there are many shaders readily available for the implementation of the same. Another way to go is to manually color the pixels based on vertex colors and lights. Gourard shading is however not favourable, since there is a vast expanse over which the specular highlight might not be visible. Usually, we can see a sharp shine on the ocean surface due to the sun. Hence, having the specular highlight is essential, and using Phong shading might make the rendering a little more realistic.

II. Theory

A. Wave Equations

Before we move on to the implementation details of this project, we need to define a few terms and talk about the equations that will lead to the construction of Gerstner Waves.

1) *Wavelength (L)*: The crest-to-crest distance for the wave
2) *Amplitude (A)*: The distance between the highest point and equilibrium
3) *Speed (S)*: The rate of displacement of the crest in the forward direction
4) *Phase Constant* (ϕ): A constant depending on the initial values of the wave motion
5) *Direction (D)*: The horizontal vector perpendicular to the wave front along which the crest travels
6) *Frequency* (ω): The number of crests of the wave that move past a given pont in unit time

7) *Q*: A parameter that controls the steepness of the waves
8) *t*: Time
9) *P(x, y, t)*: 3D position on the wave surface at time *t* for point *(x, y)* in the 2D horizontal plane

A few relations among these terms might be helpful to understand the math of the equations:

- $\omega = \sqrt{g \times \frac{2\pi}{L}}$
- $\phi = S \times \omega = S \times \frac{2}{L}$

The Gerstner wave function goes as follows:

$$\begin{aligned} P(x,y,z) = \\ &\Bigg(x + \sum (Q_i A_i \times D_i \cdot x \\ &\times cos(\omega_i D_i \cdot (x,y) + \phi_i t)), \\ &y + \sum (Q_i A_i \times D_i \cdot y \\ &\times cos(\omega_i D_i \cdot (x,y) + \phi_i t)), \\ &\sum (A_i sin(\omega_i D_i \cdot (x,y) + \phi_i t)) \Bigg) \end{aligned} \tag{1}$$

This is the equation for a wave i, as every wave can have a different Q, A, ω, ϕ and D value. Here, a value of 0 for Q_i just makes the Gerstner wave to be a regular sine wave. A large Q value might lead to loops at the crests. In practice, the parameter Q_i for each wave is derived from a general steepness parameter Q set by the artist. The formula

$$Q_i = \frac{Q}{(\omega_i A_i \times numWaves)} \tag{2}$$

can be used to derive Q_i for wave i. it is best to set a value of Q between 0 and 1 to get realistic waves.

A similar approach can be taken for selection of wavelength and amplitude values for all waves. We can choose a median wavelength and amplitude, and the wavelengths and amplitudes can be picked randomly in the range of half to double of that median value. The median value can be adjusted based on the kind of waves we wish to generate. It must be noted that for a wave of any size, the ratio of the wavelength to amplitude will be same as the ratio of the median wavelength to the median amplitude.

Controlling Q, median wavelength, median amplitude and number of waves gives us immense control on the type of waves we wish to generate.

We also need normals for shading computation for the waves. The equation is:

$$\begin{aligned} N(x,y,z) = \\ &\Bigg(- \sum (\omega_i A_i \times D_i \cdot x \\ &\times cos(\omega_i \times D_i \cdot P + \phi_i t)), \\ &- \sum (\omega_i A_i \times D_i \cdot y \\ &\times sin(\omega_i \times D_i \cdot P + \phi_i t)), \\ &1 - \sum (\omega_i \times Q_i \times \\ &\times sin(\omega_i \times D_i \cdot P + \phi_i t)) \Bigg) \end{aligned} \tag{3}$$

This equation is derived by first calculating the Binormal (B) and Tangent (T) vectors by taking derivatives of P with respect to x and y. The normal vector (N) is just a cross product of B and T.

B. Fast Fourier Transform

The fft generates the height field at discrete points $x = (nL_x/N, mL_z/M)$, where N and M are the number of vertices in the x and z axes respectively, L_x and L_z are the distances between the vertices in the x and z axes respectively, and

$\frac{-N}{2} \leq n < \frac{N}{2}$ and $\frac{-M}{2} \leq m < \frac{M}{2}$

The fft based wave height $h(\vec{x}, t)$ at a location $\vec{x} = (x, z)$, and time t is given by,

$$h(\vec{x}, t) = \sum_{\vec{k}} \tilde{h}(\vec{k}, t) exp(i\vec{k} \cdot \vec{x}) \tag{4}$$

where $\vec{k} = (k_x, k_z)$ is defined as,

$k_x = \frac{2\pi n}{L_x}$ and $k_z = \frac{2\pi n}{L_z}$

Also,

$$\tilde{h}(\vec{x}, t) = \tilde{h}_0(\vec{k})exp(i\omega(k)t) + \tilde{h}_0^*(\vec{-k})exp(-i\omega(k)t) \tag{5}$$

where $\omega(k) = \sqrt{gk}$ is the frequency and g is the gravitational acceleration constant equal to $9.8m/s^2$.

Now, we have the equation

$$\tilde{h}_0(\vec{k}) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(\vec{k})} \tag{6}$$

Where, ξ_r and ξ_i are independent Gaussian Random Variables and $P_h(\vec{k})$ is the Phillips spectrum given by,

$$P_h(\vec{k}) = A\frac{exp(-1/(kL)^2)}{k^4}|\vec{k} \cdot \vec{w}|^2 \tag{7}$$

Here, $\vec{w}$ is the wind direction and $L = \frac{V^2}{g}$ is the highest point in the largest possible wave arising from a continuous wind of speed V.

The displacement of the vertices in the x and z axes is given by the displacement vector,

$$\vec{D}(\vec{x}, t) = \sum_{\vec{k}} -i\frac{\vec{k}}{k}\tilde{h}(\vec{k}, t)exp(i\vec{k} \cdot \vec{x}) \tag{8}$$

And the Normal Vector $\vec{N}$ is given by the following equations:

$$\epsilon(\vec{x}, t) = \nabla h(\vec{x}, t) = \sum_{\vec{k}} i\vec{k}\tilde{h}(\vec{k}, t)exp(i\vec{k} \cdot \vec{x}) \tag{9}$$

and

$$\begin{aligned}\vec{N}(\vec{x}, t) &= (0, 1, 0) - (\epsilon_x(\vec{x}, t), 0, \epsilon_z(\vec{x}, t)) \\ &= (-\epsilon_x(\vec{x}, t), 0, -\epsilon_z(\vec{x}, t))\end{aligned} \tag{10}$$

III. Implementation

A. Project Framework

Our project required a real time rendering application to properly distinguish the waveforms as a function of time. To accomplish this, we decided to use the D3D11 Graphics Library for windows to create the visual window and render the vertices. This decision was primarily influenced by its nativity to the Windows (and by extension, Visual Studio) development environment, its intuitive GPU buffer system, and its simplistic rasterizer descriptor. Our current implementation works primarily on CPU to generate vertex positions and normal vectors based on the Gerstner Wave FFT computation. These values are then packed linearly into a vertex buffer that is bound to the renderer along with a shader technique (a set of shaders with specified input layouts for specified meshes).

It is important to note that, to save on GPU memory and increase cache coherency, we decided to use the built in DrawIndexed() call which utilizes single vertices and an index buffer to build triangles. We opted to keep a single list of vertices and an index buffer telling the renderer what set of three vertices makes up triangle. Instead of duplicating vertices in the buffer (where each vertex is made up of two 32 bit float3 totalling 24 bytes per vertex), we are able to reduce the duplication vertex size to just being represented by a single 32 bit integer value alongside the single set of vertices.

B. Scene setup

The mesh is generated as an N x N set of discrete vertices. The application utilizes a function that takes in inputs defining how many vertices are to be generated in the X direction, how many vertices are to be generated in the Z direction, the distance the vertices are to span in the X direction, and the distance the vertices span in the Z direction. Using these values, we are able to create a grid of vertices that encompass an area defined by the distance input parameters by create N vertices in a row, for N rows with an offset applied to each vertex. It allows for meshes to be generated in

close or far proximity based on the sizes provided. We then programmatically create the indices that represent the mesh.

Once the vertices are generated and the indices assigned, they are passed to the buffers. Since the mesh is created starting at the origin and extending in the positive X and Z directions, we had to rotate the mesh and manipulate the camera position and look at matrix to achieve the desired image. We rotated the mesh 45 degrees and lifted the camera in the Y direction by 10 units and set it to look at the center point of the mesh calculated as a function of the length of the mesh in both the X and Z directions.

C. Wave Function Generation

The simple Gerstner Wave function was implemented for a single wave. We made use of the clock() time to fetch a double time value instead of an int, to get better animation. The values of Q, A, ω, L, S, D and ϕ were tweaked based on trial and error. We had an initial idea of what some values should be. For example, Q should not be greater than 1, or there can be looping in the waves. Also, the A value should not be very high as it can lead to unrealistic simulations. Based on the grid dimensions, we were able to find reasonable A and ω values. ϕ was adjusted according to the speed of the waves we wanted.

The function itself was just outputting the new calculated vertex positions and the calculated normals in a class object. These values could then be used by the shaders.

D. FFT implementation

For simplicity in calculations we used a square grid $(N = M)$ in our FFT based implementation and the values of N and M had to be in powers of 2. We implemented a FFT class that computed the FFT for the Gerstner wave. We used the equations mentioned in the Section II.B to compute the height field, displacement and normal for the wave given the time t. We added the displacement to the grid vertices x and z components and assigned the height field to the y component, which is then returned to the renderer.

The inputs of amplitude A and the Wind direction vector were defined in the header file, and could be changed as needed to get variety in the shape and motion of the waves. All these computations of the equations were performed on the CPU.

E. Shaders

We have a shader file in our code that handles the Vertex Shader and Pixel Shader of Direct3D for vertex transformations from model space to projection space and to color the pixels. The Vertex shader is essentially just to apply the correct transformations on to the vertices of the mesh, essentially transformations from the world, view and projection spaces in DirectX. All these calculations are on the vertices, and do not deal with pixels. All the transformations occured in Vertex Shader, are fed to the Pixel Shader as an input, for further coloring the geometries.

The Pixel Shader, on the other hand, takes care of the coloring of the individual pixels on the screen. We have implemented Phong shading in our Pixel shader. We have an ambient light, as well as one directional light defined for the scene in the shader file. The directional light provides the specular and diffuse component. The specular component visible in our scene is supposed to mimic the effect of sunlight on the ocean. The normals that are computed by our Wave Generator functions are transformed to the projection space by Vertex shader, which then after transformation are passed to the Pixel shader. Pixel shader carries out the Phong shading calculations considering the vertex and normal data provided. For further improvement, we can use a texture file providing some water texture generating realistic look.

IV. Results

Here, Figure 1 shows a screen shot of our demo running Gerstner Waves with FFT in the solid rendering mode.

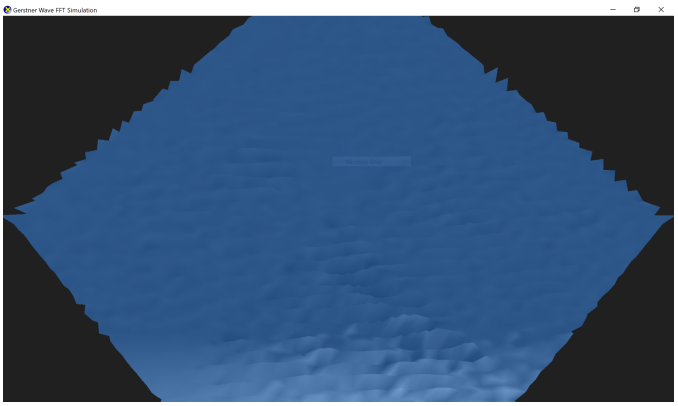

Fig. 1. Gerstner Waves (Solid)

Figure 2 is the output of the same function in the wire frame mode. The individual mesh vertices and triangles are visible here

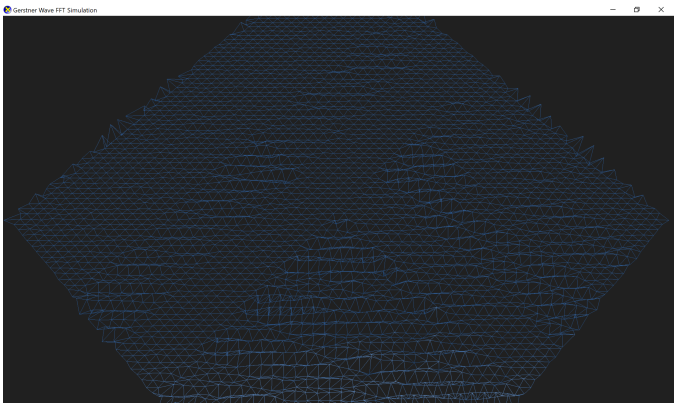

Fig. 2. Gerstner Waves (Wire Frame)

V. Issues

A. Gerstner Wave without FFT

Before implementing FFT, we worked on implementing the simple Gerstner Wave given in equation 1. However, as it is evident from Figure 3, this did not give very realistic results. In this version, we were making use of only one wave. We could add more waves, but this equation requires a number of parameters like steepness, amplitude, frequency, etc. to be set for each wave separately. These parameters give a lot of control on the structure of waves, but can be extremely difficult to figure out without knowledge of the derivations or architecture. Hence, we worked on improving the animation by using FFT to give us more realistic motion.

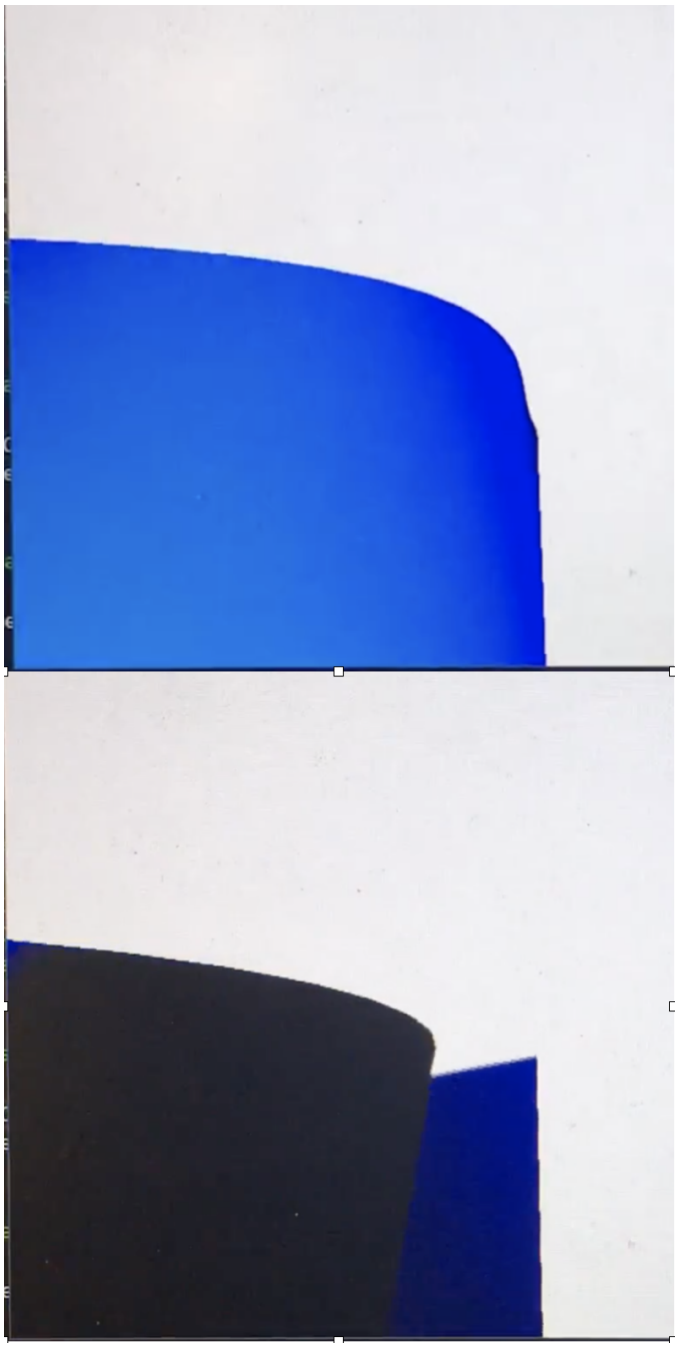

Fig. 3. Single Gerstner Wave

B. Artefacts

There are some artefacts visible at the edges of the plane. This is natural since we are trying to simulate an ocean, which essentially has an infinite

expanse, whereas in our simulation we have a plane of a given dimension. In reality, while using these waves, different meshes can be tiled next to one another and the heights would just map giving the impression of a continuous ocean with waves.

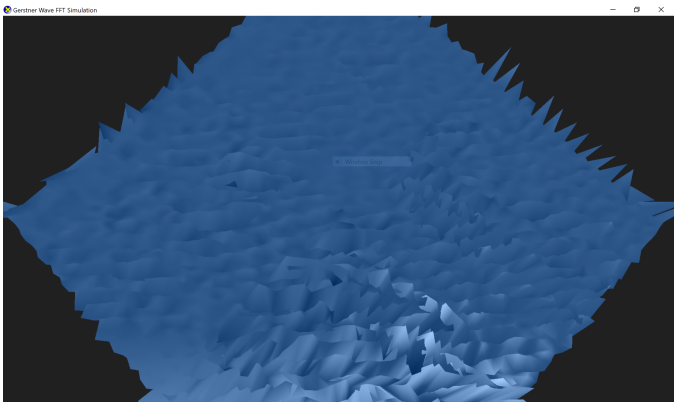

Fig. 4. Gerstner Waves with Loops (Solid)

Apart from this, we can also notice looping in the crests of the waves in some cases. This can be seen in Figures 4. This is a common problem with Gerstner waves and depends on the amplitude parameter (A) in the FFT implementation, or the steepness co-efficient (Q) in the simple Gerstner waves. If A exceeds beyond 1, what happens is that instead of getting very sharp peaks, the two edges cross across giving a loop. The problem can be fixed by choosing an appropriate A value.

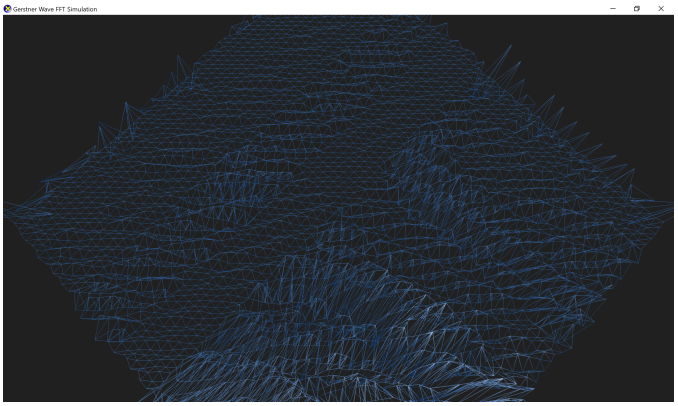

Fig. 5. Gerstner Waves with Loops (Wire Frame)

VI. Conclusion

This project while being implementation heavy, does a good job at providing the intricate details in the simulation of ocean waves. The goal was to accomplish as realistic geometry as possible. As it can be seen in the images and the demo, the focus has been on the modelling of the shape and motion rather than texturing. There are a number of ways available for realistic water texture rendering that can be integrated with this implementation of Gerstner waves.

This project was effective in teaching us about the wave function and the math behind it as well as it's wide use in Computer Graphics. We were also successful in learning to write shaders and became familiar with the DirectX framework.

Acknowledgment

We would like to thank Prof. Ulrich Neumann and the TAs of CSCI 580 for their help and support throughout the course.

References

[1] J. Tessendorf, *Simulating Ocean Water*
[2] John Van Drasek III, David Bookout, Adam Lake, *Real-Time Parametric Shallow Wave Simulation*
[3] Mark Finch, Cyan Worlds, *Effective Water Simulation from Physical Models*
[4] Alain Fournier, William T. Reeves, *A Simple Model of Ocean Waves*