České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Andrey Ufimtsev**

Studijní program: Softwarové technologie a management
Obor: Web a multimedia

Název tématu: **Modelování vodní hladiny**

Pokyny pro vypracování:

Prostudujte literaturu pojednávající o metodách generování vodní hladiny [1-5], zaměřte se zejména na metody založené na inverzní Fourierově transformaci. Na základě prostudované literatury navrhněte aplikaci, která bude schopna v reálném čase generovat časově proměnlivou vodní hladinu pomocí spektrální syntézy pro různé typy spekter.
Implementaci proveďte v C/C++ s využitím OpenGL tak, aby celá běžela na grafickém procesoru. Pro zobrazení vodní hladiny napište GLSL shadery, které budou simulovat odraz i lom světla. Generování vodní hladiny napište pomocí compute shaderů. Aplikaci doplňte o přehledné uživatelské rozhraní, které umožní měnit základní simulační parametry (např. typ spektra, velikost mřížky). Proveďte měření rychlosti implementované metody v závislosti na velikosti generované mřížky.

Seznam odborné literatury:

[1] E.Darles, B. Crespin, D. Ghazanfarpour, J.C. Gonzato: A Survey of Ocean Simulation and Rendering Techniques in Computer Graphics. Computer Graphics Forum, Vol.30, Num.1, p.43-60, Blackwell Publishing Ltd, 2011, ISSN 1467-8659.
[2] Li Tian: Ocean wave simulation by the mix of FFT and Perlin Noise. WSCG'2014 poster papers proceedings, pp. 1-4, 2014. ISBN 978-80-86943-72-5.
[3] Hao Pan, Yalin Zhang: CUDA-Based Real-Time Unbounded Ocean Rendering. International Conference on Virtual Reality and Visualization (ICVRV), pp.81–86, 2013.
[4] Jerry Tessendorf: Simulating Ocean Water. In SIGGRAPH course notes (course 47), 2001.
[5] Chen Si, Chunyong Ma, Ge Chen. Real-time Simulation of Large Bodies of Ocean Surface Based on Tessellation and Concurrent Kernels. Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering. Atlantis Press, 2013.

Vedoucí: Ing. Jaroslav Sloup

Platnost zadání: do konce letního semestru 2015/2016

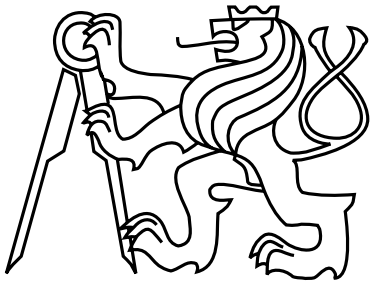
prof. Ing. Jiří Žára, CSc.
vedoucí katedry

L.S.

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 24. 3. 2015

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction



Bachelor's Project

Simulation of Ocean Surface

Andrey Ufimtsev

Supervisor: Ing. Jaroslav Sloup

Study programme: Software Technologies and Management

Branch: Web and Multimedia

May 20, 2015

Aknowledgements

I would like to thank Ing. Jaroslav Sloup, my supervisor, for patience and huge help during the whole process.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on May 22, 2015 .....

Abstract

This work addresses problems of simulating ocean surface using a spectral approach by J. Tessendorf. The surface itself is generated utilizing the Fast Fourier Transform. We will see how an FFT algorithm can be implemented on graphics processor using its power in terms of parallel computations. A produced patch of surface will be tiled to simulate a large area of ocean and illuminated using classical Fresnel reflection/refraction model. Finally, the results will be compared depending on different input parameters on several computers.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ocean simulation is an important part of the Computer Graphics field of study. Its techniques have been used in many areas like video games and movies, among which are *Titanic* (see figure 1.1) and *Waterworld*.

Figure 1.1: Picture from Titanic movie taken from [1].

Many different approaches for simulating realistic ocean surface have been proposed until now [11]. Some of them represent the ocean as a physically based motion of particals and some of them just define the surface of the ocean. However, not every approach is suitable for simulation in real time due to the high performance cost. In the chapter Previous Work we will go through the most well known techniques, compare them and choose one to use for an implementation.

The method that we will be focusing in our project has been proposed by J. Tessendorf and it is based on using a spectrum and FFT to evaluate the ocean surface. In the chapter Analysis and Design we will go through the theoretical part of the Fourier Transform and the Fast Fourier Transform needed for the later implementation. We will also go through the application design and specify the building blocks and the work flow of the application.

The Implementation chapter is dedicated to implementing the selected approach using

C++ and OpenGL compute shaders. We will go through the construction of the ocean surface geometry, implementing the shading and optimizing the critical parts of the application.

Finally, in the Results chapter we will analyze and compare the performance for different sizes of the input data and different types of implementation. We will also look at images and animations produced by the application.

Chapter 2

Previous Work

Methods for simulation of an ocean fall into two main subcategories: parametric/spectral methods and physically-based methods.

The parametric/spectral methods aim at representing water as a grid that defines the water surface. The individual points of the grid are then displaced according to the selected method. Parametric model describes a water surface with parametric equations. The spectral model uses waves spectrum and describes water surface based on the wave frequencies and amplitudes.

The physically-based methods utilize computational fluid dynamics, more specifically Navier-Stokes equations, and tend to describe a water surface as a motion of individual particles.

The parametric/spectral methods don't take into account interactions with other objects and more specifically with the bottom of the ocean, therefore they are good for simulating deep waters. To simulate water interaction with other objects or to simulate the surface of shallow water along a coastline only physically-based methods can be used.

We need to choose a method that we will be using to simulate a surface of an open ocean. Before we select one, let's go through the requirements for the method.

- The method has to produce relatively realistic results. The surface of the water should look as much close to the natural ocean surface in real world as possible. Ideally, there should not be any visible periodicity as well.
- The method has to be efficient enough for simulating large areas of ocean surface in real time. That means, that the application should be able to run at approximately 30-60 frames per second producing good looking results.
- The method doesn't have to handle interactions of the water with any other objects including the bottom of the ocean. Although this is not a requirement, but it is still important to mention, because enabling such functionality can cause noticeable computational overheard.

In the next chapters will be presented the most notable approaches. Although, this list is not full and doesn't include such methods like the one based on Euler equations and other. For the more detailed survey see [11].

2.1 Lagrangian approach

The Lagrangian approach, introduced by W. Reeves [16], describes fluid as a motion of independent particles that follow physical laws. The particles are connected to each other with springs that represent attraction or repulsion between them, this way simulating the motion of the liquid. The motion is illustrated in the figure 2.1.

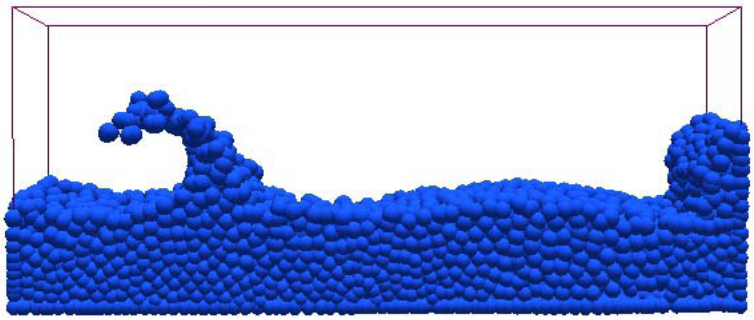


Figure 2.1: Lagrangian fluid dynamics taken from [3].

The Lagrangian approach is able to not only simulate an ocean surface, but any fluid in general. It produces very realistic results and can simulate water interaction with other objects. However, the computational cost is extremely high for large volumes of fluid due to a huge amount of particles needed to simulate it and in terms of ocean surface simulation the performance depends on the depth of the water.

The Lagrangian approach does not satisfy our requirements as an ocean simulation method and therefore will not be further considered.

2.2 Gerstner waves

Gerstner waves was one of the first methods used in computer graphics to describe an ocean surface. It found its first application in the work by Fournier and Reeves [12]. The method is based on a simplified version of the Navier-Stokes equations and describes the water surface as a circular motion of individual points on the surface. The wave profile of a wave generated with this approach is shown in the figure 2.2. Suppose that $\mathbf{x}_0$ is a point on an undisturbed surface and it is equal to (x_0, z_0) and $y_0 = 0$ is its height, A is the amplitude of a wave and t is the time, then the point displacement of the point can be calculated this way:

$$\mathbf{x} = \mathbf{x}_0 - \frac{\mathbf{k}}{k} A sin(\mathbf{k} \cdot \mathbf{x}_0 - \omega t) \tag{2.1}$$

$$y = A cos(\mathbf{k} \cdot \mathbf{x}_0 - \omega t) \tag{2.2}$$

where ω is a frequence and $\mathbf{k}$ is so called wavevector which points in the direction where the wave travels. The magnitude of the wavevector $\mathbf{k}$ is defined by equation:

$$k = \frac{2\pi}{\lambda} \tag{2.3}$$

where λ is the length of the wave.

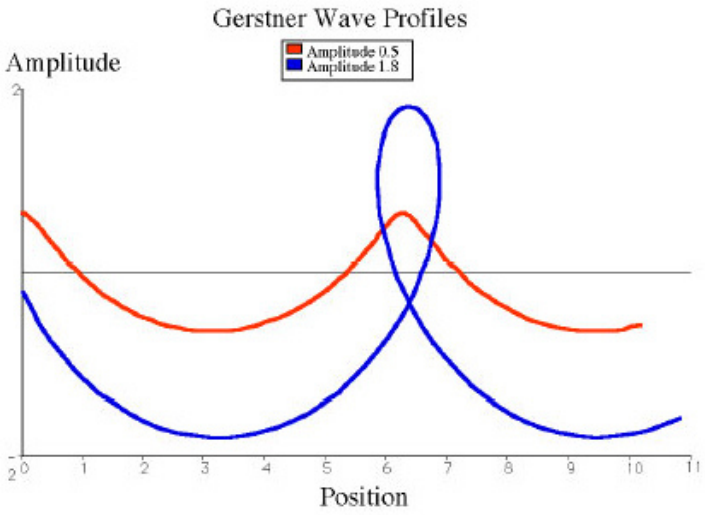


Figure 2.2: Gerstner wave profile taken from [8].

The original equation can be extended to produce more complex profiles by evaluating not just one wave, but a set of multiple waves with different frequencies and amplitudes.

$$\mathbf{x} = \mathbf{x}_0 - \sum_{i=1}^{N} \frac{\mathbf{k}_i}{k_i} A_i sin(\mathbf{k}_i \cdot \mathbf{x}_0 - \omega_i t + \phi_i) \tag{2.4}$$

$$y = \sum_{i=1}^{N} A_i cos(\mathbf{k}_i \cdot \mathbf{x}_0 - \omega_i t + \phi_i) \tag{2.5}$$

where ϕ_i is a phase.

Due to being based on parametric equations, the method is very effective in terms of simulating big areas of ocean surface and its computational cost does not depend on depth of water. Normally, Gerstner waves don't take into account interaction with other objects and the bottom of the ocean, which makes them a suitable approach for simulating a surface of a deep open ocean, but not a shallow water. Although there are extensions to this approach which address this issue [19].

2.3 Perlin noise

Perlin noise [15] is a mathematical algorithm for generating a procedural texture developed by Ken Perlin in 1983. Perlin noise is widely used in computer graphics for simulating realistic

effects of different types, such as smoke, fog, fire and etc. It is also used for procedural landscape generation.

Perlin noise algorithm is usually applied for generating 2D (see figure 2.3) or 3D textures, but it is not restricted to those dimensions. For example, it is possible to introduce the fourth dimension allowing the algorithm to dynamically change the texture based on the time.

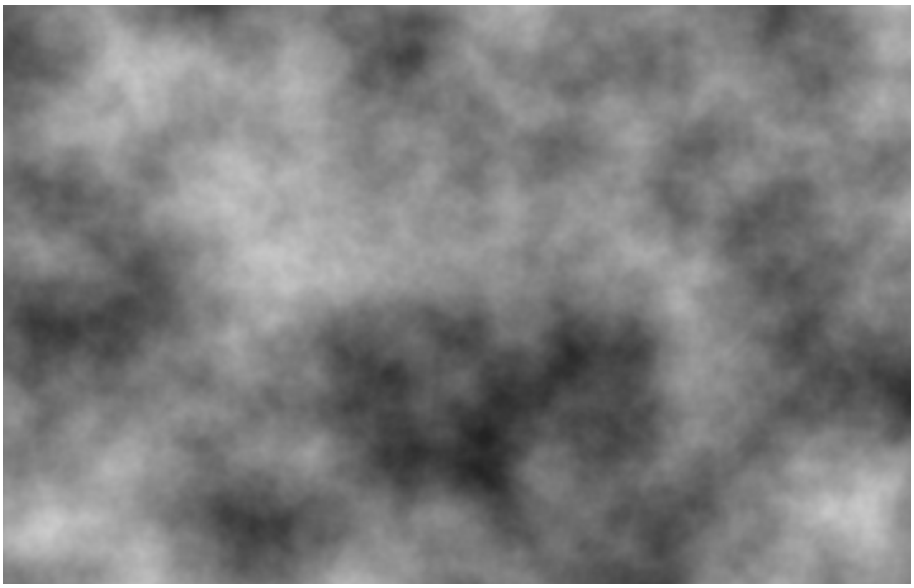


Figure 2.3: Texture generated using Perlin noise taken from [6].

The implementation of Perlin noise algorithm starts with a definition of an N-dimensional grid. Each point on the grid is a normalized N-dimensional randomly generated gradient vector. Suppose we need to implement a function noise2D(x, y), that samples the pregenerated two-dimensional grid. The first step would be to determine to which cell the point **p** defined by the vector (x, y) falls. The next step would be to find four gradient vectors that belong to the cell, and for each of them find a dot product between the gradient vector and a direction vector between the point **p** and the coordinates of the gradient vector. Finally, calculate the result using the bi-linear interpolation. The result of the function can then be used to calculate a height of the ocean surface at the point **p** by utilizing the following equation:

$$h_{perlin}(x, y) = \sum_{i=0}^{P-1} a^i \cdot noise2D(2^i x, 2^i y) \tag{2.6}$$

where P is the number of octaves used to adjust the level of details of the noise. Also, 2^i is a frequency and a^i is an amplitude of the octave i, where a is in range of $(0, 1)$. The noise2D is a function which samples the Perlin noise grid at a point (x, y). It's implementation uses the algorithm described above.

This approach is suitable for simulating an ocean surface and it is efficient enough to be applied in real time simulations as well. The example of rendering the ocean surface

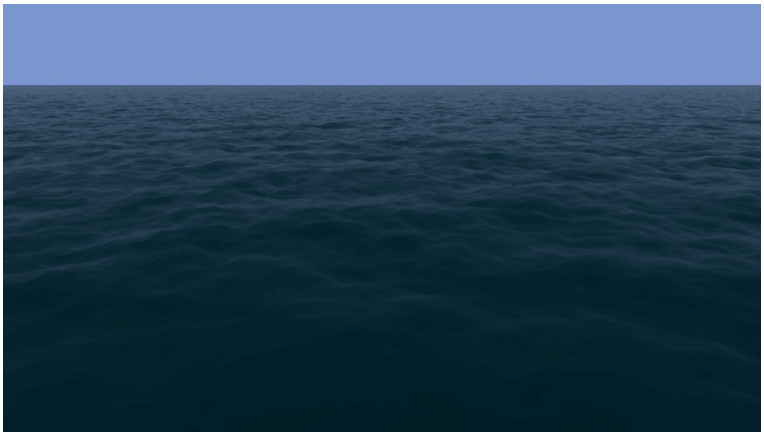


Figure 2.4: Ocean surface generated using Perlin noise approach taken from [4].

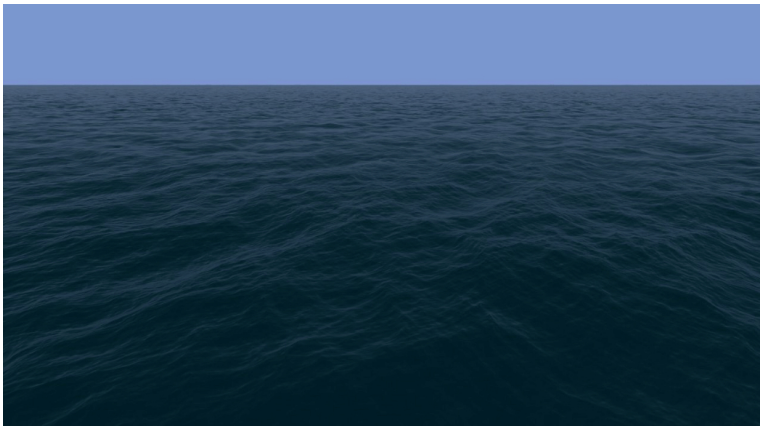


Figure 2.5: Ocean surface generated by blending Perlin noise with FFT approach taken from [4].

generated using Perlin noise is shown in the figure 2.4. Also, the Perlin noise algorithm is relatively easy to implement. Still, the approach doesn't allow simulating water interaction with other objects and the depth of the ocean is not taken into account, making it impossible to use this approach for simulating shallow waters.

Although the results produces by the Perlin noise approach are relatively realistic, the generated water surface lacks details, what becomes noticeable when observing the surface from a short distance. As the workaround the Perlin noise is sometimes mixed with some other approaches, such as the FFT approach [18]. The result of such blending is shown in the figure 2.5. The idea is to generate ocean surface using two approaches and then interpolate between two results, where the interpolation parameter depends on the distance between the position of the camera and the point on the surface that is being displaced.

2.4 FFT approach

This approach has been proposed by J. Tessendorf [17]. The approach is based on statistical model and has been widely used in Computer Graphics because of the high degree of realism.

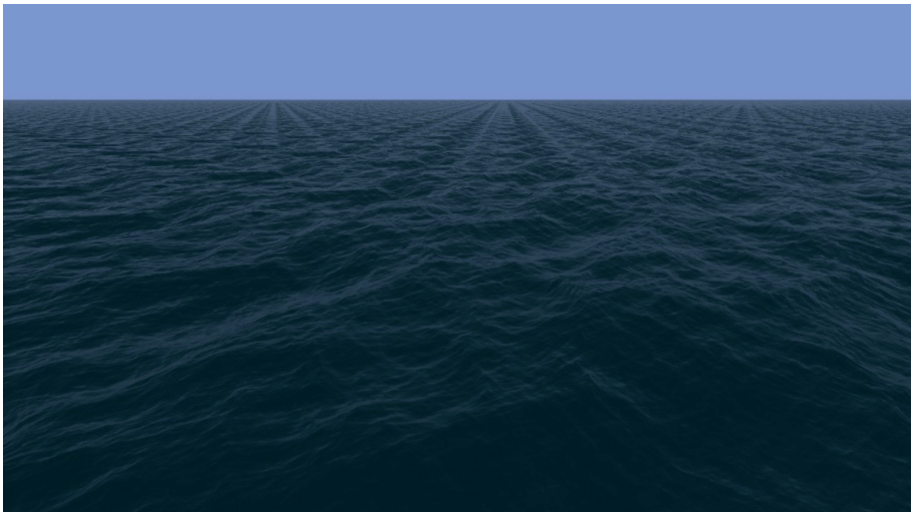


Figure 2.6: Ocean surface generated using FFT approach taken from [4].

The ocean surface is represented as a high-resolution grid. Each point of the grid is defined by a fixed horizontal position and variable vertical position. In order to create choppy looking waves the horizontal position of the points is also displaced, which will be described later in the chapter. The vertical position of a point on the surface is then defined by the following equation:

$$h(\mathbf{x}, t) = \sum_{\mathbf{k}} \tilde{h}(\mathbf{k}, t) e^{i\mathbf{k}\cdot\mathbf{x}} \tag{2.7}$$

where $\mathbf{x}$ is the horizontal position of the point, t is the time and $\mathbf{k}$ is two-dimensional vector called wavevector and defined by:

$$\mathbf{k} = \begin{bmatrix} \frac{2\pi n}{L_x} \\ \frac{2\pi m}{L_z} \end{bmatrix} \tag{2.8}$$

where n and m are integers with bounds $-\frac{N}{2} \leq n < \frac{N}{2}$ and $-\frac{M}{2} \leq m < \frac{M}{2}$. N and M is the resolution of the grid, L_x and L_z is the size of the water surface patch in meters.

The function $\tilde{h}(\mathbf{k}, t)$ takes a wavevector $\mathbf{k}$ and the time t and produces the wave height field. The equation for the height field at time equal to zero is as follows:

$$\tilde{h}_0(\mathbf{k}) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(\mathbf{k})} \tag{2.9}$$

where ξ_r and ξ_i are independent draws from a Gaussian random number generator with mean 0 and standard deviation 1. Using Gaussian distributed random numbers is not mandatory, although, according to the experimental data on ocean waves, more realistic results are achieved this way.

The function $P_h(\mathbf{k})$ returns the value of the Phillips spectrum for the wavevector $\mathbf{k}$. Phillips spectrum is a spectrum for wind-driven waves proposed by oceanographic research and it is defined by the following equation:

$$P_h(\mathbf{k}) = A\frac{e^{\frac{-1}{(kL)^2}}}{k^4}|\hat{\mathbf{k}} \cdot \hat{\omega}|^2 \tag{2.10}$$

where ω is the direction of the wind, A is a numeric constant for scaling amplitudes of the waves and L is the largest possible wave defined by the equation:

$$L = \frac{V^2}{g} \tag{2.11}$$

where V is a continuous wind speed and g is the gravitational constant.

Now in order to produce the animation, we need to use the time t to generate the Fourier amplitudes based on the current time. This is done by utilizing the following equation:

$$\tilde{h}(\mathbf{k}, t) = \tilde{h}_0(\mathbf{k})e^{i\omega(k)t} + \tilde{h}_0^*(-\mathbf{k})e^{-i\omega(k)t} \tag{2.12}$$

where $\omega(k)$ is the dispersion relation defined by the equation:

$$\omega^2(k) = gk \tag{2.13}$$

where k is the magnitude of the wavevector $\mathbf{k}$.

For the rendering purposes we also need to calculate the normal vectors for each point on the grid. This is done by performing another pass of the Fourier Transform, where we first calculate the slope vector and then subtract it from the vector $(0, 1, 0)$:

$$\varepsilon(\mathbf{x}, t) = \sum_{\mathbf{k}} i\mathbf{k}\tilde{h}(\mathbf{k}, t)e^{i\mathbf{k}\cdot\mathbf{x}} \tag{2.14}$$

$$normal(\mathbf{x}, t) = \begin{bmatrix}0\\1\\0\end{bmatrix} - \begin{bmatrix}\varepsilon(\mathbf{x}, t)_x\\0\\\varepsilon(\mathbf{x}, t)_z\end{bmatrix} \tag{2.15}$$

The steps described above are required to generate a patch of resolution $N \times M$ and size $L_x \times L_z$ of animated ocean surface. It is then relatively easy to use the produced patch for tiling, although there are a few important implementation details to mention, which will be reviewed in the Implementation chapter.

The approach described so far produces smooth looking waves with rounded peaks, what gives the ocean surface an appearance of being in good weather conditions. But even in such weather and especially during a strong wind or storm, the waves look sharp at the top and flat at the bottom. To produce such waves it is required to horizontally displace each point

on the grid given by the original position on undisturbed surface $\mathbf{x}$ at each moment of time t, which brings us to evaluating even more FFTs:

$$D(\mathbf{x}, t) = \sum_{\mathbf{k}} -i\frac{\mathbf{k}}{k}\tilde{h}(\mathbf{k}, t)e^{i\mathbf{k}\cdot\mathbf{x}} \tag{2.16}$$

Although the water surface produced by this approach looks realistic, there is still a periodic artifact due to the fact, that the approach uses tiling to copy one patch over large areas of the surface (see figure 2.6). As it was mentioned in the previous section, such artifact can be prevented by mixing the FFT approach with another method like the one based on Perlin noise. Also, as we can see, this approach doesn't take into account the interaction with other objects and it is not dependent on the depth of the ocean. It can be used in real time as well.

2.5 Comparison

In the previous section we have described the most popular approaches for simulating an ocean surface. We have seen that almost each one of them is suitable for us, but they all have some pros and cons, which we are going to review.

The Gerstner waves approach is very easy to implement and it is very efficient for simulating deep ocean. Although, the surface motion produced by this approach is not absolutely realistic.

The more realistic result is achieved using either Perlin noise or FFT. While Perlin noise approach is relatively easy to implement and it is not high performance demanding, it lacks a high level of details. The FFT approach produces more detailed surface. On the other hand, the ocean surface produced by FFT approach has a periodicity effect, while the Perlin noise doesn't have it.

For now we are more concerned about quality of the surface, therefore we will focus on the FFT approach proposed by J. Tessendorf [17]. Later this approach can be extended to blend the surface with Perlin noise.

Chapter 3

Analysis and Design

In this chapter we will go through all theoretical aspects related to the project and after this we will make the design of the application, where we will specify what its building blocks will be and how they will be connected in order to work together.

At the beginning of this chapter we will focus on the Fourier Transform. Since this is the essential part of the algorithm we are using, it needs a closer analysis. We will see how it works and how it could be applied in the project in order to generate an ocean surface. Then we will review and analyze faster algorithms to compute Fourier Transform, called Fast Fourier Transform algorithms (FFT). We will be mainly focusing on the well known Cooley-Tukey FFT algorithm. Also, we will see how this algorithm could be used with two-dimensional data.

In the section of Application design we will focus on application related topic. We will go through the structure of the application, define its building blocks and design its work flow.

3.1 Fourier Transform

The Fourier Transform [2] takes a signal as a function of time $f(t)$ and decomposes it into a set of sinusoids described by $F(\omega)$ that represent the original signal in a frequency domain. Each sinusoid is defined as a complex number, where the real part is an amplitude and the imaginary component is a phase shift. The process of decomposition is illustrated in the figure 3.1. The equation for Fourier Transform is as follows:

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt \tag{3.1}$$

Original version of the Fourier Transform works with analog signals and is not suitable for calculations done on a digital computer. Therefore, for our project we will be using another version of Fourier Transform called Discrete Fourier Transform (DFT). The equation for DFT is as follows:

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{\frac{-i2\pi kn}{N}} \tag{3.2}$$

where N is a number of samples, $x(n)$ is a sample at index n, $X(k)$ is a complex number defining an amplitude and a phase of frequency k.

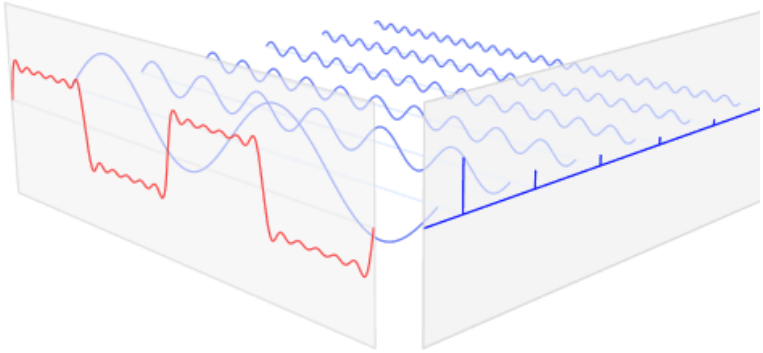


Figure 3.1: Decomposition of the signal into its frequency components taken from [9].

The result of performing the Discrete Fourier Transform is a frequency spectrum, the building blocks that would be used to construct a signal. This conversion is lossless and the produced spectrum can even be used to restore the original signal. The backward conversion of a discrete signal from a frequency domain to a time domain is called Inverse Discrete Fourier Transform and is defined by the following equation:

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{\frac{i2\pi kn}{N}} \tag{3.3}$$

It is also worth mentioning that the Fourier Transform is not only suitable for 1D signals, but for 2D data as well. For instance, a 2D version of Discrete Cosine Transform (DCT) is used in the JPEG compression algorithm.

If we think of an ocean surface as of a two-dimensional function of time, we may assume that there exists such frequency spectrum that will produce the ocean surface after applying the Inverse DFT on it. Oceanographic research has shown that Phillips spectrum yields very realistic results, although other types of spectra could be used instead. Phillips spectrum will be described in more details later.

It is evident that the naive algorithm for performing DFT has asymptotic complexity of $O(n^2)$. It will become computationally expensive for large data and therefore it is inefficient for simulating a highly detailed ocean surface in real time. Fortunately, more efficient algorithms exist. Any algorithm that is able to perform DFT in $O(n * log(n))$ is called Fast Fourier Transform (FFT). One of such algorithms will be discussed later in more details.

3.1.1 FFT

Any algorithm that can compute DFT in $O(n * log(n))$ is called Fast Fourier Transform (FFT). There are different FFT algorithms, although the one most common is Cooley-Tukey FFT algorithm [10]. It will be described here in more details.

FFT algorithms are not an approximation to the general DFT. In fact, due to the numerical instability and the much smaller amount of multiplications required, the FFT algorithms produce results that are even more accurate than the ones produces by the naive implementation.

The Cooley-Tukey algorithm was introduced in 1965 by J. W. Cooley and J. W. Tukey. It's a multi-stage algorithm that recursively breaks a larger DFT into a set of smaller DFTs. The number of smaller DFTs produced on each step depends on the variant of the algorithm. For this project we will be using the Radix-2 algorithm, which goes through $log(N)$ stages and during each stage it breaks the DFT of size M into two smaller DFTs of size M/2. Radix-4 and Radix-8 algorithms are essentially the same as Radix-2, but perform more stages at ones.

The best way to represent the FFT algorithm graphically is through so called butterfly diagram (see figure 3.2). For each stage the diagram shows how the elements are selected and how the FFT is performed on them.

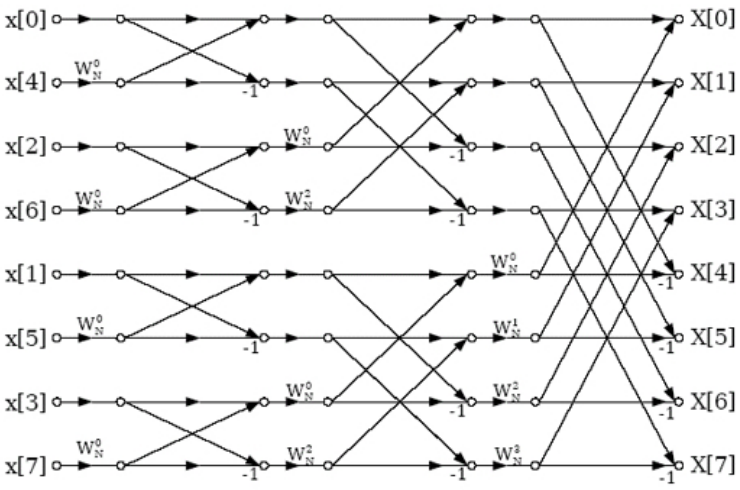


Figure 3.2: Butterfly algorithm taken from [7].

The general equation of DFT presented in the previous section can be rewritten in the following simplified form:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \tag{3.4}$$

where W_N is the twiddle factor, which is defined as follows:

$$W_N = e^{\frac{-2\pi i}{N}} \tag{3.5}$$

The following equation can now be separated into sums for odd and even elements:

$$X(k) = \sum_{n\ even} x(n) W_N^{kn} + \sum_{n\ odd} x(n) W_N^{kn} \tag{3.6}$$

$$X(k) = \sum_{r=0}^{\frac{N}{2}-1} x(2r)W_N^{k(2r)} + W_N^k \sum_{r=0}^{\frac{N}{2}-1} x(2r+1)W_N^{k(2r)} \tag{3.7}$$

$$X(k) = \sum_{r=0}^{\frac{N}{2}-1} x(2r)W_{\frac{N}{2}}^{kr} + W_{\frac{N}{2}}^k \sum_{r=0}^{\frac{N}{2}-1} x(2r+1)W_{\frac{N}{2}}^{kr} \tag{3.8}$$

$$X(k) = FFT_{even} + W_{\frac{N}{2}}^k * FFT_{odd} \tag{3.9}$$

where we have used the symetry property of twiddle factors:

$$W_N^{k(2r)} = W_{\frac{N}{2}}^{kr} \tag{3.10}$$

As any other FFT algorithm, the Cooley-Tukey algorithm has the limitation of N having to be a power of two.

In our project the ocean surface must be represented as a two-dimensional grid and therefore the two-dimensional version of FFT should be used instead. The two-dimensional FFT doesn't differ very much from its one-dimensional version and it is actually composed of a number of one-dimensional FFTs. In order to perform the FFT on two-dimensional array of data it is required to perform the FFT on each row and then on each column of the input array.

3.1.2 Fresnel factor

When an electromagnetic wave hits a surface, the energy of the incident wave is equal to the sum of the energy of the reflected wave and the energy of the refracted wave. The Fresnel factor describes this proportion.

For the polarized light, the component parallel to the plain containing the surface normal and the opposite light direction is called *p*-polarized. The other component perpendicular to the plain is called *s*-polarized. The fresnel factors for each polarized component are given by the equations:

$$f_p = \frac{tan^2(\Theta_1 - \Theta_2)}{tan^2(\Theta_1 + \Theta_2)} \tag{3.11}$$

and

$$f_s = \frac{sin^2(\Theta_1 - \Theta_2)}{sin^2(\Theta_1 + \Theta_2)} \tag{3.12}$$

where Θ_1 is the angle of incidence and Θ_2 is the angle of transmittance.

When dealing with unpolarized light we calculate a fresnel factor f by averaging two equations above:

$$f = \frac{1}{2}(\frac{tan^2(\Theta_1 - \Theta_2)}{tan^2(\Theta_1 + \Theta_2)} + \frac{sin^2(\Theta_1 - \Theta_2)}{sin^2(\Theta_1 + \Theta_2)}) \tag{3.13}$$

The relationship between the angle of incidence and the angle of refraction is described with the following equation:

$$\frac{sin\Theta_1}{sin\Theta_2} = \frac{n_2}{n_1} \tag{3.14}$$

where n_1 and n_2 are indices of refraction of each of two media.

3.2 Spectrums

The Phillips spectrum described in Previous Work chapter is a model for wind-driven weaves in a fully developed sea. Although using the Phillips spectrum is not mandatory and other spectrums can be used.

A few other wave spectrums exist, for instance the JONSWAP Spectrum [5]. This spectrum was introduced based on the data collected during the Joint Sea Wave Observation Project JONSWAP. It is a modified version of Pierson-Moskowitz spectrum [13] where the continuous development of wave spectrum is not taken into account.

The JONSWAP spectrum is defined by equation:

$$S_j(\omega) = \frac{\alpha g^2}{\omega^5} e^{-\frac{5}{4}(\frac{\omega_p}{\omega})^4} \gamma^r \tag{3.15}$$

$$r = e^{-\frac{(\omega-\omega_p)^2}{2\sigma^2\omega_p^2}} \tag{3.16}$$

where $\omega = 2\pi f$, f is frequency of the wave, $\alpha = 0.076(\frac{U_{10}^2}{Fg})^{0.22}$, $\omega_p = 22(\frac{g^2}{U_{10}F})^{\frac{1}{3}}$, $\gamma = 3.3$, $\sigma = 0.07$ for $\omega \leq \omega_p$, $\sigma = 0.09$ for $\omega > \omega_p$ and F is the distance over which the wind blows with constant speed, which is called fetch.

3.3 Application design

The application will be written for PC using C++ programming language and OpenGL. Since we will be using compute shaders, the minimum version of OpenGL is 4.3 (or 4.2 for non-Core profile).

Two main blocks of the application are rendering engine and surface generator. The rendering will be done using OpenGL shaders written is a GLSL language. The surface generator will be represented by a separate class, which will be doing all necessary calculations.

Surface generation is subdivided into two steps:

- Initialization. During this step we initialize all time independent data and precalculate necessary values.
- Update. The purpose of this step is to have all data needed for rendering to be prepared for it.

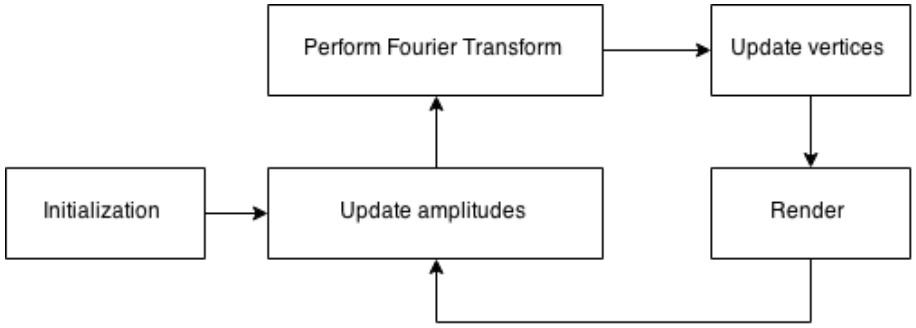


Figure 3.3: Work flow of the application.

The initialization step will be performed only once at the start of the application. The updating step is performed each frame.

During the updating step we need to go through a number of sub-tasks in order to generate an ocean surface:

- At the beginning we need to update the amplitudes.
- Then we need to perform Fourier Transform on the prepared array of amplitudes.
- Finally we need to update the vertices and normals based on the results of the Fourier Transform.

The work flow diagram is shown in the figure 3.3.

The application will also be provided with a user interface allowing the user to control necessary parameters on the fly.

Chapter 4

Implementation

In this chapter we will begin the implementation. We will start off by generating the ocean surface geometry using the naive implementation of the Fourier Transform. As Donald Knuth said, "Premature optimization is the root of all evil", we will not focus on the optimization right now, but we will try to create at least a working application, then measure the time of different blocks of execution and based on the results we will see which parts of the code require either refactoring or optimization.

After we have a working application capable of generating a mesh for the ocean surface, we will dive into the problem of shading. We will use the illumination model that is based on the Fresnel equations. Since there are no other objects in the scene, the only object reflected on the surface will be the skybox, making it very easy to implement. The rendering of the skybox will also be described in that section.

So far we will have generated one patch of the ocean surface. In the Tiling problem section we will see how the tiling of the surface could be implemented and what problems will arise during it.

Finally, in the last section we will start optimizing the code. The first step will be to rewrite the Fourier Transform computations using the FFT algorithm described in the previous chapter. And at the end of our implementation we will start utilizing the parallel computation capabilities of the GPU and implement the most time demanding parts using the compute shaders.

4.1 Surface geometry

The implementation of the ocean surface generator basically splits into two parts:

- Initialization. This is the static part of surface generation block. It doesn't depend on the current time, which means that we can precalculate the values at the beginning and then use them inside the animation later.
- Animation. This is the dynamic part of the surface generation block. It depends on the current time, therefore we have to perform this part on each frame. The values related to this part either cannot be precalculated or would cause a significant memory usage.

We will start off with the initialization first and switch to the animation later in this section.

Let's take a closer look at the equation for the Fourier Transform and analyze what data we need in order to utilize it.

$$h(\mathbf{x}, t) = \sum_{\mathbf{k}} \tilde{h}(\mathbf{k}, t) e^{i\mathbf{k}\cdot\mathbf{x}} \tag{4.1}$$

It is clearly visible that the wavevectors and the original positions of the points on the grid are static and do not require an update per each frame. Therefore, let's start by allocating the buffers for them and filling them with the required data.

As it was described in the previous chapters, the wavevector $\mathbf{k}$ is found using the equation $\mathbf{k} = \begin{bmatrix} \frac{2\pi n}{L_x} \\ \frac{2\pi m}{L_z} \end{bmatrix}$ and the vertex positions $\mathbf{x}$ on undisturbed surface are just interpolated values across the whole surface, where the range of the values in both directions is from $-\frac{L_x}{2}$ to $\frac{L_x}{2}$ and from $-\frac{L_z}{2}$ to $\frac{L_z}{2}$ respectively. For the sake of simplicity we will assume that each patch will be a square, meaning that $N = M$ and $L_x = L_z$.

Let's also look at the equation for calculating the fourier amplitudes at the time t.

$$\tilde{h}(\mathbf{k}, t) = \tilde{h}_0(\mathbf{k}) e^{i\omega(k)t} + \tilde{h}_0^*(-\mathbf{k}) e^{-i\omega(k)t} \tag{4.2}$$

As we can see, it is a sum of two expressions, where the first expression is $\tilde{h}_0(\mathbf{k}) e^{i\omega(k)t}$ and the second expression is $\tilde{h}_0^*(-\mathbf{k}) e^{-i\omega(k)t}$. There is a static part in each expression that can also be precalculated at the initialization stage of the application. Let's get to the code, where we will allocate the memory for the arrays required to hold all initial values and then fill them with the necessary data.

```
1  origins_ = new vec2[resolution * resolution];
2  wavevectors_ = new vec2[resolution * resolution];
3  baseAmplitudes_ = new BCComplex[resolution * resolution];
4  baseAmplitudeConjugates_ = new BCComplex[resolution * resolution];
5
6  for (int m = 0; m < resolution; m++) {
7      for (int n = 0; n < resolution; n++) {
8          int index = m * resolution + n;
9
10         origins_[index].x = (n - resolution / 2.0f) * (size / resolution);
11         origins_[index].y = (m - resolution / 2.0f) * (size / resolution);
12
13         wavevectors_[index].x = (2 * M_PI * (n - resolution / 2.0f)) / size;
14         wavevectors_[index].y = (2 * M_PI * (m - resolution / 2.0f)) / size;
15
16         baseAmplitudes_[index] = getBaseHeightAmplitude(wavevectors_[index]);
17         baseAmplitudeConjugates_[index] = getBaseHeightAmplitude(-wavevectors_
               [index]).conjugate();
18     }
19 }
```

Listing 4.1: Initialization of the static data.

On the lines 11 and 12 of the listing above we have used the method *getBaseHeightAmplitude*() that we haven't yet introduced. This method implements the following equation:

$$\tilde{h}_0(\mathbf{k}) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(\mathbf{k})} \tag{4.3}$$

This equation is used to calculate the time independent part of the Fourier amplitudes and can be implemented the following way:

```
1 BCComplex BCOcean::getBaseHeightAmplitude(const vec2 &wavevector) const {
2     return getRandom() * (1.0f / sqrtf(2.0f)) * sqrtf(getPhillips(wavevector))
          ;
3 }
```

Listing 4.2: Method for calculating the time independent part of the fourier amplitudes.

Computation of the time independent parts of the Fourier amplitudes requires a complex number generated with the Gaussian random number generator with mean 0 and standard deviation 1 and the value of the Phillips spectrum depending on the wavevector $\mathbf{k}$.

To generate a complex random number we will use the C++ standard library functionality.

```
1 BCComplex BCOcean::getRandom() const {
2     static default_random_engine engine;
3     static normal_distribution<float> distribution(0, 1);
4     return BCComplex(distribution(engine), distribution(engine));
5 }
```

Listing 4.3: Method for generating gaussian random complex number.

We will continue by implementing the equation for the Phillips spectrum. Let's take a look at the equation one more time.

$$P_h(\mathbf{k}) = A\frac{e^{\frac{-1}{(kL)^2}}}{k^4}|\hat{\mathbf{k}} \cdot \hat{\omega}|^2 \tag{4.4}$$

According to the description of this equation, A is a numerical constant that is used for scaling the amplitudes of the waves. Basically it can be any value and we need to find the most appropriate one, so we will set it to *0.00025*.

Let's get to the code and see, how the method for the Phillips spectrum can be implemented. At this point of time we suppose that *windSpeed_*, *waveAmplitude_* and *windDirection_* are values that are created during the ocean initialization.

```
1  float BCOcean::getPhillips(const vec2 &wavevector) const {
2      float wavevectorLength = length(wavevector);
3      if (wavevectorLength < 0.000001) return 0;
4      float k_2 = wavevectorLength * wavevectorLength;
5      float k_4 = k_2 * k_2;
6      float L = (windSpeed_ * windSpeed_) / GRAVITATIONAL_CONSTANT;
7      float L_2 = L * L;
8      float cosFactor = dot(normalize(wavevector), normalize(windDirection_));
9      float cosFactor_2 = cosFactor * cosFactor;
10
```

```
11      float spectrumValue = waveAmplitude_ * (exp(-1.0f / (k_2 * L_2)) / k_4) *
            cosFactor_2;
12
13      float l = L / 1000.0f;
14      float l_2 = l * l;
15      return spectrumValue * exp(-k_2 * l_2);
16  }
```

Listing 4.4: Method for calculating the value of the Phillips spectrum.

This concludes the initialization of the static part of the ocean implementation. Although, this is still not enough to see any result. This takes us to the next step, which is implementing the animation. The animation is done inside the method *update*() that takes the current time as its only parameter. Before we get to this method, we need to create and allocate all arrays and OpenGL buffers needed for the animated parts of the ocean. We will create three different arrays:

- *amplitudes* - this array will hold the fourier amplitudes, which we will be using to evaluate the Fourier Transform.
- *positions* - in this array we will be storing the generated positions of the vertices.
- *normals* - as the name suggests, this array will be used to store the surface normals at each vertex, which then will be interpolated for each output fragment.

The code for initialization of these arrays is very straightforward:

```
1 amplitudes_ = new BCComplex[resolution * resolution];
2 positions_ = new vec3[verticesCount_];
3 normals_ = new vec3[verticesCount_];
```

Listing 4.5: Initialization of the arrays to hold the dynamic data.

It is important to mention, that the number of the vertices, represented by the variable *verticesCount_* in the listing above, will not actually be equal to the resolution squared at the final version of the application. The reason for this will be explained in the Tiling problem section. Although for now we are not worried about the tiling and we can assume that the number of vertices per each side is equal to the resolution along each side of the surface and the total number of vertices is equal to the resolution squared.

In each frame we will be passing the positions of the vertices and their normals to the GPU. In order to do this OpenGL uses Vertex Buffer Objects (VBO). OpenGL also needs the information how the data inside the buffers is mapped onto the input parameters in the shaders. This information is represented using Vertex Array Objects (VAO). We will create a separate VBO for each type of the data and one common VAO.

```
1 glGenVertexArrays(1, &vao_);
2 glBindVertexArray(vao_);
3
4 glGenBuffers(1, &positionsVBO_);
5 glBindBuffer(GL_ARRAY_BUFFER, positionsVBO_);
6 glBufferData(GL_ARRAY_BUFFER, verticesCount_ * sizeof(vec3), NULL,
      GL_DYNAMIC_DRAW);
```

```
7
8  glEnableVertexAttribArray(shaderProgram->attributes.vertexPosition);
9  glVertexAttribPointer(shaderProgram->attributes.vertexPosition, 3, GL_FLOAT,
       GL_FALSE, 0, 0);
10
11 glGenBuffers(1, &normalsVBO_);
12 glBindBuffer(GL_ARRAY_BUFFER, normalsVBO_);
13 glBufferData(GL_ARRAY_BUFFER, verticesCount_ * sizeof(vec3), NULL,
       GL_DYNAMIC_DRAW);
14
15 glEnableVertexAttribArray(shaderProgram->attributes.vertexNormal);
16 glVertexAttribPointer(shaderProgram->attributes.vertexNormal, 3, GL_FLOAT,
       GL_FALSE, 0, 0);
```

Listing 4.6: Initialization of the arrays to hold the dynamic data.

In this listing we have used an object called *shaderProgram*. We suppose, that this object is responsible for holding the information about the shader program and the locations for all shader attributes.

Since we are going to be modifying the buffers many times, it is recommended to specify the buffer usage as *GL_DYNAMIC_DRAW*.

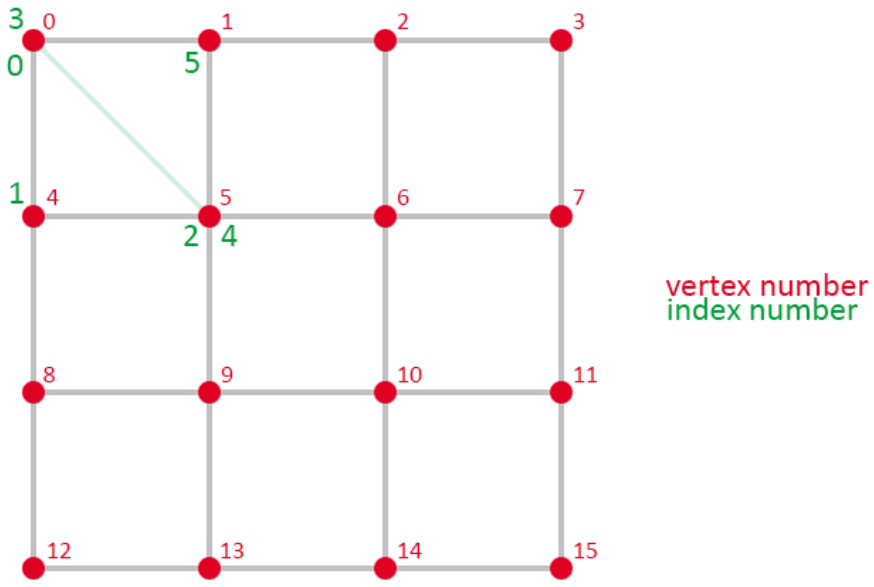


Figure 4.1: Vertex indexing.

Finally, we need to tell the GPU the order in which the vertices should be read. For this OpenGL has a special type of buffer, called Elements Buffer Object (EBO). The only rule for the order of the vertices is that the vertices inside each triangle should go in a counterclockwise order if looking from the direction the triangle is facing. This is very important if the face culling is enabled. The figure 4.1 shows how the vertex indexing could be done in our case.

Let's get to the code, where we will allocate an array to hold the vertex indices, calculate them and copy them to the GPU memory using a buffer.

```
verticesCount_ = verticesCountPerSide_ * verticesCountPerSide_;
unsigned int indicesOffset = 0;
unsigned int *indices = new unsigned int[indicesCount_];
for (int m = 0; m < resolution; m++) {
    for (int n = 0; n < resolution; n++) {
        int index = m * verticesCountPerSide_ + n;

        indices[indicesOffset++] = index;
        indices[indicesOffset++] = index + verticesCountPerSide_;
        indices[indicesOffset++] = index + verticesCountPerSide_ + 1;

        indices[indicesOffset++] = index;
        indices[indicesOffset++] = index + verticesCountPerSide_ + 1;
        indices[indicesOffset++] = index + 1;
    }
}

glGenBuffers(1, &ebo_);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo_);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indicesCount_ * sizeof(unsigned int),
    indices, GL_STATIC_DRAW);
```

Listing 4.7: Vertex indexing.

This concludes the whole initialization process. Now, when all arrays are allocated and the buffers are prepared, we will start implementing the *update()* method. The only parameter the method takes is the current time in seconds. The method itself updates the amplitudes based on the time, calculates the positions and normals of the vertices and updates the OpenGL buffers.

Let's first see how the Fourier amplitudes could be updated. We do this with a cycle over all points on the grid, where for each point the amplitude is calculated using the data prepared at the initialization step.

```
for (int m = 0; m < resolution_; m++) {
    for (int n = 0; n < resolution_; n++) {
        int index = m * resolution_ + n;
        amplitudes_[index] = getHeightAmplitude(wavevectors_[index],
            baseAmplitudes_[index], baseAmplitudeConjugates_[index], time);
    }
}
```

Listing 4.8: Updating the amplitudes.

The method *getHeightAmplitude()* is the one that does the actual calculation. It's implementation is based on the equation 4.2.

```
BCComplex BCOcean::getHeightAmplitude(const vec2 &wavevector, const BCComplex
    &baseAmplitude, const BCComplex &baseAmplitudeConjugate, float time) const
    {
    float dispersion = getDispersion(wavevector);
    float cosdt = cos(dispersion * time);
    float sindt = sin(dispersion * time);
```

```
5     BCComplex c1 = BCComplex(cosdt, sindt);
6     BCComplex c2 = BCComplex(cosdt, -sindt);
7     return baseAmplitude * c1 + baseAmplitudeConjugate * c2;
8 }
```

Listing 4.9: Method for updating the amplitudes.

On the second line of the listing above we use a method called *getDispersion()*, which takes a wavevector and evaluates the dispersion relation defined by the equation 2.13. The implementation of the method is straightforward.

```
1 float BCOcean::getDispersion(const vec2 &wavevector) const {
2     return sqrt(length(wavevector) * GRAVITATIONAL_CONSTANT);
3 }
```

Listing 4.10: Method evaluating the dispersion relation.

The *GRAVITATIONAL_CONSTANT* is set to be *9.8*.

Let's return to the *update()* method. When the Fourier amplitudes have been updated, the next step is to use them to compute the vertex positions and their normals.

```
1 for (int m = 0; m < resolution_; m++) {
2     for (int n = 0; n < resolution_; n++) {
3         int srcVertexIndex = m * resolution_ + n;
4         int destVertexIndex = m * verticesCountPerSide_ + n;
5 
6         vec3 normal;
7         vec3 displacement;
8         evaluateVertex(origins_[srcVertexIndex], &displacement, &normal);
9 
10        positions_[destVertexIndex] = vec3(origins_[srcVertexIndex].x, 0.0f,
              origins_[srcVertexIndex].y) + displacement;
11        normals_[destVertexIndex] = normal;
12    }
13 }
```

Listing 4.11: Method evaluating the dispersion relation.

On the line 8 we use a not yet introduces method, called *evaluateVertex()*. This method is the one that implements the equation 4.1 by performing the Fourier Transform. As it was said earlier, at this point of time we will not be implementing the Fast Fourier Transform algorithm leaving it for the future optimization. Although we will implement the naive algorithm for evaluating the Fourier Transform. This naive implementation is essentially just performing a sum of the amplitudes inside a cycle. In this case we can clearly use the double cycle to go through all amplitudes, which will not be the case when we will be implementing the Fast Fourier Transform. As said before, the algorithm itself will be implemented inside the *evaluateVertex()* method, where the input parameter is the original position of the vertex on the undisturbed surface and the output parameters are the two vectors. The *outputDisplacement* is a 3D vector that represents the displacement of the vertex relatively to its original position and *outNormal* is the normal vector of the vertex. Given that, we can implement the method the following way:

```
1 void BCOcean::evaluateVertex(const vec2 &origin, vec3 *outDisplacement, vec3 *
      outNormal) const {
```

```
2     BCComplex height;
3     vec3 slope;
4     vec2 displacement;
5
6     for (int m = 0; m < resolution_; m++) {
7         for (int n = 0; n < resolution_; n++) {
8             int index = m * resolution_ + n;
9
10            vec2 wavevector = wavevectors_[index];
11            float wavevectorLength = length(wavevector);
12            vec2 wavevectorNormalized = wavevector / wavevectorLength;
13
14            float dotProduct = dot(origin, wavevector);
15            BCComplex expDotProduct(cos(dotProduct), sin(dotProduct));
16            BCComplex hTildeExp = amplitudes_[index] * expDotProduct;
17
18            height = height + hTildeExp;
19
20            slope = slope + vec3(-wavevector.x * hTildeExp.im, 0, -wavevector.
                  y * hTildeExp.im);
21
22            if (wavevectorLength > 0.000001) {
23                displacement = displacement + vec2(wavevectorNormalized.x *
                      hTildeExp.im, wavevectorNormalized.y * hTildeExp.im);
24            }
25        }
26    }
27
28    *outNormal = normalize(vec3(0, 1, 0) - slope);
29    *outDisplacement = vec3(displacement.x * CHOPPINESS_FACTOR, height.re,
          displacement.y * CHOPPINESS_FACTOR);
30 }
```

Listing 4.12: Method for computing the vertex data.

In the listing above we first calculate the height value for each wavevector and then according to the equations 2.14 and 2.16 use this value in the calculation of the horizontal displacement and the slope vector. In order to produce the normal vector we need to subtract the sum of the slope vectors from the vector $(0, 1, 0)$. The horizontal displacement is scaled according to the constant *CHOPPINESS_FACTOR*, that is set to be *-1.0*.

The whole code presented above is wrapped into a class *BCOcean*. Inside the constructor we must specify the resolution of the surface grid, the size of the grid in meters, the wave amplitude scaling factor and a 2D vector that defines the wind direction and its speed.

This concludes the generation of the ocean surface using the naive algorithm for evaluating the Fourier Transforms implemented on CPU. For now it should be enough for the testing purposes, although we still need to implement the rendering. With that being said, let's move to the next section.

4.2 Shading

There are many different shading models and choosing one or another depends on such factors as performance requirement, scene complexity and the desired look of the result. For

this implmenetation we will choose the classical fresnel reflection and refraction model to render the surface. Most of the time implementing this model would be resource demanding, because we need to take into account reflection and refraction which are relatively difficult to implement using the classical resterization. But since our scene consists only of the ocean surface, we can easily sample the cube map of the skybox to get the reflection color and use the color of the ocean as the refraction color.

4.2.1 Skybox

We will be using the skybox to simulate the distant environment like the sky, the clouds and the sun. The skybox itself is just a cube with 6 different textures on its inner sides. For the convenience, each texture will be named after the side of the cube this texture belongs to. For example, the texture for the positive X side of the cube can be named *posx*. The idea of naming the textures is shown on the figure 4.2. It is also important to notice that the images have to be flipped vertically in order to use them in OpenGL as a cube map.

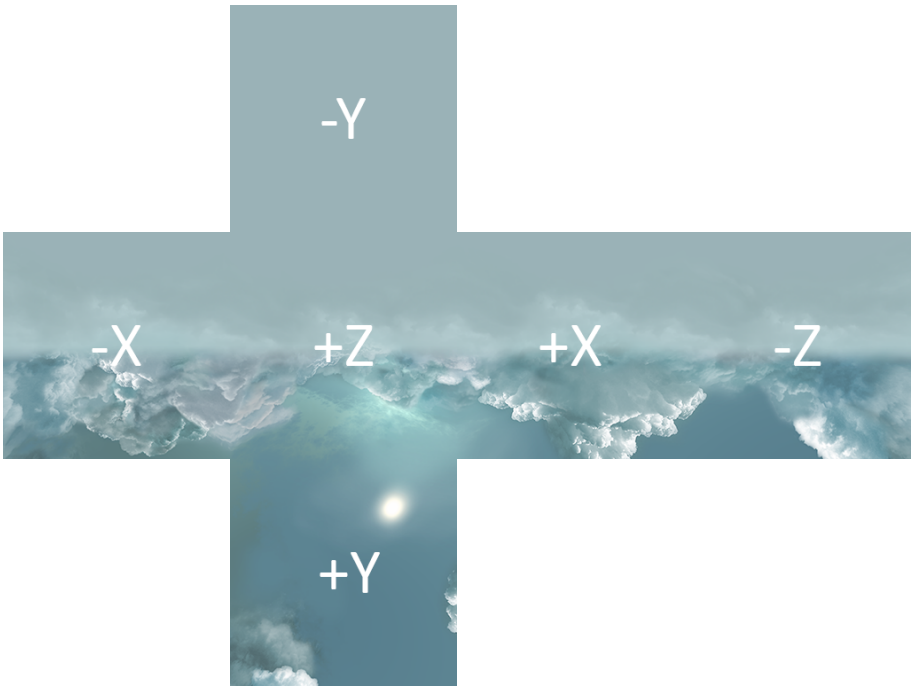


Figure 4.2: Cubemap.

To create a skybox we need to perform two steps:

- Create the cube map, where we will generate a single texture of OpenGL specific type *GL_TEXTURE_CUBE_MAP*.
- Create the cube mesh on which the texture will be mapped.

The code for creation of the cube map is straightforward. We first reserve a texture name in the GPU memory, then we set the rendering parameters of the texture and finally we copy the texture data to the GPU.

```
1  glGenTextures(1, &id_);
2  glBindTexture(GL_TEXTURE_CUBE_MAP, id_);
3
4  glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
5  glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
6  glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
7  glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
8  glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
9
10 glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGB,
11     imageForPositiveX->getWidth(), imageForPositiveX->getHeight(),
12     0, GL_BGR, GL_UNSIGNED_BYTE, imageForPositiveX->getData());
13 glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 0, GL_RGB,
14     imageForNegativeX->getWidth(), imageForNegativeX->getHeight(),
15     0, GL_BGR, GL_UNSIGNED_BYTE, imageForNegativeX->getData());
16 glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Y, 0, GL_RGB,
17     imageForPositiveY->getWidth(), imageForPositiveY->getHeight(),
18     0, GL_BGR, GL_UNSIGNED_BYTE, imageForPositiveY->getData());
19 glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, 0, GL_RGB,
20     imageForNegativeY->getWidth(), imageForNegativeY->getHeight(),
21     0, GL_BGR, GL_UNSIGNED_BYTE, imageForNegativeY->getData());
22 glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Z, 0, GL_RGB,
23     imageForPositiveZ->getWidth(), imageForPositiveZ->getHeight(),
24     0, GL_BGR, GL_UNSIGNED_BYTE, imageForPositiveZ->getData());
25 glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, 0, GL_RGB,
26     imageForNegativeZ->getWidth(), imageForNegativeZ->getHeight(),
27     0, GL_BGR, GL_UNSIGNED_BYTE, imageForNegativeZ->getData());
```

Listing 4.13: Generating the cube map.

In the listing above we assume that variables like *imageForPositiveX*, *imageForNegativeX* etc. are objects that represent the texture images with the complementary information like their width and height.

The texture parameters are chosen to get a smooth looking texture in cases when the size of one pixel on the texture doesn't correspond to the size of one pixel on the screen.

The cube mesh itself is prepared in the similar way as any other mesh. We first create an array of vertices, then we create a buffer on the GPU and copy the data to it. In case of the cube mesh we need to specify neither normals, since the cube will not be affected by any light in the scene, nor texture coordinates.

The important thing to keep in mind when creating vertices for the cube mesh is that the triangles should be facing inside the cube, therefore the vertices should be specified in the clockwise order if we are looking at the cube from outside.

Just like any other object, the skybox is rotated according to the rotation of the camera, although unlike any other object the skybox is not translated and it is placed at the center of the scene in camera view space. To achieve such transformation we need to cut the translation part of the view matrix and fill it with zeros.

```
mat4 skyboxMVPMatrix = window->getProjectionMatrix() * mat4(mat3(camera->
    getViewMatrix()));
glUniformMatrix4fv(BC_SHADER_UNIFORM_MVP_MATRIX, 1, GL_FALSE, value_ptr(
    skyboxMVPMatrix));
```

Listing 4.14: Creating model-view-projection matrix for skybox.

Finally, inside the fragment shader we set the output color using the built-in function *texture()*.

```
color = texture(cubeMapSampler, fragmentPosition);
```

Listing 4.15: Sampling the cube map.

In order to correctly draw the skybox we need to disable the depth test. This way anything that is rendered after the skybox will be rendered on top of it.

4.2.2 Fresnel reflection model

We will be using the classical Fresnel model to simulate reflection of a skybox on the ocean surface. To calculate proper reflection and refraction angles we need to know the refraction indices of the media. The refraction index of air is 1.000293 and the refraction index of water is 1.3330, making the ratio to be approximately 0.75.

Let's implement the computation of Fresnel factors from equation 3.13. As it is described in the article [14], the Fresnel factors may be precalculated and stored inside a 1D texture. It will save for us some performance. The Fresnel factor is then will be looked up by $|n * e|$, where n is a normal vector and e is a camera direction.

Let's get to the code and generate a texture for storing Fresnel factors.

```
GLubyte fresnelData[256];
for (int i = 0; i < 256; i++) {
    float t = (float)i / 256.0f;
    float theta_i = acos(t);
    float theta_t = asin(sin(theta_i) * 0.75);
    float f = 0.5 * ((pow(sin(theta_t - theta_i), 2) / pow(sin(theta_t +
        theta_i), 2)) + (pow(tan(theta_t - theta_i), 2) / pow(tan(theta_t +
        theta_i), 2)));
    fresnelData[i] = 256.0 * f;
}
fresnelTexture = new BCTexture1D(fresnelData, 256, GL_RED, GL_RED,
    GL_UNSIGNED_BYTE);
```

Listing 4.16: Storing Fresnel factors in 1D texture.

The data inside the array is stored in byte format. The variables *theta_i* and *theta_t* stand for angle of incidence and angle of transmittance respectively.

Now we can use the texture inside the fragment shader to lookup a Fresnel factor. The color of reflection will be calculated by sampling the cube map. The color of the refraction will be left as a color of the ocean, because there are no objects under the surface and the depth of the water is too significant for the bottom of the ocean to be visible.

To create an effect of a light transfering through the water, we will change the color of the waves. This will not produce physically correct results, but still will make the overall

appearance more realistic. This way the color of a point on the ocean surface is dependend on its height. Having a color of a wave and a color of the deep ocean water we can interpolate between those two values to get the result color.

```
vec3 evaluateSurfaceColor() {
    float waveGradientRange = 5;
    float waveHeightClamped = (clamp(fragmentPosition.y, -waveGradientRange /
        2, waveGradientRange / 2));
    float t = (waveHeightClamped + waveGradientRange / 2) / waveGradientRange;

    vec3 waterDeepColor = vec3(16.0 / 255.0, 40.0 / 255.0, 64.0 / 255.0);
    vec3 waterWaveColor = vec3(16.0 / 255.0, 72.0 / 255.0, 88.0 / 255.0);
    return mix(waterDeepColor, waterWaveColor, t);
}
```

Listing 4.17: Function for calculating the color of the surface.

This color is then used to evaluate the final color of the fragment:

```
float t = dot(fragmentNormalNormalized_V, cameraDirectionNormalized_V);
float f = texture(fresnelSampler, t).x;

vec3 refractColor = evaluateSurfaceColor();
vec3 reflectColor = vec3(texture(cubeMapSampler,
    reflectedCameraDirectionNormalized_W));

return mix(refractColor, reflectColor, f);
```

Listing 4.18: Evaluating the final color of the fragment.

4.3 Tiling problem

Up to this point we have a functioning application capable of generating one patch of an ocean surface. In order to create a full ocean we need to tile patches accross the desired area. Although the approach we have been using so far produces a seamless patch, there is still a problem that we need to take into account.

As we can see in the figure 4.3, two patches put together miss a connection between them. Simply moving them closer to each other will produce a seam, since the opposite sides are not identical to each other. Although, connecting the pathces with one extra row and an extra column of vertices will result in a perfect seamless surface.

We will have to change the existing implementation to update a few buffers and the cycles that go through the vertices. For instance, when calculating an index of a vertex in 1D array we need to take into account the new amount of vertices per side.

Finally, after performing the Fourier Transform and updating the vertices, we need to calculate the positions and normals for one extra row and an extra column. The normals of the vertices are simply equal to the normals of the opposite side. To obtain correct positions we need to add the size of the patch to either x or z coordinate of a point on the other side.

```
for (int m = 0; m < resolution_; m++) {
    int srcIndex = m * verticesCountPerSide_;
    int destIndex = srcIndex + resolution_;
```

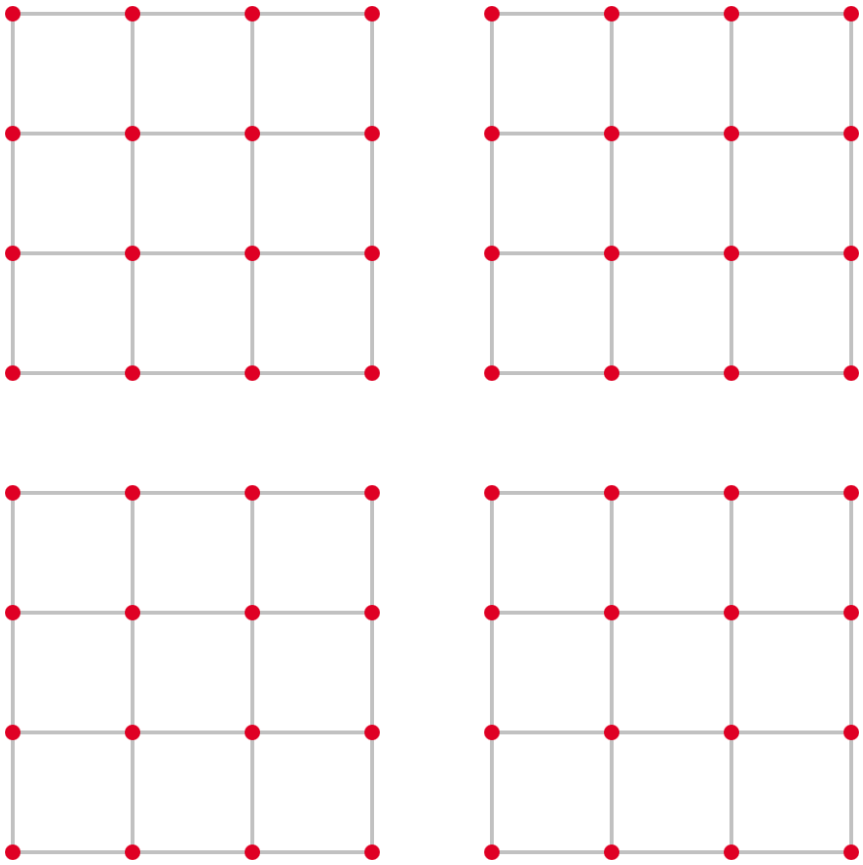


Figure 4.3: Gaps between patches produced by incorrect tiling.

```
4       positions_[destIndex] = positions_[srcIndex] + vec3(size_, 0, 0);
5       normals_[destIndex] = normals_[srcIndex];
6   }
7
8   for (int n = 0; n < verticesCountPerSide_; n++) {
9       int srcIndex = n;
10      int destIndex = verticesCountPerSide_ * resolution_ + n;
11      positions_[destIndex] = positions_[srcIndex] + vec3(0, 0, size_);
12      normals_[destIndex] = normals_[srcIndex];
13  }
```

Listing 4.19: Updating extra vertices for correct tiling.

4.4 Optimization

At this point the application works as expected, the ocean surface is generated and rendered correctly, although the frame rate falls very fast as the grid resolution increases. We will

try to increase the performance by optimizing the most time demanding parts. One of these parts is clearly the naive implementation of the Fourier Transform. In the next section we will rewrite this block of code implementing the Fast Fourier Transform. After we are done with this, we will try to use the parallel computation capabilities of a graphics card and transfer the evaluation of the FFT to the compute shaders.

4.4.1 FFT

In this section we will create a new class for the FFT computation. The computation itself will be done inside the only method of the class and it will be called *compute()*. This method is only applicable for 1D FFT computations. Therefore, we will need to call this method on each row and then on each column of the input array. Finally, we will use result to calculate the vertex positions and normals.

Let's go back and analyze the butterfly algorithm represented by the figure 3.2. For the natural access order the input data must be rearranged before usage. The algorithm for rearranging is essentially swapping values, where the index of one value is equal to the bit-reversed representation of the index of another value. The diagram for bit-reversal permutation is presented in the figure 4.4.

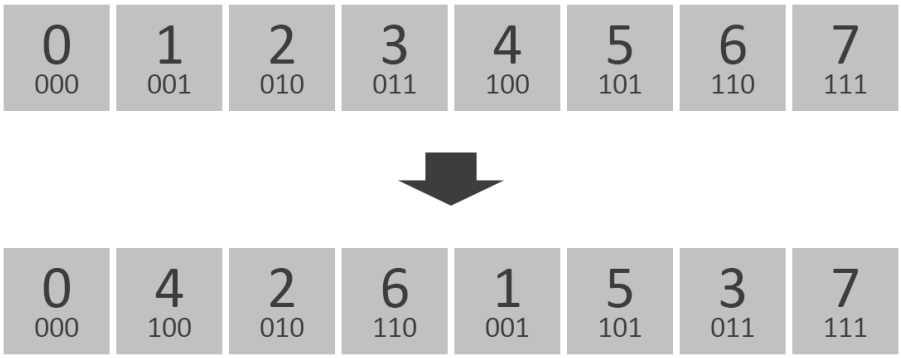


Figure 4.4: Bit-reversal permutation.

The computation of the bit-reversals on the fly is relatively expensive, therefore we will create a buffer which will hold the precalculated bit-reversed indices.

```
int target = 0;
for (int i = 0; i < resolution; i++) {
    bitReversedIndices_[i] = target;
    int mask = resolution;
    while (target & (mask >>= 1)) {
        target &= ~mask;
    }
    target |= mask;
}
```

Listing 4.20: Precalculating bit-reversed indices.

The process of performing the FFT is split into several stages. The amount of stages is equal to $log_2 N$, where N is the size of the input array. It is important to keep in mind that FFT only works with the data with the size of power of two elements. For each stage we perform a number of simple additions and multiplications. The two elements that we perform the operations on we will call *pair* and *match*. The basic idea is loop through all stages from 0 to $log_2 N - 1$ and for each stage loop through all *pair*-elements finding their respective *match*-elements and doing the computations on them. We will also use the following variables:

- *step* - the distance between the *pair*-element and the *match*-element.
- *jump* - the distance between the first elements of two sequential groups.
- *inner* - the index of the element inside the current group.

All of these values are changed depending on the current stage. Let's assume that the size of the input array is 8 so that it corresponds to the figure 3.2, then we can use the table 4.1 to analyze how the variables are changed depending on the current stage.

Stage	Step	Jump	Inner
0	1	2	0
1	2	4	0..1
2	4	8	0..3

Table 4.1: Parameters for different FFT stages.

If we take a closer look at the equation 3.2 we can see, that the exponential part of the equation does not depend on the input data. This fact makes it possible to precompute the exponentials once and use them later for any input array. Let's do this in the code:

```
1  exponentials_ = new BCComplex*[log2Resolution_];
2  for (int stage = 0, count = 1; stage < log2Resolution_; stage++, count <<= 1)
       {
3      exponentials_[stage] = new BCComplex[count];
4      for (int i = 0; i < count; i++) {
5          exponentials_[stage][i] = BCComplex(
6              cos(PI_2 * i / (count << 1)),
7              sin(PI_2 * i / (count << 1))
8          );
9      }
10 }
```

Listing 4.21: Precalculating exponentials.

In the listing above we allocate an array for storing the exponentials, called the twiddle factors, and then compute them for each stage, since the sizes of FFTs performed during each stages are different.

Having prepared all necessary data, we can implement the *compute()* method. This method will take three parameters. The first parameter is an array of complex numbers on which the FFT should be performed. The second and the third parameters are *offset* and

stride parameters respectively. These parameters specify how the data should be read from the input array. Assuming that i is the index among the elements we need to process, the actual index of this element inside the input array can be calculated using the following equation:

$$internalIndex = i * stride + offset \tag{4.5}$$

This leads us to the actual implementation of the Fast Fourier Transform inside the method *compute()*:

```
void BCFFT::compute(BCComplex *data, int offset, int stride) {
    for (int i = 0; i < resolution_; i++) {
        data_[i] = data[offset + bitReversedIndices_[i] * stride];
    }

    for (int step = 1, stage = 0; step < resolution_; step <<= 1, stage++) {
        int jump = step << 1;
        for (int inner = 0; inner < step; inner++) {
            for (int pair = inner; pair < resolution_; pair += jump) {
                int match = pair + step;

                BCComplex factor = exponentials_[stage][pair % step];
                BCComplex product = data_[match] * factor;

                data_[match] = data_[pair] - product;
                data_[pair] = data_[pair] + product;
            }
        }
    }

    for (int i = 0; i < resolution_; i++) {
        data[offset + i * stride] = data_[i];
    }
}
```

Listing 4.22: Method for performning FFT.

In the listing above the elements are first copied into an internal array at the bit-reversed indices, then the loop through all stages is performed computing the small FFTs. After the computation is done, the elements are copied back to the original array.

Finally, we will update the project in the way so that it will use the new FFT class. We will start by allocating five arrays for the FFT input data, among which will be one array for the height amplitudes, two arrays for slope vector amplitudes and two arrays for horizontal displacement amplitudes. These arrays will be filled each frame following the similar steps as were shown in the method *evaluateVertex()*. We basically split this method into three steps, where during the first step we calculate the amplitudes, then we perform the FFTs and finally we update the vertex data.

```
for (int m = 0; m < resolution_; m++) {
    for (int n = 0; n < resolution_; n++) {
        int index = m * resolution_ + n;

        vec2 wavevector = wavevectors_[index];
```

```
6           float wavevectorLength = length(wavevector);
7           vec2 wavevectorNormalized = wavevector / wavevectorLength;
8
9           heightAmplitudes_[index] = getHeightAmplitude(wavevectors_[index],
                baseAmplitudes_[index], baseAmplitudeConjugates_[index], time);
10          xSlopeAmplitudes_[index] = BCComplex(0, wavevector.x) *
                heightAmplitudes_[index];
11          ySlopeAmplitudes_[index] = BCComplex(0, wavevector.y) *
                heightAmplitudes_[index];
12          if (wavevectorLength > 0.000001f) {
13              xDisplacementAmplitudes_[index] = BCComplex(0, -
                    wavevectorNormalized.x) * heightAmplitudes_[index];
14              yDisplacementAmplitudes_[index] = BCComplex(0, -
                    wavevectorNormalized.y) * heightAmplitudes_[index];
15          } else {
16              xDisplacementAmplitudes_[index] = BCComplex(0.0f, 0.0f);
17              yDisplacementAmplitudes_[index] = BCComplex(0.0f, 0.0f);
18          }
19      }
20  }
```

Listing 4.23: Updating amplitudes.

We perform the FFT first on every row and then on every column of each input array. With the aid of the offset/stride indexing we can do this very easily:

```
1  for (int m = 0; m < resolution_; m++) {
2      fft_->compute(heightAmplitudes_, m * resolution_, 1);
3      fft_->compute(xSlopeAmplitudes_, m * resolution_, 1);
4      fft_->compute(ySlopeAmplitudes_, m * resolution_, 1);
5      fft_->compute(xDisplacementAmplitudes_, m * resolution_, 1);
6      fft_->compute(yDisplacementAmplitudes_, m * resolution_, 1);
7  }
8
9  for (int n = 0; n < resolution_; n++) {
10     fft_->compute(heightAmplitudes_, n, resolution_);
11     fft_->compute(xSlopeAmplitudes_, n, resolution_);
12     fft_->compute(ySlopeAmplitudes_, n, resolution_);
13     fft_->compute(xDisplacementAmplitudes_, n, resolution_);
14     fft_->compute(yDisplacementAmplitudes_, n, resolution_);
15 }
```

Listing 4.24: Performing FFTs.

Finally we go through all vertices to update their positions and their normals:

```
1  for (int m = 0; m < resolution_; m++) {
2      for (int n = 0; n < resolution_; n++) {
3          int srcVertexIndex = m * resolution_ + n;
4          int destVertexIndex = m * verticesCountPerSide_ + n;
5
6          if (((n + m) & 1) == 1) {
7              heightAmplitudes_[srcVertexIndex] = heightAmplitudes_[
                   srcVertexIndex] * -1.0f;
8              xSlopeAmplitudes_[srcVertexIndex] = xSlopeAmplitudes_[
                   srcVertexIndex] * -1.0f;
```

```
9               ySlopeAmplitudes_[srcVertexIndex] = ySlopeAmplitudes_[
                    srcVertexIndex] * -1.0f;
10              xDisplacementAmplitudes_[srcVertexIndex] =
                    xDisplacementAmplitudes_[srcVertexIndex] * -1.0f;
11              yDisplacementAmplitudes_[srcVertexIndex] =
                    yDisplacementAmplitudes_[srcVertexIndex] * -1.0f;
12          }
13
14          positions_[destVertexIndex] = vec3(
15              origins_[srcVertexIndex].x + xDisplacementAmplitudes_[
                    srcVertexIndex].re * CHOPPINESS_FACTOR,
16              heightAmplitudes_[srcVertexIndex].re,
17              origins_[srcVertexIndex].y + yDisplacementAmplitudes_[
                    srcVertexIndex].re * CHOPPINESS_FACTOR
18          );
19
20          normals_[destVertexIndex] = normalize(vec3(0.0f - xSlopeAmplitudes_[
                srcVertexIndex].re, 1.0f, 0.0f - ySlopeAmplitudes_[srcVertexIndex
                ].re));
21      }
22  }
```

Listing 4.25: Updating vertices.

This concludes the implementation of the FFT algorithm on the CPU. We can notice a great increase in performance and now we can run the application with higher grid resolutions. The results will be shown later in the Results chapter. For now we will continue with the optimization and see how the code functionality implemented so far could fit into the parallel computations using OpenGL compute shaders.

4.4.2 Parallel computation with compute shaders

The power of graphics card is in efficiency of parallel computations. Until the compute shaders were introduced, the graphics cards were design to work with such data as vertices, primitives or fragments. Each execution of a shader doesn't depend on any other execution of the same shader enabling a graphics card to perform the executions at the same time. The compute shaders were introduced in OpenGL version 4.2 and became a part of the core profile in version 4.3. The compute shaders allow running parallel computations on large amounts of data represented as 1-, 2- or 3-dimensional arrays.

There are many ways how to perform the FFT in parallel, although we will start with what seems to be the naive implementation. Each compute shader execution will compute the whole FFT of exactly one row and during the second pass the FFT will be performed on each column.

To pass the data to the GPU memory we will be using the buffers of the OpenGL type *GL_SHADER_STORAGE_BUFFER*. The data will be copied in rearranged order and we will need to perform the rearranging the second time between two different executions of the compute shader.

For this FFT implementation we will create a new class so we could compare it later to the one we created before. The structure of the new class will not change significantly except

for the method *compute()*, where we will have to update the code for the changes described in this section. Let's start with copying data to the GPU memory in rearranged order.

```
glBindBuffer(GL_SHADER_STORAGE_BUFFER, dataBuffer_);
BCComplex *buffer = (BCComplex *)glMapBuffer(GL_SHADER_STORAGE_BUFFER,
    GL_WRITE_ONLY);
for (int m = 0; m < resolution_; m++) {
    for (int n = 0; n < resolution_; n++) {
        int nReversed = bitReversedIndices_[n];
        int srcIndex = m * resolution_ + n;
        int destIndex = m * resolution_ + nReversed;
        buffer[destIndex] = data[srcIndex];
    }
}
glUnmapBuffer(GL_SHADER_STORAGE_BUFFER);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);
```

Listing 4.26: Copying data to the GPU memory.

Here we assume that *dataBuffer_* is a buffer created just like any other OpenGL buffer. We map the buffer to the main application memory, then we loop through all elements of the input array and copy them at the bit-reversed indices in the mapped buffer.

In the next step we execute the FFT shader for columns:

```
fftVerticalShaderProgram_->use();
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, dataBuffer_);
glUniform1i(BC_FFT_SHADER_UNIFORM_RESOLUTION, resolution_);
glDispatchCompute(resolution_ / BC_FFT_SHADER_SIZE_X, 1, 1);
glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
```

Listing 4.27: Executing FFT for columns.

Here we assume that *fftVerticalShaderProgram_* as well as *fftHorizontalShaderProgram_* used later are shader programs that perform FFT on columns and rows respectively. We pass the size of data using uniform parameters, then we bind the prepared buffer and dispatch the compute shaders.

As mentioned earlier, we need to rearrange the data between two FFT executions. This is done very similar way we do it when copy the initial data to the buffer.

```
glBindBuffer(GL_SHADER_STORAGE_BUFFER, dataBuffer_);
buffer = (BCComplex *)glMapBuffer(GL_SHADER_STORAGE_BUFFER, GL_READ_WRITE);
for (int n = 0; n < resolution_; n++) {
    for (int m = 0; m < resolution_; m++) {
        int mReversed = bitReversedIndices_[m];
        if (m < mReversed) {
            int leftIndex = m * resolution_ + n;
            int rightIndex = mReversed * resolution_ + n;
            BCComplex tmp = buffer[leftIndex];
            buffer[leftIndex] = buffer[rightIndex];
            buffer[rightIndex] = tmp;
        }
    }
}
glUnmapBuffer(GL_SHADER_STORAGE_BUFFER);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);
```

Listing 4.28: Rearranging data.

When the data is rearranged in the bit-reversed order, we perform the FFTs one more time, but this time on rows:

```
fftHorizontalShaderProgram_->use();
glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, dataBuffer_);
glUniform1i(BC_FFT_SHADER_UNIFORM_RESOLUTION, resolution_);
glDispatchCompute(resolution_ / BC_FFT_SHADER_SIZE_X, 1, 1);
glMemoryBarrier(GL_SHADER_STORAGE_BARRIER_BIT);
```

Listing 4.29: Executing FFT for rows.

Finally, we map the buffer in order to copy the data back to the output array, which is the same as the input array in this case:

```
glBindBuffer(GL_SHADER_STORAGE_BUFFER, dataBuffer_);
buffer = (BCComplex *)glMapBuffer(GL_SHADER_STORAGE_BUFFER, GL_READ_ONLY);
memcpy(data, buffer, resolution_ * resolution_ * sizeof(BCComplex));
glUnmapBuffer(GL_SHADER_STORAGE_BUFFER);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0);
```

Listing 4.30: Writing output.

The compute shaders are written in the same GLSL language we write any other type of OpenGL shader. The structure of the shader is also almost the same with a few differences. When writing a compute shader, we need to set its local execution size which will be then used to group single executions together. At this moment there is no particular requirement for the size, so we will set it to $(16, 1, 1)$.

We will have to work with complex numbers that are not natively supported by GLSL. Therefore, we will add our own functionality for creating, multiplying, adding and subtracting complex numbers:

```
struct complex {
    float re;
    float im;
};

complex complex_new(float re, float im) {
    complex result;
    result.re = re;
    result.im = im;
    return result;
}

complex complex_multiply(complex c1, complex c2) {
    return complex_new(c1.re * c2.re - c1.im * c2.im, c1.re * c2.im + c1.im *
        c2.re);
}

complex complex_add(complex c1, complex c2) {
    return complex_new(c1.re + c2.re, c1.im + c2.im);
}
```

```
20
21 complex complex_subtract(complex c1, complex c2) {
22     return complex_new(c1.re - c2.re, c1.im - c2.im);
23 }
```

Listing 4.31: GLSL complex numbers functionality.

In the last step we perform one FFT on a selected column. The algorithm is absolutely the same as the one we did for the CPU implementation, although instead of precomputing the exponentials we calculate them on the fly.

```
1  int m = int(gl_GlobalInvocationID.x);
2
3  for (int step = 1, stage = 0; step < resolution; step <<= 1, stage++) {
4      int jump = step << 1;
5      for (int group = 0; group < step; group++) {
6          for (int pair = group; pair < resolution; pair += jump) {
7              int match = pair + step;
8
9              int pairMapped = m * resolution + pair;
10             int matchMapped = m * resolution + match;
11
12             float f = 6.283185 * group / (step << 1);
13             complex factor = complex_new(cos(f), sin(f));
14             complex product = complex_multiply(data[matchMapped], factor);
15
16             data[matchMapped] = complex_subtract(data[pairMapped], product);
17             data[pairMapped] = complex_add(data[pairMapped], product);
18         }
19     }
20 }
```

Listing 4.32: Performing FFT on columns.

The shader for performing FFT on rows is very similar with the only difference for how the indices *pairMapped* and *matchMapped* are calculated:

```
1 int pairMapped = pair * resolution + n;
2 int matchMapped = match * resolution + n;
```

Listing 4.33: Index mapping in FFT shader for rows.

As it will be presented in the Results section, this naive implementation doesn't work as good as expected. It works even slower than the CPU version. There are two main problems with it:

- The GPU reads global data efficiently when it its reading is coalesced. This is not the case for our shaders.
- Reading of a value in global memory is very expensive operation. If the same value is used at least two times inside a shader, it should first be stored in the local memory.

Instead of having separate shaders for rows and columns we can write only one common FFT shader and transpose the data between executions. This way the algorithm for computing the 2D FFT is as follows:

- Perform 1D FFTs;
- Transpose data;
- Perform 1D FFTs;
- Transpose data;

The transposition of the elements can be also done on GPU. We will utilize the local memory to make the readings from the global memory coalesced. The input array is first subdivided into 2D blocks of data with the maximum available size of the work group, then the data is read sequentially and stored in transposed order to the local memory and then it is sequentially copied back to the global memory. We will set the size of the work group to $(32, 32, 1)$. Here is the implementation of the compute shader for transposition:

```
uint inputID = gl_GlobalInvocationID.x * resolution + gl_GlobalInvocationID.y;
uint outputID = (gl_WorkGroupID.y * gl_WorkGroupSize.y + gl_LocalInvocationID.
    x) * resolution + gl_WorkGroupID.x * gl_WorkGroupSize.x +
    gl_LocalInvocationID.y;

sharedData[gl_LocalInvocationID.y][gl_LocalInvocationID.x] = inputData[inputID
    ];
barrier();

outputData[outputID] = sharedData[gl_LocalInvocationID.x][gl_LocalInvocationID
    .y];
```

Listing 4.34: Compute shader for transposition.

The *barrier()* function blocks the execution of the current work item until all work items from the current work group have completed everything before the call of the *barrier()* function.

Now we need to minimize the access to the global memory inside the FFT shader. At this point we perform one full FFT inside one work item, which means that inside one FFT execution the operations are done sequentially. What we can do is to split the FFT execution to more working items, where the whole work group will be dedicated to perform one FFT on a single row. The entire row will first be stored in the local memory, then $\frac{N}{2}$ of work items will be executed where each work item will go through $log_2 N$ stages performing FFT on a pair of elements storing the results in the local memory and finally the produced values will be copied to the global memory. This way we ensure that the access to the global memory is minimized.

```
int rowNumber = int(gl_GlobalInvocationID.x);
int itemNumber = int(gl_GlobalInvocationID.y);

sharedData[itemNumber * 2] = data[rowNumber * resolution + itemNumber * 2];
sharedData[itemNumber * 2 + 1] = data[rowNumber * resolution + itemNumber * 2
    + 1];
barrier();

for (int i = 0, step = 1; i < log2Resolution; i++, step <<= 1) {
    int inner = itemNumber % step;
```

```
10      int pair = (itemNumber / step) * (step << 1) + inner;
11      int match = pair + step;
12
13      float f = 6.283185 * inner / (step << 1);
14      complex factor = complex_new(cos(f), sin(f));
15      complex product = complex_multiply(sharedData[match], factor);
16
17      complex pairElement = sharedData[pair];
18
19      sharedData[match] = complex_subtract(pairElement, product);
20      sharedData[pair] = complex_add(pairElement, product);
21
22      barrier();
23  }
24
25  data[rowNumber * resolution + itemNumber * 2] = sharedData[itemNumber * 2];
26  data[rowNumber * resolution + itemNumber * 2 + 1] = sharedData[itemNumber * 2
        + 1];
```

Listing 4.35: FFT in compute shader.

The performance we have achieved so far still lags behind the performance of CPU implementation. The speed measurement shows that there is a great performance lost during mapping of the GPU buffers to the main application memory. We need to think if we can minimize the amount of times we access the contents of the buffers from C++ code. The possible perfect solution would be to eliminate any buffer access from application code and leave all operations to perform inside GPU. Let's see if we can do this.

First of all let's get rid of the need to map the buffer in order to perform rearrangement inside the FFT code. In order to do this we will implement a new compute shader specially for this purpose. We will name the new shader *rearrangeComputeShader*.

```
1  uint rowNumber = gl_GlobalInvocationID.x;
2  uint itemNumber = gl_GlobalInvocationID.y;
3
4  sharedData[reverseBits(itemNumber)] = data[rowNumber * resolution + itemNumber
       ];
5  barrier();
6
7  data[rowNumber * resolution + itemNumber] = sharedData[itemNumber];
```

Listing 4.36: Compute shader for rearranging.

Inside this shader we go through all rows of the input array and for each row we rearrange elements inside it by swapping them with their pair elements located at bit-reversed indices. Since accessing the global memory on GPU is very expensive operation, we are calculating the bit-reversals on the fly using the function *reverseBits()*:

```
1  uint reverseBits(uint x) {
2      x = (((x & 0xaaaaaaaa) >> 1) | ((x & 0x55555555) << 1));
3      x = (((x & 0xcccccccc) >> 2) | ((x & 0x33333333) << 2));
4      x = (((x & 0xf0f0f0f0) >> 4) | ((x & 0x0f0f0f0f) << 4));
5      x = (((x & 0xff00ff00) >> 8) | ((x & 0x00ff00ff) << 8));
6      return ((x >> 16) | (x << 16)) >> 32 - log2Resolution;
7  }
```

Listing 4.37: Function for bit-reversal.

At this point the method *compute()* for performing the 2D FFT maps the buffer only two times: once to copy the input array to the buffer and once to copy the buffer data back to the input array, which is then used as output. Let's see if we can avoid copying data to the buffer. In order to do this we will modify the *compute()* method to accept an OpenGL buffer, which will be used directly to perform computations. We will also be saving the results back to this buffer and return it as output data. This way the *compute()* method will no longer map the buffer to the main application memory.

Let's update the main *BCOcean* class to correspond to the changes. It is obvious that we can no longer pass the C++ arrays to the FFT, therefore we will need to convert them to OpenGL buffers. This way instead of having arrays *heightAmplitudes_*, *xSlopeAmplitudes_*, *ySlopeAmplitudes_*, *xDisplacementAmplitudes_* and *yDisplacementAmplitudes_*, we will have buffers *heightAmplitudeBuffer_*, *xSlopeAmplitudeBuffer_*, *ySlopeAmplitudeBuffer_*, *xDisplacementAmplitudeBuffer_* and *yDisplacementAmplitudeBuffer_*. The buffers will be created the same way any other buffer is created in OpenGL.

Since this data no longer belongs to the main application memory, we need to write a new compute shader for updating the buffers on each frame. Many *BCOcean* methods have to be moved to the GPU as well, but their implementation stays the same. Let's get to the code and implement the *amplitudeUpdateComputeShader* for updating the amplitude buffers:

```
1  vec2 wavevetor = getWavevector();
2  float wavevectorLength = length(wavevetor);
3  vec2 wavevetorNormalized = normalize(wavevetor);
4
5  complex heightAmplitude = getHeightAmplitude(wavevetor);
6  complex xSlopeAmplitude = complex_multiply(complex_new(0.0f, wavevetor.x),
       heightAmplitude);
7  complex ySlopeAmplitude = complex_multiply(complex_new(0.0f, wavevetor.y),
       heightAmplitude);
8
9  complex xDisplacementAmplitude = complex_new(0.0f, 0.0f);
10 complex yDisplacementAmplitude = complex_new(0.0f, 0.0f);
11
12 if (wavevectorLength > 0.000001f) {
13     xDisplacementAmplitude = complex_multiply(complex_new(0.0f,
           wavevetorNormalized.x), heightAmplitude);
14     yDisplacementAmplitude = complex_multiply(complex_new(0.0f,
           wavevetorNormalized.y), heightAmplitude);
15 }
16
17 heightAmplitudes[gl_GlobalInvocationID.x * resolution + gl_GlobalInvocationID.
       y] = heightAmplitude;
18 xSlopeAmplitudes[gl_GlobalInvocationID.x * resolution + gl_GlobalInvocationID.
       y] = xSlopeAmplitude;
19 ySlopeAmplitudes[gl_GlobalInvocationID.x * resolution + gl_GlobalInvocationID.
       y] = ySlopeAmplitude;
20 xDisplacementAmplitudes[gl_GlobalInvocationID.x * resolution +
       gl_GlobalInvocationID.y] = xDisplacementAmplitude;
21 yDisplacementAmplitudes[gl_GlobalInvocationID.x * resolution +
       gl_GlobalInvocationID.y] = yDisplacementAmplitude;
```

Listing 4.38: Compute shader for updating amplitudes.

The data is ready for the FFT and not a single buffer mapping have been done so far. We will now do the same with the output of the FFT, which results are used to update the vertices and their normals.

Once again, the functionality that is responsible for updating the vertices must be moved to a new compute shader, which we will call *vertexUpdateComputeShader*. This shader will use results from FFT computation and update the vertex buffer objects *positionsVBO* and *normalsVBO* directly.

```
float factor = 1.0f;
if (((gl_GlobalInvocationID.x + gl_GlobalInvocationID.y) & 1) == 1) {
    factor = -1.0f;
}

uint index = gl_GlobalInvocationID.x * resolution + gl_GlobalInvocationID.y;

complex heightAmplitude = heightAmplitudes[index];
complex xSlopeAmplitude = xSlopeAmplitudes[index];
complex ySlopeAmplitude = ySlopeAmplitudes[index];
complex xDisplacementAmplitude = xDisplacementAmplitudes[index];
complex yDisplacementAmplitude = yDisplacementAmplitudes[index];

heightAmplitude = complex_multiply(heightAmplitude, factor);
xSlopeAmplitude = complex_multiply(xSlopeAmplitude, factor);
ySlopeAmplitude = complex_multiply(ySlopeAmplitude, factor);
xDisplacementAmplitude = complex_multiply(xDisplacementAmplitude, factor);
yDisplacementAmplitude = complex_multiply(yDisplacementAmplitude, factor);

vec2 origin = getOrigin();
vec3 position;
position.x = origin.x + xDisplacementAmplitude.re;
position.y = heightAmplitude.re;
position.z = origin.y + yDisplacementAmplitude.re;

vec3 normal = normalize(vec3(0.0f, 1.0f, 0.0f) - vec3(xSlopeAmplitude.re, 0.0f
    , ySlopeAmplitude.re));

positions[gl_GlobalInvocationID.y * (resolution + 1) + gl_GlobalInvocationID.x
    ] = vec4(position, 1.0f);
normals[gl_GlobalInvocationID.y * (resolution + 1) + gl_GlobalInvocationID.x]
    = vec4(normal, 0.0f);
```

Listing 4.39: Compute shader for updating vertices.

Finally, we move the code that fills gaps for correct tiling to a compute shader as well.

```
int executionID = int(gl_GlobalInvocationID.x);

int srcIndex;
int destIndex;

srcIndex = executionID * (resolution + 1);
destIndex = srcIndex + resolution;
```

```
8  positions[destIndex] = positions[srcIndex] + vec4(size, 0.0f, 0.0f, 0.0f);
9  normals[destIndex] = normals[srcIndex];
10
11 srcIndex = executionID;
12 destIndex = (resolution + 1) * resolution + srcIndex;
13 positions[destIndex] = positions[srcIndex] + vec4(0.0f, 0.0f, size, 0.0f);
14 normals[destIndex] = normals[srcIndex];
```

Listing 4.40: Compute shader for correct patch tiling.

This way the ocean surface is generated entirely on the GPU and the data never reaches the main application memory. As the performance measurement shows, this approach to the problem of parallel computation on GPU is extremely efficient and increases the frame rate from 5 up to more than 10 times comparing to when the data was mapped periodically from GPU memory and back. The results will be shown in the next chapter.

Chapter 5

Results

In this chapter we present the final results of the project. In the figure 5.1 is shown the rendering of the ocean with grid resolution of 256×256.

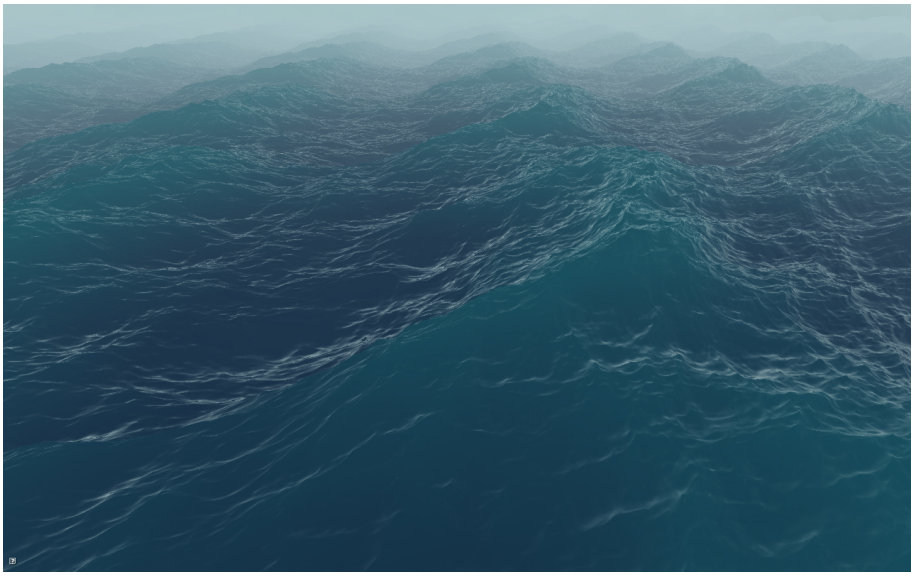


Figure 5.1: Final rendering with resolution of 256x256.

Also, in the figures 5.2 and 5.3 are presented frames of the animation with time step of 0.5 seconds. The grid resolution used in those figures is 256×256 and 32×32 respectively.

During the implementation we went through four of its stages, where during the first stage we were implementing the ocean surface generation using a naive algorithm for Fourier Transform. At the second stage we optimized the code by implementing the Fast Fourier Transform, but all calculations were still done on CPU. The third stage was dedicated to using the compute shaders and transferring the FFT computations to the GPU. The first implementation of compute shaders appeared to be running even slower than the CPU version, therefore we had to dive deeply into how GPU accesses its memory and optimize the existing compute shaders, what became the fourth stage of the implementation.

Results will be compared on different computers. For convenience, a list of used computers with their parameters is provided in the table 5.1.

Computer name	Processor	Graphics card
PC1	Intel Core i7-4790K	NVIDIA GeForce GTX 650
PC2	Intel Core i3-540	NVIDIA GeForce GTX 960
PC3	Intel Core i5-2500K	NVIDIA GeForce GTS 450

Table 5.1: List of computers used for performance test.

In the table 5.2 are presented the final results of the application performance after testing it on different computers. As the table suggests, the greatest increase in performance was achieved by implementing the Fast Fourier Transform algorithm. This proves that $O(n * log(n))$ is dramatically faster than $O(n^2)$ for large n. The second increase in performance was achieved after the proper implementation of compute shaders.

	Resolution	PC1	PC2	PC3
CPU without FFT	32	23	12	19
	64	1	1	1
CPU with FFT	32	896	871	886
	64	804	388	670
	128	205	101	174
	256	47	23	39
	512	9	4	8
	1024	2	1	1
GPU with FFT (Naive)	32	219	117	164
	64	90	36	77
	128	30	7	24
	256	7	1	5
	512	1	0	1
	1024	0	0	0
GPU with FFT	32	853	970	425
	64	618	957	336
	128	360	759	234
	256	164	438	124
	512	49	168	39
	1024	11	50	13

Table 5.2: Number of frames per second on different computers with different resolution.

It is also clear, that the performance fall with increasing the grid resolution is less significant when using compute shaders than doing the computations on CPU.

Finally, we will measure how much time in seconds needs an invocation of each compute shader presented in the application. We will choose the resolution of the grid to be 512×512. The performance test will be done on only one computer PC1. The results are presented in the table 5.3.

From this table we can see, that the most time demanding part is updating amplitudes

amplitudeUpdateComputeShader	0.002064
vertexUpdateComputeShader	0.001035
rearrangeComputeShader	0.000217
fftComputeShader	0.000420
transposeComputeShader	0.000818
tilingComputeShader	0.000028

Table 5.3: Time needed for different compute shaders to perform their work.

and vertices. Although, it is important to notice, that per each frame only one *amplitudeUpdateComputeShader* and only one *vertexUpdateComputeShader* is called, when *rearrangeComputeShader*, *fftComputeShader* and *transposeComputeShader* are called ten times each per each frame.

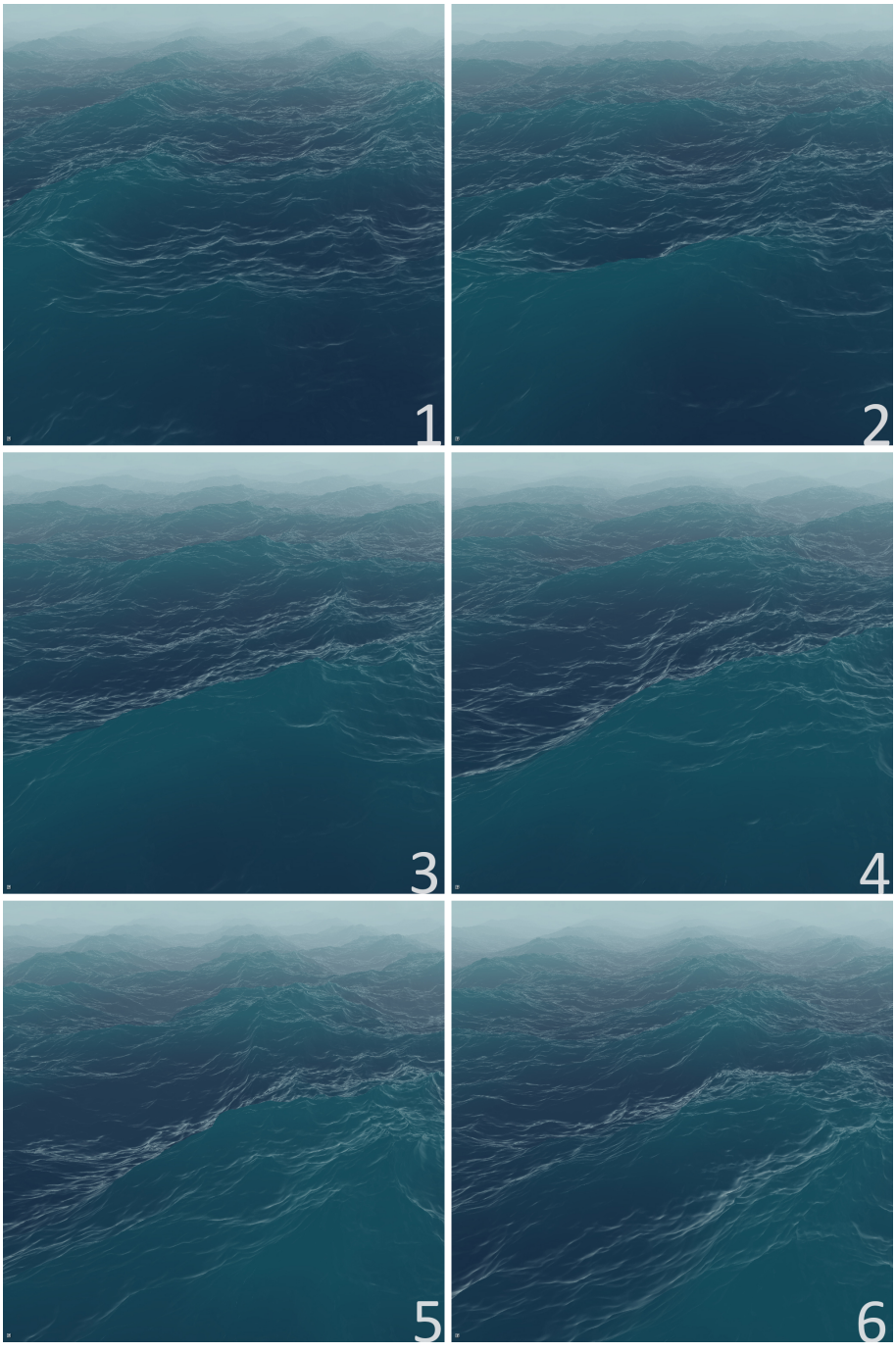


Figure 5.2: Animation frames with 256x256 resolution grid.

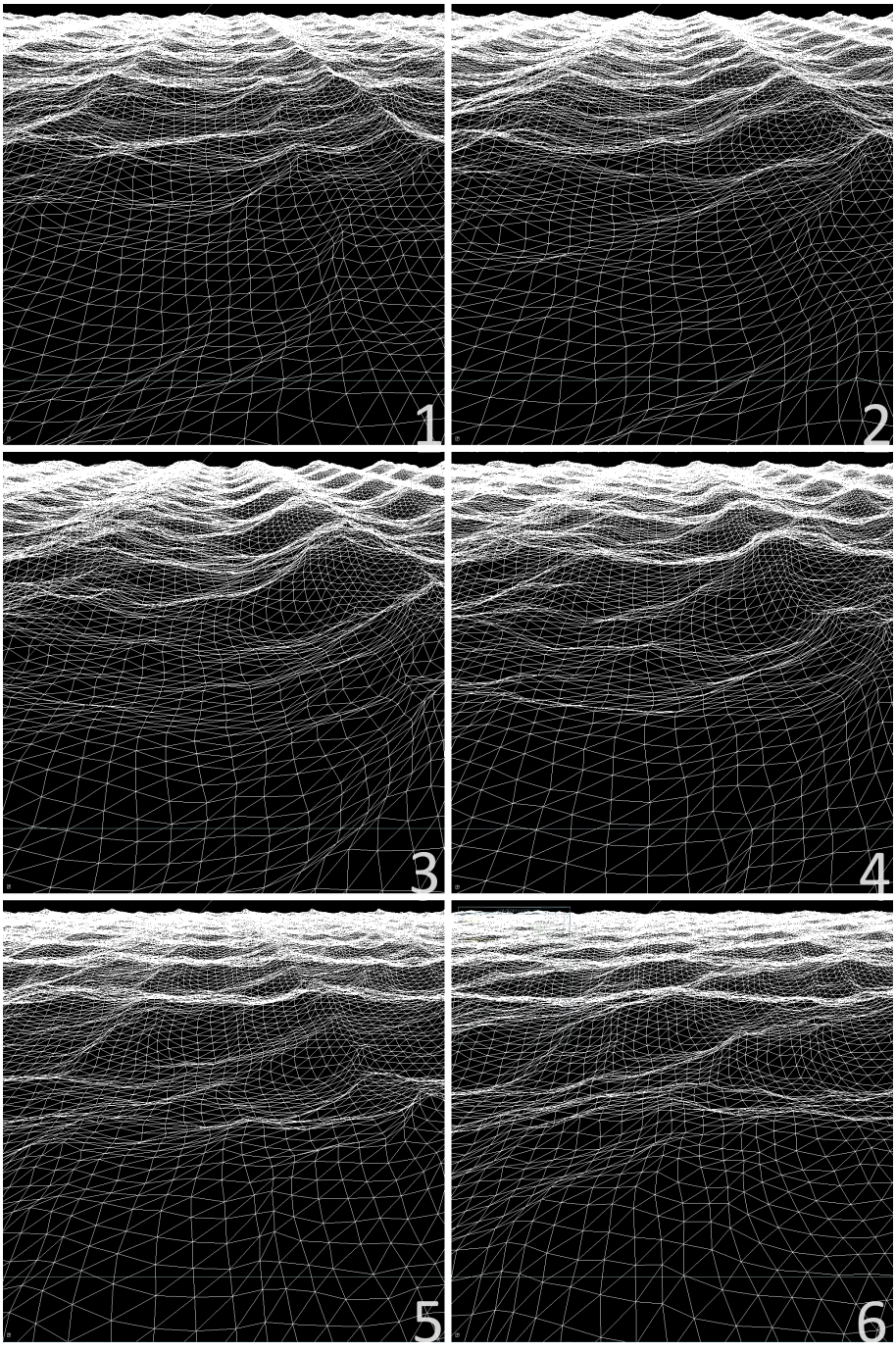


Figure 5.3: Animation frames with 32x32 resolution grid in wireframe mode.

Chapter 6

Conclusion

During the project we were able to create an application that simulates the ocean surface in real time. We used the FFT to evaluate sums on amplitudes. We were able to achieve a performance boost by placing computations to the graphics processor using OpenGL compute shaders. We looked how a patch of generated surface can be tiled across bigger areas of ocean. Finally, we implemented a Fresnel reflection and refraction model to render the scene.

The most challenging task was to make the compute shaders to work fast or at least at the speed of the CPU implementation. We have seen how buffer mapping and reading from the global GPU memory affects the overall performance.

6.1 Future work

Although, the project is finished and the expected results are achieved, there is still more functionality can be added. One of the main issues at this point is a periodicity artifact due to the patch tiling. Although it can be hidden with more intense fog, it is still better to add a random factor to the surface shape. This is achieved by modifying the surface with a noise. As described earlier, the Perlin noise could be used to blend the results of FFT with it.

The simulation works on an interactive frame rate. Although, the performance can drop very much if we try to extend the application by adding other objects to the scene or using the ocean simulator inside other more complex applications. This is why the optimization is still required. As we increase the patch amount during the tiling process, the frame rate will drop. To avoid it, we could decrease the level of detail for distant patches.

The visual appearance of the ocean is also a subject for improvement. At this point we use the Fresnel reflection/refraction model, although the color obtained by refracting the light is assumed to be a color of the ocean, which doesn't produce physically correct results. To improve the appearance we could implement a correct refraction or even use a ray tracing. Also, more realistic appearance could be achieved by adding ocean foam and sprays.

Bibliography

[1] From the archive: Titanic (1997). https://johnkennethmuir.wordpress.com/2012/04/11/from-the-archive-titanic-1997/.

[2] Introduction to fourier transform. http://www.thefouriertransform.com.

[3] Lagrangian fluid dynamics. http://www.glowinggoo.com/sph/.

[4] Ocean surface simulation. https://developer.nvidia.com/sites/default/files/akamai/gamedev/files/sdk/11/OceanCS_Slides.pdf.

[5] Ocean wave spectra. http://www.wikiwaves.org/Ocean-Wave_Spectra.

[6] Sozdanie protsedurnich landshaftov. http://habrahabr.ru/post/170521/.

[7] The story of the fast fourier transform. http://www.versci.com/fft/index.html.

[8] Water mathematics. https://habibs.wordpress.com/water-mathematics/.

[9] Wikipedia: Fourier transform. http://en.wikipedia.org/wiki/Fourier_transform.

[10] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[11] E. Darles, B. Crespin, D. Ghazanfarpour, and J. C. Gonzato. A survey of ocean simulation and rendering techniques in computer graphics. *Computer Graphics Forum*, 30(1):43–60, 2011.

[12] A. Fournier and W. T. Reeves. A simple model of ocean waves. *Computer Graphics*, 20(4):75–84, 1986.

[13] W. J. P. Jr. and L. Moskowitz. A proposed spectral form for fully developed wind seas based on the similarity theory of s. a. kitaigorodskii. *Journal of Geophysical Research*, 1964.

[14] H. Pan and Y. Zhang. Cuda-based real-time unbounded ocean rendering. *International Conference on Virtual Reality and Visualization*, pages 81–86, 2013.

[15] K. Perlin. An image synthesizer. *ACM SIGGRAPH Computer Graphics*, 19:287–296, 1985.

[16] W. Reeves. Particle systems: A technique for modeling a class of fuzzy objects. *Proceedings of the ACM SIGGRAPH*, pages 359–375, 1983.

[17] J. Tessendorf. Simulating ocean water. *ACM SIGGRAPH course notes*, 2001.

[18] L. Tian. Ocean wave simulation by the mix of fft and perlin noise. *WSCG: Poster Papers Proceedings*, pages 1–4, 2014.

[19] K. Yrjölä and T. Larsson. Real-time generation of plausible surface waves. In *The Annual SIGRAD Conference*, November 2007.