TUHH
Hamburg University of Technology

Research Project Thesis

Realtime GPGPU FFT Ocean Water Simulation

Institute of Embedded Systems

Author:
Fynn-Jorin Flügge

Supervisor:
Prof. Dr. Karl-Heinz Zimmermann

October 16, 2017

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Winsen, October 16, 2017

Fynn Flügge

Contents

List of Figures

List of Abbreviations

FFT Fast Fourier Transform

IFFT Inverse Fast Fourier Transform

DFT Discrete Fourier Transformation

GPU Graphical Processing Unit

CPU Central Processing Unit

GPGPU General Processing on GPU

CGI Computer Generated Imagery

FLOPS floating point operations per second

Chapter 1

Introduction

The Fast Fourier Transform (FFT) has a wide usage in signal processing applications, since the Discrete Fourier Transformation is essential for generating the spectrum or frequency domain of a signal resp. for computing the discrete convolution and correlation of signals. The IEEE magazine lists the FFT in the top ten of the most influential algorithms in the 20th century and defines it as "the most ubiquitous algorithm in use today to analyze and manipulate digital or discrete data". [1] Actually the publication "An Algorithm for the machine calculation of complex Fourier series" by J.W. Cooley and J.W. Tukey in 1965 introduced a new era in digital signal processing. [2]

In contrast to the common usage of FFT's this elaboration presents it's application in a different manner. With FFT's it is possible to simulate statistical based CGI water with a high degree of realism. Already in the early nineties in blockbuster movies such as *Titanic* and *Waterworld*, the CGI water was generated with FFT's. However, to this times it was not possible to animate FFT generated water on common computer hardware in realtime. Only when computer architectures became more powerful in the early 2000s the first video games used realtime FFT generated water. But there was still a big gap in the degree of realism between video games and CGI movies. While *Titanic* and *Waterworld* used a 2D-FFT with a resolution of 2048x2048 [3, p. 2], the pioneering game *Crysis* by *Crytek* with it's famous *CryEngine* applied a resolution of just 64x64, which lead to a big loss of quality compared to the high resolution CGI water in *Titanic* and *Waterworld*. The FFT's in *Crysis* were generated on CPU on a dedicated thread [4, p. 15]. Thereby large resolutions became a bottleneck in realtime water rendering, since one FFT (three FFT's when Choppy-displacement is applied, more in chapter 4.2.9) has to be computed for each frame separately. Since a 2D-FFT is well parallelizable, a solution to this performance issue is shifting the computations to the GPU. In contrast to a sequential processing CPU with just a few cores, a GPU consists of a parallel architecture with thousands of small cores to handle lots of tasks concurrently.

Already in the year 2001, once *NVIDIA* launched with it's *GeForce 3* series the first GPU's with programmable shaders, the basis for GPU computing were laid. It still lasted until 2003, the starting point of the GPGPU-century, when the Stanford University presented it's *BrookGPU*, the first GPGPU-API. Soon *NVIDIA* and *AMD* released their low-level-APIs *CUDA* and *STREAM*, but dedicated for their *GeForce*- and *Radeon*-Chips respectively. Due to dedications to *GeForce* (*CUDA*) and *Radeon* (*STREAM*) the two API's were not relevant in commercial usage for the market such as the video game industry. However, *CUDA* became very popular in researching area. In 2008 the *Khronos Group* released the platform-independent API *OpenCL*, which supports *GeForce*- and *Radeon*-Chips both and many more graphic processors such as the *Cell*-Processor used by *Sony* in it's *Playstation 3*. With *OpenCL* it is even possible to use it in parallel with *OpenGL* or *DirectX*. With *OpenGL 4.3* and *Direct3D 11* releases, Compute-Shaders were introduced, which made it easier to use GPGPU directly in graphic software without relying on *OpenCL*. [5] [6]

Meanwhile GPGPU has a wide range of usage in almost every kind of software for accelerating complex parallelizable computations as illustrated in figure 1.1. Figure 1.2 shows the performance advantage of GPU's towards CPU's in FLOPS and GBit/s up to 2014. While the *Intel Ivy Bridge* architecture reaches around 600 GFLOPS, the *Geforce 780 Ti* performs with 5400 GFLOPS. In 2017 *NVIDIA's Geforce GTX 1080 Ti* reaches a performance of 11 TFLOPS. [7]

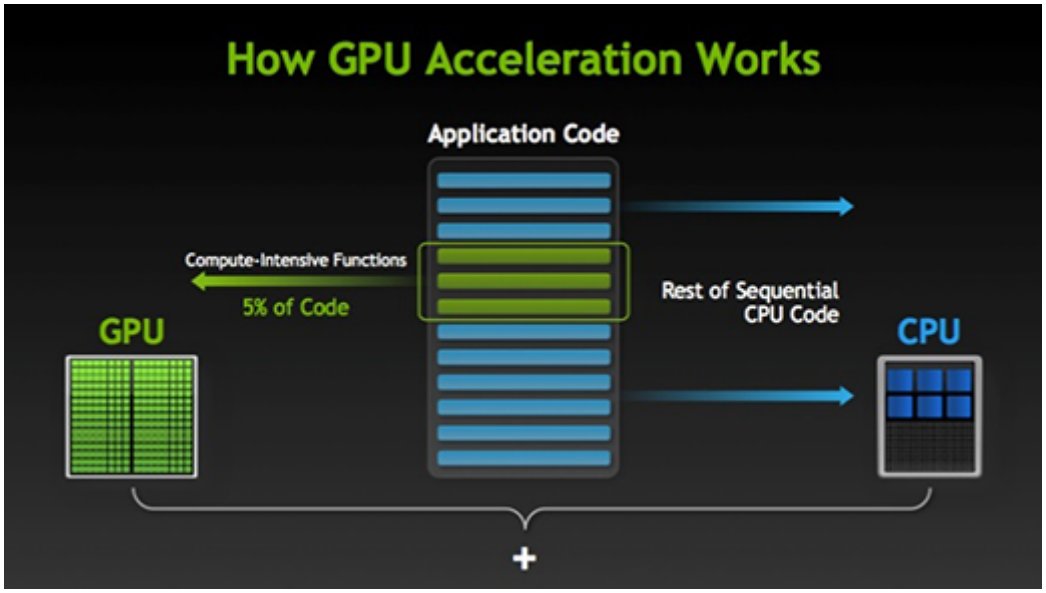

Figure 1.1: Graphical representation how the GPU accelerates the code execution by shifting parallelizable computations to the GPU. [8]

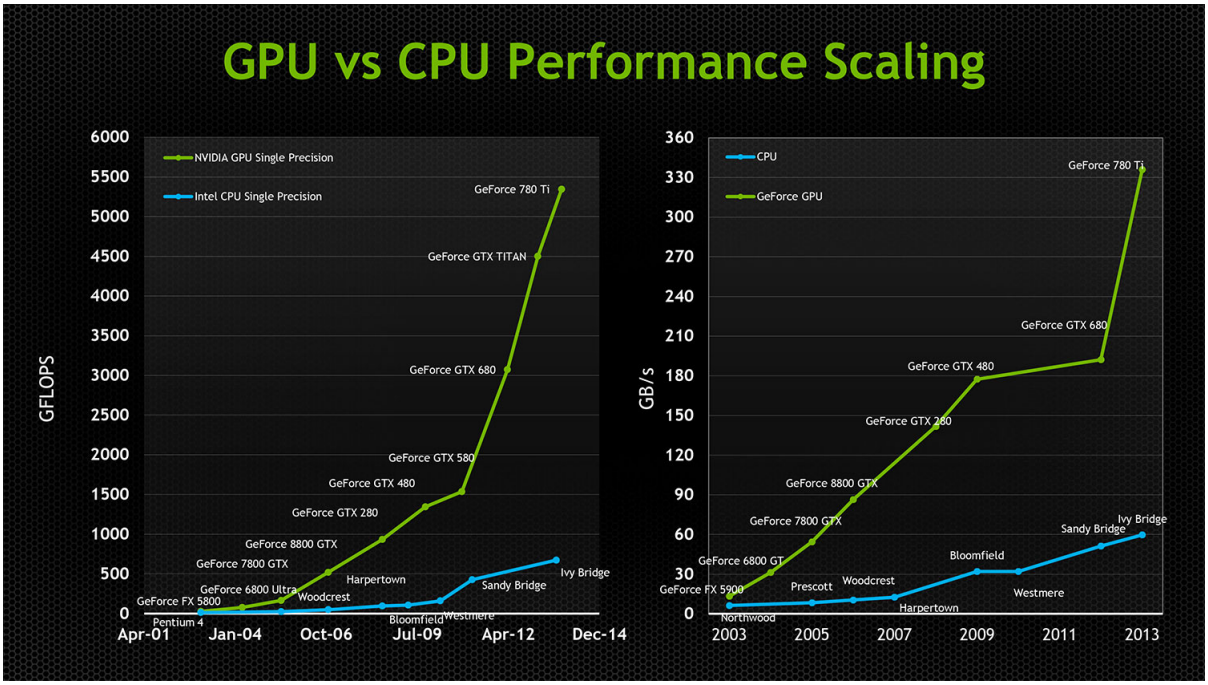

Figure 1.2: Comparism of CPU and GPU performance measured in GFLOPS and GB/s over the past years. [9]

In modern Game Engines GPGPU became not only essential for FFT generated water, but also for several post-processing effects[1] and deferred shading[2]. Figures 1.3 and 1.4 showing rendered scenes with GPGPU generated post-processing effects. This elaboration targets the development of a realtime CGI ocean renderer with a GPGPU FFT. The next chapter presents the Radix-2 Cooley-Tukey FFT-algorithm. Following this, it is demonstrated how the Cooley-Tukey algorithm is used to animate realtime ocean water based on Jerry Tessendorf's paper *Simulating Ocean Water* with an *OpenGL* GPGPU implementation. Figure 1.5 shows a rendered screenshot of a realtime FFT generated ocean surface.
All rendered demos and screenshots presented in this elaboration were produced with *Oreon Engine* [10].

[1] special effects that are applied on the rendered scene image

[2] screen-space shading technique, where light and shadows are added to the rendered scene image after the geometry processing has finished

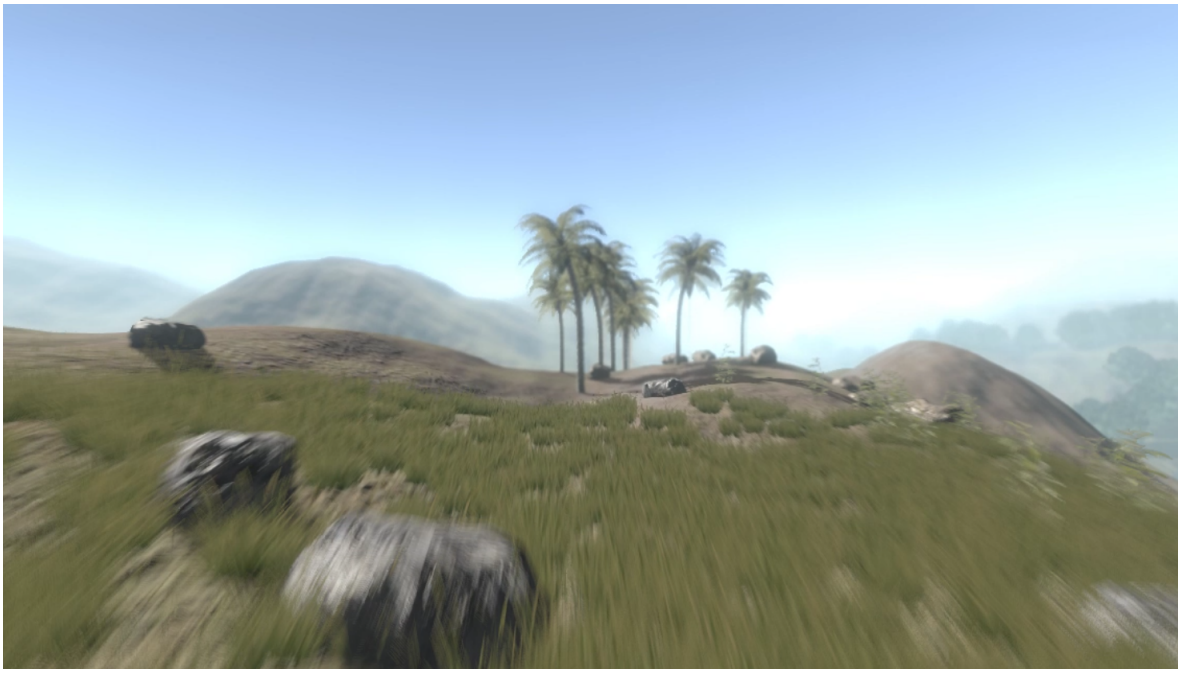

Figure 1.3: Screenshot of a rendered Scene with post processing bloom, depth of field blur and motion blur effect.

Figure 1.4: Screenshot of a rendered Scene with sun light scattering and lens flare effect.

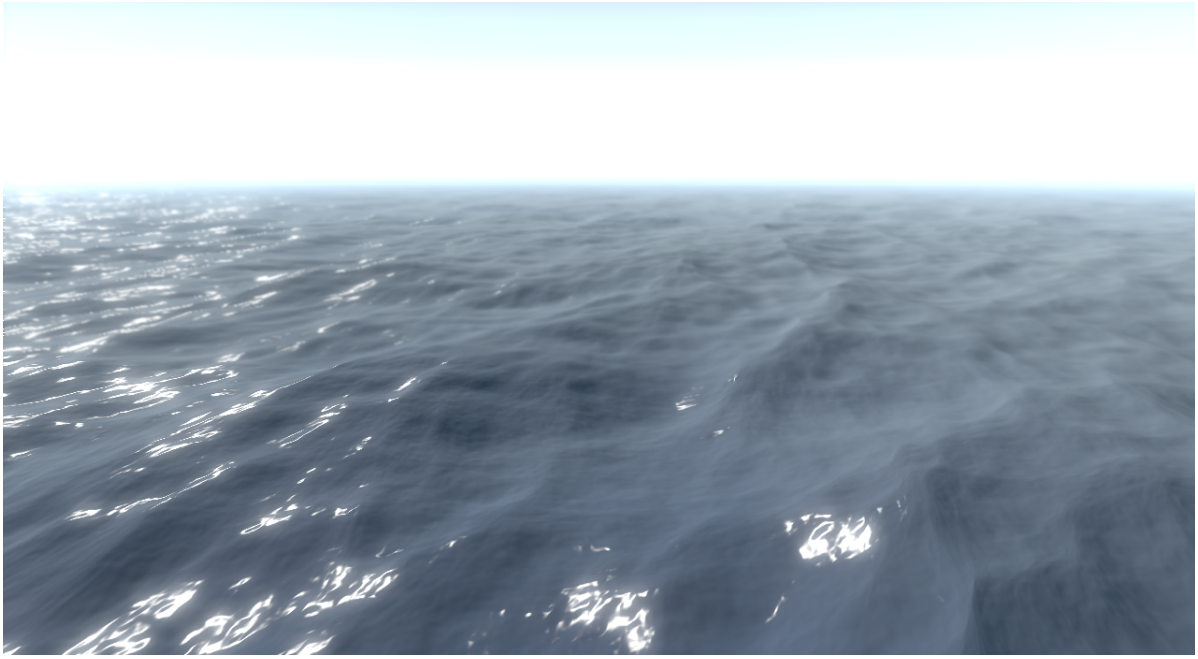

Figure 1.5: Screenshot of FFT Generated Water.

Chapter 2

Cooley-Tukey Fast Fourier Transform Algorithm

This chapter introduces the Cooley-Tukey FFT-algorithm for computing the Discrete Fourier Transformation (DFT). The DFT of a discrete signal $\mathbf{f}$ with N samples

$$\mathbf{f} = (\mathbf{f}[0], \mathbf{f}[1], ..., \mathbf{f}[N-1]) \tag{2.1}$$

is defined by the N-tuple

$$\mathbf{F} = (\mathbf{F}[0], \mathbf{F}[1], ..., \mathbf{F}[N-1]) \ , \tag{2.2}$$

where

$$\mathbf{F}[k] = \sum_{n=0}^{N-1} \mathbf{f}[n] e^{\frac{-2\pi i k n}{N}} \quad , n = 0, 1, ..., N-1 \ . \tag{2.3}$$

Computing $\mathbf{F}$ of a discrete signal $\mathbf{f}$ with N samples in the naive way would require N complex multiplications and $N-1$ complex additions for each element of $\mathbf{F}$, which total time is proportional to $\mathcal{O}(n^2)$. The FFT algorithm reduces the complexity to $\mathcal{O}(n \log n)$. For large N this leads to a massive performance advantage. One limitation of the FFT is, that the DFT must be of even order and for most efficiency N must be a power of 2. An option to execute FFT's on signal data without N^2 samples is by adding zeros until the closest power of 2 is reached. This technique is known as "Zero Padding". [11, pp. 254, 279-281, 291]

2.1 Four-Point DFT

The basic concept of the FFT is to exploit the structure of the DFT by smartly rearranging the products. Before the general procedure of the FFT algorithm will be presented in chapter 2.2, the computation of the DFT with $N = 4$ samples is demonstrated.

Since

$$e^{-i\pi/2} = -i \,, \tag{2.4}$$

$\mathbf{F}[k]$ can be simplified for $N = 4$ to

$$\mathbf{F}[k] = \sum_{n=0}^{3} = (-i)^{kn}\, \mathbf{f}[n] \,, \tag{2.5}$$

which leads to

$$\mathbf{F}[k] = \mathbf{f}[0] + (-i)^k\, \mathbf{f}[1] + (-1)^k\, \mathbf{f}[2] + i^k\, \mathbf{f}[3] \,. \tag{2.6}$$

Hence, $\mathbf{F}^\top$ is given by

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} \mathbf{f}^\top = \begin{bmatrix} \mathbf{f}[0] + \mathbf{f}[1] + \mathbf{f}[2] + \mathbf{f}[3] \\ \mathbf{f}[0] - i\mathbf{f}[1] - \mathbf{f}[2] + i\mathbf{f}[3] \\ \mathbf{f}[0] - \mathbf{f}[1] + \mathbf{f}[2] - \mathbf{f}[3] \\ \mathbf{f}[0] + i\mathbf{f}[1] - \mathbf{f}[2] - i\mathbf{f}[3] \end{bmatrix} , \tag{2.7}$$

which takes $\mathcal{O}(n^2)$ multiplications for solving $\mathbf{F}$. [11, p. 281]
Rearranging the summands of $\mathbf{F}[k]$ to

$$\begin{bmatrix} \mathbf{f}[0] + \mathbf{f}[2] + \mathbf{f}[1] + \mathbf{f}[3] \\ \mathbf{f}[0] - \mathbf{f}[2] - i\mathbf{f}[1] + i\mathbf{f}[3] \\ \mathbf{f}[0] + \mathbf{f}[2] - \mathbf{f}[1] - \mathbf{f}[3] \\ \mathbf{f}[0] - \mathbf{f}[2] + i\mathbf{f}[1] - i\mathbf{f}[3] \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} \begin{bmatrix} \mathbf{f}[0] \\ \mathbf{f}[2] \\ \mathbf{f}[1] \\ \mathbf{f}[3] \end{bmatrix} \tag{2.8}$$

exposes a reduction of the computation costs by precomputing the subexpressions $\mathbf{f}[0] + \mathbf{f}[2], \mathbf{f}[0] - \mathbf{f}[2], \mathbf{f}[1] + \mathbf{f}[3]$ and $\mathbf{f}[1] - \mathbf{f}[3]$, since

$$\begin{bmatrix} \mathbf{f}[0] + \mathbf{f}[2] + \mathbf{f}[1] + \mathbf{f}[3] \\ \mathbf{f}[0] - \mathbf{f}[2] - i\mathbf{f}[1] + i\mathbf{f}[3] \\ \mathbf{f}[0] + \mathbf{f}[2] - \mathbf{f}[1] - \mathbf{f}[3] \\ \mathbf{f}[0] - \mathbf{f}[2] + i\mathbf{f}[1] - i\mathbf{f}[3] \end{bmatrix} = \begin{bmatrix} (\mathbf{f}[0] + \mathbf{f}[2]) + (\mathbf{f}[1] + \mathbf{f}[3]) \\ (\mathbf{f}[0] - \mathbf{f}[2]) - i(\mathbf{f}[1] - \mathbf{f}[3]) \\ (\mathbf{f}[0] + \mathbf{f}[2]) - (\mathbf{f}[1] + \mathbf{f}[3]) \\ (\mathbf{f}[0] - \mathbf{f}[2]) + i(\mathbf{f}[1] - \mathbf{f}[3]) \end{bmatrix} . \tag{2.9}$$

The FFT exploits the possibility to save operations by rearranging the components and reusing intermediate results of the DFT. The next section presents the execution of the FFT for N=4.

2.2 Four-Point-FFT

A two-point DFT consisting of two Complex multiplies and adds can be sketched with the following two-point butterfly diagram:

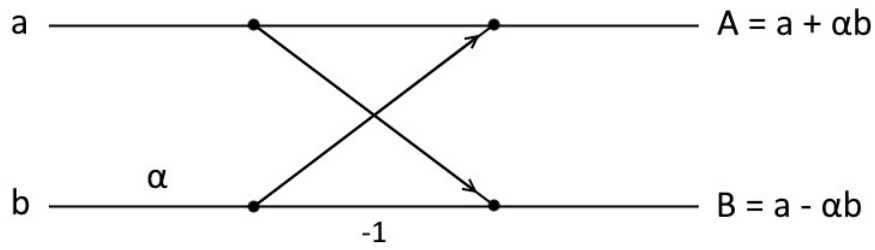

Figure 2.1: A two-point butterfly diagram consists of a single butterfly operation. A butterfly operation takes two complex numbers **a** and **b** as input and a fixed complex number ***a*** and outputs the complex values **A** and **B**. [12]

Therefore, the subexpressions $\mathbf{f}[0] + \mathbf{f}[2]$ and $\mathbf{f}[0] - \mathbf{f}[2]$ can be depicted as a butterfly operation as follows:

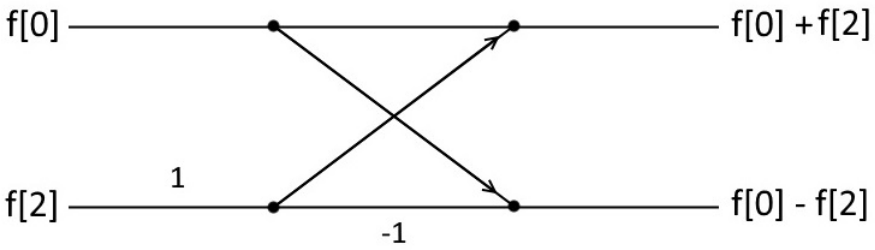

Figure 2.2: Two-point butterfly diagram with f[0], f[2] input.

The butterfly operation for $\mathbf{f}[1] + \mathbf{f}[3]$ and $\mathbf{f}[1] - \mathbf{f}[3]$ is similar:

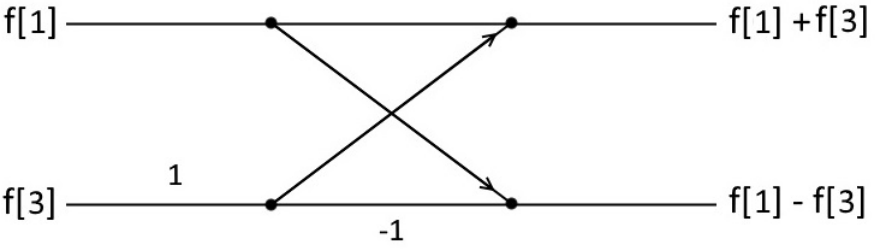

Figure 2.3: Two-point butterfly diagram with f[1], f[3] input.

With divide and conquer the output of the previous two butterfly operations can be used for two further butterfly operations to obtain $\mathbf{F}^\top$:

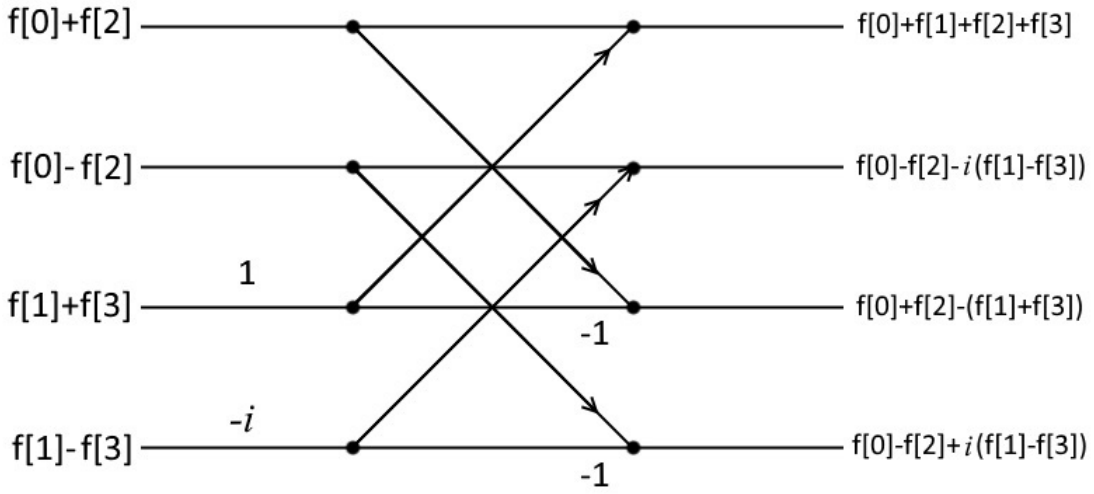

Figure 2.4: Four-point butterfly diagram second stage.

Solving the DFT for N=4 with butterfly operations reduces the computation cost to $\mathcal{O}(n \log n)$, since eight complex multiplies and adds are executed (two for each sample $\mathbf{f}[k]$). [12]
Combining figures 2.2,2.3 and 2.4 to one graph results in the butterfly diagram for the four-point FFT:

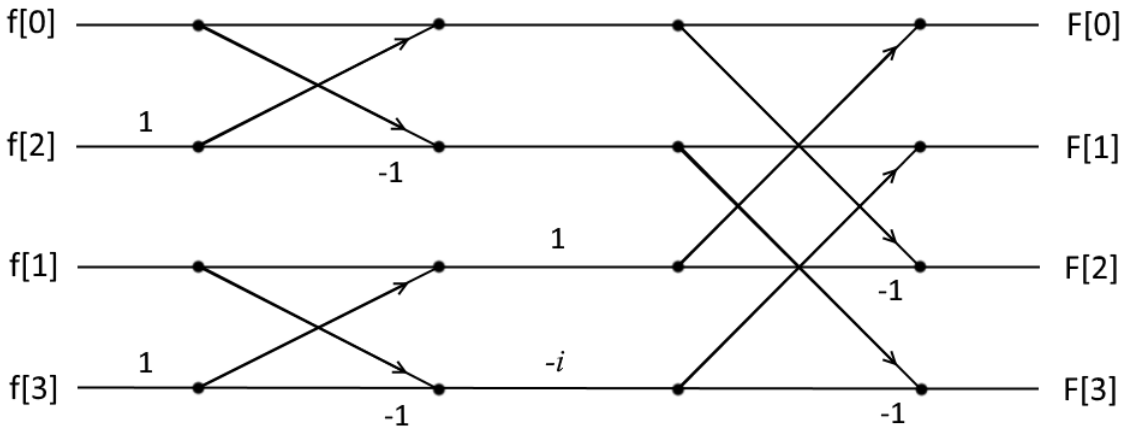

Figure 2.5: Four-point butterfly diagram.

2.3 From 4-Point to N-Point FFT

The strategy of the FFT is that each component of the DFT is not computed separately, which would result in unnecessary repetitions of a (relative to N) large number of computations. Instead, the FFT does it's computation in stages. Figures 2.2 and 2.3 are the butterfly operations of the first stage, while figure 2.4 is the second stage of the four-point FFT. Each stage takes N complex numbers from the previous stage's output (resp. $\mathbf{f}$ in the first stage) and executes $N/2$ butterfly operations. The output of these butterfly operations are N complex numbers as input for the next stage (resp. $\mathbf{F}$ in the last stage). Since one stage involves $N/2$ butterfly operations with two complex additions, two complex multiplications and one sign inversion (resp. multiplication by -1) for each butterfly operation, one stage can be executed in $\mathcal{O}(n)$ time. [12] [11, pp. 285-286]

As seen in chapter 2.2 , FFT's can be well illustrated with butterfly diagrams. Due to figure 2.5, it is obvious that the four-point FFT consists of two stages ($\log_2 N$ for $N = 4$) with 2 butterfly operations for each each stage ($N/2$ for $N = 4$). For a general formulation of the FFT the simplification $e^{-i\pi/2} = -i$, from chapter 2.1 is not helpful. Therefore, a new notation is introduced, the twiddle factor [11, pp. 281-282] [13]:

$$W_N = e^{-i\frac{2\pi}{N}} \; . \tag{2.10}$$

The matrix representation of the four-point DFT by using the twiddle factor is given by

$$\begin{bmatrix} W_4^0 & W_4^0 & W_4^0 & W_4^0 \\ W_4^0 & W_4^1 & W_4^2 & W_4^3 \\ W_4^0 & W_4^2 & W_4^4 & W_4^6 \\ W_4^0 & W_4^3 & W_4^6 & W_4^9 \end{bmatrix} \mathbf{f}^\top, \tag{2.11}$$

and with rearrangement of the components of $\mathbf{f}^\top$

$$\begin{bmatrix} W_4^0 & W_4^0 & W_4^0 & W_4^0 \\ W_4^0 & W_4^2 & W_4^1 & W_4^3 \\ W_4^0 & W_4^4 & W_4^2 & W_4^6 \\ W_4^0 & W_4^6 & W_4^3 & W_4^9 \end{bmatrix} \begin{bmatrix} \mathbf{f}[0] \\ \mathbf{f}[2] \\ \mathbf{f}[1] \\ \mathbf{f}[3] \end{bmatrix} . \tag{2.12}$$

Due to the Nth roots of unity of W_N^k for k=0,...,N-1 the matrix representation can be simpliefied to

$$\begin{bmatrix} W_4^0 & W_4^0 & W_4^0 & W_4^0 \\ W_4^0 & W_4^2 & W_4^1 & W_4^3 \\ W_4^0 & W_4^0 & W_4^2 & W_4^2 \\ W_4^0 & W_4^2 & W_4^3 & W_4^1 \end{bmatrix} \begin{bmatrix} \mathbf{f}[0] \\ \mathbf{f}[2] \\ \mathbf{f}[1] \\ \mathbf{f}[3] \end{bmatrix} , \tag{2.13}$$

since

$$e^{i\frac{2\pi k}{N}} = e^{i\frac{2\pi N + k}{N}} \; , \tag{2.14}$$

and thus

$$W_N^k = W_N^{k+mN} \quad \forall m \in \mathbb{Z} \; . \tag{2.15}$$

as defined in [11, pp. 281-282].

The equivalent butterfly diagram to figure 2.5 with twiddle factors is depicted in figure 2.6. Alternatively the sign-inversions in the butterfly operations can be replaced by twiddle factors regarding to the Nth root of unity, since

$$-W_N^k = W_N^{k+\frac{N}{2}} \quad . \tag{2.16}$$

The alternative butterfly diagram with replaced sign inversions is illustrated in figure 2.7.

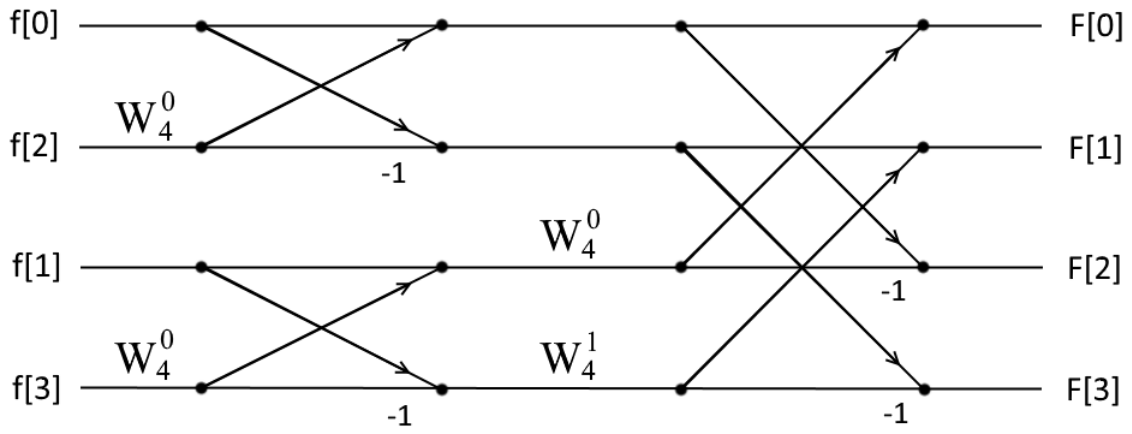

Figure 2.6: Four-point butterfly diagram A.

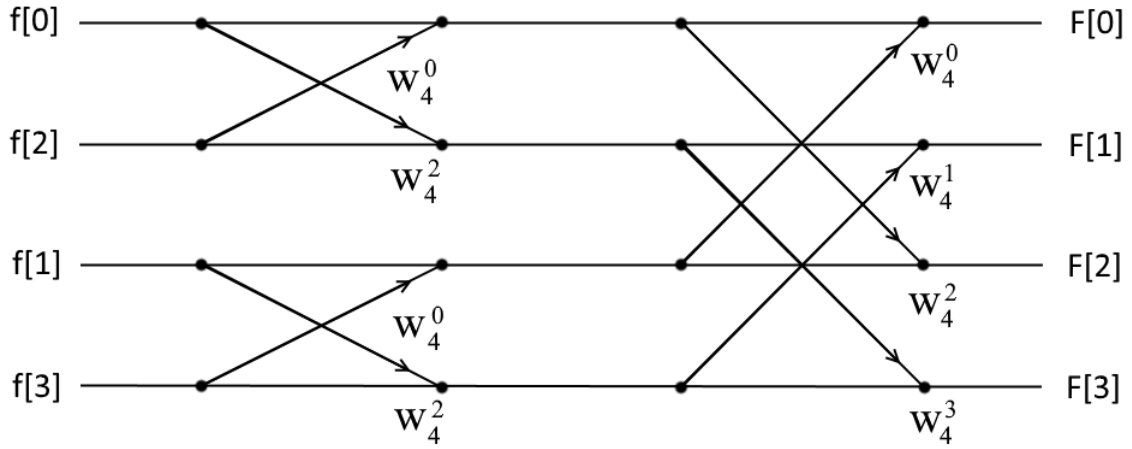

Figure 2.7: Four-point butterfly diagram B.

The FFT-implementation of this elaboration uses butterfly diagrams similar to 2.7. More on this will be presented in chapter 4.2. Due to the twiddle factors W_N^k, the multiplicators within a butterfly-graph can be expressed in a gerneric way, which facilitates the construction of butterfly graphs for $N > 4$. But in order to build up the butterfly grah for $N > 4$ it is necessary to rearrange the DFT matrix resp. the input vector $\mathbf{f}^\top$ as done in the butterfly diagrams 2.5, 2.6 and 2.7.

Referring to the eight-point FFT buttefly diagram in figure 2.8, the rule for arranging the components of $\mathbf{f}^\top$ can be extracted. By inspecting the input samples it is apparent that the components $\mathbf{f}^\top$ in the initial stage are ordered bit-reversed. [11, pp. 289-291]

The exponents k of the twiddle factors W_N have the form

$$k = n \cdot \frac{N}{2^{stage}} \ , \tag{2.17}$$

where n is the vertical index. Further the number of butterfly stages can be generically determined to $\log_2 n$ stages, since the butterfly-span is doubled after each stage to a maximum span of $N/2$ in the last stage. Thus, the total execution time of the FFT with $\log_2 n$ stages and $\mathcal{O}(n)$ for each stage is $\mathcal{O}(n \log n)$. Due to the reason that the butterfly-span is doubled after each stage, the Cooley-Tukey FFT algorithm is a divide and conquer algorithm. Since the algorithm can only be applied on DFT's with N as a power of two, it is also called Radix-2 FFT algorithm. [11, p. 288]

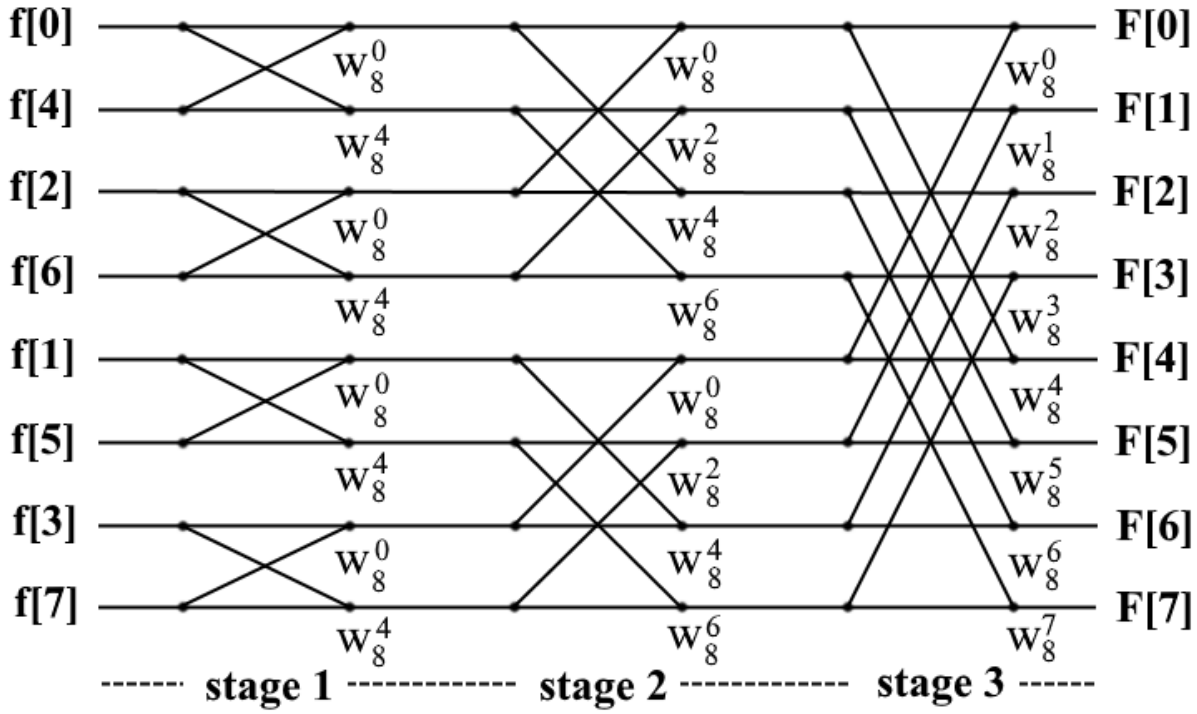

Figure 2.8: Eight-point butterfly diagram. [12]

Chapter 3

The Statistical Ocean Waves Model

This chapter introduces the statistical ocean waves model. The following presented approach snythesizes a square surface of ocean waves from a Fast Fourier Transform on a statistical, emperically-based oceanographic spatial spectrum of the ocean surface. The generated patch can be tiled seamlessly over a large area. What the FFT produces, is the synthesized height displacement from octaves of sinusoidal waves at each point of the square at time t. The rapid FFT algorithm makes it possible to computationally decompose the ocean height field in trivial time. The height $h(\mathbf{x}, t)$ at a horizontal position $\mathbf{x} = (x, z)$ at time t is defined in [3, p. 6] by the sum of sinusoids with complex amplitudes:

$$h(\mathbf{x}, t) = \sum_{\mathbf{k}} \tilde{h}(\mathbf{k}, t) \exp(i\mathbf{k} \cdot \mathbf{x}) \ . \tag{3.1}$$

The wavevector $\mathbf{k}$ is a two-dimensional horizontal vector, which points in the direction of travel of the wave. The components k_x and k_z of $\mathbf{k} = (k_x, k_z)$ are defined in [3, p. 6] as

$$k_x = (2\pi n/L) \tag{3.2}$$

and

$$k_z = (2\pi m/L) \tag{3.3}$$

where n and m have bounds:

$$-N/2 \leq n, m < N/2 \ . \tag{3.4}$$

L is the horizontal dimension of the patch. The height amplitude Fourier components $\tilde{h}(\mathbf{k}, t)$ are generated by gaussian random numbers and a spatial spectrum. Oceanographic research showed with reference to statistical analysis, photographic and radar measurements of the ocean surface that the wave height amplitudes $\tilde{h}(\mathbf{k}, t)$ are approximately independent gaussian fluctuations with the Phillips spectrum $P_h(k)$ defined in [3, p. 6] by:

$$P_h(\mathbf{k}) = A \frac{\exp(-1/(kL)^2)}{k^4} |\hat{k} \cdot \hat{w}|^2 \tag{3.5}$$

where

$$L = V^2/g \ .$$

V is the wind speed, w is the direction of the wind and $g = 9.8m/sec^2$, the gravitational constant of the earth. The factor $|\hat{k} \cdot \hat{w}|^2$ removes waves that are directed perpendicular to the wind direction vector.
For suppressing very small waves with $l << L$ the Phillips spectrum can be multiplied with the factor $exp(-k^2l^2)$.
The Fourier amplitudes are denoted as

$$\tilde{h}_0(\mathbf{k}) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(\mathbf{k}} \tag{3.6}$$

where ξ_r and ξ_i are independent numbers of a gaussian normal distribution with mean 0 and standard deviation 1. [3, p. 6]
Finally the dispersion relation

$$w^2(k) = gk \tag{3.7}$$

is integrated into the equation of the Fourier amplitudes. [3, p. 5] The dispersion relation is the relationship between frequencies and magnitude of the water wavevectors k_i, where g is the gravitational constant.
With the dispersion relation, the Fourier amplitudes at time t are denoted in [3, p. 6] by

$$\tilde{h}(\mathbf{k}, t) = \tilde{h}_0(\mathbf{k}) \exp(iw(k)t) + \tilde{h}_0^*(-\mathbf{k}) \exp(-iw(k)t) \quad . \tag{3.8}$$

Chapter 4

Generating The Height Field

4.1 Preparing the IFFT-Equation

In order to synthesize the ocean height field with the FFT, the DFT-equation is derived from (3.1). First, two variables k, l are defined with the bounds

$$0 < k, l < N - 1 \; . \tag{4.1}$$

Now $\mathbf{k}$ can be redefined with k and l as

$$\mathbf{k} = \left(\frac{2\pi k - \pi N}{L}, \frac{2\pi l - \pi N}{L} \right) \; . \tag{4.2}$$

Since the Fourier components $\tilde{h}(\mathbf{k}, t)$ are the spectral components of $h(\mathbf{x}, t)$, the inverse FFT (IFFT) must be applied on $\tilde{h}(\mathbf{k}, t)$ to obtain the ocean height field as the height amplitudes of the spatial domain. [3, p. 6]

Consequently, the IFFT-equation as defined in [12, p. 272] is denoted by

$$h(n, m, t) = \frac{1}{N \cdot N} \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} \tilde{h}(k, l, t) \exp\left(i \frac{2\pi kn - \pi Nn}{N} \right) \exp\left(i \frac{2\pi lm - \pi Nm}{N} \right) \; . \tag{4.3}$$

With $e^{i\pi} = -1$ the equation can be simplified to

$$h(n, m, t) = \frac{1}{N \cdot N} \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} \tilde{h}(k, l, t)(-1)^n \exp\left(i \frac{2\pi nk}{N} \right) (-1)^m \exp\left(i \frac{2\pi ml}{N} \right) \; . \tag{4.4}$$

By rearranging (4.4) the following 2D-IFFT equation is obtained:

$$h(n, m, t) = \frac{1}{N \cdot N} (-1)^n \sum_{k=0}^{N-1} \left[(-1)^m \sum_{l=0}^{N-1} \tilde{h}(k, l, t) \exp\left(i \frac{2\pi ml}{N} \right) \right] \exp\left(i \frac{2\pi nk}{N} \right) \; . \tag{4.5}$$

4.2 Computing the IFFT on GPU

This section presents the GPGPU implementation of the 2D-IFFT computation with *OpenGL* Compute Shaders. Compute Shaders were introduced in *OpenGL 4.3* and are intended for computing data on GPU rather than drawing into the framebuffer[1]. The processing sequence of the Ocean simulation consists of various GPGPU stages and a final rendering stage. 2D textures with 32 bit RGBA format serve as data buffers for the initial, intermediate and output stages of the FFT. Overall five textures are used as data buffers during the FFT execution. The role of the different textures used for data storage are explained below. The output of the FFT resp. the ocean height field is written into a further texture.
Section 4.2.1 gives an overview over the implementation of the IFFT algorithm. The following sections starting from 4.2.2 give a deeper insight into implementation details.

4.2.1 IFFT Algorithm

The IFFT implementation consists of five phases. The initial phase produces the spectrum textures containing the $\tilde{h}_0(\mathbf{k})$ and $\tilde{h}_0(-\mathbf{k})$ values. Phase two generates the texture holding the height amplitude Fourier components $\tilde{h}(\mathbf{k}, t)$.
The third phase executes N horizontal 1D FFT's with the Fourier components texture as input vectors. Each row of the input texture serves as an input vector for the 1D FFT's. The output texture of the horizontal 1D FFT's is the input for the next phase.
In the fourth phase N vertical 1D FFT's are executed with the output from phase three as input vectors. Each column of the input texture serves as an input vector for the 1D FFT's. The final phase multiplies the amplitudes by $(-1)^m$ and $(-1)^n$ and inverts the result with a mulitiplication by $1/N^2$.

```
Initial Spectrum h̃0(k) and h̃0(−k) shaderpass;
while isRunning do
    Fourier components h̃(k, t) shaderpass;
    pingpong := 0;
    for i=0 to i<log2N do
        horizontal butterfly shaderpass for stage i;
        pingpong := pingpong++ mod 2;
    end
    for i=0 to i<log2N do
        vertical butterfly shaderpass for stage i;
        pingpong := pingpong++ mod 2;
    end
    Inversion and permutation shaderpass;
end
```

Algorithm 1: 2D-IFFT Algorithm

[1]The framebuffer is the data buffer for the pixels of the device display

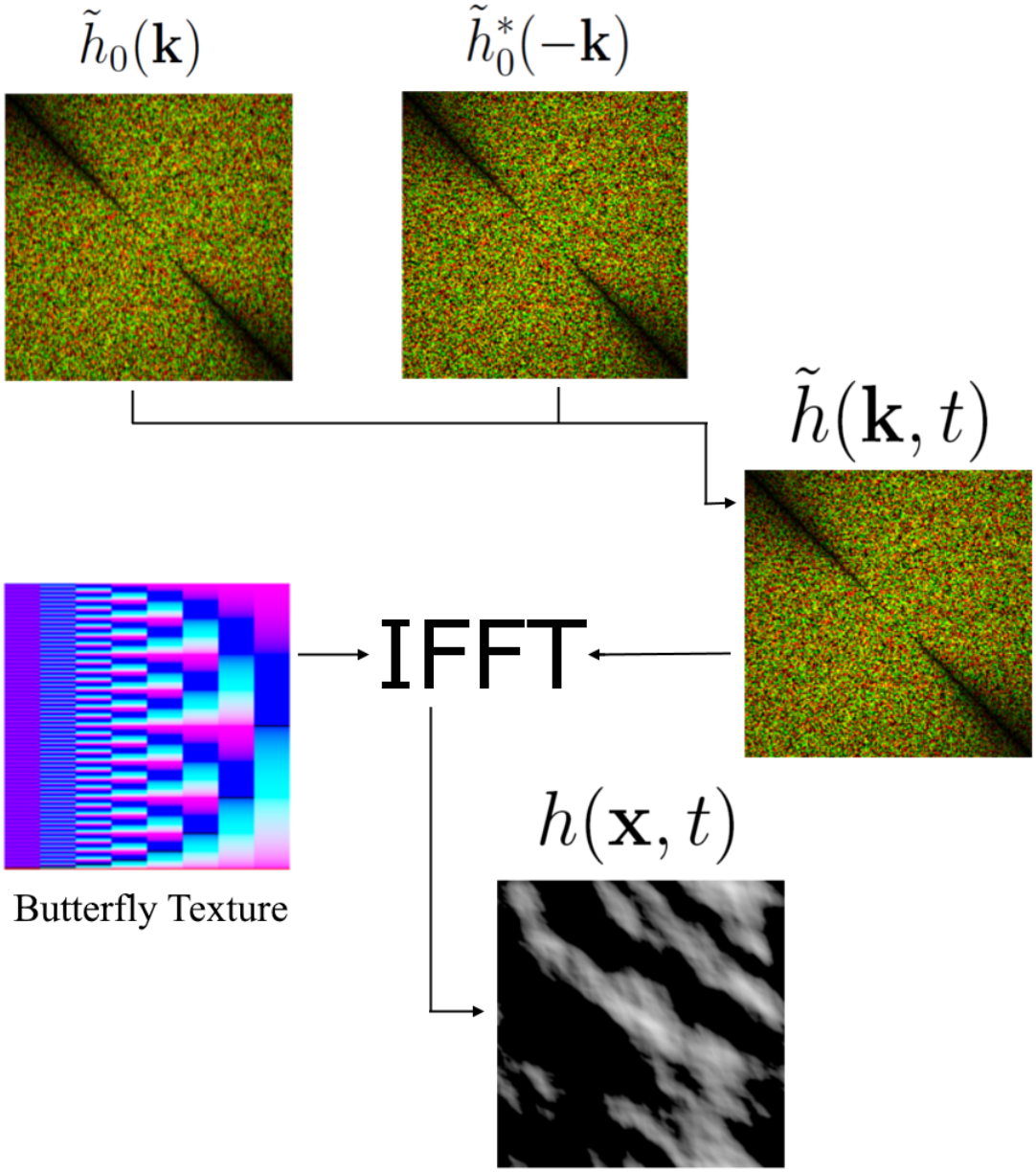

Figure 4.1: Two textures holding the values of $\tilde{h}_0(\mathbf{k})$ and $\tilde{h}_0(-\mathbf{k})$ and a further texture (butterfly texture) holding informations to perform the butterfly operations are precomputed and stored in the GPU memory. At each time t the texture holding $\tilde{h}(\mathbf{k},t)$ is updated. The IFFT procedure fetches all required data from the precomputed butterfly texture and the texture with $\tilde{h}(\mathbf{k},t)$ values. The ouput is a greyscale image containing the ocean height field.

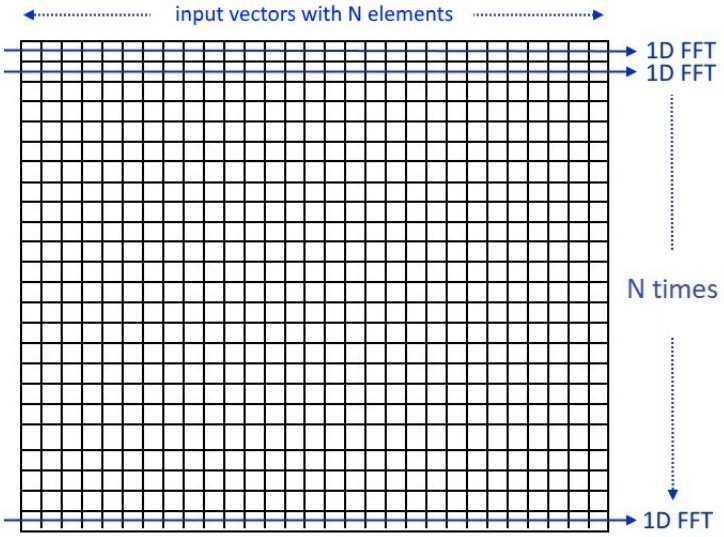

Figure 4.2: Illustration how the horizontal 1D-FFT's are performed on the texture pixels. Each pixel row serves as input vector for the FFT stages.

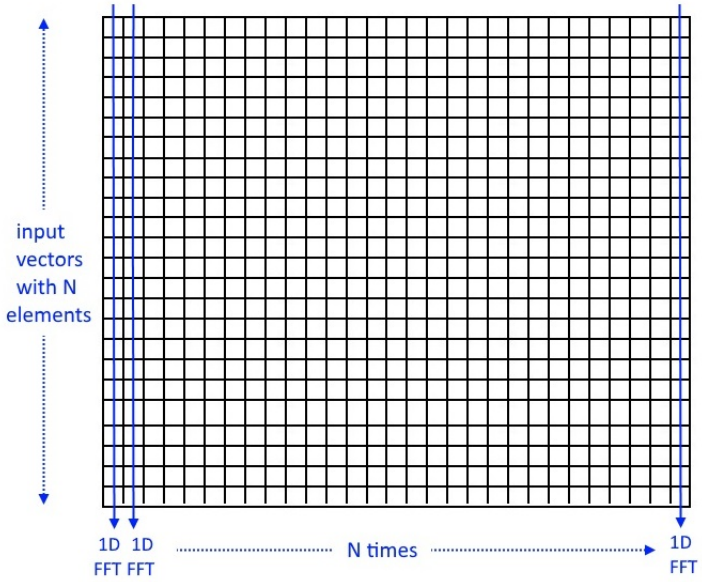

Figure 4.3: Illustration how the vertical 1D-FFT's are performed on the texture pixels. Each pixel column serves as input vector for the FFT stages.

4.2.2 Compute Space

The compute space consists of a three dimensional space of work groups. Each work group itself has a three dimensional invocation space. The dimensions of the work group space with their invocation spaces must be specified by the user. Each invocation within the different work groups holds a uniquely defined three dimensional index vector, the *gl_GlobalInvocationID*. In the IFFT implementation the compute space is arranged in a way that each work group invocation computes exactly one pixel of the output texture. There are various possibilities to arrange the work groups.
For example, with $N = 256$ a work group space with dimension $(1 \times 1 \times 1)$ with invocation space $(256 \times 256 \times 1)$ can be specified. Another option would be a work group space with dimension $(16 \times 16 \times 1)$ and invocation space of $(16 \times 16 \times 1)$. For very large N it should be considered that only the indiviual invocations within one work group are executed in parallel. Figure 4.4 illustrates the work group space with it's invocation spaces. [14, pp. 625-626]

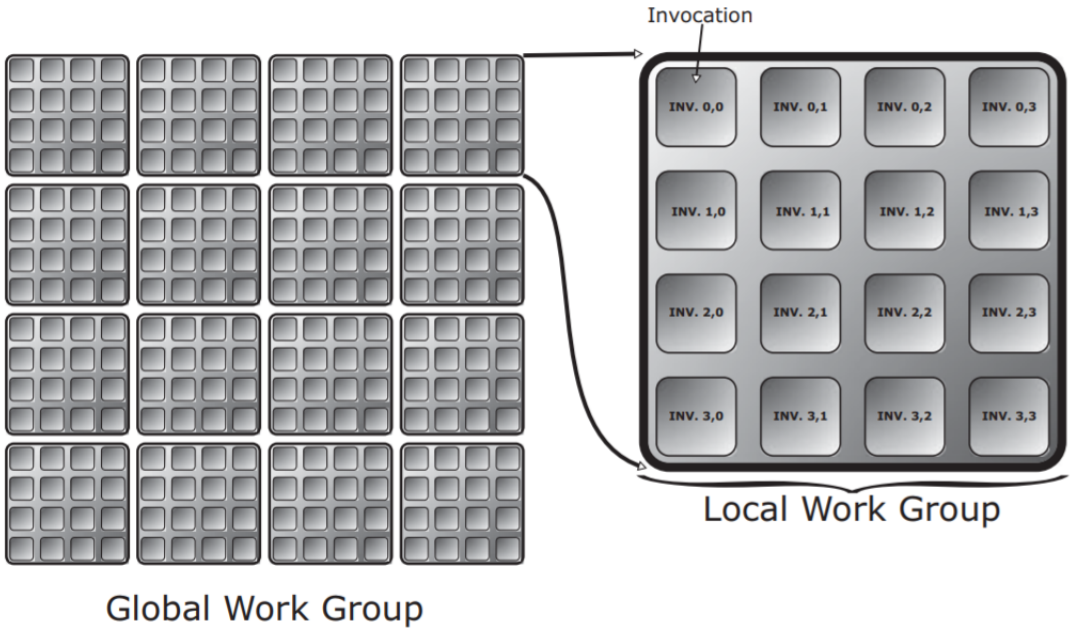

Figure 4.4: Illustration of a compute space consisting of 16 resp. $(4 \times 4 \times 1)$ work groups with 16 resp. $(4 \times 4 \times 1)$ invocation each, which results in a 16×16 Compute Space with overall 256 invocations. [14, p. 626]

4.2.3 $\tilde{h}_0(\mathbf{k})$ and $\tilde{h}_0(-\mathbf{k})$ Spectrum Textures

One texture is needed for storing the initial spectrum from (3.6). The gaussian random numbers ξ_r and ξ_i are generated from two independent noise textures with the Box-Muller method. The real part of $\tilde{h}_0(\mathbf{k})$ is stored in the red channel while the imaginary part is stored in the green channel. A further texture is used to store $\tilde{h}_0^*(-\mathbf{k})$ with gaussian random numbers ξ_r and ξ_i generated from two further noise textures. Figure 4.5 shows the rendered $\tilde{h}_0(\mathbf{k})$ spectrum texture . The texture with $\tilde{h}_0^*(-\mathbf{k})$ looks similar.

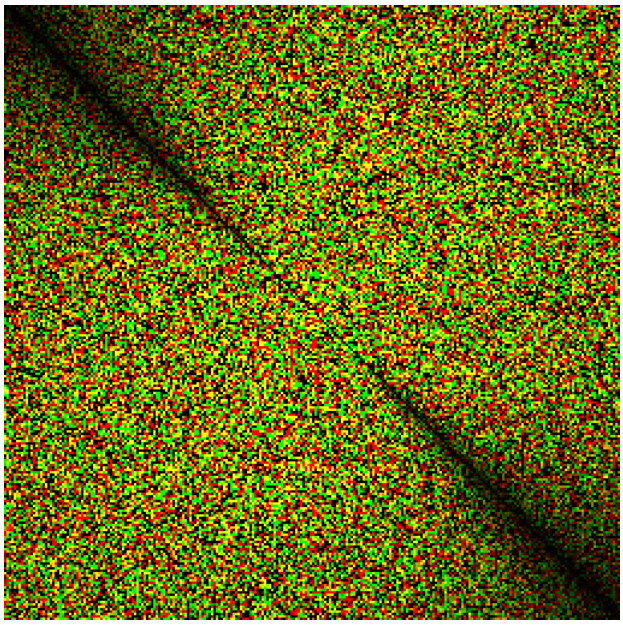

Figure 4.5: $\tilde{h}_0(\mathbf{k})$ texture
Rendered image of the initial spectrum $\tilde{h}_0(\mathbf{k})$ texture with the setting:
$N = 256, L = 1000, l = 0.5, w = (1, 1), V = 40, A = 4$

4.2.4 $\tilde{h}(\mathbf{k}, t)$ Fourier Components Texture

The time dependent $\tilde{h}(\mathbf{k}, t)$ Fourier components from equation (3.8) are stored in another texture. Again the real part is written in the red channel and the imaginary part is stored in the green channel. The time dependent Fourier components texture looks similar to figure 4.5.

4.2.5 Ping-Pong Texture

The ping-pong texture is used for storing the output of the FFT computation stages. Actually two ping-pong textures are needed, but the Fourier components texture from the previous section 4.2.4 serves as the second ping-pong texture after the first FFT stage. The role of each ping-pong texture switches from input to output respectively after each stage.
In the first stage the Fourier components texture serves as the input samples while the output samples are stored in the ping-pong texture. In the second stage the role of the textures are swapped. Now the ping-pong texture containing the output data of the first stage serves now as the input for the second stage, while the other ping-pong texture resp. the input texture of the first stage serves as the output buffer for the second stage.

4.2.6 Butterfly Texture

The butterfly texture stores all needed information to process the butterfly computations at the FFT stages. For the butterfly informations all four color channels of the texture are used. Unlike the previous textures, the size of the butterfly texture is $N \times log_2\, N$, since the horizontal dimension of the texture indicates the current FFT stage (out of $log_2\, N$ stages), while the vertical dimension indicates the index of the butterfly output value within the FFT stages.
The butterfly texture provides all necessary information for all FFT stages to perform the required butterfly operations. The red (real part) and green (imaginary part) channels holds the twiddle factor of the related butterfly. The input sample indices are stored in the blue and alpha channel. Figures 4.6 and 4.7 show the stored information related to a simple butterfly operation.

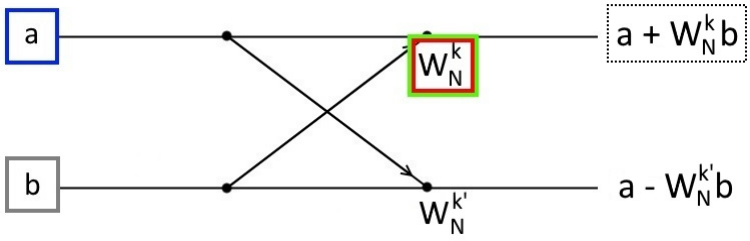

Figure 4.6: In roder to perform the top wing butterfly operation to obtain $a+W_N^k b$, the twiddle factor is read from red and green channel while the input sample indices are read from the blue and alpha channel. The blue channel holds the index of the top input sample and the alpha channel holds the index of the bottom input sample.

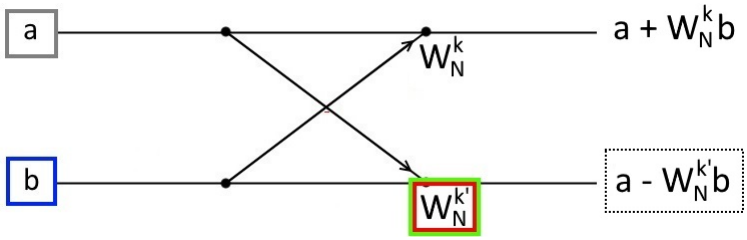

Figure 4.7: In roder to perform the bottom wing butterfly operation to obtain $a - W_N^k b$, the twiddle factor is read from red and green channel while the input sample indices are read from the blue and alpha channels. The blue channel holds the index of the bottom input sample and the alpha channel holds the index of the top input sample.

The butterfly texture is generated by computing the information data for each output sample resp. each pixel dependent on the three variables N, the horizontal pixel coordinate (the FFT stage) and the vertical pixel coordinate (the output sample index). With pixel coordinates (x, y) the twiddle factor W_N^k can be computed by just figure out the exponent k with

$$k = \frac{y \cdot N}{2^{x+1}} \; mod \; N \; , \tag{4.6}$$

since W_N is constant.
The butterfly span is determined with 2^x. If the inequality

$$y \; mod \; 2^{x+1} < 2^x \tag{4.7}$$

evaluates to true, the current coordinate is the output of a top butterfly wing, otherwise bottom. With these information the second input sample index (stored in the alpha-channel) can be determined by adding the butterfly span to the current y-value of the current pixel coordinate if the inequality (4.7) evaluates to true, otherwise the butterfly span will be substracted. The following image shows a rendered butterfly texture for $N = 256$:

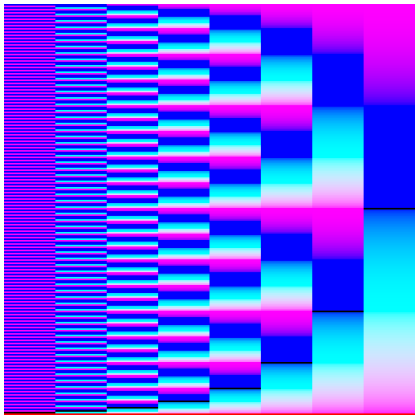

Figure 4.8: Butterfly texture for $N = 256$ with resolution 256×8. For better visualization the the texture is streched onto a quad.

4.2.7 Butterfly Shader

The Butterfly Compute-Shader is the core of the FFT implementation. The butterfly shader is used for both horizontal and vertical 1D-FFT executions. The uniform variable *direction* indicates whether horizontal or vertical butterfly operations are perfomed. Two further uniform variables *stage* and *pingpong* are used to determine the current stage of the FFT execution and which pingpong texture serves as input and output buffer. Depending on *direction*, *stage* and *pingpong*, the specific butterfly operations are executed. Listing 4.1 shows the GLSL code of the horizontal butterfly operation.

```
1  // fetch butterfly information data
2  vec4 data = imageLoad(butterflyTexture,
3                        ivec2(stage, coord.x)).rgba;
4  // fetch first butterfly input sample
5  vec2 p_ = imageLoad(pingpong0,
6                      ivec2(data.z, coord.y)).rg;
7  // fetch second butterfly input sample
8  vec2 q_ = imageLoad(pingpong0,
9                      ivec2(data.w, coord.y)).rg;
10 // create complex numbers and Twiddle factor
11 complex p = complex(p_.x,p_.y);
12 complex q = complex(q_.x,q_.y);
13 complex w = complex(data.x,data.y);
14
15 // perform butterfly operation
16 H = add(p,mul(w,q));
```

Listing 4.1: Horizontal Butterfly Operation

The uniform variable *stage* indicates which horizontal coordinate is read from the butterfly texture. The 2-dimensional vector *coord* specifies the global index of the compute space invocation resp. the *gl_GlobalInvocationID*. As indicated in 4.2.2 each compute shader invocation writes to exactly one pixel in the output image buffer. Hence, the index of the output sample value is equivalent to the *gl_GlobalInvocationID*. The butterfly information data is fetched from the butterfly texture in line two. The first parameter of *imageLoad* specifies the texture name, the second parameter specifies the pixel coordinates. As mentioned in 4.2.6 the horizontal space of the butterfly texture represents the FFT stage. Hence, the horizontal coordinate of the fecthed pixel is the *stage* uniform variable. Since listing 4.1 shows a horizontal butterfly operation the vertical coordinate of the fetched pixel is the x-value resp. the horizontal value of the *coord* vector.
The input sample values for the butterfly operation are fetched in line 5 and 8 from the first pinpong texture. Depending on the uniform *pingpong* the first or second pingpong texture is used to read the input values. The butterfly texture stores the index of the top and bottom butterfly input sample index in the blue and alpha channel. Therefore the x-coordinate of the fetched pixels are the z- and w-values from the fetched butterfly information data respectively. Accordingly the y-coordinate of the fetched pixel is the

vertical coordinate of the *coord* vector. The necessary complex numbers are initialized in lines 11, 12 and 13 in order to perform the butterfly operation in line 16.
Listing 4.2 shows the similar GLSL code of the vertical butterfly operation. The only differences are the coordinates of the fetched pixels in line 2,5 and 8. Due to vertical 1D FFT's, the y-coordinate of the fetched pixel from the butterfly texture is the y-value of the *coord* vector. Further the fetched input samples lie on the same pixel column of the texture. Accordingly the x-coordinates of the fetched pixel are the x-value of the *coord* vector and the y-coordinates are the z- and w-values from the fetched butterfly information data.

```
1  // fetch butterfly information data
2  vec4 data = imageLoad(butterflyTexture,
3                        ivec2(stage, coord.y)).rgba;
4  // fetch first butterfly input sample
5  vec2 p_ = imageLoad(pingpong0,
6                      ivec2(coord.x, data.z)).rg;
7  // fetch second butterfly input sample
8  vec2 q_ = imageLoad(pingpong0,
9                      ivec2(coord.x, data.w)).rg;
10 // create complex numbers and Twiddle factor
11 complex p = complex(p_.x,p_.y);
12 complex q = complex(q_.x,q_.y);
13 complex w = complex(data.x,data.y);
14
15 // perform butterfly operation
16 H = add(p,mul(w,q));
```

Listing 4.2: Vertical Butterfly Operation

4.2.8 The Ocean Height Field $h(\mathbf{x}, t)$

The result of the 2D-IFFT is the greyscale ocean wave height field shown in figure 4.9. Brighter means higher and darker means lower height while negative height is represented in black, since the color values of rendered images ranges only from 0 to 1.0. By a more accurate consideration of the height field, it does not look really "wavy". To align the waves more with the wind direction, the factor $|\hat{k} \cdot \hat{w}|^2$ from equation (3.5) can be changed to $|\hat{k} \cdot \hat{w}|^k$ for any arbitrary positive k. The higher the exponent k in $|\hat{k} \cdot \hat{w}|^k$ will be set, the more the waves will be aligned with the wind direction. [3, p. 7]

Figure 4.10 shows the greyscale ocean wave height field with the factor $|\hat{k} \cdot \hat{w}|^8$. The initial spectrum texture from 4.2.3 with $|\hat{k} \cdot \hat{w}|^8$ is shown in figure 4.11.

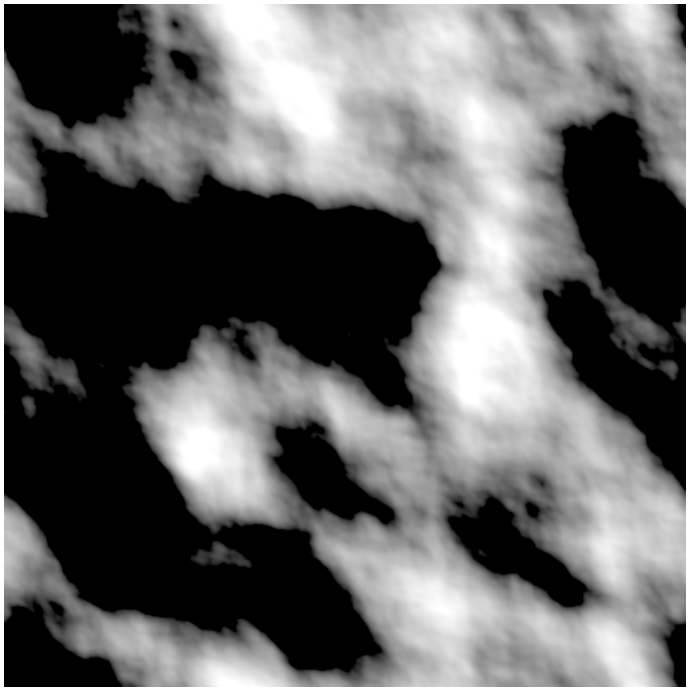

Figure 4.9: Ocean wave height field with $|\hat{k} \cdot \hat{w}|^2$ factor.

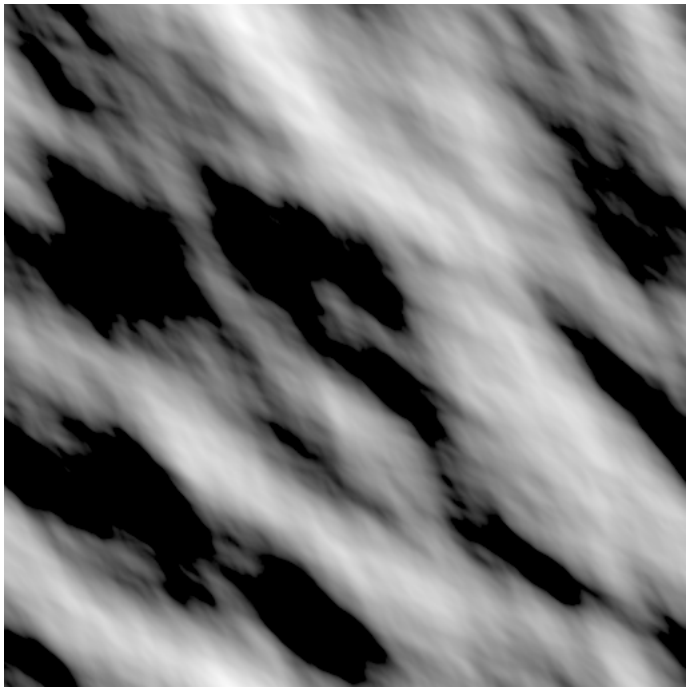

Figure 4.10: Ocean wave height field with $|\hat{k} \cdot \hat{w}|^8$ factor.

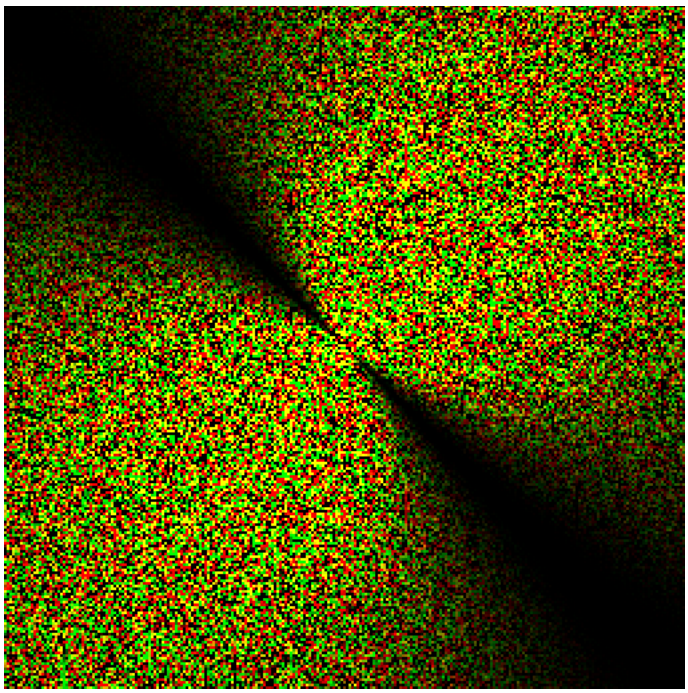

Figure 4.11: $\tilde{h}_0(\mathbf{k})$ texture with $|\hat{k} \cdot \hat{w}|^8$ factor.

4.2.9 Choppy Waves

Up to this point one last thing needs to be solved to let the ocean surface appear more realistic. Using the wave height field from figure 4.10 for displacement of the y-component generates ocean waves with rounded peaks as illustrated in figure 4.12. Since naturally ocean waves have sharp wave peaks even with a weak wind, the ocean surface in figure 4.12 does not look realistic.

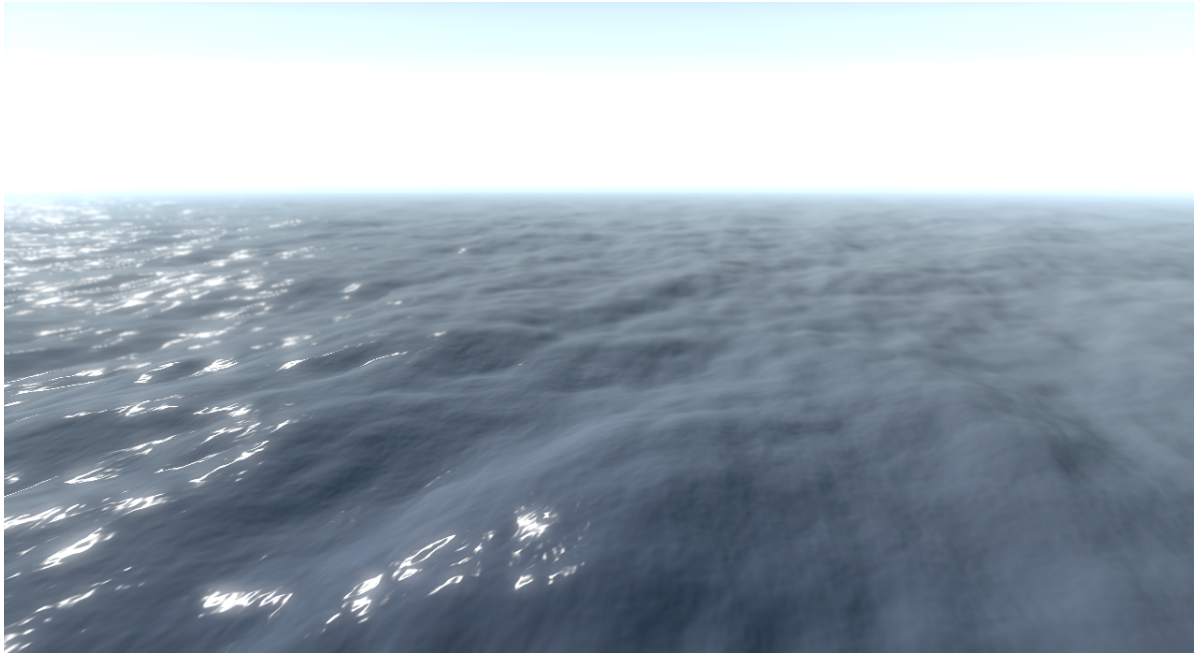

Figure 4.12: Rendered ocean surface without choppy waves.

To solve this issue a horizontal displacement in both x- and z-direction needs to be applied besides the displacement of the y-component. Hence, two further 2D-IFFT's are performed to generate the height fields for the horizontal displacement. For the horizontal height field generation the following Fourier amplitudes as defined in [3, p. 10] are used:

$$\mathbf{D}(\mathbf{x}, t) = \sum_{\mathbf{k}} -i\frac{\mathbf{k}}{k}\tilde{h}(\mathbf{k}, t)\exp(i\mathbf{k}\cdot\mathbf{x}) \ . \tag{4.8}$$

The only thing which needs to be adjusted compared to the vertical height field generation, is to multiply the elements of the height amplitude Fourier components $\tilde{h}(\mathbf{k}, t)$ by the factor

$$-i\frac{\mathbf{k}}{k} \ . \tag{4.9}$$

Figure 4.13 and 4.14 are the corresponding choppy waves textures to figure 4.10. Figure 4.15 demonstrates a way more realistic appearing rendered ocean surface with height and choppy displacement.

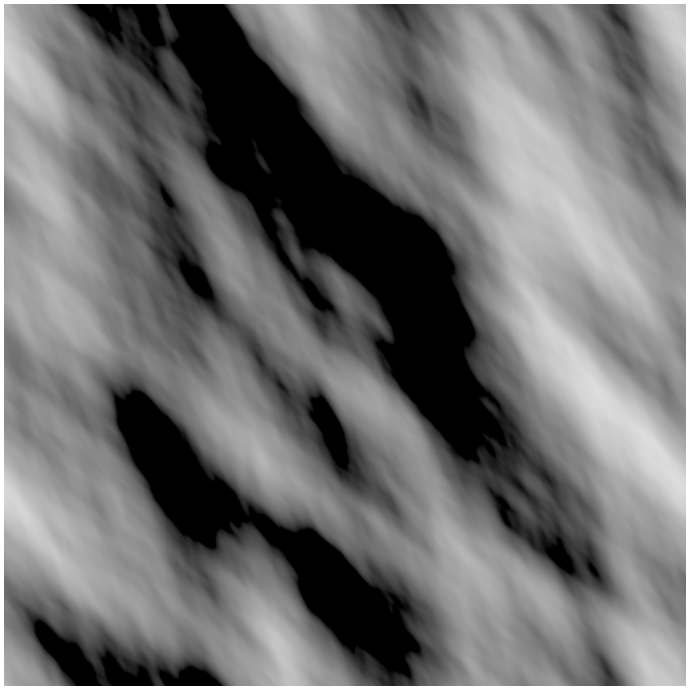

Figure 4.13: Horizontal choppy wave height field of the x-component.

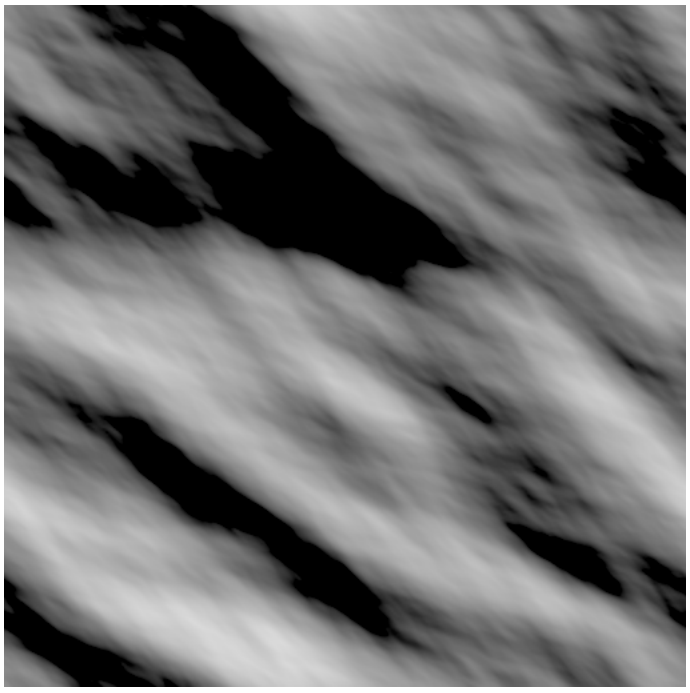

Figure 4.14: Horizontal choppy wave height field of the z-component.

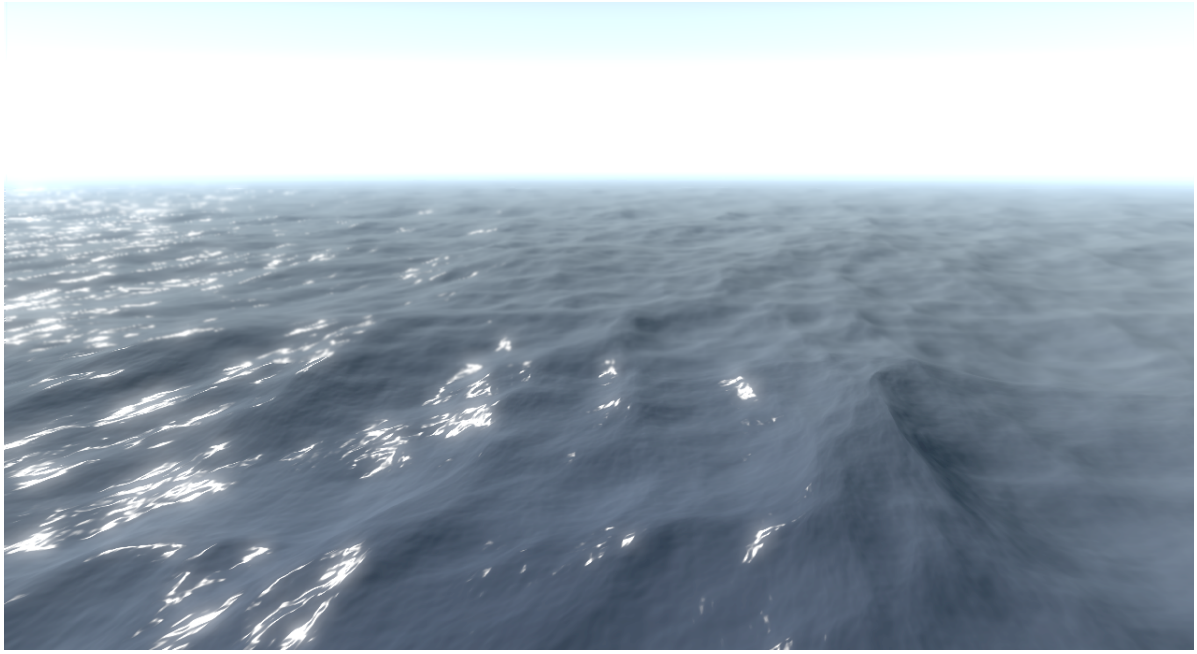

Figure 4.15: Rendered ocean surface with choppy waves.

Chapter 5

Outlook

With the 2D-IFFT implementation not only dynamic ocean surfaces can be generated, but also huge terrain landscapes. By superimposing many precomputed IFFT-generated height fields with different scalings a natural terrain can be produced.
Very huge and detailed terrain landscapes can be rendered in realtime by using an efficient Quadtree algorithm and the IFFT generated height fields. Screenshots of rendered realtime landscapes with different height field generations are shown in figures 5.1 and 5.2.

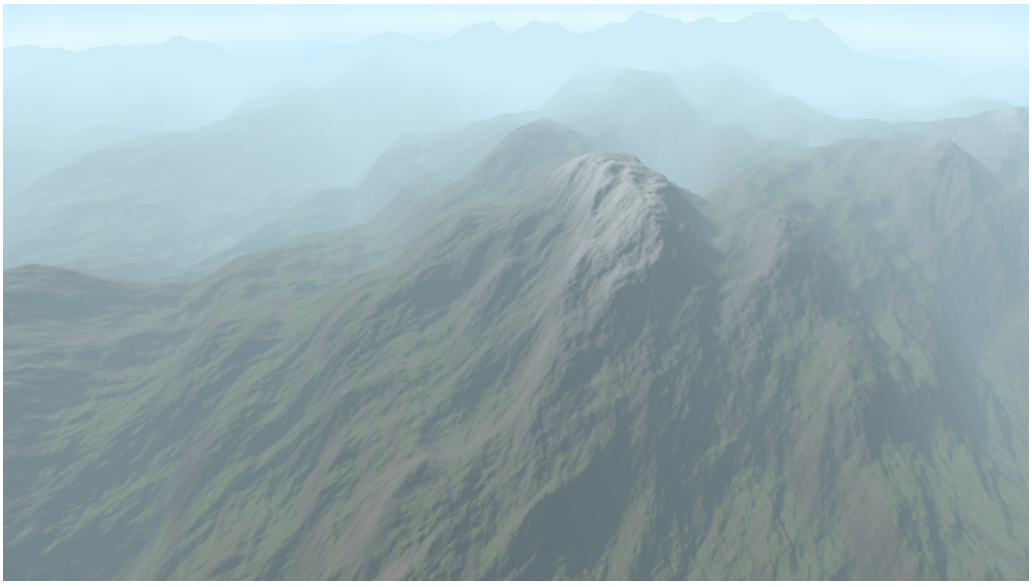

Figure 5.1: FFT generated landscape A

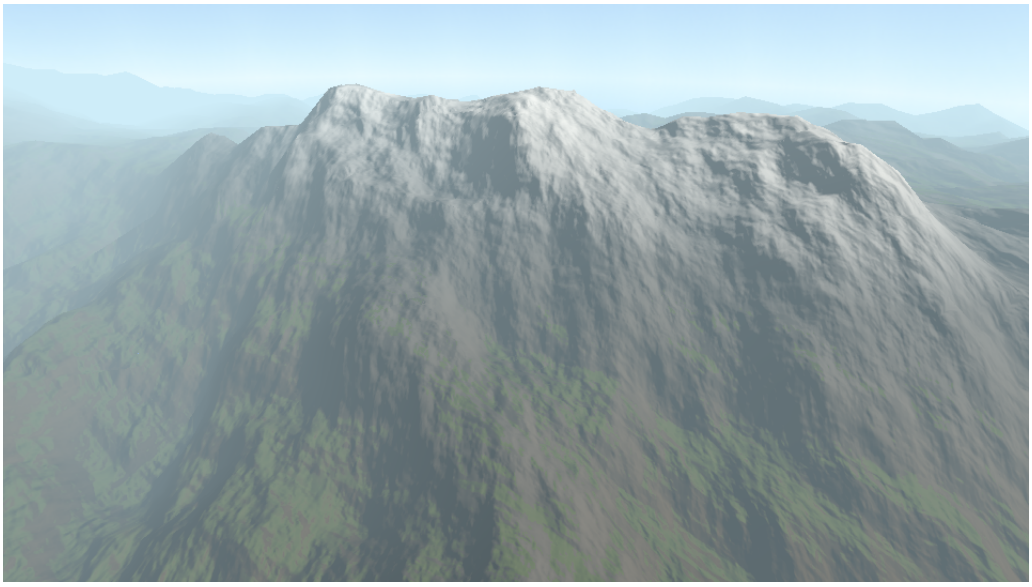

Figure 5.2: FFT generated landscape B

Appendix A

$\tilde{h}_0(\mathbf{k})$ and $\tilde{h}_0(-\mathbf{k})$ Compute Shader

```
#version 430 core
#define M_PI 3.14159265358979323384626433832795

layout (local_size_x = 16, local_size_y = 16) in;

layout (binding = 0, rgba32f) writeonly uniform image2D tilde_h0k;

layout (binding = 1, rgba32f) writeonly uniform image2D tilde_h0minusk;

uniform sampler2D noise_r0;
uniform sampler2D noise_i0;
uniform sampler2D noise_r1;
uniform sampler2D noise_i1;

uniform int N;
uniform int L;
uniform float A;
uniform vec2 windDirection;
uniform float windspeed;

const float g = 9.81;

// Box-Muller-Method
vec4 gaussRND()
{
        vec2 texCoord = vec2(gl_GlobalInvocationID.xy)/float(N);

        float noise00 = clamp(texture(noise_r0, texCoord).r, 0.001, 1.0);
        float noise01 = clamp(texture(noise_i0, texCoord).r, 0.001, 1.0);
        float noise02 = clamp(texture(noise_r1, texCoord).r, 0.001, 1.0);
        float noise03 = clamp(texture(noise_i1, texCoord).r, 0.001, 1.0);

        float u0 = 2.0*M_PI*noise00;
        float v0 = sqrt(-2.0 * log(noise01));
        float u1 = 2.0*M_PI*noise02;
        float v1 = sqrt(-2.0 * log(noise03));

        vec4 rnd = vec4(v0 * cos(u0), v0 * sin(u0), v1 * cos(u1), v1 * sin(u1));

        return rnd;
}


void main(void)
{
        vec2 x = vec2(gl_GlobalInvocationID.xy) - float(N)/2.0;

        vec2 k = vec2(2.0 * M_PI * x.x/L, 2.0 * M_PI * x.y/L);

        float L_ = (windspeed * windspeed)/g;
        float mag = length(k);
        if (mag < 0.00001) mag = 0.00001;
        float magSq = mag * mag;

        //sqrt(Ph(k))/sqrt(2)
        float h0k = clamp(sqrt((A/(magSq*magSq))
        * pow(dot(normalize(k), normalize(windDirection)), 6.0)
        * exp(-(1.0/(magSq * L_ * L_)))
        * exp(-magSq*pow(L/2000.0,2.0)))/ sqrt(2.0), -4000, 4000);

        //sqrt(Ph(-k))/sqrt(2)
        float h0minusk = clamp(sqrt((A/(magSq*magSq))
        * pow(dot(normalize(-k), normalize(windDirection)), 6.0)
        * exp(-(1.0/(magSq * L_ * L_)))
        * exp(-magSq*pow(L/2000.0,2.0)))/ sqrt(2.0), -4000, 4000);

        vec4 gauss_random = gaussRND();
```

```
        imageStore(tilde_h0k, ivec2(gl_GlobalInvocationID.xy),
                                vec4(gauss_random.xy * h0k, 0, 1));

        imageStore(tilde_h0minusk, ivec2(gl_GlobalInvocationID.xy),
                                vec4(gauss_random.zw * h0minusk, 0, 1));
}
```

Listing A.1: $\tilde{h}_0(\mathbf{k})$ and $\tilde{h}_0(-\mathbf{k})$ Compute Shader

Appendix B

Fourier Components $\tilde{h}(\mathbf{k}, t)$ Compute Shader

```
#version 430 core
#define M_PI 3.1415926535897932384626433832795

layout (local_size_x = 16, local_size_y = 16) in;

layout (binding = 0, rgba32f) writeonly uniform image2D tilde_hkt_dy;

layout (binding = 1, rgba32f) writeonly uniform image2D tilde_hkt_dx;

layout (binding = 2, rgba32f) writeonly uniform image2D tilde_hkt_dz;

layout (binding = 3, rgba32f) readonly uniform image2D tilde_h0k;

layout (binding = 4, rgba32f) readonly uniform image2D tilde_h0minusk;

uniform int N;
uniform int L;
uniform float t;

struct complex
{
        float real;
        float im;
};

complex mul(complex c0, complex c1)
{
        complex c;
        c.real = c0.real * c1.real - c0.im * c1.im;
        c.im   = c0.real * c1.im + c0.im * c1.real;
        return c;
}

complex add(complex c0, complex c1)
{
        complex c;
        c.real = c0.real + c1.real;
        c.im   = c0.im   + c1.im;
        return c;
}

complex conj(complex c)
{
        complex c_conj = complex(c.real, -c.im);

        return c_conj;
}

void main(void)
{
        vec2 x = ivec2(gl_GlobalInvocationID.xy) - float(N)/2.0;

        vec2 k = vec2(2.0 * M_PI * x.x/L, 2.0 * M_PI * x.y/L);

        float magnitude = length(k);
        if (magnitude < 0.00001) magnitude = 0.00001;

        float w = sqrt(9.81 * magnitude);

        vec2 tilde_h0k_values = imageLoad(tilde_h0k,
                                          ivec2(gl_GlobalInvocationID.xy)).rg;

        complex fourier_cmp = complex(tilde_h0k_values.x, tilde_h0k_values.y);
```

```
        vec2 tilde_h0minusk_values = imageLoad(tilde_h0minusk,
                                               ivec2(gl_GlobalInvocationID.xy)).rg;

        complex fourier_cmp_conj = conj(complex(tilde_h0minusk_values.x,
                                                tilde_h0minusk_values.y));

        float cos_w_t = cos(w*t);
        float sin_w_t  = sin(w*t);

        // euler formula
        complex exp_iwt = complex(cos_w_t, sin_w_t);
        complex exp_iwt_inv = complex(cos_w_t, -sin_w_t);

        // dy
        complex h_k_t_dy = add(mul(fourier_amp, exp_iwt),
                               mul(fourier_amp_conj, exp_iwt_inv));

        // dx
        complex dx = complex(0.0,-k.x/magnitude);
        complex h_k_t_dx = mul(dx, h_k_t_dy);

        // dz
        complex dy = complex(0.0,-k.y/magnitude);
        complex h_k_t_dz = mul(dy, h_k_t_dy);

        imageStore(tilde_hkt_dy, ivec2(gl_GlobalInvocationID.xy),
                                       vec4(h_k_t_dy.real, h_k_t_dy.im, 0, 1));
        imageStore(tilde_hkt_dx, ivec2(gl_GlobalInvocationID.xy),
                                       vec4(h_k_t_dx.real, h_k_t_dx.im, 0, 1));
        imageStore(tilde_hkt_dz, ivec2(gl_GlobalInvocationID.xy),
                                       vec4(h_k_t_dz.real, h_k_t_dz.im, 0, 1));
}
```

Listing B.1: Fourier Components $\tilde{h}(\mathbf{k}, t)$ Compute Shader

Appendix C

Butterfly Texture Compute Shader

```
#version 430 core
#define M_PI 3.14159265358979323384626433832795

layout (local_size_x = 1, local_size_y = 16) in;
layout (binding = 0, rgba32f) writeonly uniform image2D butterflyTexture;

layout (std430, binding = 1) buffer indices {
        int j[];
} bit_reversed;

struct complex
{
        float real;
        float im;
};

uniform int N;

void main(void)
{
        vec2 x = gl_GlobalInvocationID.xy;
        float k = mod(x.y * (float(N)/ pow(2,x.x+1)), N);
        complex twiddle = complex( cos(2.0*M_PI*k/float(N)), sin(2.0*M_PI*k/float(N)));

        int butterflyspan = int(pow(2, x.x));

        int butterflywing;

        if (mod(x.y, pow(2, x.x + 1)) < pow(2, x.x))
                butterflywing = 1;
        else butterflywing = 0;

        // first stage, bit reversed indices
        if (x.x == 0) {
                // top butterfly wing
                if (butterflywing == 1)
                        imageStore(butterflyTexture, ivec2(x),
                                        vec4(twiddle.real, twiddle.im,
                                                bit_reversed.j[int(x.y)],
                                                bit_reversed.j[int(x.y + 1)]));
                // bot butterfly wing
                else
                        imageStore(butterflyTexture, ivec2(x),
                                        vec4(twiddle.real, twiddle.im,
                                                bit_reversed.j[int(x.y - 1)],
                                                bit_reversed.j[int(x.y)]));
        }
        // second to log2(N) stage
        else {
                // top butterfly wing
                if (butterflywing == 1)
                        imageStore(butterflyTexture, ivec2(x),
                                        vec4(twiddle.real, twiddle.im,
                                                x.y, x.y + butterflyspan));
                // bot butterfly wing
                else
                        imageStore(butterflyTexture, ivec2(x),
                                        vec4(twiddle.real, twiddle.im,
                                                x.y - butterflyspan, x.y));
        }
}
```

Listing C.1: Butterfly Texture Compute Shader

Appendix D

Butterfly Compute Shader

```
#version 430 core
#define M_PI 3.1415926535897932384626433832795

layout (local_size_x = 16, local_size_y = 16) in;

layout (binding = 0, rgba32f) readonly uniform image2D butterflyTexture;

layout (binding = 1, rgba32f) uniform image2D pingpong0;

layout (binding = 2, rgba32f) uniform image2D pingpong1;

uniform int stage;
uniform int pingpong;
uniform int direction;

struct complex
{
        float real;
        float im;
};

complex mul(complex c0, complex c1)
{
        complex c;
        c.real = c0.real * c1.real - c0.im * c1.im;
        c.im   = c0.real * c1.im + c0.im * c1.real;
        return c;
}

complex add(complex c0, complex c1)
{
        complex c;
        c.real = c0.real + c1.real;
        c.im   = c0.im   + c1.im;
        return c;
}


void horizontalButterflies()
{
        complex H;
        ivec2 x = ivec2(gl_GlobalInvocationID.xy);

        if(pingpong == 0)
        {
                vec4 data = imageLoad(butterflyTexture, ivec2(stage, x.x)).rgba;
                vec2 p_ = imageLoad(pingpong0, ivec2(data.z, x.y)).rg;
                vec2 q_ = imageLoad(pingpong0, ivec2(data.w, x.y)).rg;
                vec2 w_ = vec2(data.x, data.y);

                complex p = complex(p_.x,p_.y);
                complex q = complex(q_.x,q_.y);
                complex w = complex(w_.x,w_.y);

                H = add(p,mul(w,q));

                imageStore(pingpong1, x, vec4(H.real, H.im, 0, 1));
        }
        else if(pingpong == 1)
        {
                vec4 data = imageLoad(butterflyTexture, ivec2(stage, x.x)).rgba;
                vec2 p_ = imageLoad(pingpong1, ivec2(data.z, x.y)).rg;
                vec2 q_ = imageLoad(pingpong1, ivec2(data.w, x.y)).rg;
                vec2 w_ = vec2(data.x, data.y);

                complex p = complex(p_.x,p_.y);
                complex q = complex(q_.x,q_.y);
                complex w = complex(w_.x,w_.y);
```

```
                H = add(p,mul(w,q));

                imageStore(pingpong0, x, vec4(H.real, H.im, 0, 1));
        }
}

void verticalButterflies()
{
        complex H;
        ivec2 x = ivec2(gl_GlobalInvocationID.xy);

        if(pingpong == 0)
        {
                vec4 data = imageLoad(butterflyTexture, ivec2(stage, x.y)).rgba;
                vec2 p_ = imageLoad(pingpong0, ivec2(x.x, data.z)).rg;
                vec2 q_ = imageLoad(pingpong0, ivec2(x.x, data.w)).rg;
                vec2 w_ = vec2(data.x, data.y);

                complex p = complex(p_.x,p_.y);
                complex q = complex(q_.x,q_.y);
                complex w = complex(w_.x,w_.y);

                H = add(p,mul(w,q));

                imageStore(pingpong1, x, vec4(H.real, H.im, 0, 1));
        }
        else if(pingpong == 1)
        {
                vec4 data = imageLoad(butterflyTexture, ivec2(stage, x.y)).rgba;
                vec2 p_ = imageLoad(pingpong1, ivec2(x.x, data.z)).rg;
                vec2 q_ = imageLoad(pingpong1, ivec2(x.x, data.w)).rg;
                vec2 w_ = vec2(data.x, data.y);

                complex p = complex(p_.x,p_.y);
                complex q = complex(q_.x,q_.y);
                complex w = complex(w_.x,w_.y);

                H = add(p,mul(w,q));

                imageStore(pingpong0, x, vec4(H.real, H.im, 0, 1));
        }
}

void main(void)
{
        if(direction == 0)
                horizontalButterflies();
        else if(direction == 1)
                verticalButterflies();
}
```

Listing D.1: Butterfly Compute Shader

Appendix E

Inversion and Permutation Compute Shader

```
#version 430 core

layout (local_size_x = 16, local_size_y = 16) in;

layout (binding = 0, rgba32f) writeonly uniform image2D displacement;

layout (binding = 1, rgba32f) readonly uniform image2D pingpong0;

layout (binding = 2, rgba32f) readonly uniform image2D pingpong1;

uniform int pingpong;
uniform int N;

void main(void)
{
        ivec2 x = ivec2(gl_GlobalInvocationID.xy);

        float perms[] = { 1.0, -1.0 };
        int index = int(mod((int(x.x + x.y)),2));
        float perm = perms[index];

        if(pingpong == 0)
        {
                float h = imageLoad(pingpong0, x).r;
                imageStore(displacement, x, vec4(perm*(h/float(N*N)),
                                                 perm*(h/float(N*N)),
                                                 perm*(h/float(N*N)),
                                                 1));
        }
        else if(pingpong == 1)
        {
                float h = imageLoad(pingpong1, x).r;
                imageStore(displacement, x, vec4(perm*(h/float(N*N)),
                                                 perm*(h/float(N*N)),
                                                 perm*(h/float(N*N)),
                                                 1));
        }
}
```

Listing E.1: Inversion and Permutation Compute Shader

Bibliography

[1] J. Dongarra and F. Sullivan. Guest editors introduction to the top 10 algorithms. *Computing in Science & Engineering*, 2(1):22–23, 2000.

[2] M. Heideman, D. Johnson, and C. Burrus. Gauss and the history of the fast fourier transform. *IEEE ASSP Magazine*, 1(4):14–21, 1984.

[3] Jerry Tessendorf. Simulating Ocean Water. 1999.

[4] Tiago Sousa. Crysis 2 & CryEngine 3. (http://crytek.com/assets/Crysis-2-Key-Rendering-Features.pdf, visited September 18, 2016).

[5] http://www.nvidia.com/content/gtc-2010/pdfs/2275_gtc2010.pdf, visited September 18, 2016.

[6] Mark Harris. A Brief History of GPGPU. http://cs.unc.edu/xcms/wpfiles/50th-symp/Harris.pdf, visited September 19, 2016.

[7] http://www.gamestar.de/artikel/nvidia_geforce_gtx_1080_ti,3310460.html, visited September 18, 2016.

[8] http://www.nvidia.com/object/what-is-gpu-computing.html, visited September 18, 2016.

[9] http://www.digitalstorm.com/unlocked/upcoming-nvidia-driver-to-\give-amd-mantle-a-run-for-its-money-massive-performance-boost-idnum209, visited September 18, 2016.

[10] https://github.com/oreonengine/oreon-engine.

[11] Brad Osgood. The Fourier Transform and its Applications. (https://see.stanford.edu/materials/lsoftaee261/book-fall-07.pdf, visited September 18, 2016), 2007.

[12] http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html, visited September 18, 2016.

[13] https://www.dsprelated.com/showcode/232.php, visited September 19, 2016.

[14] Dave Shreiner, Graham Sellers, John M. Kessenich, and Bill M. Licea-Kane. *OpenGL Programming Guide The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley, 8. edition, 2013.