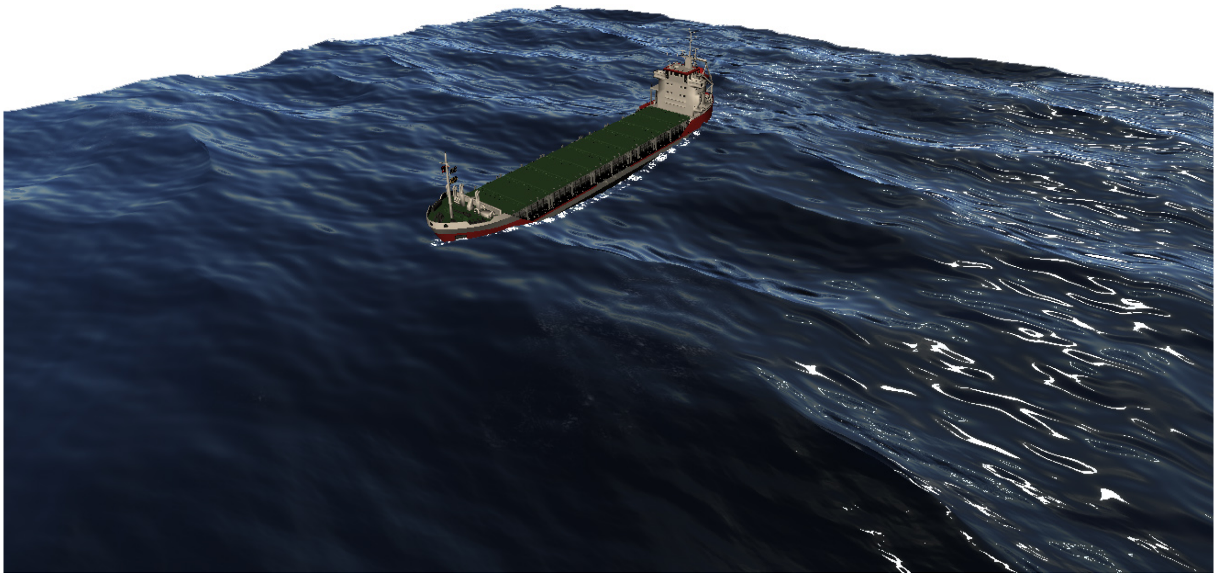
DEEP WATER OCEAN SURFACE MODELLING WITH SHIP SIMULATION

MASTER THESIS AT NTNU
MARINE TECHNOLOGY



Thorvald C. Grindstad and Runar J. F. Rasmussen

Supervisor Associate Professor Håvard Holm

ABSTRACT

The scope of this work is to implement a realistic looking ocean simulator with a ship floating on the surface. Foam, and ship wake is included to add to the realism. The application is designed to run on a normally equipped laptop with an ATI graphics card and Windows as the operation system. All calculations except the wake generation are performed in real time.

The program, named 'OceanSimulator', is written in C++ with OpenGL for visualization. The waves are represented with the JONSWAP spectrum and calculated using fast fourier transformation algorithm. The ship movement is calculated by use of transfer functions given in a RAO text file.

The graphics card is programmed directly by use of shaders to get high flexibility regarding visual effects. Two different lighting approaches of the ocean are explored, cube mapping, and parameterization.

Acknowledgements

First and foremost we would like to thank our supervisor Associate Professor Håvard Holm for help and guidance during our work. No matter when we came to your office with our problems, you always had time to help us out. We would also like to thank you for letting us work with what we really enjoy.

We wouldn't be doing this kind of work if it wasn't for PhD Stein Inge Dale. You introduced us to visualization and graphics programming with your course at NTNU that we both really enjoyed. We also got a lot of help on several occasions when meeting you and Sigurd Pettersen at Ceetron. We are very grateful that both of you would set aside time for us.

Thanks to Renato Skejic for being very helpful when learning us how calculate the wake from a ship.

Contents

Introduction

During the last decade, there has been a revolution in the use of graphics card. Tools for using and instructing the graphics card have improved a lot, at the same time as the performance drastically increases. Calculating and visualizing complex problems can now be done on a normal laptop, and that is the essence of the work presented in this report.

With overall increasing performance for computers, more advanced simulators are being developed each year. Marine and offshore industry sees the benefits this technology gives. Doing a simulation before performing the real operations can be beneficial. The crew that is supposed to carry out the work may need training, and doing the operations in a simulator can be a valuable exercise. The simulator provides a safe environment, and the tasks can be done over and over again. An example of this is the vessel winch simulator developed by Rolls Royce and Offshore Simulator Centre. This is the first anchor-handling simulator developed, and gives the crew a safe environment to practice. The simulator was developed as a result of the "Bourbon Dolphin" accident.

The goal for this work is to develop a simulator for the ocean sea, with a ship moving on the surface. The calculations are performed in real-time, on a laptop with sufficient powerful graphics card. The main concern with this work has been to make the program mathematically correct without compromising the frame rate. The sea wave pattern is calculated by integrating the JONSWAP spectrum to find the amplitude at distinct wave frequencies and directions. Then the inverse fast fourier transform is applied to calculate the ocean elevation. OpenGL code makes the basis for visualizing the result. The code is optimized with respect to common computer architecture to provide realistic visualization at a high frame rate. To get the required visual effects, shaders have been used to program the graphics card. Two different approaches regarding the visualization of the ocean surface have been tested. These two approaches are by use of a skybox, and by use of a parametrization. The ship movement is calculated by combining the sea wave pattern with a transfer function file. The wake generated by the ship is added based on the hull geometry.

Background Theory

The following chapter will explain theory and methods that have been used in the implementation of the simulator. This includes how to calculate the wave pattern on the ocean surface and ship response. A description for ship generated wake for calm water is also provided. The chapter explains how OpenGL operates, including an explanation of lighting and shaders.

1.1 Symbols

ζ_i:	wave height, one wave
ζ:	wave height, sum of all waves
$s(\omega)$:	energy curve for wave distribution
ω:	wave frequency
θ:	wave direction relative to main wave direction
E_i:	energy of single wave
ρ:	density water
g:	gravity
k:	wave number, determines the wave length
ϕ:	phase

1.2 Coordinate System

Three coordinate systems are used.

(x, y, z, t)	Cartesian coordinate system. Used to describe the water elevation on CPU
(x, z, y, t)	Cartesian coordinate system. Used to describe the water elevation on GPU
(ω, θ)	Frequency domain. For describing wave height or energy with respect to frequency and direction

Within marine engineering the z-axis is normally upward. However, OpenGL uses a right-handed coordinate system with the y-axis pointing upwards. Except the for switch of axis, the two coordinate systems are equal. Meaning that $(x, y, z, t)_{marine} = (-x, z, -y, t)_{opengl}$. The marine coordinate system is used when calculating the wave pattern and the surface elevation. The OpenGL coordinates are used when drawing objects and waves on to the screen.

1.3 Wave Conventions

This section explains the technique used to describe the wave pattern mathematically. The equations are found in Faltinsen (1990).

A simple sine shaped wave is written as:

$$\zeta_i(x,t) = A_i \cos(\omega t - kx + \phi) \tag{1.1}$$

For two dimensional waves, the equation is:

$$\zeta_i(x,y,t) = A_i \cos(\omega t - k_x x - k_z z) \tag{1.2}$$

$$k = \sqrt{k_x^2 + k_y^2} \tag{1.3}$$

In cylindrical coordinates:

$$\zeta_i = A_i \cos(\omega t - k\cos(\theta)x + k\sin(\theta)z) \tag{1.4}$$

The relationship between wave period and wave length:

$$\lambda = 1.56 * T^2 \tag{1.5}$$

Of which it follows that

$$omega = \sqrt{\frac{2\pi g}{\lambda}}; \tag{1.6}$$

1.3.1 Energy Distribution

Wave height and wave period for a given sea state is represented by an energy curve. The energy curve describes how much energy the sea state contains for different frequencies. The formulas can be found in Myrhaug (2007)

$$E_i = \frac{1}{2}\rho g \zeta_i^2 \tag{1.7}$$

$$E = \rho g \sum_{i=0}^{n} E_i \tag{1.8}$$

$$E = \rho g \int_0^\infty S(\omega)\,\mathrm{d}\omega \tag{1.9}$$

$$E_i = \rho g \int_{\omega_{i-0.5}}^{\omega_{i+0.5}} S(\omega)\,\mathrm{d}\omega \tag{1.10}$$

Combining the first and last equation, the wave height for one wave can be represented by:

$$\zeta_i = \sqrt{2\int_{\omega_{i-0.5}}^{\omega_{i+0.5}} S(\omega)\,\mathrm{d}\omega} \tag{1.11}$$

$$\zeta_i \;= \sqrt{2S(\omega_i)\,\Delta\omega} \tag{1.12}$$

The equations above can only represent waves coming from one direction. A new parameter, θ, describing direction is included. By multiplying the equation above with cos^{2s} the equation will be able to represent waves from multiple directions. 'S' is a positive integer, describing the spread of the incoming waves. High value of 'S' gives low spread.

$$\zeta_i = \sqrt{2S(\omega_i)\,\frac{\cos^{2s}(\theta_j)}{C_1}\Delta\omega\Delta\theta} \tag{1.13}$$

$$C_1 = \int_{-0.5\pi}^{0.5\pi} \cos^{2s}(\theta)\,\mathrm{d}\theta \tag{1.14}$$

By summing up all the individual waves, the total surface elevation is given by:

$$\zeta(x,z,t) = \sum_{i=0}^{n} \zeta_i(x,z,t) \tag{1.15}$$

1.3.2 Fast Fourier transformation

Fast Fourier Transformation (FFT) is an algorithm for solving Discrete Fourier Transformation (DFT). DFT is used to transform a function from regular cartesian coordinates into the frequency domain. From (1.13) the wave height in frequency domain can be calculated. Inverse FFT is used to transfer from frequency to cartesian coordinates. The cartesian coordinates can then be transferred to the graphic card. Further explanation of the FFT algorithm is found in Kreyszig (2006).

The mathematical definition of the inverse fast fourier transform is:

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N}kn} \tag{1.16}$$

For 2 dimensional waves the inverse FFT is:

$$x_{n_1,n_2} = \frac{1}{N}\sum_{j=0}^{N-1} X_{j,k} \sum_{k=0}^{N-1} e^{\frac{2\pi i}{N}(jn_1+kn_2)} \tag{1.17}$$

From the mathematical definition above, it follows that the input argument for an inverse FFT algorithm is a matrix containing complex number. Each element in this matrix is describing one single wave. Each wave has a unique combination of wave length and direction. For time state 't', the input argument will be described as:

$$input(i,j) = \zeta_{i,j}e^{\phi+\omega t+k_x x_0+k_y y_0} \tag{1.18}$$

Let n_1 and n_2 denote the row and column number in the input matrix. Let L be the physical length of the domain. Then the total wave length of a single wave is given by (1.19).

$$\lambda = \frac{L}{\sqrt{n_1^2 + n_2^2}} \tag{1.19}$$

$$\theta = \arctan(\frac{n1}{n2}) \tag{1.20}$$

From wave theory, a single sine wave can be writen as:

$$\zeta(x,t) = Real(\zeta_0 e^{\phi+\omega t}) = Real(\zeta_0(\cos(\phi + \omega t0) + i\sin(\phi + \omega t)) \tag{1.21}$$

The last equation is directly applicable as input argument to the inverse FFT algorithm. The input argument for the inverse FFT algorithm is then calculated by dividing the sea state into a predetermined set of frequencies and directions. The next section describes how the amplitude of the different single waves are estimated.

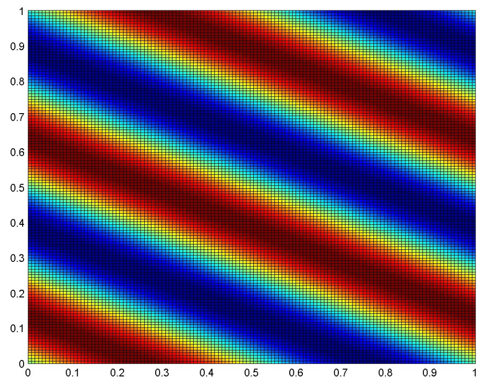


Figure 1.1: The real result of an inverse fast fourier transform. Element number (3, 2) in input matrix is given value 1. All other elements in matrix are given value 0.

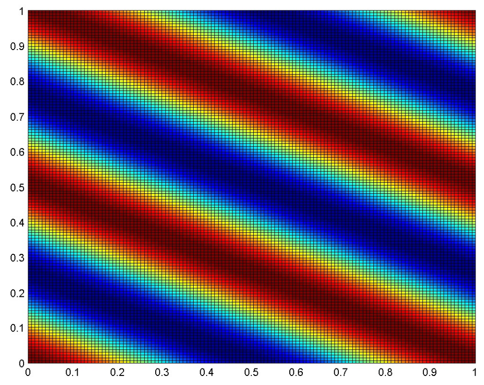


Figure 1.2: The imaginary part of an inverse fast fourier transform. Element number (3, 2) in input matrix is given value 1. All other elements in matrix are given value 0.

1.3.3 Integration

Using inverse FFT algorithm, only a predetermined set of individual waves are available. Figure 1.3 displays all available combinations of frequencies and directions for a 64 x 64 system.

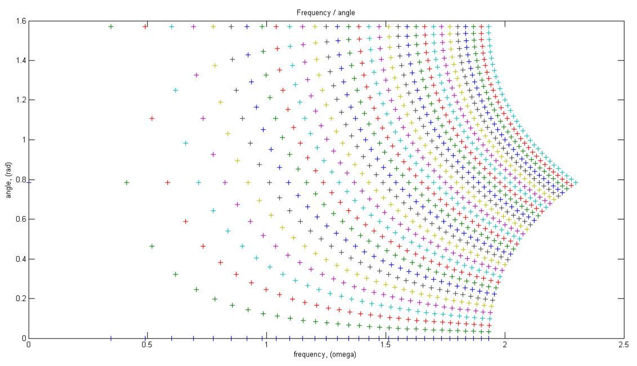


Figure 1.3: Plot of possible angles and frequencies on a 64 x 64 inverse FFT transformation

The wave height in absolute value for each unique wave, is given by:

$$H(i,j) = S(\omega(i,j), \theta(i,j))dA \tag{1.22}$$

$S(\omega(i,j), \theta(i,j))$ is the wave spectral function, which is described in previous chapter. dA is the area surrounding the wave number i, j. The value of dA is found by calculating the vectors spanning from the given node, to its surrounding nodes. Then taking 1/4 of the cross product of these vectors.

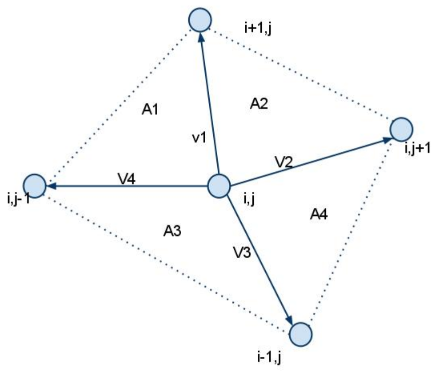


Figure 1.4: Node (i, j) in frequency domain

The area surrounding node i, j can be described as:

$$dA = \frac{1}{2}(A1 + A2 + A3 + A4) \tag{1.23}$$

$$dA = \frac{1}{4}(|V1 \times V2| + |V2 \times V3| + |V3 \times V4| + |V4 \times V1|) \tag{1.24}$$

PROOF

This paragraph proofs that the integration technique explained in section 1.3.3 is valid.

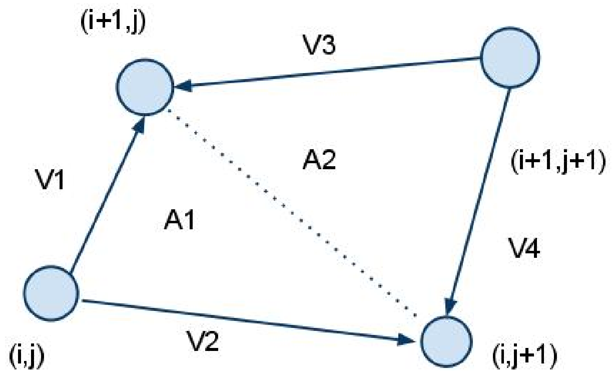


Figure 1.5: Node (i, j) in frequency domain

ion is valid.

The area of the figure above is
$A = A1 + A2$
$A = \frac{1}{2}|V1 \times V2| + \frac{1}{2}|V3 \times V4|$
$\frac{1}{2}A = \frac{1}{2 \cdot 2}|V1 \times V2| + \frac{1}{2 \cdot 2}|V3 \times V4|$

which proofs that the area belonging to two diametral nodes sums up to be $\frac{1}{2}$ of the total area of the figure. The sum of the area belonging to all the nodes therefore has to be equal to the total area of the figure.

1.4 Ship Motion by Use of Transfer Functions

The wave elevation is calculated by superposition of a series of simple sine shaped waves. Chapter 1.3 explains how the single waves are calculated. The movement of the ship is known for a single sine shaped wave. Adding together the movement of the ship for each single wave will give a good estimate of the ship response. How the ship reacts to each wave is calculated by a transfer function. A transfer function is a function which operates in the frequency domain. If the input value is a simple sine shape, the function value is the movement of the ship. A transfer function can be written as:

$$y(s) = h(s)u(s) \tag{1.25}$$

For the purpose of this report. A transfer function can be described as

$$y(\omega) = h(\omega)\zeta(\omega) \tag{1.26}$$

$\zeta(\omega)$: input signal
$h(\omega)$: transfer function
$y(\omega)$: response
s: frequency
ω: frequency
All values are of complex numbers.

$$y(\omega) = |y(\omega)|\, e^{\phi_y i} \tag{1.27}$$

$$h(\omega) = |h(\omega)|\, e^{\phi_h i} \tag{1.28}$$

$$\zeta(\omega) = |\zeta(\omega)|\, e^{i(\omega t + k_x(\omega)x_0 + k_y(\omega)y_0)} \tag{1.29}$$

$\zeta(\omega)$ is the wave height, $h(\omega)$ is the transfer function. x_0 and y_0 is denoting the center of the ship. For this case the transfer function is a complex number given by an amplitude and a phase. The real value of $y(\omega)$ is the response on ship. One transfer function is given for each of the 6 DOF.

1.5 Wake

The wake is calculated based on the method described by E.O. Tuck and Scullen (1999). A more detailed explanation of the mathematics is found in Faltinsen (2005). The method calculates the surface elevation for a ship with constant speed on flat water. The method is based on potential theory.

$$\zeta(x, y) = real(\int_{-pi/2}^{pi/2} A(\theta)e^{-i\omega(\theta)}d\theta \tag{1.30}$$

$$\omega(\theta) = k(\theta)[x\cos(\theta) + y\sin(\theta)] \tag{1.31}$$

$$k = \frac{g}{u^2}\frac{1}{\cos(\theta)^2} \tag{1.32}$$

$$A(\theta) = \frac{2}{\pi}k(\theta)^2 \iint Y(x, z)exp(k(\theta)Z + ik(\theta))dxdz \tag{1.33}$$

(1.30) describes the surface elevation. (1.33) describes the wave amplitude with respect to frequency. (1.31) describes the heading of the wave with respect to frequency. Y(x, z) is half the beam of the ship at position (x,z), below water line. z is assumed to be negative below water line.

The integral has to be computed numerically. An analytic solution of the $A(\theta)$ integral exists only for idealised system, and is difficult to compute. However, for a ship model consisting of flat planes, an analytical solution can be found for $A(\theta)$.

(1.33) is becoming extremely oscillating when θ approaches $\pm\frac{\pi}{2}$. Solving the integral of $A(\theta)$ will therefore require a very fine discretization. The number of elements needed is in the matter of thousands. Computing the wake is therefore time consuming. $A(\theta)$ is plotted for a ship of length 320 meter in Figure 1.6.

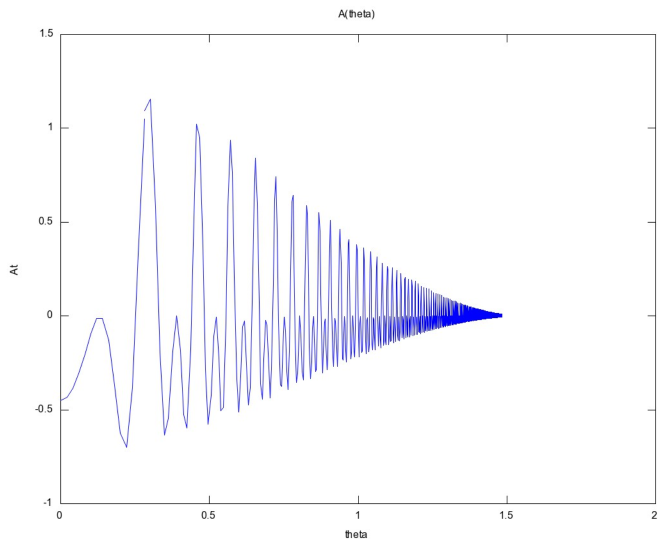


Figure 1.6: $A(\theta)$ for a 320 meter long ship with speed 3 m/s

The graph below shows the $A(\theta)$ function for a ship with length 64 meter. The speed is 3 m/s, which is the same as the graph above. Notice the longer oscillations on the function in Figure 1.7 compared to the graph in Figure 1.6 above.

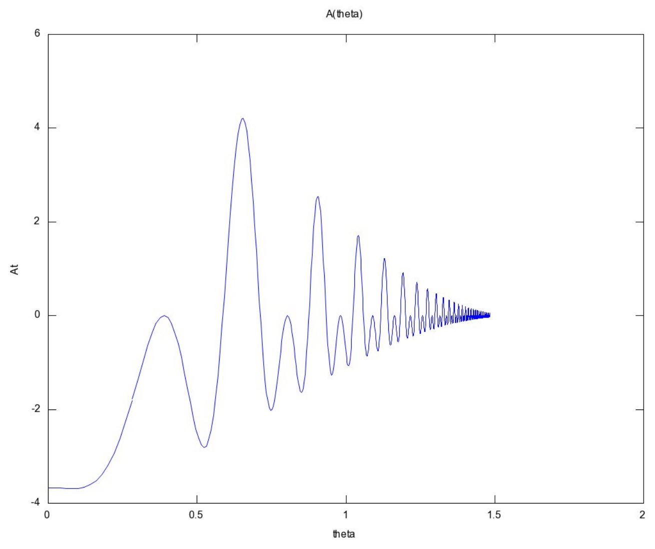


Figure 1.7: $A(\theta)$ for a 64 meter long ship with speed 3 m/s

By choosing the values of θ carefully, inverse fast fourier transformation can be applied. The constraint is that the total domain length is a multiple of the wave length given by the $\omega(\theta)$ value, (1.31). The constraint has to be for filled for only one dimension. The equations below shows how the θ values are chosen. For the fourier transformation the expression $e^{-i\omega(\theta)}$ has to correspond to a wave length of $\frac{L_d}{i}$. Where i is an integer with value less than the vector length used. L_d is the domain length. The equation below show how the θ value is defined.

$$2\pi i = L_d \cos(\theta) k(\theta) \tag{1.34}$$

$$2\pi i = L_d \frac{k0}{\cos(\theta)} \tag{1.35}$$

$$\theta_i = \arccos(\frac{L_d k0}{2\pi i}) \tag{1.36}$$

By using inverse FFT (IFFT) the computation time will be $O(N_x, N_y, N_\theta) = N_y N_\theta \log(N_\theta)$ provided that $N_x \leq N_\theta$. N_x and N_y denotes the grid size while N_θ denote number of sampling points. By not using IFFT the computational time will be $O(N_x, N_y, N_\theta) = N_x N_y N_\theta$. When N is in the size of thousands, the difference in computation time becomes significant.

IFFT is not able to cover the whole specter of θ. The largest θ value possible to represent is $\theta_n = \arccos(\frac{L_d k0}{2\pi N})$. The smallest value of θ is limited by $\Delta\theta$. The IFFT method is therefore unable to capture the upper and lower range of θ. Alternative integration technique has to be implemented for these ranges.

Since the $A(\theta)$ function is highly oscillatory it is practical to keep the $\Delta\theta$ below a fraction of the length of an oscillation when doing numerical integration. In Figure 1.8 a graph displaying the $\Delta\theta$ value for a 320 m ship with speed of 6 knot is shown. The Figure shows two plots. One viewing the $\Delta\theta$ using IFFT. The other one shows the limit when 10 sampling point per cycle is required. The graph shows that IFFT keeps the $\Delta\theta$ is within the margin. If the ship would be smaller, the oscillations would be smaller. Thus resulting in fewer sampling points required.

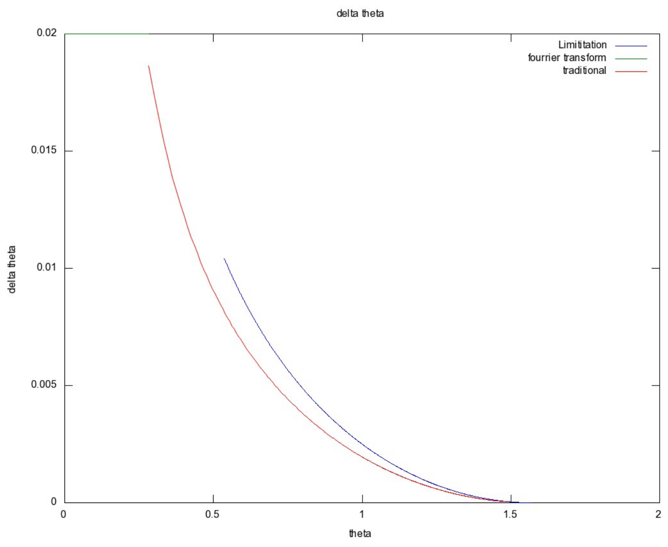


Figure 1.8: Computational domain for wake

1.6 Graphics Card Programming with OpenGL

When developing an application for displaying 3D computer graphics, both the CPU (processor) and the GPU (graphics card) have to work together. Traditional programming languages like C++ and fortran are only applicable to the CPU. Therefore a new type of programming language has to be used on the GPU. There are mainly two different programming languages available for the GPU, DirectX and OpenGL. Both DirectX and OpenGL work as a set of commands which is sent from the CPU to the GPU. These commands include information of the objects we want to display. The GPU then organizes the commands and generates the final picture displayed.

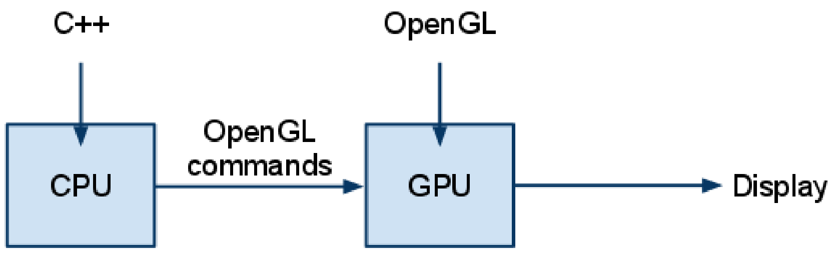


Figure 1.9: The graphics device (GPU) is controlled by CPU by use of OpenGL instructions

OpenGL is an open source API and works on most operation systems. This includes Microsoft, Apple and most unix platforms. DirectX is restricted to only operate on Microsoft platforms, including Windows OS and Xbox. 'Ocean Simulator' is based on OpenGL due to its open source concept.

1.6.1 Shaders

This section describes how OpenGL operates in its most basic form. A more comprehensive explanation is given in Appendix A.

A graphics card uses points, lines and planes to describe 3D objects. A 3D object will therefore consist of a series of polygons and lines in a 3D space. The polygons and lines are called primitives. The objective of the CPU is to send the primitives to the GPU. The CPU also sends information in terms of textures. The textures are then mapped on one or more primitives. A texture is a vector or a matrix consisting of number. A texture can represent a picture or a data set.

OpenGL has options for programming directly on the GPU, called shader programming. By using shaders, each polygon or line can be manipulated individually. There are two types of shaders, called vertex shader and fragment shader. The vertex shader has ability to change the position of the primitives. The vertex shader can also send single values to the fragment shader. The values are then linear interpolated by the fragment shader. The fragment processor determines how the color changes over each polygon.

When using shaders, there is no small limitation to what data can be sent to the GPU. The data is sent in terms of textures or single values like float. These data are called uniforms, and will not be changed by the shaders. Instead the shader uses the uniform values to change the vertex and fragment values. The picture below describes how the GPU is programmed.

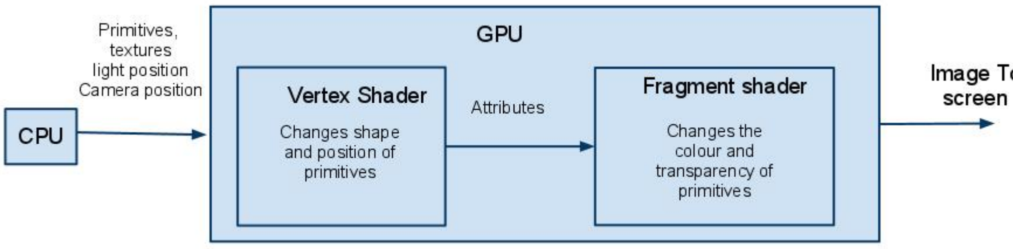

Primitives, textures light position Camera position
CPU
GPU
Vertex Shader
Changes shape and position of primitives
Attributes
Fragment shader
Changes the colour and transparency of primitives
Image To screen

Figure 1.10: Programming the GPU can be devided into vertex and fragment manipulation

1.7 Colors and Lighting

On a computer screen, each pixel emits different amounts of red, green and blue light. These are called the R, G and B values. Mixing different intensities of the parameters gives all kinds of colors. The R, G, and B values can range from 0.0 (none) to 1.0 (full intensity). For example, R = 1.0, G = 0.0 and B = 0.0 gives the brightest possible red. If all values are zero, the pixel is black, and if all values are 1.0, the pixel is white.

The final color of an object depends on what lighting is used in the scene, and how the object in the scene reflects or absorbs the light. The formula below shows how the reflected color is calculated:

$$color_{surrounding} = [r_s, g_s, b_s] \tag{1.37}$$

$$color_{object} = [r_o, g_o, b_o] \tag{1.38}$$

$$color_{reflected} = [r_s \cdot r_o, g_s \cdot g_o, b_s \cdot b_o] \tag{1.39}$$

The direction of the light reflected depends on the property of the object material. Normally we divide the reflection into specular and diffuse. By specular means shiny, metal like reflection, while diffuse reflection spreads the light equally in all directions. Diffuse reflection is not dependent on the angle of the incoming light. The difference between specular and diffuse reflection is further explained in Appendix B.

In addition to light directly reflected from a light source, light reflected more than once is also simulated. This kind of reflection is called ambient light. Assuming that the concentration of ambient light is equal everywhere, the ambient color setting can be calculated with

$$color_{ambient} = k_a \cdot color_{object} \tag{1.40}$$

where k_a is the ambient constant.

In this work, two approaches have been explored regarding lighting, *Cube mapping* and *parameterization*. With the cube mapping approach, images taken from outdoor environment is used for reflection, see Appendix C. The parameterization approach uses mathematical formulas to calculate the reflected color.

Implementation

2.1 Overview

The main goal is to make a realistic representation of the ocean wave pattern, capable of running on a laptop with a decent graphics card. There are several ways to implement a program like this regarding workload balance, each with their own strengths and weaknesses. With the hardware available today there are roughly two main strategies to choose amongst. The first strategy is to let the CPU do all the calculations regarding the wave field, and let the GPU take care of the rendering with the requested effects like lightning and reflections. This requires communication and data transfers between the CPU and the graphics device. Thanks to new powerful GPUs it is also possible to do all the calculations on the device itself. The strength of this approach is that there is no need to transfer any data between the CPU and the GPU, thus removing the transfer rate bottleneck. Compared to the CPU, the GPU has an enormous capacity. As a result, computing the water elevation on the GPU will be less time consuming. But the algorithms are more complicated to implement. With a correct setup of the first strategy described, the bandwidth bottleneck is not a compromising factor for the performance, and it does not require the latest graphics card. In addition, the domain size of the FFT calculations is relatively small so the computational time of this domain is small compared to the total time needed to draw one frame. Testing on two computers, the computation time for FFT was 4% and 7% of total computation time. It is therefore not much performance to gain by doing the computations on the GPU. The cpu is therefore chosen to perform this work.

2.1.1 Libraries

The application itself is written in C++, and uses OpenGL for rendering. In addition to self produced code, two external libraries have been used. One for the wave calculations, and one for loading 3D models. The fourier transformations used in the wave calculations are implemented with the Fastest Fourier Transform in the West (FFTW) library *FFTW* (2011). To load the ship model itself the library Open Asset Import (ASSIMP) is used.

2.1.2 Application Design

Figure 2.1 gives an overview over the most important classes, attributes and methods. The class *Main* takes care of setting up the windows environment, with OpenGL as graphical

API. It also holds the ocean related classes needed for the simulation and visualization. The *Camera* class uses mouse and keyboard input in order for the user to move around in the 3D environment. The wave calculations, which give us the heightmap and normal maps are implemented in the class *WaveHeight*. *ModelObject* is a class used to load 3D models, in this case the ship model. Ship movement according to the ocean waves are implemented in *Transfer* which reads a transfer file containing transfer data, and calculates the movement in all the six degrees of freedom. The *Wake* class calculates the wake with different vessel velocities based on the hull of the ship. To add realism to the scene, *Foam* is added when certain criteria are met.

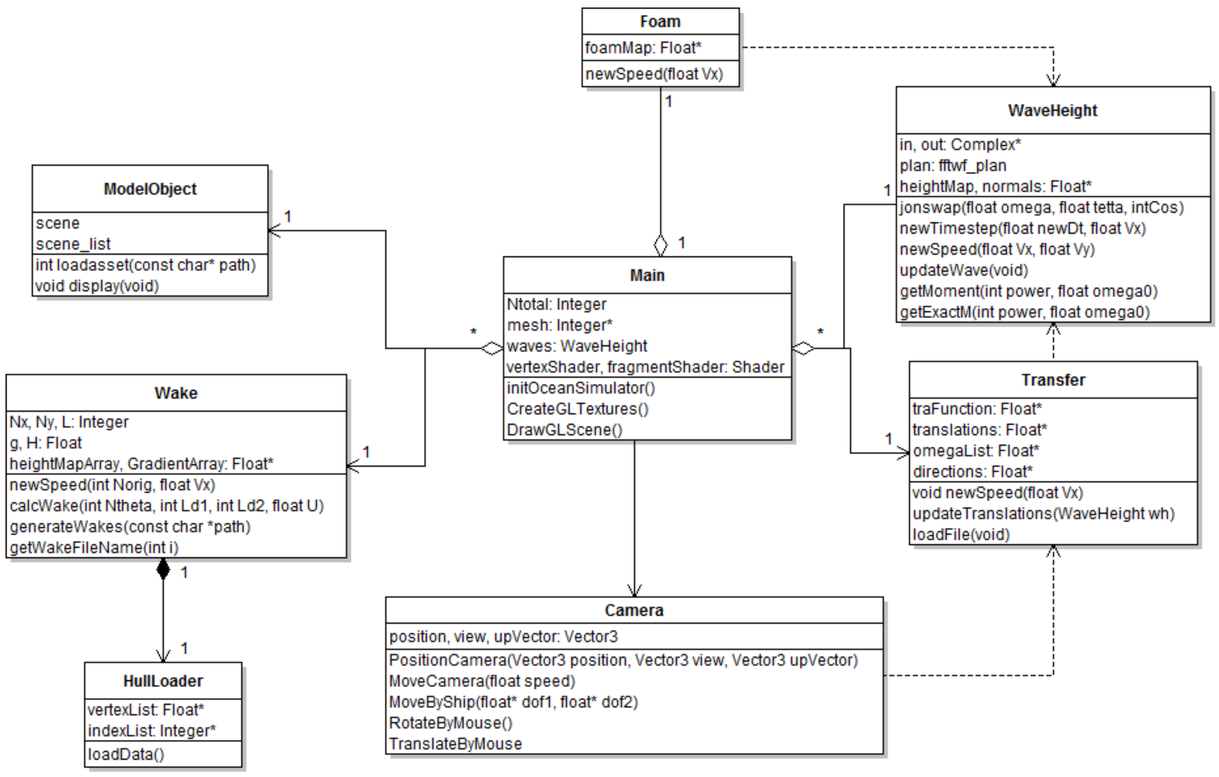

Foam
foamMap: Float*
newSpeed(float Vx)
WaveHeight
in, out: Complex*
plan: fftwf_plan
heightMap, normals: Float*
jonswap(float omega, float tetta, intCos)
newTimestep(float newDt, float Vx)
newSpeed(float Vx, float Vy)
updateWave(void)
getMoment(int power, float omega0)
getExactM(int power, float omega0)
ModelObject
scene
scene_list
int loadasset(const char* path)
void display(void)
Main
Ntotal: Integer
mesh: Integer*
waves: WaveHeight
vertexShader, fragmentShader: Shader
initOceanSimulator()
CreateGLTextures()
DrawGLScene()
Wake
Nx, Ny, L: Integer
g, H: Float
heightMapArray, GradientArray: Float*
newSpeed(int Norig, float Vx)
calcWake(int Ntheta, int Ld1, int Ld2, float U)
generateWakes(const char *path)
getWakeFileName(int i)
Transfer
traFunction: Float*
translations: Float*
omegaList: Float*
directions: Float*
void newSpeed(float Vx)
updateTranslations(WaveHeight wh)
loadFile(void)
Camera
position, view, upVector: Vector3
PositionCamera(Vector3 position, Vector3 view, Vector3 upVector)
MoveCamera(float speed)
MoveByShip(float* dof1, float* dof2)
RotateByMouse()
TranslateByMouse
HullLoader
vertexList: Float*
indexList: Integer*
loadData()
1
1
1
*
1
1
*
1
1
1

Figure 2.1: Class diagram for OceanSimulator

Figure 2.2 shows the initialization stage of the program. All classes are instantiated with correct parameters and are ready for further usage while running in the main loop. Three *WaveHeight* objects are instantiated with parameters regarding domain, height, spread and period. The 3D environment is configured with OpenGL, and the viewpoint (camera) is given a starting position and a view vector. The 3D model of the ship gets loaded and placed in the center of the scene. Setting up buffers for data transfer between the CPU and GPU is the last stage before the program is ready to run.

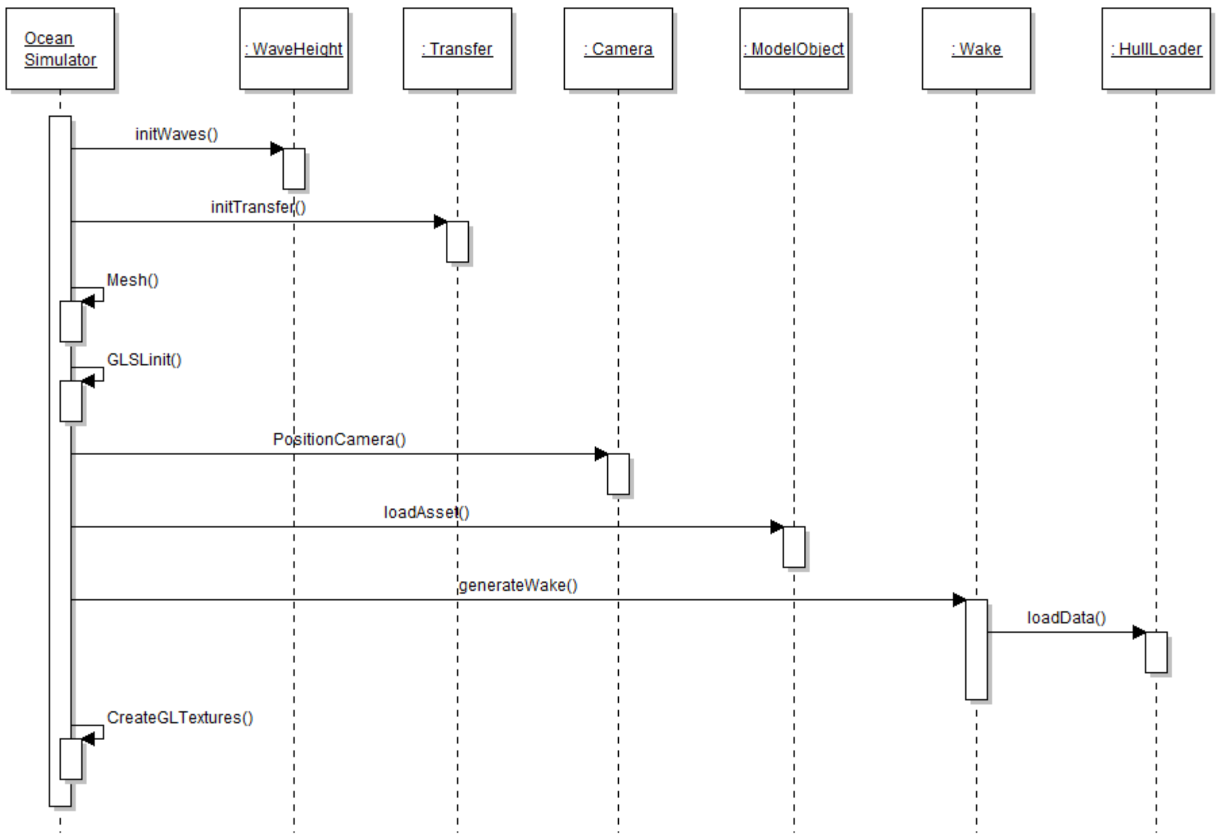


Figure 2.2: Sequence diagram for the setup stage

Figure 2.3 is a simplified diagram for the main loop. The loop starts by calculating the next step of the wave pattern in the *WaveHeight* objects. Once this is done, the translations and rotations for the ship are calculated based on the data output from the wave patterns. The foam calculations are also based on the *WaveHeight* objects. The foam object requests the amplitudes and uses these to calculate where the foam should appear, and returns this information in a 128 x 128 mesh. The new data is then sent to the GPU and finally drawn to the screen with *SwapBuffers*.

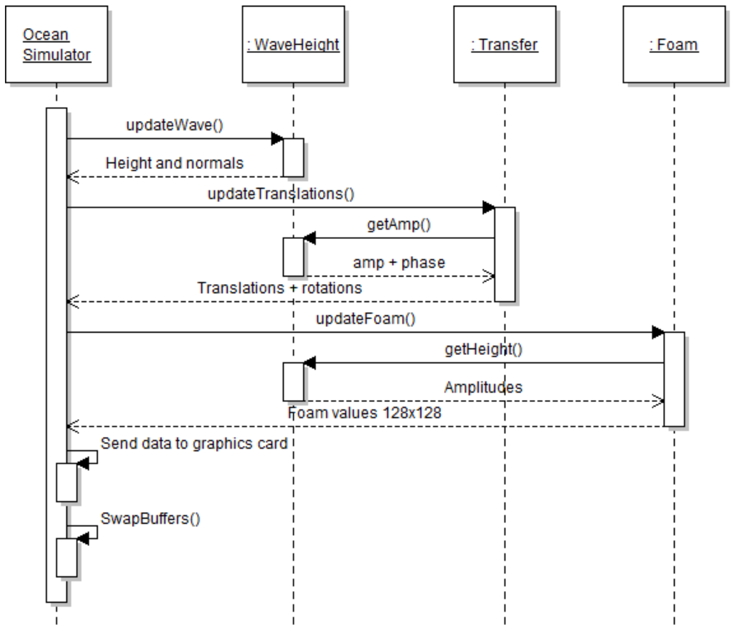


Figure 2.3: Main loop

In addition to the events in the main loop there are two triggered events that may occur, time scaling and keyboard inputs.

The program is scaled with time in order to get a constant speed regardless of the frame rate achieved on a given computer. The frame rate on one specific computer can also vary quite much as it depends on the current field of view. For instance, turning the camera away from the ship model only looking at the ocean can double the frame rate. The time scaling is implemented by measuring the time spent on doing all the calculations and the drawing of the scene, which essentially is the time spent on one iteration in the main loop. When a change in frame rate is detected the *WaveHeight* objects are updated with the new time and scaled properly. This ensures that the waves move with constant velocity without any glitches.

Keyboard and mouse inputs can trigger a whole set of function calls. These inputs are mainly used to change wave parameters, and controlling the camera. The keyboard mapping is described in Figure 4.3.

2.2 Generating Wave

Generating a realistic wave elevation, is done by combining Jonswap spectrum with inverse fast fourier transformation (IFFT). Explanation of the method is done in 1.3.2. IFFT is used due to its running time. Not using a frequency transformation for calculating the wave height would lead to a vast increase in computation time. Ocean Simulator uses two dimensional IFFT. One dimensional IFFT could be used for wave generation. Even though it gives a slower solution, computation time will still scale as $N^2 log(N)$. N different one dimensional IFFT has to be computed. Then only the wave length will be restricted, not the direction. Since the current 2D IFFT version is giving satisfying results, the 1D IFFT solution has not been further investigated. The basis for the waves is three 64 x 64 IFFT transformation on different domain sizes. For an IFFT algorithm the physical domain size and the size of the matrix limit both the longest and shortest wave possible to represent. By introducing three different domain sizes, waves up to 512 meter are calculated, without compromising the details of the smallest waves. The results from the different IFFT transformations add together to the final elevation. The computation time for IFFT is $O(N^2 log(N))$. Having three 64 by 64 domain requires less computation than computing one more detailed IFFT. To avoid duplication of singular waves, each domain is given an upper and a lower limit for the wave length.The physical sizes of these domains are:

Name	Domain size	Shortest wave	Largest wave
small waves	32 m	- m	15 m
medium wave	128 m	15 m	74 m
big waves	512 m	74 m	- m

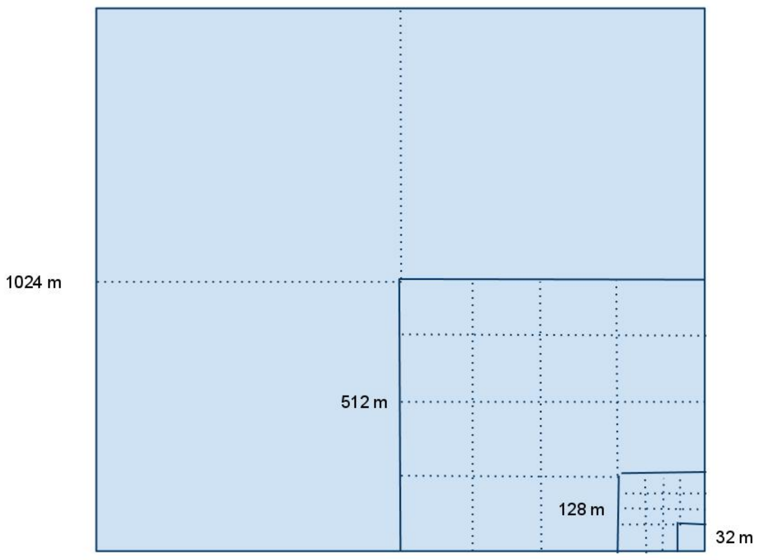


Figure 2.4: Shows the size of the different domain. The three domains are added together to calculate the final wave elevation

One drawback is that the domain sizes are a multi plum of each other, resulting in a repeating pattern. How to fix this problem is explaind in 5.1.

2.2.1 Updating wave

Each iteration the wave pattern has to be updated. By assuming that the FPS rate is constant, the wave pattern can be updated by multiplying each wave component with a complex value each iteration. A wave component is the same as the input value for the IFFT . This update factor is a complex number of length 1, with an angle equal to $dt \cdot \omega$. The ω value can be found from (1.6). The updating factor also includes the ship speed. By adding the ship speed to the updating factor, the ship position will be constant relative to the wave domain. The C++ code below shows how the updating factor is found.

```
void WaveHeight::newSpeed(float Vx)
{
    //dt= newDt;
    // omega :wave frequency
    // dt    : time step
    // Vx    : speed x direction
    for (int i=1;i<N/2;i++)
    {
        for (int j=1;j<N/2;j++)
        {
            w_inc[i+j*N][0]=cos(dt*omega(i,j)+2.0*pi*Vx/L*j*dt);
            w_inc[i+j*N][1]=sin(dt*omega(i,j)+2.0*pi*Vx/L*j*dt);

            w_inc[N-i+j*N][0] = w_inc[i+j*N][0];
            w_inc[N-i+j*N][1] = w_inc[i+j*N][1];
        }
    }
}
```

2.2.2 Choppy Waves

The Fast Fourier Transformation creates normal sin shaped waves, which will be the case for a non windy condition. For windy condition realistic waves will have sharper crests and less curved trough. Ocean Simulator uses the method described by Jensen and Golias (2001).

The basis of the method is to change the position the x and z position of each vertex according to the equations given below. The z coordinate is kept constant. The equations are given for OpenGL coordinate system, meaning that the Y axis is upward.

$$X = X + \frac{C_1 T1}{H0} \frac{(\Delta\zeta)}{\Delta x} \tag{2.1}$$

$$Z = Z + \frac{C_1 T1}{H0} \frac{\Delta(\zeta)}{\Delta x} \tag{2.2}$$

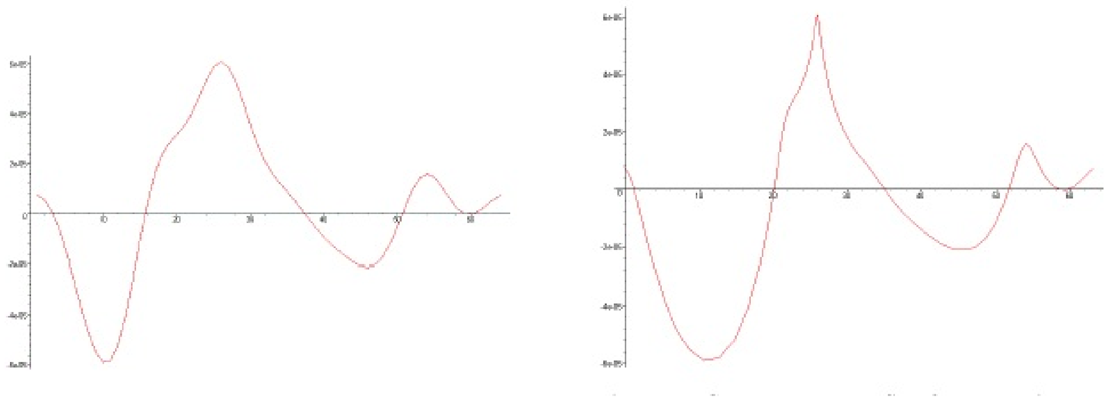


Figure 2.5: Wave profile before and after modification

These calculations are done in the vertex shader. The calculation of the heightmap is thereby not affected by this manipulation. Another positive benefit of this method is that the distance between the vertexes is smaller on the wave crest than on the wave trough. The wave crest is most sensitive to visual quality with respect to vertex density. Smaller distance between the vertexes on the crest results in better graphic without increasing the vertex density.

Quality Check

The total energy level in our final sea state is to be one concrete value, only depending on the square of the wave height. Both left and right side of the equation below are calculated in the program. The relationship between H_i and $S(\omega)$ and $T1$ is explained in 1.3.1

$$T1 = 2\pi \frac{m_0}{m_1} \tag{2.3}$$

$$m_i = \int_{\omega=0}^{\inf} \omega^i S(\omega) \delta\omega \tag{2.4}$$

$$m_{i,discrete} = \sum_{k=0}^{N} \omega_k^i 0.5 H_k^2 \tag{2.5}$$

$$m_{0error} = 1 - \frac{(m_{0discrete})}{m_0} \tag{2.6}$$

$$T1_{error} = 1 - \frac{T1_{discrete}}{T1} \tag{2.7}$$

$$\sigma_{error} = 1 - \frac{stdev(H_i)}{stdev(S(\omega))} \tag{2.8}$$

The graph below shows the relative error

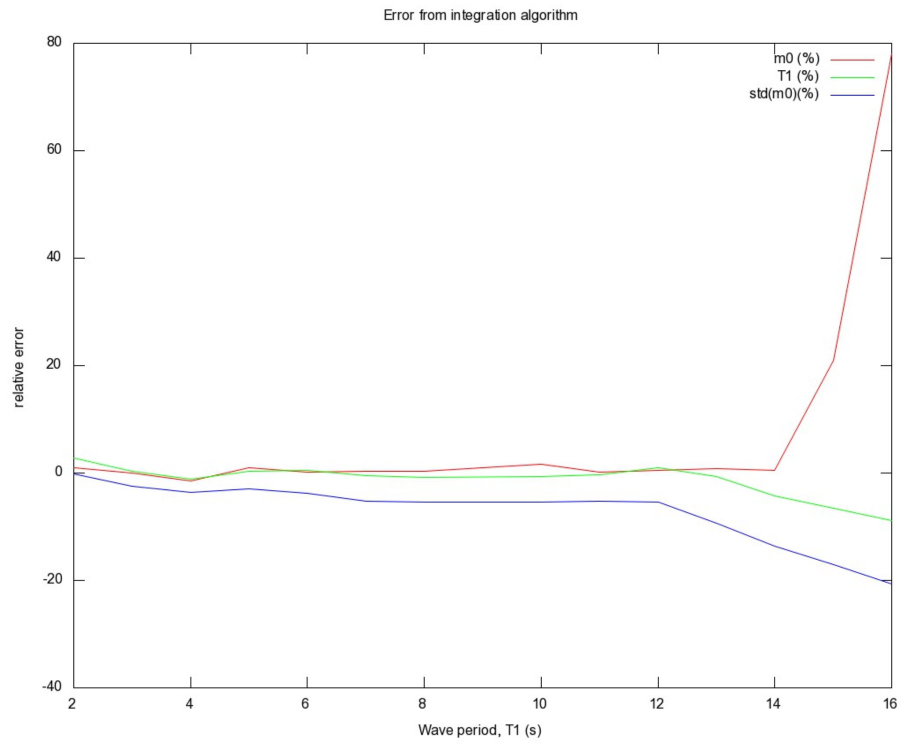


Figure 2.6: relative error. $m_{0error}, T1_{error}, variance_{error}$

The graphs indicate an error of less than ±2% for the T1 and m0. The error for stdev(m0) is 0.1 % for a period of 2 sec. But the error increases for larger periods. The error with respect to T1, is stable when T1 is in the range { 2 s - 12 s}. The longest possible wave to be represented has a period of 18 s. For the range between 12.5 s and 18 s, only 4 single waves are calculated. It is therefore impossible to give a good representation of a sea state with peak period over 12.5 s. For the same reason, the standard deviation therefore starts to drop for T1 over 12 sec.

2.3 Generating Wake

The algorithm for generating wake is based on the theory from 1.5. The implementation of the wake function has only been tested visually. No systematic testing or error estimating has been done. It is therefore no guarantee that there is no smaller error in the code. Visually the results seem promising. It is therefore to believe that the algorithm has no major fault. The picture below shows the result for a 32 meter ship at a speed of 2 m/s. The data has been calculated using an algorithm written in Matlab, but the same algorithm has been implemented in the c++ code . The kelvin angle can be seen from the picture. The wave length on the graph fits well with a wave having a speed of 2 m/s.

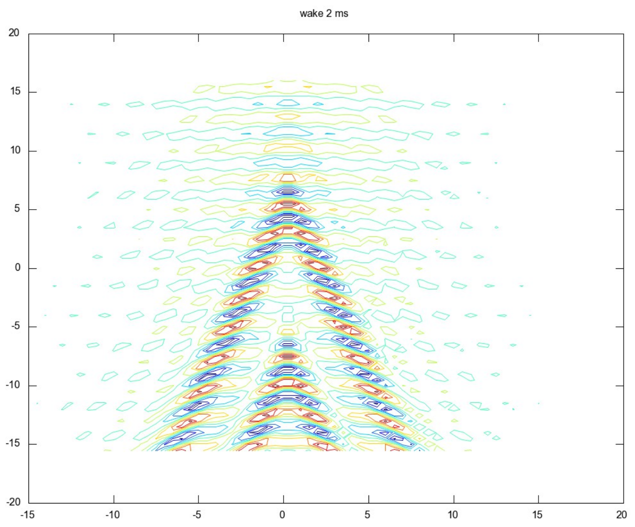


Figure 2.7: Wake from a 32 m long ship at 2 m/s

Using Ifft with a domain size of 1024 meter and 2048 sized IFFT, the largest θ value possible to represent is then 1.4839, which is 0.945π. No attempt to capture the remaining 5.5 percent has been done. An upper limit of 0.02 rad is set for $\Delta\theta$. For the range where the IFFT algorithm gives an $\Delta\theta$ larger than 0.02, the trapezoidal rule has been used. The computation time is measured to 26 sec on a two year old 2.2 GHZ 64 bit Intel processor.

Picture below shows the sampling points for a 32 meter long vessel. Notice how the density of the sampling points increases as theta approaches $\pi/2$. Where the sampling points is marked with '+', the basic trapezoidal are used for integration. The pictures shows that the number of samplings points

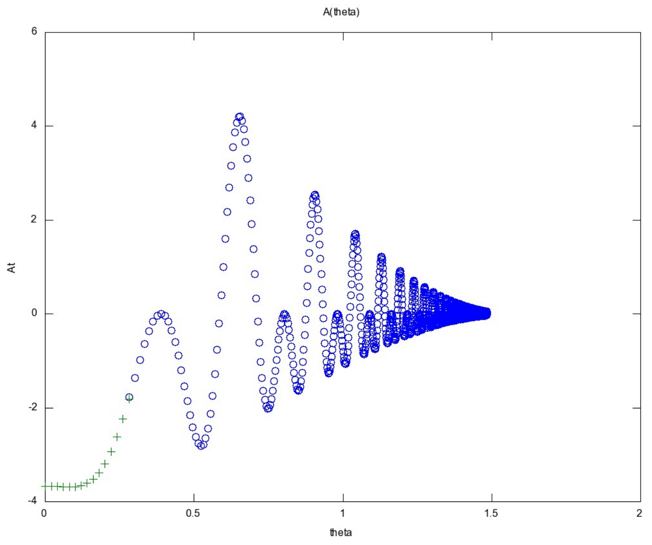


Figure 2.8: A(θ), for a 64 meter ship with speed 3 m/s

The picture below 2.9 displays a fraction of the samplings points for A(θ). The ship length is here 320 meter. Having a 320 more sampling points is needed to get a smooth integration.

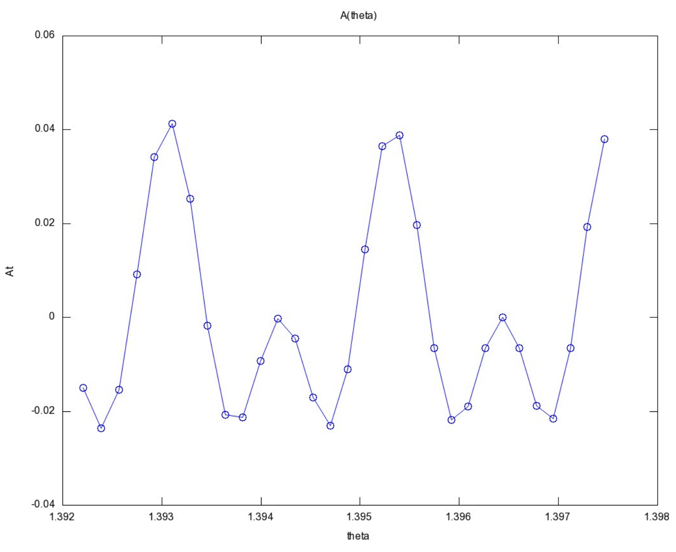


Figure 2.9: section of A(θ), for a 320 meter ship with speed 3 m/s

Only a simple ship model is used when calculating the waves. The shape of the ship model is described below. The depth of the model is constant 16 m over the whole length. A method for reading a 3D. model and calculating the wake is made. But due to limited time, no testing of this method has been done. Therefore only the simplified model is activated. The algorithm for calculating the wake based on the ship shape can be done by activating the method called 'calcAt(...)' in class file 'Wake.cpp' in the program.

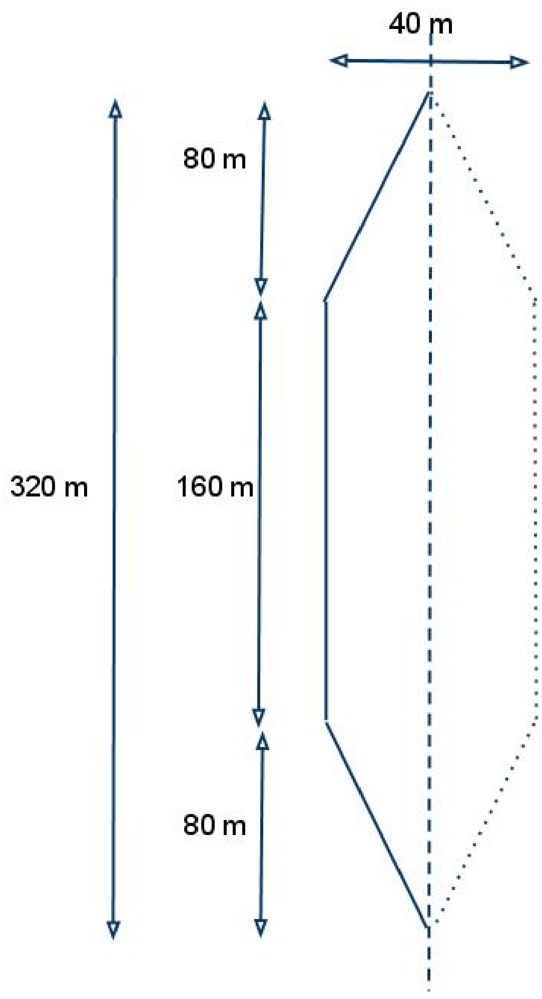


Figure 2.10: Simplified ship model. Depth is 16 m

Picture below 2.11 shows the results. In the upper left of the computational domain some noise can be seen. The noise is due to the discretization. A finer discretization will reduce this problem.

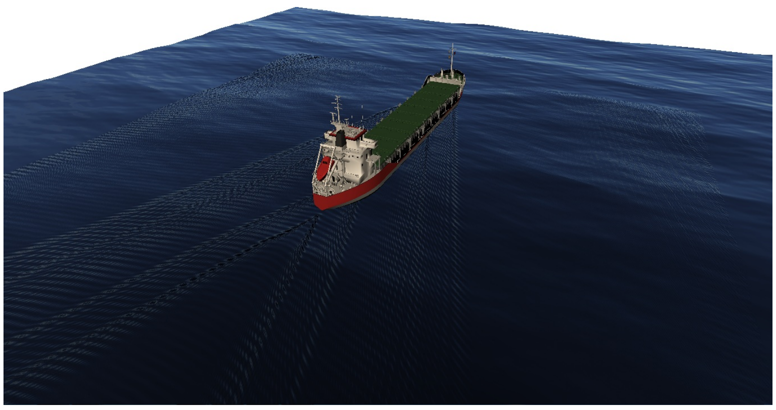


Figure 2.11: Wake calculated using simplified ship model. The visual signature of the wake is increased

Solving the integral for the whole area is time consuming, and requires several minutes of computation time. The wake is therefore computed in advance and stored on an ascii file. One ascii file for each speed of a set of predetermined speed is made. Running the program, the ascii file is then loaded when needed. To reduce the amount of data needed to be loaded, the wave elevation is only computed for a part of the domain. Due to symmetry the only left half of the domain is computed and then mirrored on the right side. The area computed is illustrated below. The size of the grid corresponds to a density of 2 sampling points per meter. A wave having a speed of 2 m/s will be 2.6 m long. To visually represent a wave at least 4 sampling points per wave length is required. 2 m/s is therefore the lowest possible speed to be represented.

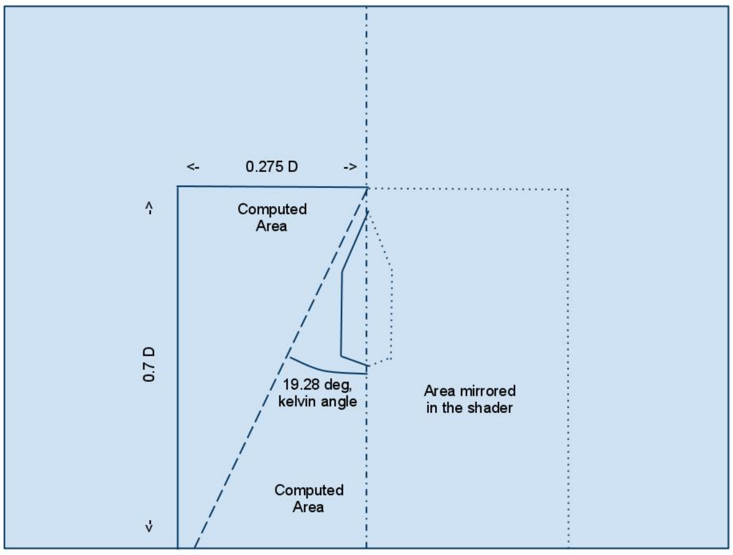


Figure 2.12: Computational domain for wake

2.4 Heightmap and Normal Maps

Before anything can be animated or rendered on the screen, the graphics device needs its data. As mentioned earlier all the calculations are done on the CPU, and the results need to be transferred to the GPU. The best way of doing this is by structuring the data in a *heightmap*, also called heightfield.

A heightmap is represented by an image where each pixel, instead of representing a RGB color, represents a single displacement value. Heightmaps are often used in geographic information systems to describe terrain. It is also heavily used in the rendering of terrain in video games, and it is a perfect way to store the elevation of an ocean surface.

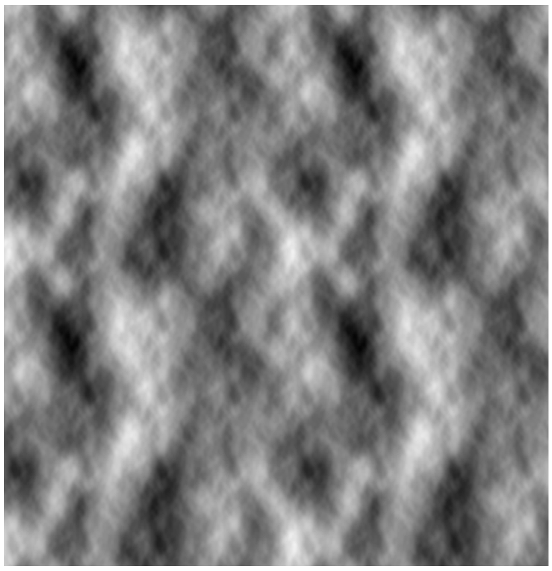


Figure 2.13: Heightmap

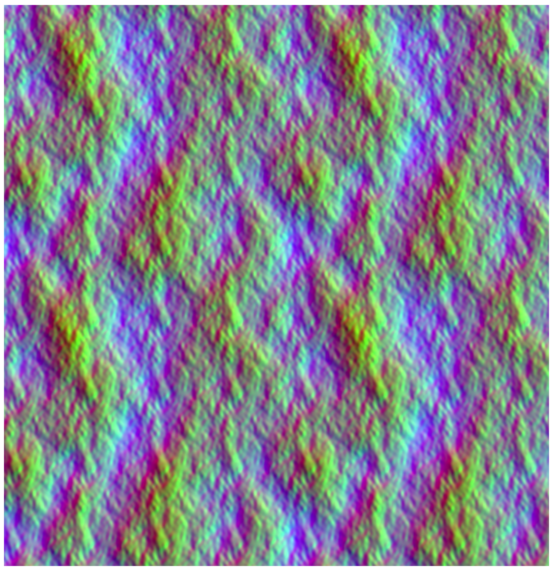


Figure 2.14: Normal map

When visualizing the data in heightmap directly the result is a greyscale image, as shown in Figure 2.13. Black represents the minimum height, and white represents the maximum height. In Figure 2.15 the data from the heightmap have been used to elevate the nodes in a 3D mesh.

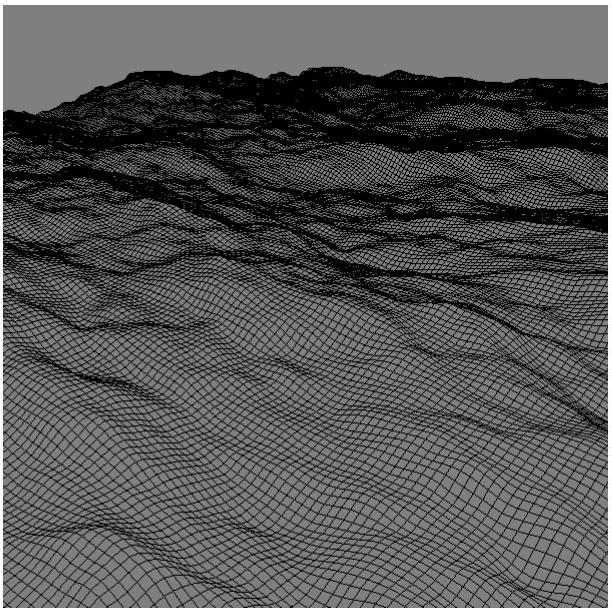


Figure 2.15: Elevated mesh

It is not only elevation data that can be structured and sent this way. Normally an image texture consists of RGB colors. That means each pixel can store three values (Red, Green and Blue). Instead of storing RGB values, it is possible to store whatever kind of values that is needed for the shader. To make possible for light and reflection calculations, the normal vector for the surface is needed in order to get the direction each polygon is facing.

A normal is a vector consisting of three values, and it is possible to use all of the RGB values in a texture to store the normal. The red, green and blue channels of the map represent the X, Y, and Z values of the normal vector. The Z value is the height, while the X, and Y values are the directions.

In Figure 2.14 the first component in a normal vector has been stored at the red location, and the second component has been stored at the green location, and at last the height is stored in the blue location. The result is a *normal map*. All the maps in this section are taken from the application with JONSWAP spectrum.

2.5 Foam

There is no exact answer to when foam emerge. When foam is turned on, Ocean Simulator adds foam to those area where the surface elevation is more than 60 % of H0. When foam is added, the repeating pattern of the wave elevation is easy to detect. To avoid this repeating pattern, the domain size of the different wave systems can be manipulated. Chapter 5.1 gives a further explanation.

The Foam value are calculated two different way and then added together.

- A 128 x 128 matrix is calculated on cpu, describing the foam level on an area equal the largest heightmap. If wave height is higher than 60 % of the H0, a value of 0.03 is added each iteration. The foam level is fading by 1-3% for each iteration.

- On shader, foam is shown for every wave higher than 60 % of H0. This foam has a higher intensity than the foam given by the 128 by 128 matrix.

To reduce the number of texture units used, the foam is calculated by using the gradients from the smallest and medium sized waves.

The foam around the ship is created when loading the wake. The third parameter in the matrix describing the wake is used to describe the foam intensity. The foam from the ship is only calculated once and then shuffled to the GPU. The structure of the foam is calculated on the GPU, using the same technique as when calculating foam from the waves.

2.6 Shaders

This section describes how shaders have been used to visualize the ocean. There are two different types of shaders. The vertex shader sets the position of each vertex. The fragment shader describes the color setting on each fragment. There are two different implementations of the fragment shader.

2.6.1 Vertex Shader

The vertex shader is run for every vertex in the mesh. The vertex shader sets the correct elevation of each node, based on the data given from the heightmaps. This includes the heightmap from the ocean, and the heightmap based on the wake contribution.

As described in figure 2.4 the ocean consists of three textures mixed together. These span over different size domains, and they are repeated several times. Based on the vertex position from the mesh, the correct values from the textures must be chosen. A texture is clamped on a domain from $[0, 0]$ to $[1, 1]$. In order to pick the correct heightmap value, the vertex position has to be converted to a texture coordinate. The code snippet below calculates the texture coordinates for the medium domain and the big domain. 'v' is the position of the vertex.

```
mediumCord.x = mod(v.x, 128.0f)/128.0f;
mediumCord.y = mod(v.z, 128.0f)/128.0f;
bigCord.x = mod(v.x, 512.0f)/512.0f;
bigCord.y = mod(v.z, 512.0f)/512.0f;
```

Once the texture coordinates are created, they can be used to pick the the values stored in the textures. First the displacement values are fetched. Only the textures describing the two largest domains are used. The smallest domain is only used for surface details, and not displacement.

```
v.y = texture2D(heightmapBig, bigCord)[0];
v.y += texture2D(heightmapMedium, mediumCord)[0];
```

Second, the normals for the vertex are fetched, and sent to the fragment shader for lighting calculations. Fetching a texture to vertex shader is less time consuming than fetching a texure to the fragment shader. The grid of the biggest waves is coarser than the grid of the vertices. Fetching the gradient of the biggest waves can be done in vertex shader without loosing data. But the two smaller wave systems have to be loaded in fragment shader.

```
normal.xyz = texture2D(gradientBig, bigCord).xyz;
```

The waves look more realistic with sharper wave crest. The waves are therefore manipulated according to 2.2.2 making the waves more choppy.

```
v.x -= 10*Tc/H0*normal.x
v.z -= 10*Tc/H0*normal.z
Tc := T1*T1,  calculated on CPU.
```

If the ship is displayed, the following code is adding the wake to the elevation data. The code also reads the foam generated by the ship.

```
shipCord.x = abs(v.x/(281.6f)-1.822);
shipCord.y = v.z/(716.0f);

wakeFoam = 0.0f;
if ( shipCord.x<1.0f &&  shipCord.y<1.0f)
{
wakeFoam = texture2D(gradientShip, shipCord).z;
v.y += texture2D(heightmapShip, shipCord)[0];
}
```

2.6.2 Fragment Shaders

The fragment shader sets the final color of each pixel. Normals are added together from each of the wave textures, and the fresnel effect is calculated. Two different fragment shaders have been made. One version using the skybox approach, and one using parameterized approach. An option in Ocean Simulator decides which one to use. Both shaders start with loading the texture values and calculating the gradients.

First gradients from the different textures are loaded.

```
    finalNormal.x = normal.x + 0.8f*tex.x + gradSh.x + tex2.x;
    finalNormal.y = 1.0f;
    finalNormal.z = normal.z + 0.8f*tex.y + gradSh.y + tex2.y;

    finalNormal = normalize(finalNormal);
```

Then the view vector is calculated. The view vector describes the direction from viewpoint to one fragment.

```
vec3 view = normalize(gl_ModelViewMatrixInverse[3].xyz - pos);
```

To be able to calculate the intensity of the light, the reflection vector is calculated.

```
    vec3 reflection;
    reflection = (reflect(view, finalNormal));
```

The fresnel factor is calculated to simulate the diffraction pattern. The calculation of fresnel factor is done according to B.3.

```
cosinus = dot(view, finalNormal);
sinvec  = cross(view, finalNormal);
sinus = sqrt(1.0f-0.25f*dot(sinvec,sinvec));
fresnel = (cosinus - 2.0f*sinus)/(cosinus + 2.0f*sinus);
fresnel = fresnel*fresnel;
```

Foam

When simulating foam, further manipulation of the color is needed. The following code is only executed when foam is simulated. See chapter 2.5 for further explanation.

The foam is simulated by combining a CPU generated matrix with a hight check in shader. First an 'if sentence' makes a check for simulating whether a wave is breaking or not. If a wave is breaking, the color is changed according to the code below. FoamMap is a simple picture used to give texture to the breaking wave.

```
vec4 foam2 = texture2D(foamMap2,  foamCord);

if(wakelessHeight > 0.6f*H0 && normal.z<-0.01)
{
finalColor.x = foam2.x+0.4;
finalColor.y = foam2.y+0.4;
finalColor.z = foam2.z+0.4;
}else{
finalColor.x = 0.0;
finalColor.y = 0.0;
finalColor.z = 0.0;
}
```

A breaking wave tends to create foam which fades away after some minutes. To simulate this effect a foamMap is calculated on the CPU. The texture of the foam is made by combining gradients of the the smallest waves width texture foamMap2.

```
    float foamI = abs(tex.x)*25/H0;;

    finalColor.x += (foam)*(foam2.y*foamI);
    finalColor.y += (foam)*(foam2.y*foamI);
    finalColor.z += (foam)*(foam2.y*foamI);
```

Foam generated by the ship is calculated using similar technique. But the foam intensity is given by the same textures as describe the gradients of the wake.

```
float foamI = abs(tex.x*tex2.x)*7500/H0/H0;

if (shipCord.x<1.0f &&  shipCord.y<0.99f)
{
finalNormal.x += gradSh.x;
finalNormal.z += gradSh.y;
finalColor.x += foamI*wakeFoam;
finalColor.y += foamI*wakeFoam;
finalColor.z += foamI*wakeFoam;
}
```

Parameter

In the parameter shader, position of the sun is set directly in the fragment shader.

```
    lightDirection.x = -1.0f;
    lightDirection.y = -0.4f;
    lightDirection.z = -0.0f;
    lightDirection=normalize(lightDirection);
```

based on the reflection vector, the intensity of diffuse and specular light is calculated. The max() and min() functions make sure that the quotients are 0 or larger.

```
    Rs = max(dot(reflection,lightDirection.xyz), 0);
    Rd = -min(dot(finalNormal,lightDirection.xyz), 0);
    Ra = max(dot(reflection,view), 0);
```

An if sentence separates the area where the sun is directly reflected to viewpoint. For the fragments reflecting the sun directly, a clear bright color is given. For all the other fragments, a modified version of phong formula gives the lighting. . The fresnel component is the only component separating the shader from the Phong formula *Phong shading* (2011).

```
    if (Rs >= 0.9903f)
    {
        finalColor.x = 0.99f;
        finalColor.y = 0.79f;
        finalColor.z = 0.65f;
    }
    else
    {
        finalColor.x = 0.03f*Rd*fresnel+0.02f*Ra+0.13f*Rs*fresnel+0.02f;
        finalColor.y = 0.04f*Rd*fresnel+0.07f*Ra+0.25f*Rs*fresnel+0.03f;
        finalColor.z = 0.05f*Rd*fresnel+0.10f*Ra+0.4f*Rs*fresnel+0.04f;
    }
```

Skybox

Using the Skybox shader instead of the Parameter shader, the color setting is made by the code given below.

```
finalColor += textureCube(skyBox, reflection).zyxw;

if(finalColor.x<0.9f)
{
```

```
finalColor *=fresnel;
}
```

The textureCube command gives the reflecting color from a surrounding skybox. The skybox principle is explaind in C.

2.7 Ship Motion

How the ship moves in the waves is calculated by using transfer function. Chapter 1.4 explains how a transfer function works.

In naval engineering, a common way of describing a transfer function is by using Ascii file. The file describes the transfer function for distinct frequencies and directions. The value of the transfer function is given by two numbers, one amplitude and one phase shift. The file gives one transfer function for every combination of a direction, frequency and DOF. In the beginning of the file, the available frequencies and directions are listed. Interpolation can be used to find the transfer function for directions and frequencies in between the given values.

Ocean Simulator uses the transfer function closest to the original wave. Ocean Simulator operates on several different coordinate system. Making algorithms as simple as possible is therefore crucial for debugging. A more mathematical correct representation would be to use linear interpolation. As the algorithm for calculating the transfer function is implemented, using linear interpolation will not make the algorithm more complicated. But due to the vast number of single waves, the implemented method gives good results. Rewriting the class to use linear interpolation has therefore not been prioritized.

A matrix describes the relationship between each single wave and the transfer function. For each single wave, the matrix has a reference to the nearest transfer function. The pseudo code below describes how the algorithm works.

```
N  :          Grid size.
reDof[]:      relationship between single wave and transfer function.
wave[] :      input values for ifft. Complex number.
transfer[] : transfer function. Complex number.
DOF[]  :      calculated ship position.

for k = 0:sizeof(DOF)
    for i = 0_N
        for j=0:N
            closest_dir = reDof[i,j,1]
            closest_omega = reDof[i,j,0]
            DOF[k] += real ( transfer[closest_dir,closest_omega] * wave[i,j] )
        end
    end
end
return DOF
```

Results

3.1 Performance

3.1.1 IFFT

The asymptotic behaviour of the computation time for a IFFT algorithm is given below.

$$O(N) = N * N * log(N); \tag{3.1}$$

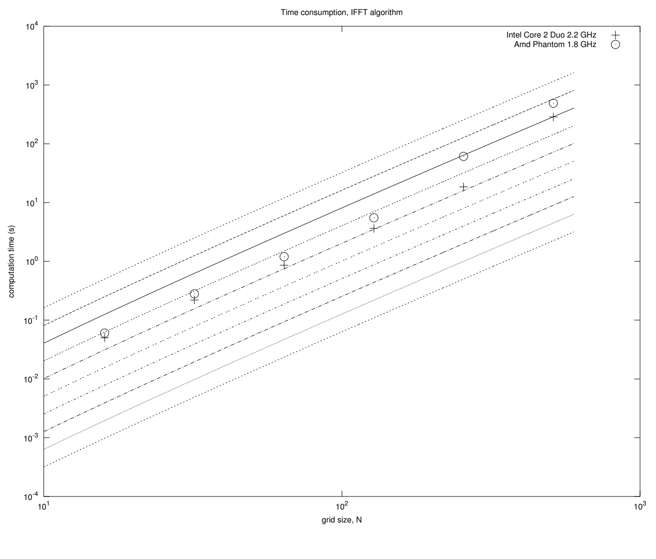


Figure 3.1: Time consumption. The lines show the asymptotic behaviour of the complexity.

Figure 3.1 shows the computation time on two different mediocre CPU's. For grid size below 256, both CPU's run through the algorithm quite fast. The CPU time is not significant for the total performance of the program. But when the grid size exceeds 256, both processors use significantly longer time. Both processors have a L1 cache of 512 kB, which is the same size as a 256 by 256 matrix of complex floating point numbers. The limitation of L1 cache can therefore explain the jump in time consumption when grid size exceeds 256 by 256 elements. But for the purpose of Ocean Simulator, a grid size of 64 by 64 is enough. The IFFT algorithm is therefore not a limiting factor.

3.1.2 FPS vs. Grid Size

For the simulation to run smoothly, a frame rate above 50 is required. The graphs below show how the frame rate responds to a change in the number of vertices. As expected, the relationship between the computation time and the total number of vertices seems to be linear.

The performance has been tested on a 2 year old computer with a ATI 4000 - series GPU. The tests have been done without displaying the ship and without simulating foam. The results are varying with a factor of 2 based on other programs running in Windows. To establish the reason for this variation, more testing is needed.

'Ocean Simulator' is optimized to display waves without effects like foam. Including foam therefore reduces the frame rate significantly. Including a ship in addition results in a reduction of the FPS of approximately one half.

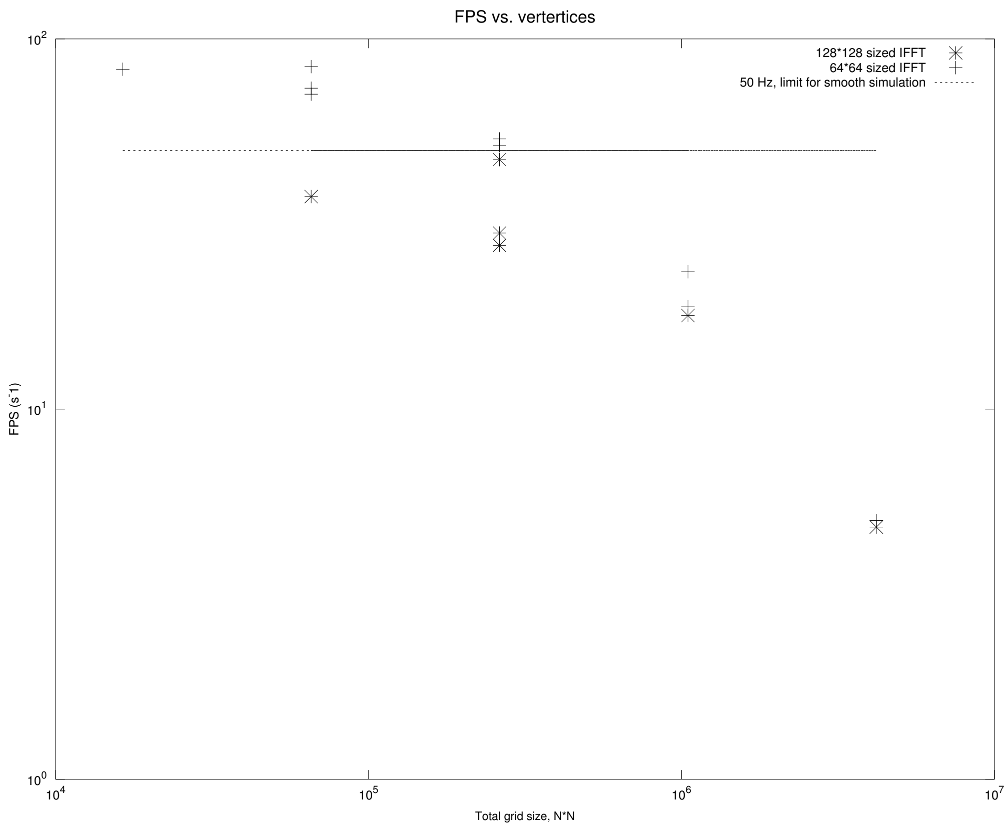


Figure 3.2: The frame rate vs. total number of unique vertices.

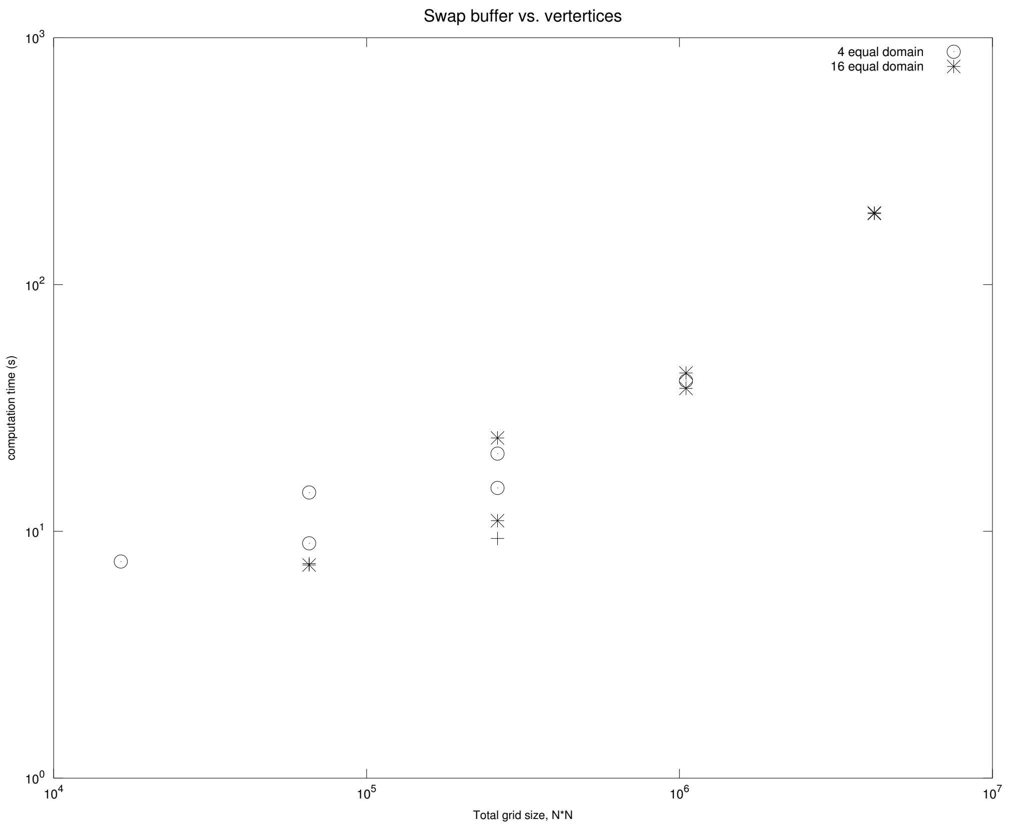


Figure 3.3: Time consumption for swappbuffer

D.N.	N Vertex	N IFFT	FPS	T IFFT	T texture	T swapB
2	64	64	82.8	0.901	0.518	7.55
2	128	64	73.6	0.669	0.00	8.944
2	256	64	51.5	0.873	0.00	14.997
4	64	64	70.9	1.282	0.855	7.306
4	128	64	53.7	1.527	0.983	11.042
4	256	64	23.5	0.921	0.746	37.960
4	512	64	5.0	1.237	0.816	195.12
2	128	128	37.5	3.738	1.397	14.379
2	256	128	29.9	3.888	1.621	20.652
2	512	64	18.9	3.62	1.387	40.746
4	128	128	27.7	3.646	1.359	23.904
4	256	128	17.9	3.556	1.393	43.815
4	512	128	4.8	4.528	2.333	193.917
2	128	64	84.2	0.894	0.491	7.401
2	256	128	47.2	3.653	0.992	9.353

Explanation Table

D.N.:	number of repetition of largest wave system.
N Vertex:	Number of vertexes for one repetition of largest wave
N IFFT:	Size of matrix for fast fourier transformation.
FPS:	Frame rate.

```
T IFFT:    Time needed for all inverse fast fourier transformation
T texture: Time consumption updating all textures.
T swapB:   Time consumption Swap Buffer.
```

3.2 Skybox and Parameterization

In the application two different lighting shaders were implemented. The skybox approach, and the parameterization approach. The visual results are shown in Figure 3.4 and Figure 3.5. Both approaches give a nice looking sea, but the skybox approach gives the most realistic representation. This conclusion might not be a big surprise since the skybox uses pictures of the real sky. All the details in a picture are difficult to represent mathematically with a formula. Both shaders run with the same execution time giving no difference in the overall performance.

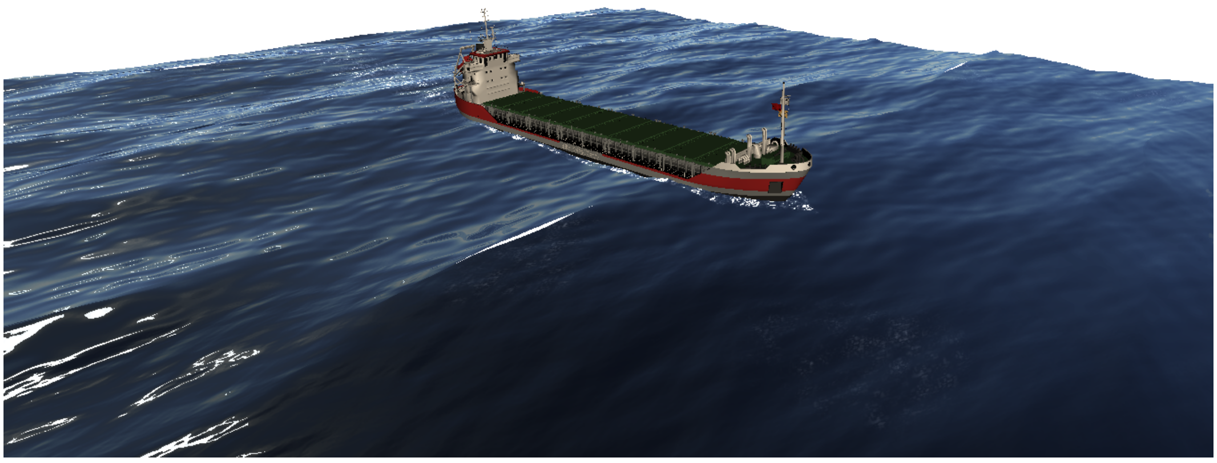


Figure 3.4: Skybox (cube mapping)

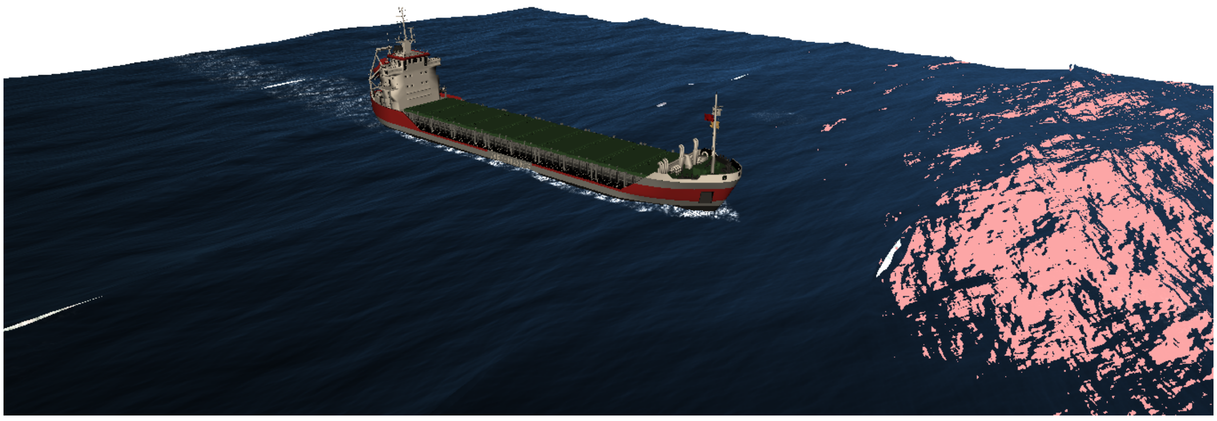


Figure 3.5: Parameterization

3.3 Repetition

With a short wave length, it is easy to see the repetitive pattern of the wave. Figure 3.6 below shows the wave pattern with a T1 of 4 s, which gives a wave length of approximately 25 meter. Ocean Simulator uses three heightmap with dimensions of 32, 128 and 512 meter. Since the 128 is a multiple of 32, the two smaller height map produce a repetitive pattern for every 128 meter.

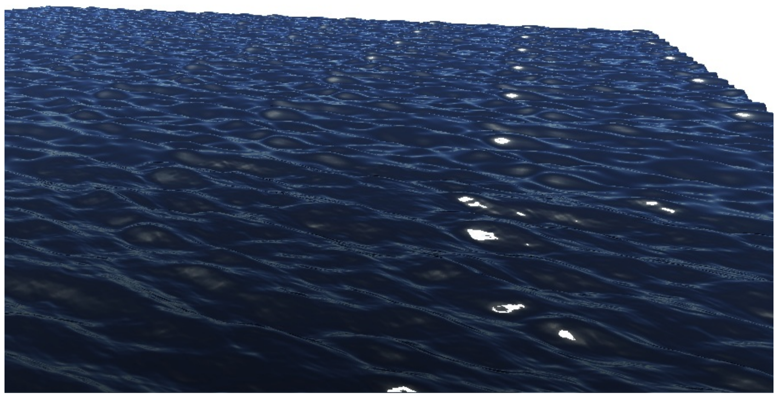


Figure 3.6: Repetitive pattern of waves

A solution is to implement a new height map and change the physical size of their domain, at which they are not a multiple of each other. E.g.. Having 4 heightmaps with sizes 24, 53, 189 and 551 meter will eliminate this problem.

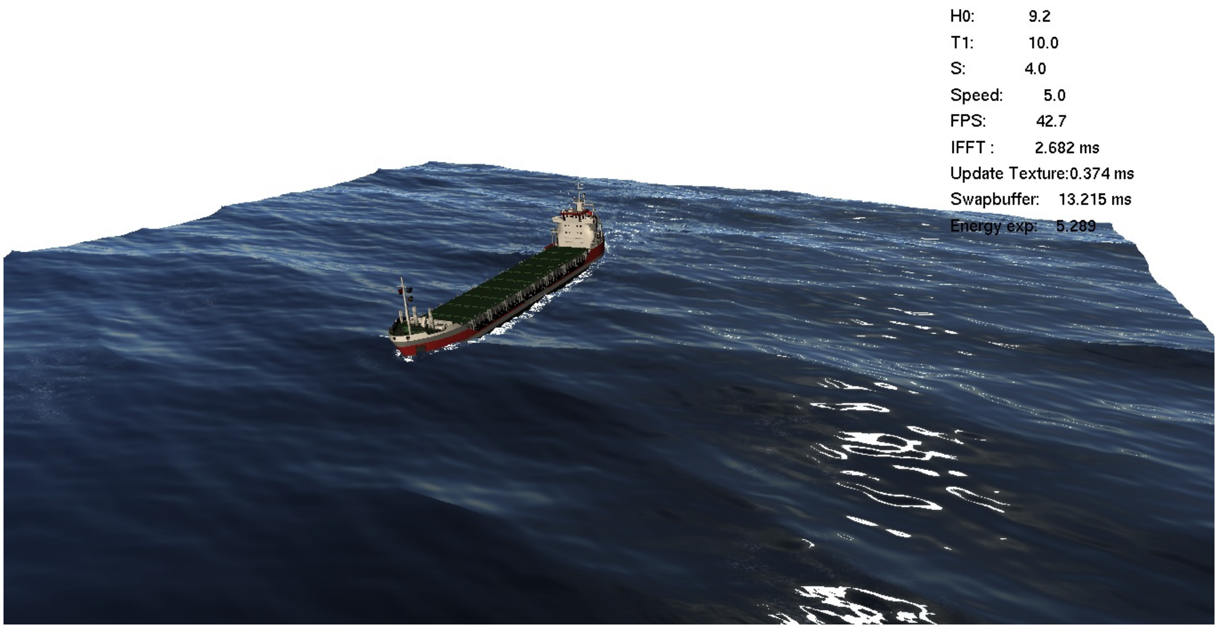
The OceanSimulator Application

To run OceanSimulator, double click on the file 'oceanSimulator.exe'. For the program to run properly, the requirements below have to be met. Most graphics card with less than 2 year of age will meet these requirements.

Requirements:

- Graphics card from ATI (may run on some Nvidia devices)
- Support for OpenGL 3.2 or newer version.
- Support for at least 16 texture units.

Figure 4.1 and Figure 4.2 shows the different program interfaces. They are accessed by pressing 'B'. Figure 4.1 shows the first interface giving information about the different wave parameters, and time spent on the different calculation and data transfer tasks. Figure 4.2 shows the second interface displaying plots for the ship motion.



```
H0:          9.2
T1:          10.0
S:           4.0
Speed:       5.0
FPS:         42.7
IFFT :       2.682 ms
Update Texture:0.374 ms
Swapbuffer:   13.215 ms
Energy exp:   5.289
```

Figure 4.1: Application with the first interface showing wave parameters and time spent on different tasks

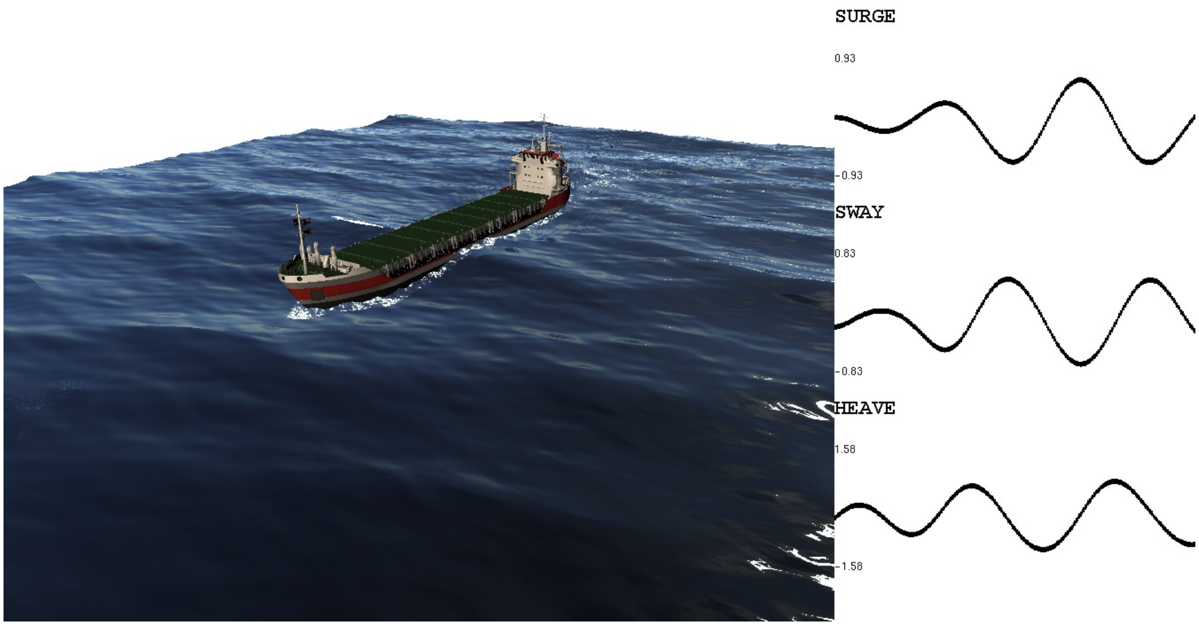


Figure 4.2: Application with the second interface showing plots of the ship movement

Operation of Ocean Simulator is based on keystroke and mouse input. An overview of the different keyboard inputs can be seen in Figure 4.3. Note that the keys 'W', 'S' , 'E' , 'D' will change purpose when a single wave is generated. Keyboard arrow 'UP' and 'DOWN' zooms the view in, and out, respectively. The mouse is used to look around in the scenery. Holding right mouse button down enables the user to strafe left, right, up and down with the mouse. Left clicking pauses the program and unlocks the mouse so other tasks can be performed on the desktop. 'ESC' shuts down and exits the program.

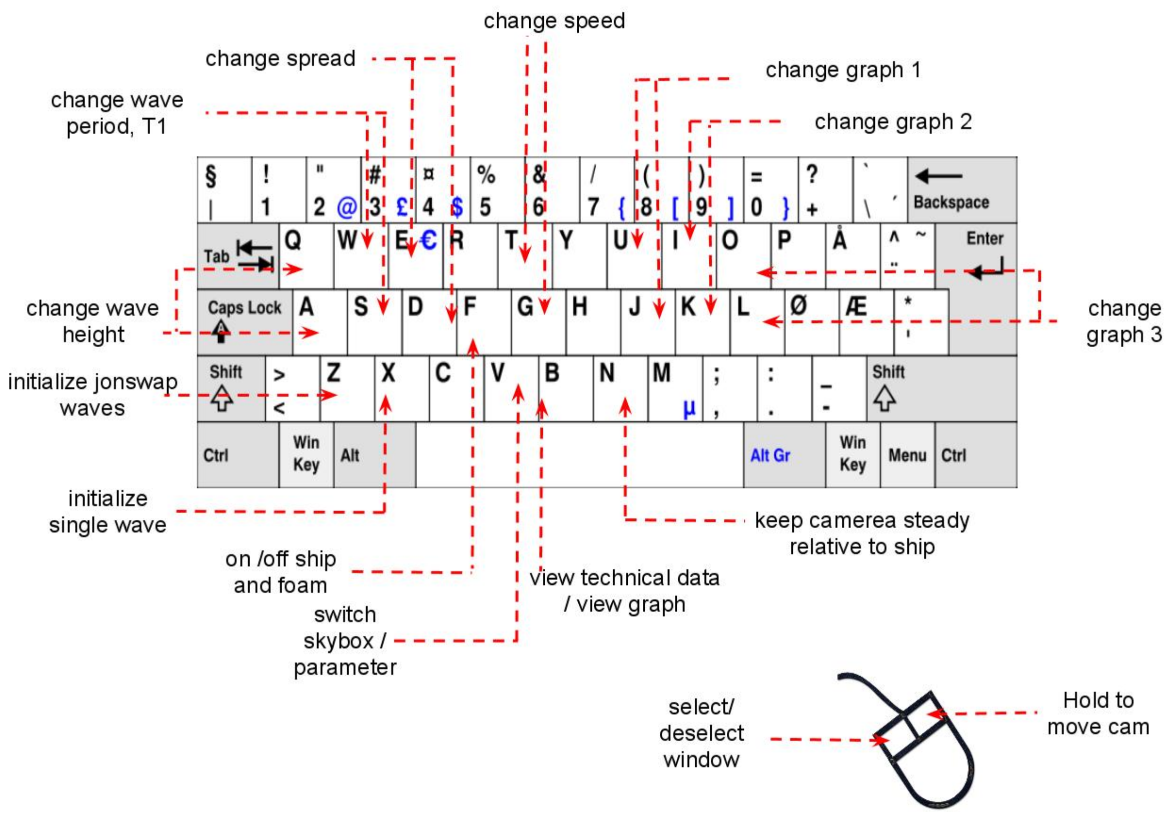


Figure 4.3: User input

Further work

5.1 Generating Wake

The algorithm for generating wake has an option for implementing a 3D model of a ship. But this method is not combined with the use of IFFT algorithm. Which means a computation time of one hour instead of 30 sek. In addition the algorithm which calculates the wake based on an external 3D model, does so by using numerical integration of function 1.33. If the numerical integration was switched with an analytical model, the algorithm would be both faster and more correct. The method below describes how to implement an analytical model.

The method described here assumes that a model consists of triangles, or can be represented by triangles. The shape of a triangle can be represented by four linear functions, as shown in picture 5.1 below. The first step is then to dividing the triangle into two new triangles separated by a vertical line going through the second vertex. Then $A(\theta)$ can be found by analytical integration over each of the two triangles 5.2. y(x,z) is the function describing the plane surface.

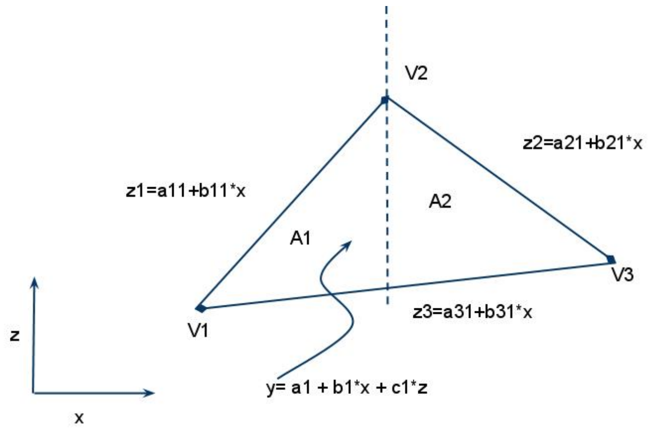


Figure 5.1: A triangle can be represented as equations

$A(\theta)$ from each primitive can then be computed by adding together the contribution from each area.

$$A(\theta) = A(\theta)_{A1} + A(\theta)_{A2} \tag{5.1}$$

From chapter 1.5 we have the formula for $A\theta$

$$A(\theta) = \frac{2}{pi} * k(\theta)^2 \int\int Y(x,z)exp(k(\theta) * Z + i * k(\theta)))dzdx \tag{5.2}$$

$$A(\theta)_{A1} = \frac{2}{pi} * k(\theta)^2 \int_{V1.x}^{V2.x} \int_{z3(x)}^{z1(x)} Y(x,z)exp(k(\theta) * Z + i * k(\theta)))dzdx \tag{5.3}$$

$$A(\theta)_{A2} = \frac{2}{pi} * k(\theta)^2 \int_{V1.x}^{V2.x} \int_{z3(x)}^{z2(x)} Y(x,z)exp(k(\theta) * Z + i * k(\theta)))dzdx \tag{5.4}$$

A brief attempt to estimating the error has been done. The IFFT method is combined with the trapezoidal rule for integration, the error estimate for the trapezoidal rule is applicable. The proof for the error formula is given in R. L. Burden (2005).

$$Error = \sum \frac{(\Delta\theta_i)^3}{12} f''(\theta_i) \tag{5.5}$$

The maximum wave amplitude found for a 320 meter long ship at speed of 10 knots is 0.45 meter. The maximum error found from the formula is 0.1 m. Which gives an maximum error of approximately 20 %. From Figure 2.9 it is clear that a more clever integration technique than the trapezoidal rule has to be implemented if the number of sampling points is not increased. The error estimate can explain the noise emerging in the outer areas of the wake domain. Since the computation time is around 30 sec for one wake field, increasing the accuracy of the method seems feasible. One solution for reducing the error could be to to increasing vector for the IFFT by a factor of 8. Thus increasing the number of sampling points by a factor of 8. Simultaniusly increasing the computational domain by a factor of 8 in x direction. The step size in frequency domain will then be redused by a factor of 8. Leaving the error estimate with a value of $(\frac{1}{8})^2 = 1.5\%$ of current value. Since a bigger area will be computed, only 1/8 of the computed needs will be used in the program. Increasing the computational domain with a factor of 8 is expected to increase computation time with a factor of 10.

In the current configuration, the IFFT algorithm will not capture the last 5.5 % of the domain of $A(\theta)$. Integrating the $A(\theta)$ function over the last 0.055% , will affect the wave height by at most 10^-4m. Compared to a wake amplitude of 0.45 meter, this is a small error. The value 10^-4 is found by doing the following integration.

$$error = \int_{0.95*pi/2}^{\pi/2} abs(A(\theta))d\theta \tag{5.6}$$

Another small problem with the wake algorithm is that it produces texture with a size of 700 by 1700. Textures are more efficiently handled when it is at a size of 2^n. In addition, some older versions of OpenGL drivers may not support textures which do not have a size of 2^n. Testing the program on an older computer, the textures may therefore be interpreted wrongly. Increasing the texture size to 1024 by 2048 will therefore increase the performance and stability of the program. But longer loading time is to be expected.

5.2 Tessellation

Tessellation is a technique for dynamically varying the number of vertices. Typically increasing the number of vertices near the view point, and decreasing the number of vertices for objects far away.

The mesh implemented in the application is static. The size of each polygon is the same whether viewing from a distance far away or near the surface. This can cause a problem when looking closely at the elevated surface. If the node size in the mesh is big enough, it might be possible to see that the surface consists of squares. Another visual aspect to notice is that the surface itself looks relatively flat. The elevation data density is greater than the node density. A solution for how to use more of the height data could simply be to increase the number of nodes. The downside of increasing the number of nodes is that it increases workload on the GPU, and would drop the frame rate significantly. The best solution would be to implement a tessellation algorithm.

Tessellation techniques are used to divide a polygon into a suitable sized structure. In graphics programing this is implemented by introducing the geometry shader which builds the vertices and polygons. With a proper algorithm in the geometry shader, the size of the polygons could be determined based on the distance from the camera to the surface. This is known as adaptive subdivision tessellation, and an example of this technique is shown in Figure 5.2. Polygons near the camera would be small, making the surface more detailed where it is important, and polygons far away would be bigger. The result would be a fine grid near the camera and a coarse grid at distance. With this technique the visualization would be improved, only suffering a small performance loss when the same number of vertices as in the static approach is used.

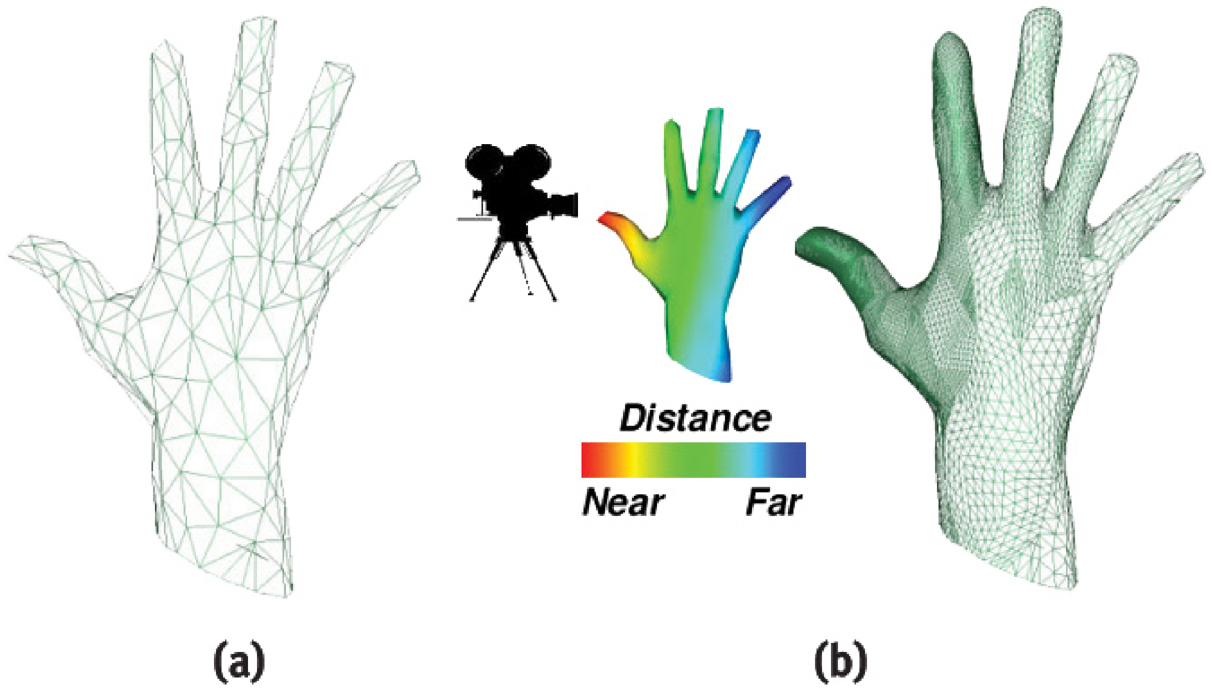


Figure 5.2: (a) The base mesh for a hand. (b) Refined mesh based on distance from camera. (Nguyen, 2007)

Figure 5.3 shows the current elevation mesh. Keeping the density of the vertices constant results in more vertices than possible to represent on screen. The high number of vertices may also result in a distorted picture, reducing the visual quality. A clever implementation will use less resources on computing the areas far away, without compromising with the quality.

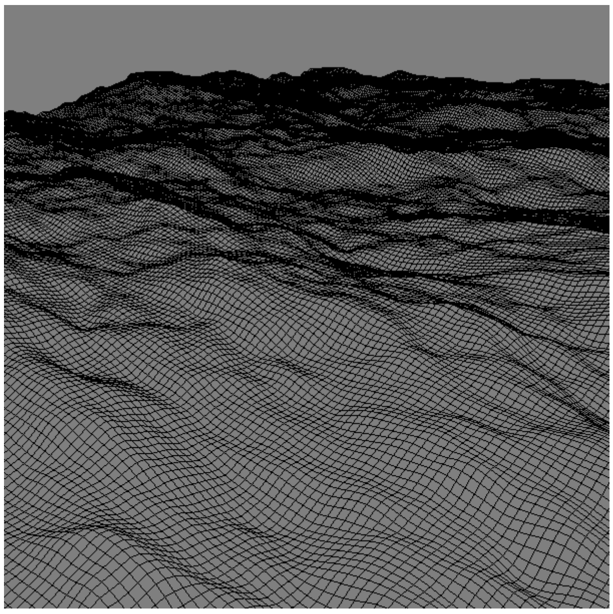


Figure 5.3: Elevated mesh

Bibliography

E.O. Tuck, L. L. and Scullen, D. (1999), ‘Sea wave pattern evaluation’.

Faltinsen, O. (1990), *Sea Loads on Ship and Offshore Structures*, first edn, Cambridge University Press.

Faltinsen, O. (2005), *HYDRODYNAMICS OF HIGH-SPEED MARINE VEHICLE*, first edn, Cambridge University Press.

FFTW (2011), `http://www.fftw.org/`.

Jensen, L. S. and Golias, R. (2001), ‘Deep-water animation and rendering’, `http://www.gamasutra.com/gdce/2001/jensen/jensen_01.htm`.

Kreyszig, E. (2006), *Advanced Engineering Mathematics*, ninth edn, John Wiley and Sons (WIE).

Myrhaug, D. (2007), ‘Marin dynamikk, uregelmessig sjø’, Kompendium.

Nguyen, H. (2007), *GPU Gems 3*, first edn, Addison-Wesley Professional.

Phong shading (2011), `http://en.wikipedia.org/wiki/Phong_shading`.

R. L. Burden, J. D. F. (2005), *Numerical Analysis*, eighth edn, Thomson Brooks/Cole.

Rost, R. J. (2006), *OpenGL® Shading Language, Second Edition*, Addison Wesley Professional.

Appendix A

Opengl, Graphics Language

When developing an application capable of producing 2D and/or 3D computer graphics, there are two major application programming interfaces (APIs). One of these is the Windows specific version, called DirectX. DirectX generally targets the Microsoft Windows platform, including the Xbox consoles. Second, there is OpenGL (Open Graphics Library). OpenGL is a cross-language, cross-platform API. The OpenGL API is available under an open source license, and implementations exist on most platforms. The idea behind OpenGL is to have an open source platform for programming on graphics card. The usage of OpenGL has increased greatly the last years mostly due to the smartphone revolution where the Microsoft Windows platform (DirectX) has a very limited amount of the market share.

The graphics card gets its instructions from the central processing unit (CPU), and executes the given commands. All well known programming languages supports OpenGL.

A.1 Functionality

DirectX and OpenGL have to be supported in the graphics processing unit (GPU) in order to work. Nearly all GPU vendors have support for both of these APIs. The OpenGL interface consists of more than 700 distinct commands that allow the user to specify operations needed to produce interactive three-dimensional applications.

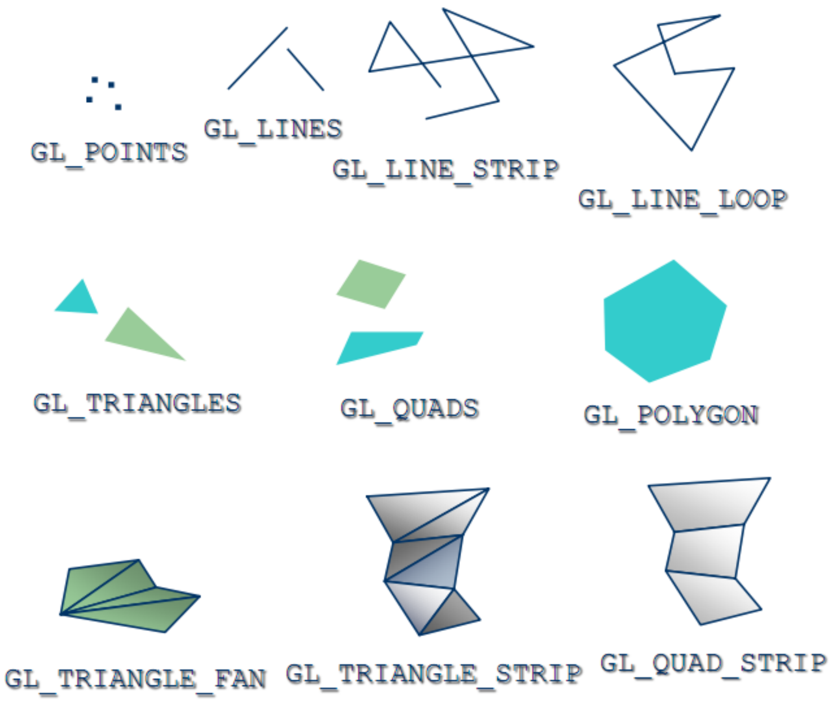


Figure A.1: Overview of the geometric primitives

It does not provide high-level commands for describing models of three-dimensional objects, like a boat model for instance. Every model is built up with a set of geometric primitives consisting of points, lines and polygons. Figure A.1 shows all the primitives in OpenGL. The primitives are specified by their vertices, and OpenGL converts these primitives to images on the screen. A vertex may be specified in either 2D (x, y) or 3D (x, y, z). The coordinates are used for positioning in space. Different numbers of vertices give different geometric primitives. Two vertices are needed for a line, three vertices for a triangle and four for a quad.

A.2 The OpenGL Shading Language

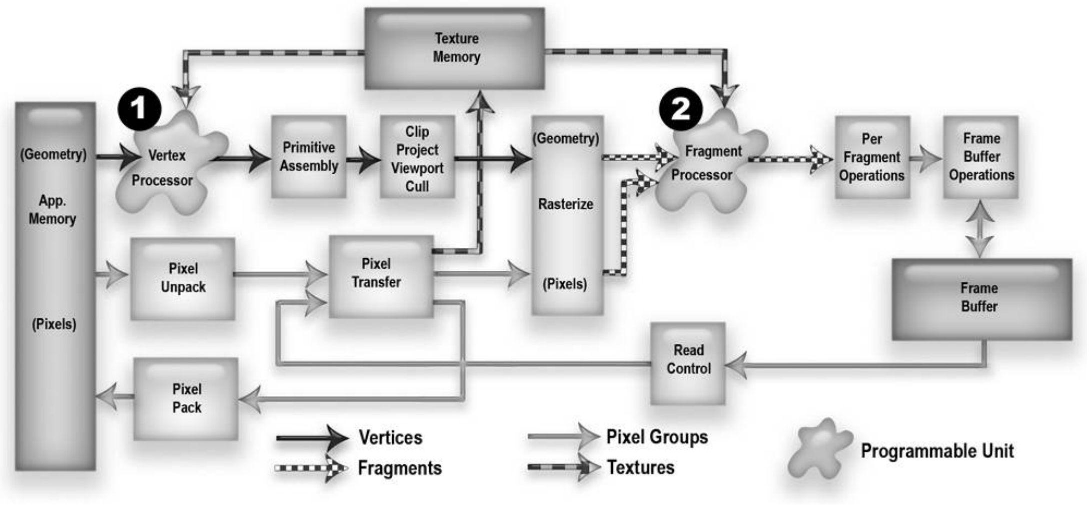


Figure A.2: OpenGL rendering pipeline (Rost, 2006)

Figure A.2 is a detailed overview of the rendering pipeline in OpenGL. The rendering pipeline consists of several stages that is required to get the image of a 3D model to the screen. The important stages here are the vertex processor, and fragment (pixel) processor, numbered with 1, and 2, respectively. The vertex processor assembles the geometry, and the fragment processor performs coloring of the geometry surface. These stages are fully programmable, and a program running on either the fragment or vertex processor is called a *shader*. To program a shader the OpenGL Shading Language (GLSL) is used. With C as its basis, functions like for loops, subroutine calls, and branching (if-statements) are supported. In addition the language has a rich set of types, including vector and matrix types which are heavily used in typical 3D graphics operations.

To run the shader, a shader program must be created and the source code has to be sent to the GPU. The steps are shown in Figure A.3. When the applications is running, the instructions (shader) for the vertex processor is run on every vertex in the 3D model, with one vertex in, and one updated vertex out. The same concept applies for the fragment processor. For every pixel that needs to be drawn on the 3D model surface, the fragment processor performs its instructions and the result is one updated pixel ready to be drawn on the screen. The vertex and fragment shader is described further in Section A.3 and Section A.4.

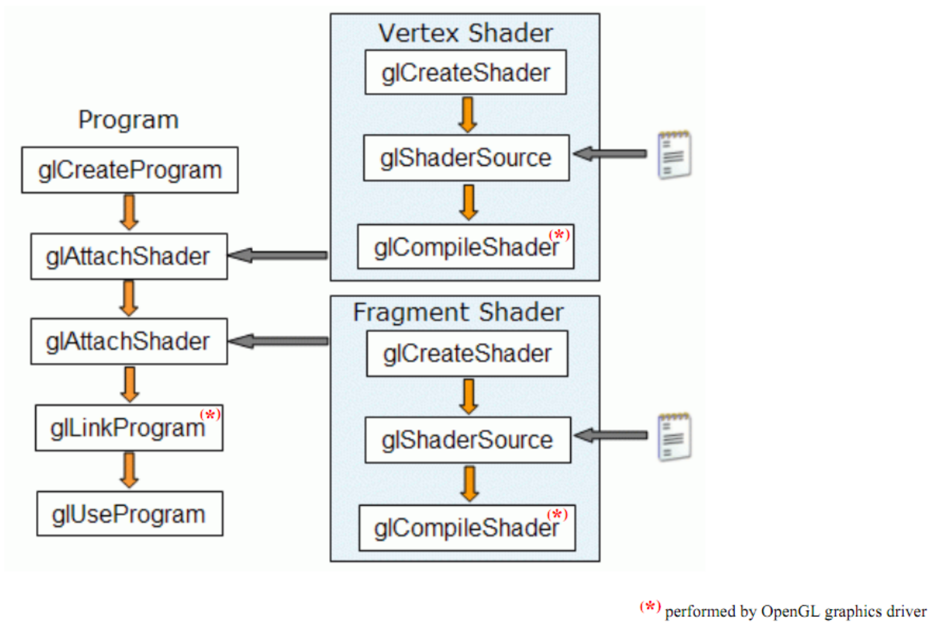


Figure A.3: Steps in creating and sending shaders to the graphics device

In order to get a realistic looking ocean, both the vertex and the fragment stages needs to be programmed properly. To get correct animation of the waves, the vertex processor needs its instructions. In order to get the water looking realistic with light reflections and optical effects, the proper algorithms has to be implemented in the fragment processor.

A.3 Vertex Processor

First and foremost a vertex is described by its location in space, as mentioned earlier, but it also holds other values like color, texture coordinates and normals. The vertex shader cannot change the data types but they can change the value of the data. The vertex processor performs operations such as vertex transformation, normal transformation and normalization, texture coordinate generation, texture coordinate transformation, lightning, and color material application. Because of its general purpose programmability mathematical equations and implementations can be used in any order to manipulate these data to give the desired effect. The same applies to the fragment shader.

Figure A.4 shows the different data values that are used as inputs to the vertex processor, and the values that are produced by the vertex processor. There are two types of attribute variables, the built in and the user defined. The built in variables consist of the vertex coordinates, normals, colors, and texture coordinates, i.e. variables that describe the geometry. But it is also possible to define your own variables, called uniforms. The uniform variables are available in both the vertex and fragment shader, and can be updated from the CPU when it is needed. It can store integers, floats, boolean, matrix values, and arrays of these.

With the vertex processors capability to read from texture memory, displacement mapping algorithms, among other things, can be implemented. This is a key feature regarding our implementation of the ocean surface.

Variables can also be passed from the vertex processor to the fragment processor. These variables are called varying variables, and both built-in and user-defined varying variables are supported. They are called varying because each of these variables usually are different for each vertex.

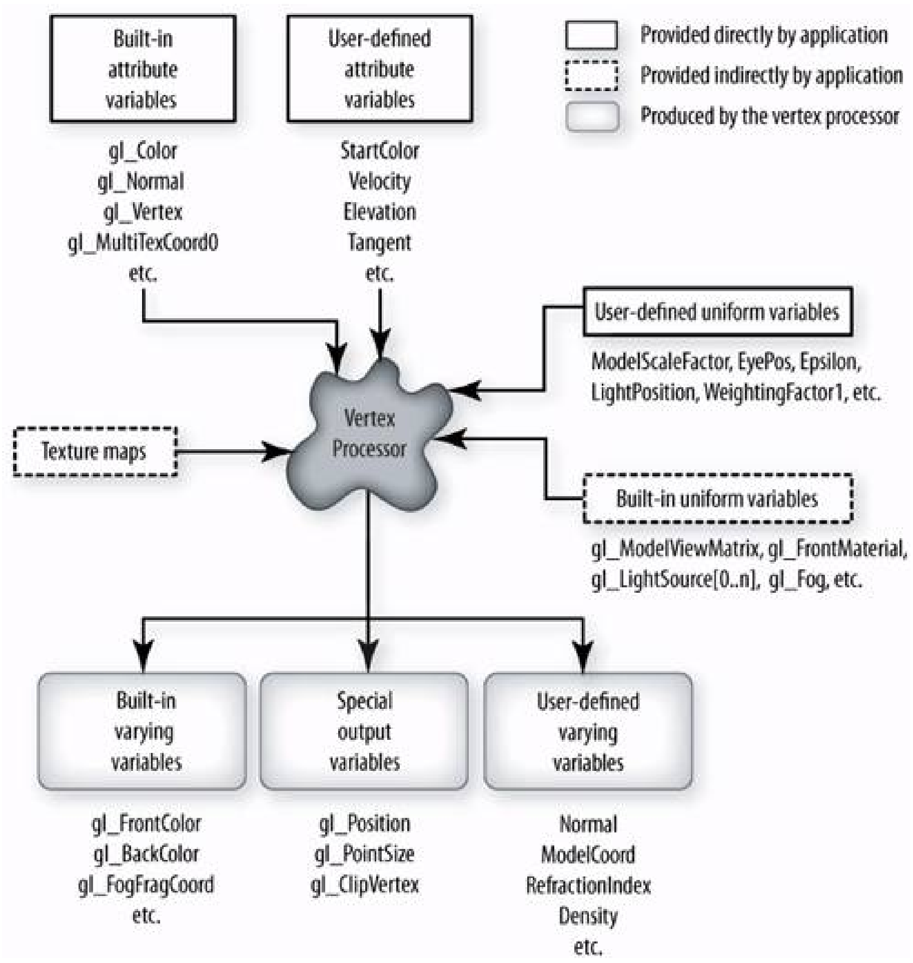
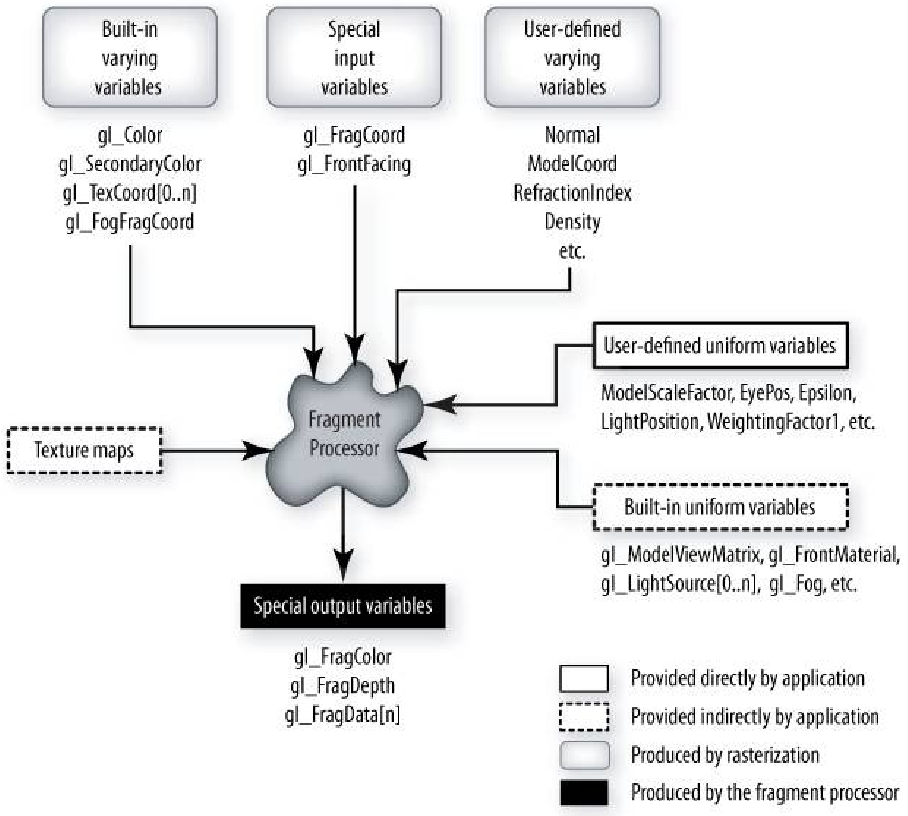

Built-in attribute variables
gl_Color
gl_Normal
gl_Vertex
gl_MultiTexCoord0
etc.
User-defined attribute variables
StartColor
Velocity
Elevation
Tangent
etc.
Provided directly by application
Provided indirectly by application
Produced by the vertex processor
User-defined uniform variables
ModelScaleFactor, EyePos, Epsilon,
LightPosition, WeightingFactor1, etc.
Texture maps
Vertex Processor
Built-in uniform variables
gl_ModelViewMatrix, gl_FrontMaterial,
gl_LightSource[0..n], gl_Fog, etc.
Built-in varying variables
gl_FrontColor
gl_BackColor
gl_FogFragCoord
etc.
Special output variables
gl_Position
gl_PointSize
gl_ClipVertex
User-defined varying variables
Normal
ModelCoord
RefractionIndex
Density
etc.

Figure A.4: Vertex processor inputs and outputs (Rost, 2006)

A.4 Fragment Processor

The fragment processor operates on fragment values and their associated data. It performs traditional graphics operations such as operations on interpolated values, texture access, texture application, fog, and color sum. These operations are performed on each fragment generated by the rasterization of points, lines, polygons, pixel rectangles and bitmaps. If a texture is enabled, the fragment processor will read the image from the texture memory and apply it to the polygon while performing traditional operations such as pixel zoom, scale and bias, color table lookup, convolution, and color matrix.

Figure A.5 gives a representation of which inputs and outputs that are processed.



Built-in varying variables
gl_Color
gl_SecondaryColor
gl_TexCoord[0..n]
gl_FogFragCoord
Special input variables
gl_FragCoord
gl_FrontFacing
User-defined varying variables
Normal
ModelCoord
RefractionIndex
Density
etc.
User-defined uniform variables
ModelScaleFactor, EyePos, Epsilon,
LightPosition, WeightingFactor1, etc.
Texture maps
Fragment Processor
Built-in uniform variables
gl_ModelViewMatrix, gl_FrontMaterial,
gl_LightSource[0..n], gl_Fog, etc.
Special output variables
gl_FragColor
gl_FragDepth
gl_FragData[n]
Provided directly by application
Provided indirectly by application
Produced by rasterization
Produced by the fragment processor

Figure A.5: Fragment processor inputs and outputs (Rost, 2006)

The fragment shader computes color, depth, and arbitrary values. Different lightning, reflection, and shadow effects are typically added here. When the fragment has got its final color, it is sent to the framebuffer and displayed on the screen.

Appendix B

Optical Effects On The Water Surface

To get a realistic rendering of the water surface there are many components that must be considered. Water reflects its surroundings, and the reflection changes depending on the viewing angle. When observing at a glancing angle (eyes at level with the water surface) there are reflections and specularity on the surface. If the water's orientation reflects sunlight directly there will appear very bright spots on these locations. When looking straight down from above the ocean, or a pool for instance, there is not much reflected light on the water surface. The reason for this is that water is a transmissive medium, and at a certain angle the incoming ray will not be reflected, but it will continue through the water. If the reflectivity of the water surface is low, light coming from below the surface can be visible to the observer. This light is either caused by scattered light from the water volume, or reflection from the water bottom. The scattered light is a result of unabsorbed light from water molecules, and impurities in the water. The impurities in the water determine how much of the light is scattered, and what color the water gets. For instance muddy water is brown, and algae can cause the water to become green.

B.1 Reflection

There are basically two kinds of reflections. Reflection of light is either *specular* or *diffuse* which depends on the material of the reflecting surface. Specular reflection is a mirror like reflection where light from a single incoming ray is reflected into a single outgoing ray. The angle of the incoming ray is equal to the angle of the outgoing ray. Diffuse reflection is a ray where light from a single source is reflected at many angles rather than just one angle. Figure B.1 shows how the incoming light can be reflected, either in a diffuse or specular manner.

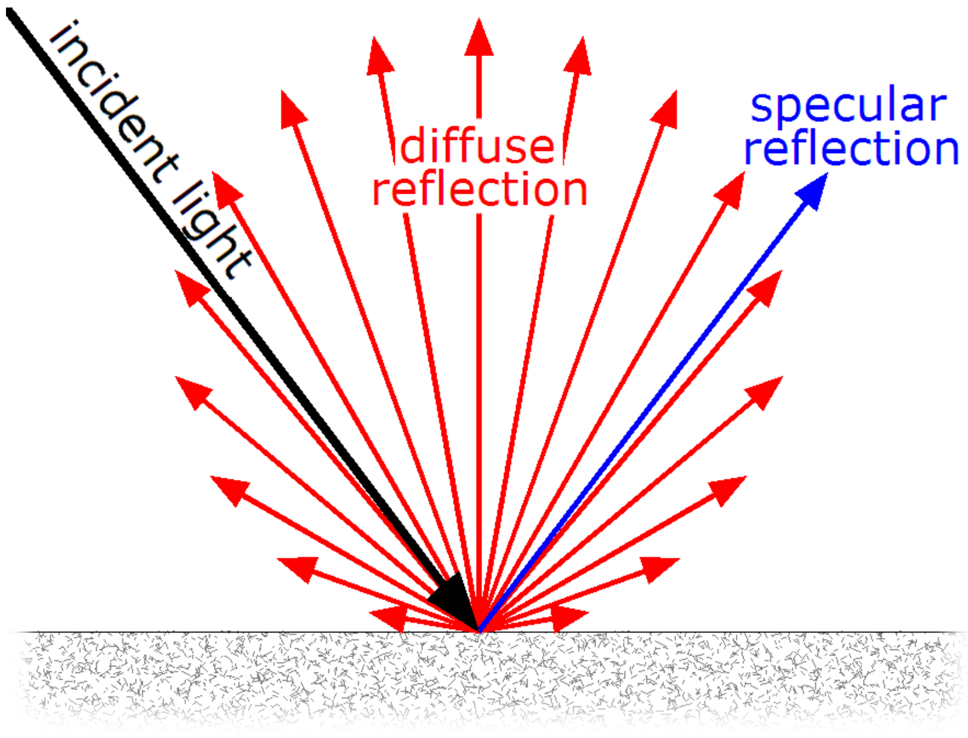


Figure B.1: Diffuse and specular reflection

B.2 Refraction

If the incoming ray is not reflected, it may either be absorbed by the material, or it will pass through the material. When a wave travels from one medium to another with different densities, the speed will change. This results in a change in the direction of the wave, also called refraction. Snell's law, (B.1), describes refraction where the angle of incidence θ_1 is relative to the angle of refraction θ_2. n_1 and n_2 are the refractive indices of the two media. These parameters expresses the slowdown of the light in the given media compared to the light speed in vacuum. For air and water, the value of these indices are 1 and 1.333.

$$\frac{sin(\theta_1)}{sin(\theta_2)} = \frac{v_1}{v_2} = \frac{n_2}{n_1} \tag{B.1}$$

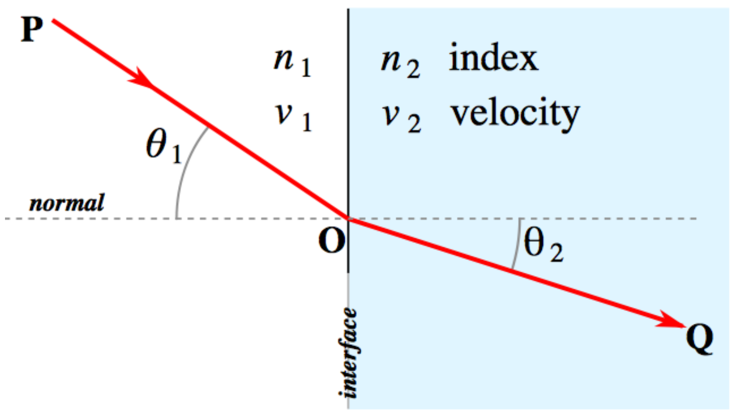


Figure B.2: Refraction of light between two media with different densities (Wikipedia:Refraction)

Figure B.2 is an example of how the light would bend when entering a material with greater density than its origin.

B.3 Fresnel

Reflection and refraction may occur at the same time if the material allows it. Water is both a transmissive medium and a great specular reflector. The Fresnel equations describe this behaviour.

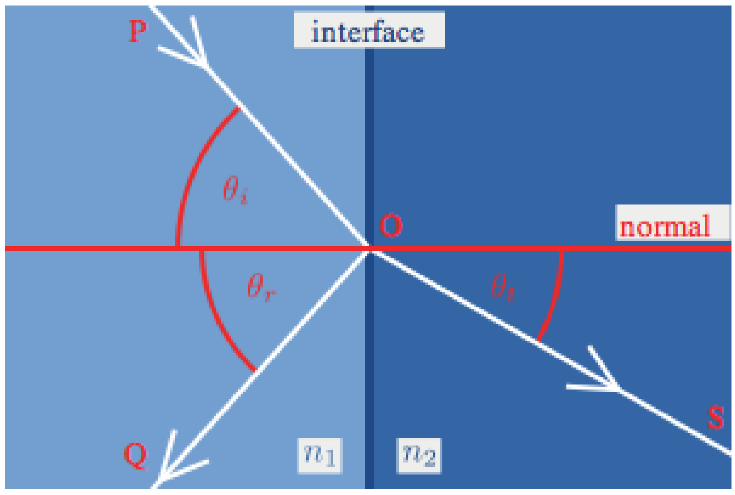


Figure B.3: Incoming light P gets reflected Q, and refracted (Wikipedia:Fresnel_equations)

In Figure B.3 symbols and behaviour of an incoming light ray is shown. The initial power of the incoming ray is split between the reflection and the refraction. Reflectance R is the fraction of initial power that is reflected, and transmittance T is the fraction of the initial power that is refracted. If the light is s-polarized the reflection coefficient is given by:

$$R_s = \left(\frac{n_1 cos(\theta_i) - n_2 cos(\theta_t)}{n_1 cos(\theta_i) + n_2 cos(\theta_t)} \right)^2 \tag{B.2}$$

If the light is p-polarized the reflection coefficient is given by:

$$R_p = \left(\frac{n_1 cos(\theta_t) - n_2 cos(\theta_i)}{n_1 cos(\theta_t) + n_2 cos(\theta_i)} \right)^2 \tag{B.3}$$

Due to conservation of energy the transmission coefficient is given by:

$$T = 1 - R \tag{B.4}$$

The reflection coefficient for unpolarized light is:

$$R = \frac{R_s + R_p}{2} \tag{B.5}$$

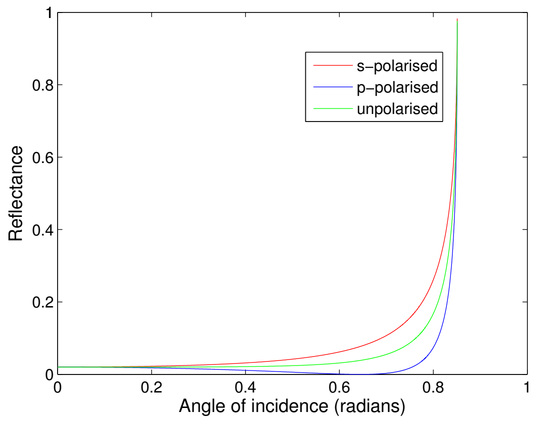


Figure B.4: Reflectance of water to air

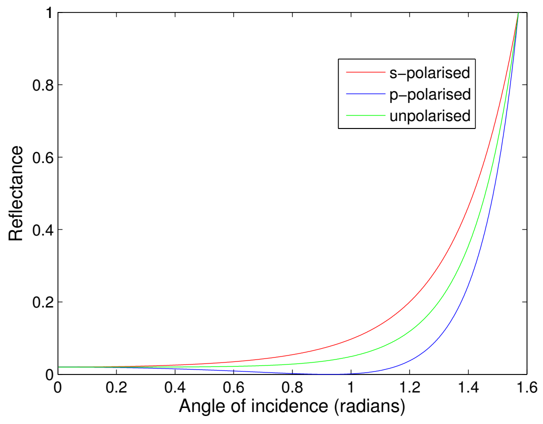


Figure B.5: Reflectance of air to water

Figure B.4 and Figure B.5 shows the reflectance at different angles of water-to-air, and air-to-water, respectively. The graphs in Figure B.4 converge to a critical angle. This angle can be found with (B.6). Total internal refraction occurs beyond this angle.

$$\theta_c = \arcsin(\frac{n_2}{n_1}) \tag{B.6}$$

Appendix C

Cube mapping

Environment mapping is an technique to render an object as if it were perfectly reflective, so the colors on the surface are those reflected to the eye from its surroundings. The technique is an approximation based on the assumption that the environment is far away comparing to the reflective object.

With a cube map, also called a skybox, the surroundings are represented by images, and these images are mapped onto a cube.

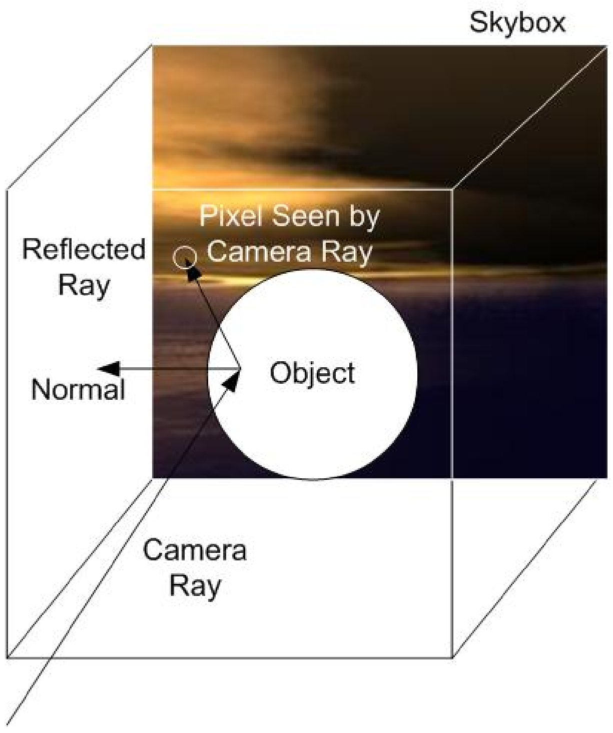


Figure C.1: An object in a cube map (Wikipedia:Cube_mapping)

In Figure C.1 an object is placed in a cube map environment. The reflective vector is calculated based on the viewing vector and the surface normal. The direction of the reflection vector determines which pixel from the cube map that is chosen. Based on the material properties of the object, the material uses the reflected value to calculate its final color.

Cube maps are supported in the graphics hardware, and integrated into shading languages. When the cube map is loaded into the GPU, a single function call with the cube map and the reflective vector as input is enough to get the reflected color.