Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Zsolt Mátyás

WATER SURFACE RENDERING USING SHADER TECHNOLOGIES

Consultant
Tamás Rajacsics

BUDAPEST, 2006

CONTENTS

ABSTRACT

In this paper I describe various real-time water rendering approaches using the graphics hardware. Not only is the scientific background presented, but the advantages and disadvantages of the different solutions as well. The key optical characteristics of water are reflection and refraction. Although, the complex optical behavior and physical interactions can be calculated absolutely accurate, the computational capacity of today's graphic cards is limited; the optimal compromise between realism and accuracy can be different depending on the target platform and required result. I present a wide range from completely simple to incredibly complex alternatives which can be basis for interactive water rendering. The most commonly used approaches are based on Perlin noise, Fast Fourier Transformations, Navier-Stokes Equations or on Particle System. The two demo applications try to demonstrate the main ideas.

Keywords

Water mathematics, Water shader, Graphics hardware, HLSL, Optical effects, Reflection, Refraction, Critical angle, Fresnel term, Surface waves, Gerstner waves, Gird, Projection, Fast Fourier Transformation, Navier-Stokes equations, Pixel shader, Vertex Shader, Level of Detail.

1 INTRODUCTION

Innovation is perhaps the most significant in the computer world, especially among the different kind of graphics cards. The brand new shader technologies made several techniques possible which became basis of new water rendering techniques in movies, computer games and other kind of terrain creation. These new water simulations make the virtual realities more and more alive.

To render realistic water surfaces three components need to be addressed:

- Representation of the water
- Optical behavior
- Wave motion

These components absolutely depend on each other. The solutions for optical behavior and wave motion simulation can be selected depending on the water representation after taking into account the possible approaches and the desired result as well.

The first attempts of water rendering were visually not too convincing, as computational power of the CPUs was much more limited than today, and no graphics card existed with 3D support. Since that, the evolution of GPUs introduced numerous innovations which not only boosted but also changed the rendering methods. Though, today's CPUs are much faster, exploitation of the parallel computation method of the GPUs instead of using the CPU can result much more realistic real-time animations.

One of the first key points in the water rendering literature was *Jerry Tessendorf: Simulating Ocean Water* (2001). His approach combined a complex water representation and a practical ocean wave algorithm. The results published in that paper were used several times later, for example, in *Deep-Water Animation and Rendering* (2001) by Jensen and Robert Golias. They demonstrated the application of another wave simulation method (based on Fast Fourier Transformations) and discussed several smaller ingredients of natural water rendering as well. New approaches were published in *Interactive Animation of Ocean Waves* (2002) by Damien Hinsinger, Fagrice Neyret and Marie-Paule Cani, and in *Real-Time Water Rendering – Projected Grid Concept* (2004) by Claes Johanson as well. Their techniques and optimizations reduce the necessary computations and enhance the real-time visual results by this. The latest games and other graphics applications are really convincing, they can create extremely spectacular water scenes. The only drawback is that, because of commercial reasons, they don't publish the details of their techniques. In the following chapters I will discuss the basics of the latest water rendering approaches which can be extended to realistic water rendering under diverse conditions.

In chapter 2, I try to summarize the main scientific laws, equations and methods which are the most important for water rendering. Chapter 3 attempts to introduce the basics of the graphics hardware focusing its operating method. The most commonly used alternative approaches for real-time water rendering are described in chapter 4, ranging from basic and simple methods to extremely complex solutions. Chapter 5 and 6 describes the steps I used in my demo applications: one of them is a lake water shader, the other focuses on ocean waves.

I encourage you to check the homepage of this project and comment the articles as well. Also several animations, numerous screen-shots and lots of references can be found on the site: http://habibs.wordpress.com.

2 WATER MATHEMATICS

2.1 Introduction

You have seen many different kinds of water in your life already, but not everyone knows what makes natural water look realistic. Different natural water types have several properties in common but they are absolutely different in others. The key ingredients of natural waters are:

- Reflecting the objects visible above the surface
- Refracting the objects under the surface
- Multiple reflections and refractions
- Appropriate ratio between reflection and refraction: the Fresnel term
- Color modification, dirtier water - water-fog
- Moving surface – various kind of wave motion
- Specular reflection
- Deep water phenomena including caustics, spray, Godrays, foam and the Kelvin wedge

2.2 Optical Effects of Water Surfaces

2.2.1 Reflection

Reflection is the phenomena when the wave front changes direction at an interface between two different media, in the way that the wave front returns into the medium from which it originated. In some aspects the surface of the water acts like a perfect mirror. Electromagnetic waves of light are reflected on the surface. The physical law for this: the angle of reflection equals the angle of incidence, and we measure these angles to the normal-vector of the surface, namely the two angles α and β are equal, as shown on Figure 2-1:

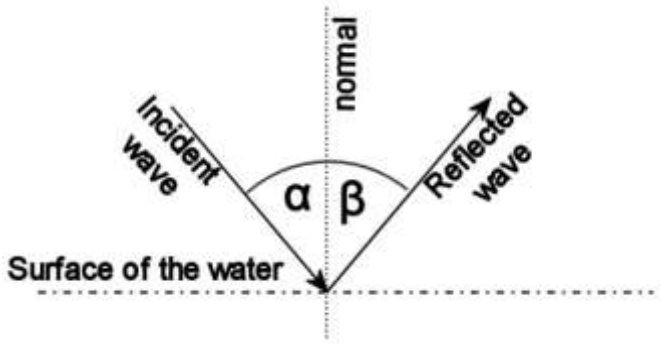


Figure 2-1: The law of reflection. The angle of incident (α) and reflectance (β) are equal.

Taking into consideration only the reflection behavior of the water it is easy to calculate the color of the pixels on the water surface. Mirroring the position of the camera to the plane of the surface gives the exact location of the virtual view: just determine which color has the object which is visible through every pixel of the water from the virtual view. This idea is visualized on Figure 2-2:

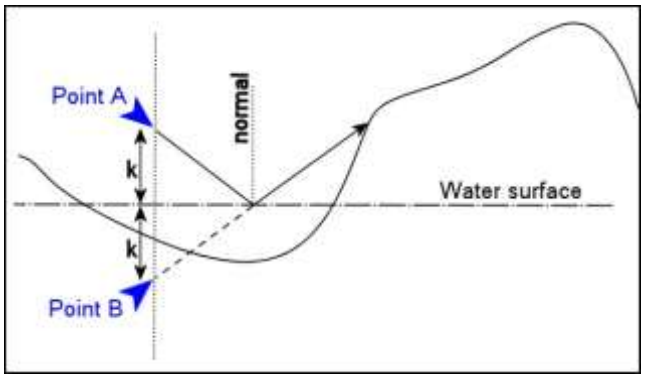


Figure 2-2: Getting the reflected colors on the water surface.

If the camera is in point A, the perceived color on the water surface will be the color of the object visible from point B through the same intersection point. Point B is exactly the same far from the plane of the water as point A - the two distances are marked with letter "k" on the figure.

2.2.2 Refraction

The speed of electromagnetic waves is different in different media. The change happening when it passes from one medium to another causes the phenomenon of refraction. The Snell's law (named after the Dutch mathematician Willebrord Snellius) describes the relationship between the angle of incidence and refraction: the ratio between them is a constant depending on the media or more exactly the ratio of the sines of the angles equals the ratio of velocities in the two media or equals the opposite ratio of the refraction indices of the media:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{v_1}{v_2} = \frac{n_2}{n_1}$$

or

$$n_1 \sin\theta_1 = n_2 \sin\theta_2 \ .$$

Where Vs are the velocities, Ns are the refraction indices and θs are the angle of incidence and refraction. These angles are measured in respect to the normal-vector of the boundary between the two media.

According to this law the direction of the wave can be refracted towards or from the normal line, depending on the relative refraction-indices of the media. Figure 2-3 shows an example:

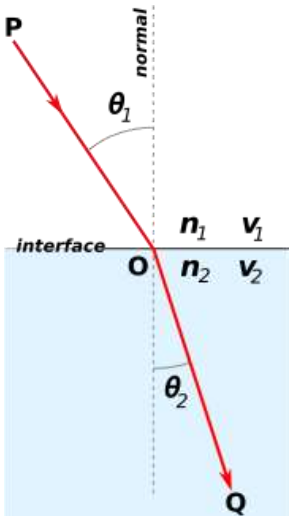


Figure 2-3: The Snell's law. In this case the velocity is lower in the second medium (V2<V1), the ray in the second medium is closer to the normal-line.

For our intended use, we need the values of these indices for only two materials. The index of refraction for air is 1, for water is 4/3.

2.2.3 The Third Dimension

The previously discussed examples and definitions were only two-dimensional, but for the real world, everything needs to be formed into 3D. The following equations describe the direction after refraction in 3D. For explanation, see [MFGD]. Let **s** be the incoming ray of light (vector), **t** the transformed ray of light, **n** the normal-vector of the surface, η_1 and η_2 are the velocity indices. The transformed vector has two components: one parallel and one perpendicular to *n*. This can be written in the following form:

$$\mathbf{t} = -\mathbf{n}\cos(\theta_2) + \mathbf{m}\sin(\theta_2)$$

To calculate the two coefficients, we need to use the fact that only the angle along the surface changes, and not the entire direction. **m** can be defined as follows:

$$\mathbf{m} = \mathrm{perp}_{\mathbf{n}}\mathbf{s} / \sin(\theta_1) = \mathbf{s} - (\mathbf{n} \bullet \mathbf{s})\,\mathbf{n} / \sin(\theta_1)$$

Using the previous equations [MFGD] described the following result:

$$\mathbf{t} = -\mathbf{n}\left(\sqrt{1-\frac{\eta_1^{\,2}}{\eta_2^{\,2}}\left(1-(\mathbf{n}\bullet\mathbf{s})^2\right)}+\frac{\eta_1}{\eta_2}(\mathbf{n}\bullet\mathbf{s})\right)-\mathbf{s}\frac{\eta_1}{\eta_2}$$

This equation contains the possibility to have negative square root, which means that the equation is not defined for every angles and coefficients. The physical reason for this is described in the following paragraph.

2.2.4 Critical Angle

There is one more important phenomenon that I need to mention in connection with refraction. If the light is coming from a media with lower velocity, the angle will change from the normal, so the angle between the normal and the beam will be bigger in the target media.

This means that to get 90 degree in the target medium a smaller angle is enough in the source medium. This specific angle is called critical angle. The critical and any higher angle results a refraction vector which is parallel to the surface of the media. If this happens, the wave will be refracted along the border of the media, it will not intrude the target medium, the so-called total internal reflection will occur. This critical angle is about 50 degree at a water-air boundary, which is shown on Figure 2-4:

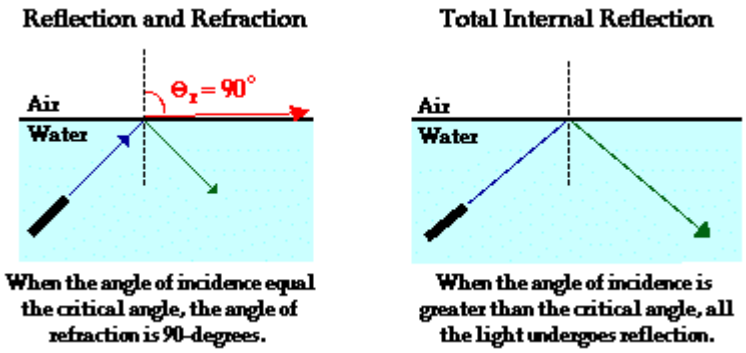


Figure 2-4: The critical angle

2.2.5 Multiple Reflection and Refraction

The light beams are reflected and refracted on the surface of the water, but to a certain amount transformed light beams meat the air-water border again and reflection and refraction happens newly. This is illustrated on Figure 2-5:

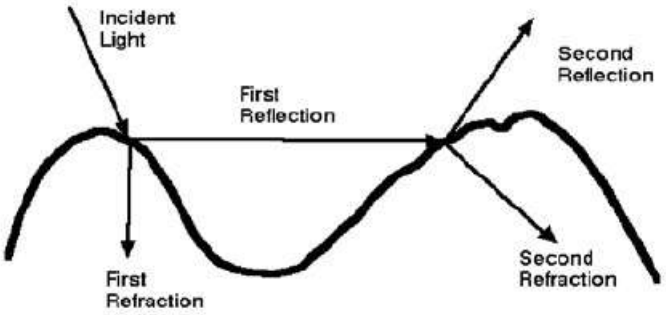


Figure 2-5: Multiple reflection and refraction.

2.2.6 The Reflection-Refraction Ratio: the Fresnel Term

The first two sections described reflection and refraction. They both happen to electromagnetic waves on the border of different media like on Figure 2-6:

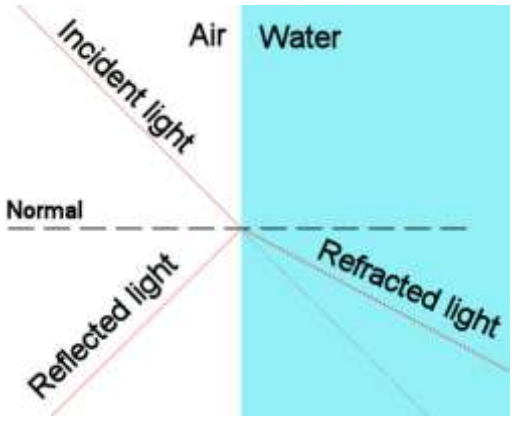


Figure 2-6: Both reflection and refraction happen on media boundaries.

But how to get the accurate ratio between reflection and refraction? Augustin-Jean Fresnel (/freI'nɛl/) worked out the laws of optics in the early 19th century. His equations give the degree of reflectance and transmittance at the border of two media with different density. Derivation of them is outside the scope of this paper.

The wave has two components: a parallel and a perpendicular. E_i is the amplitude of the incident wave, E_r and E_t are the amplitudes of the reflected and the transmitted wave. θ_i, θ_r and θ_t are the angles between the surface normal and the beam of incidence, refraction and transmittance.

$$\mathbf{r = E_r/E_i}$$
$$\mathbf{t = E_t/E_i}$$

For the perpendicular component the equations are the following:

$$\mathbf{r = [nicos(\theta i) - ntcos(\theta t)]/[nicos(\theta i) + ntcos(\theta t)]}$$
$$\mathbf{t = [2nicos(\theta i)]/[nicos(\theta i) + ntcos(\theta t)]}$$

The next equations are showing the properties of the parallel components:

$$\mathbf{r = [n_i cos(\theta_i) - n_t cos(\theta_i)]/[n_i cos(\theta_t) + n_t cos(\theta_i)]}$$
$$\mathbf{t = [2n_i cos(\theta_i)]/[n_i cos(\theta_t) + n_t cos(\theta_i)]}$$

The more elegant version of them are:

$$\mathbf{r_{perpendicular} = -[sin(\theta_i) - sin(\theta_t)]/[sin(\theta_i) + sin(\theta_t)]}$$

and

$$\mathbf{r_{parallel} = [tan(\theta_i) - tan(\theta_t)]/[tan(\theta_i) + tan(\theta_t)]}$$

If the light is polarized to have only perpendicular component, we call it S-polarized. Similarly, if it has only parallel components, it is called P-polarized. Figure 2-7 shows example coefficients depending on the angles for both S and P polarized cases:

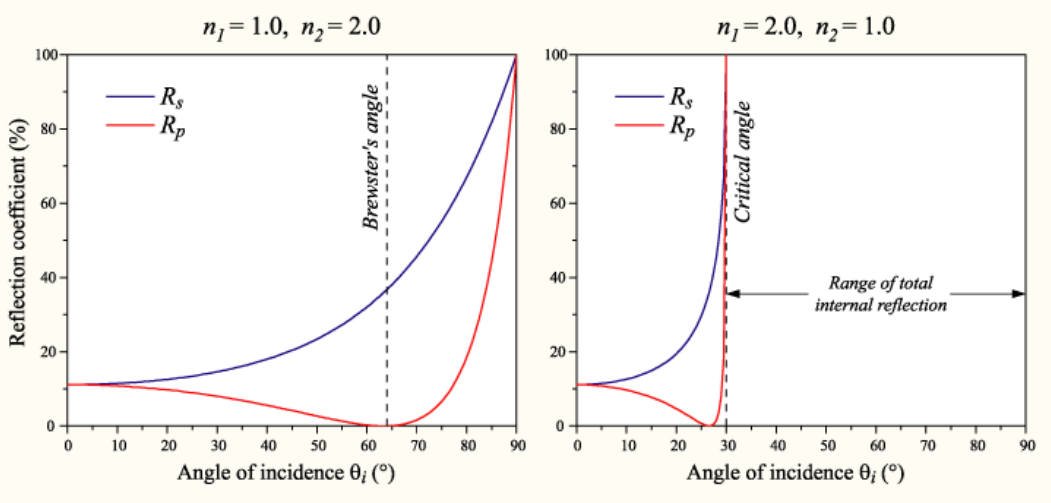


Figure 2-7: Example Fresnel ratios. As visible on the figure, in case of moving from denser medium to a less dense one (on the right), the reflection coefficient is 1 above the critical angle. This phenomenon is known as total internal reflection, as mentioned earlier.

2.3 Moving Water

2.3.1 About Waves

Describing ocean waves is a huge challenge. It has several different components and their cooperation results in a very complex system. Basically there are two different kinds of mechanical wave motion: longitudinal and transverse. The direction of oscillation relative to the wave motion distinguishes them. If the oscillation is parallel to the wave motion it is called longitudinal wave, as shown on Figure 2-8:

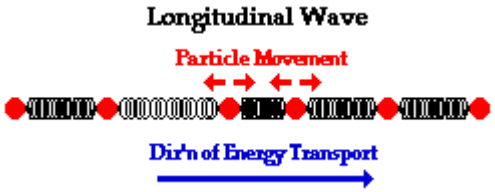


Figure 2-8: Longitudinal wave

If the oscillation is perpendicular to the wave motion it is called transverse wave, as visualized on Figure 2-9:

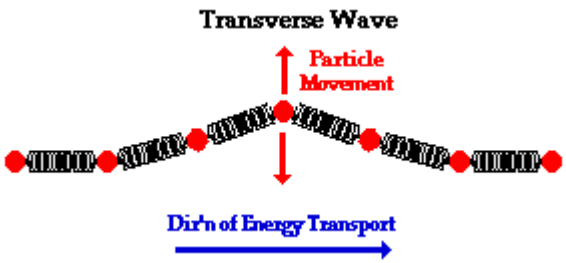


Figure 2-9: Transverse waves

Examples are, for example, sound wave in air for longitudinal waves and the motion of a guitar-string if you play the guitar for transverse waves. For both kinds of waves amplitude is the maximum displacement of a wave from the equilibrium and wavelength is the shortest length between two points of the wave which are in the same wave-phase. The frequency shows the number of wave cycles in a second. It is easy to calculate the speed of a wave from these data: wave speed equals the product of frequency and wavelength.

The blowing wind and gravity together forms the ocean waves which propagate along the surface of the water and air. This compound system has both longitudinal and transverse components and makes water particles move in a circular path. The closer to the surface a particle is, the bigger the radius of the motions becomes. This kind of wave is called surface wave and is illustrated on Figure 2-10:

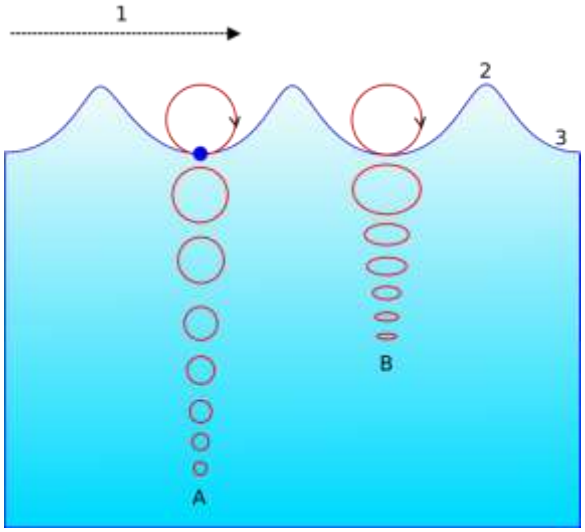


Figure 2-10: Surface waves.

At point ***A***, where the water is deeper, the path is circle, at point ***B***, where the water is shallow, the path of the motion becomes elliptic with decreasing water depth. The arrow ***#1*** shows the direction of propagation. ***#2*** shows the crest of the wave, ***#3*** shows the wave trough.

The following equation describes the dispersion relationship of the surface waves (for more details, see [TSWHP]):

$$\omega^2 = gk + \frac{\gamma k^3}{\rho},$$

where ω is the angular frequency, g is the gravitational acceleration, k is the wavenumber, γ is the surface tension, and ρ is the density. Solving for ω gives

$$\omega = \sqrt{gk + \frac{\gamma k^3}{\rho}}.$$

2.3.2 Compound Systems – Summation of Different Waves

The previously discussed theoretical background is enough to describe the motion of ocean waves, but we need to use more components with different amplitudes and wavelengths to get a more realistic result, as shown on Figure 2-11:

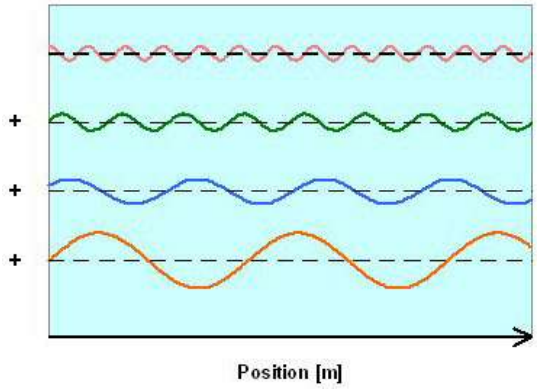


Figure 2-11: Waves with different amplitudes and wavelengths.

The sum of the components results the next wave on Figure 2-12:

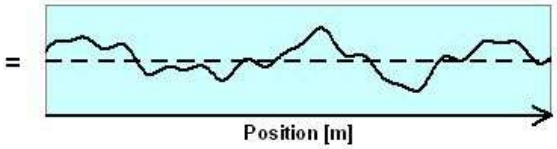


Figure 2-12: Sum of waves.

In the tree-dimensional world different components have not only different amplitudes and wavelength, but different directions as well. The dominant direction will be the one with bigger amplitude and longer period. Figure 2-13 shows three components:

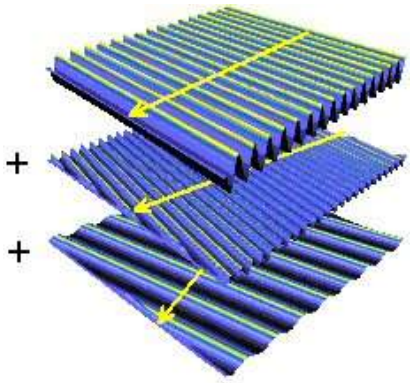


Figure 2-13: Three-dimensional waves.

And the sum of the components can approximate the real ocean surface realistic, as they do on Figure 2-14:

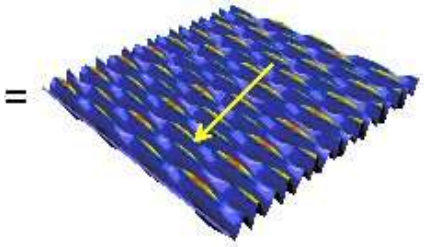


Figure 2-14: Sum of three-dimensional waves.

2.3.3 Gerstner Waves

A Czech scientist, Jozef Gerstner obtained the first exact solution describing water waves of arbitrary amplitudes in 1802. His model also describes the cyclonidal movement of the surface waves. In this model the water depth is large compared to the wave length. The resulted curve is also called trachoid.

Displacements are defined with the following equations:

$$x = X_0 - (k/k_0) * A \sin(k * X_0 - \omega t)$$
$$y = A \cos(k * X_0 - \omega t)$$

where X_0 is the undisturbed surface point, A is the wave amplitude, k is the wave vector and k_0 is the magnitude.

Gerstner waves become close to sinusoidal if the amplitudes are very small, but they break if the amplitudes are bigger, as visualized on Figure 2-15:

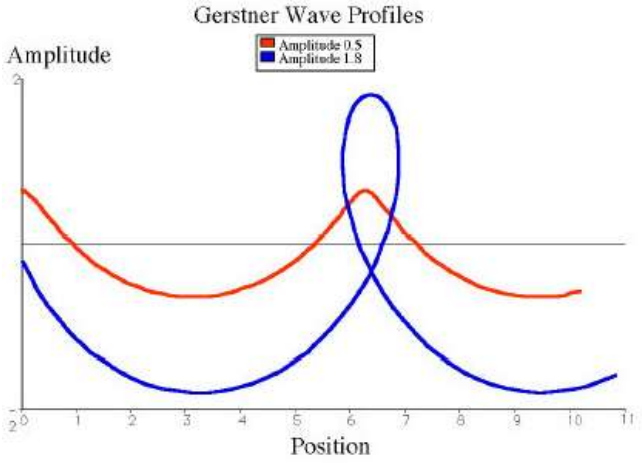


Figure 2-15: Gerstner waves with different amplitudes.

These qualities allow Gerstner waves to describe various surface waves under different conditions.

For more details, see [IAoOW] or [GW].

2.3.4 The Navier-Stokes Equations

Navier-Stokes Equations (NSE) are nonlinear partial differential equations and describe the motion of incompressible viscose fluids. In NSE there are three types of forces acting:

- **Gravity**: $F_g = \rho G$, where ρ is the density and G is the gravitational force (9.81 m/s^2).
- **Pressure forces**: These forces act inwards and normal to the water surface.
- **Viscose forces**: These are forces due to friction in the water and acts in all directions on all elements of the water.

The time dependant chaotic, stochastic behavior of fluids is called turbulence. Navier-Stokes equations are thought to describe the phenomena, but it is not answered yet, how to decide whether smooth, physically reasonable solutions exist for the equations. Actually a 1,000,000

dollar prize is offered to whoever makes preliminary progress toward a mathematical theory which will help in the understanding of this phenomenon. Further discussion is outside the scope of this paper. The Navier-Stokes equations are given by:

$$(1) \qquad \frac{\partial}{\partial t}u_i + \sum_{j=1}^{n} u_j \frac{\partial u_i}{\partial x_j} = \nu \Delta u_i - \frac{\partial p}{\partial x_i} + f_i(x,t) \qquad (x \in \mathbb{R}^n, t \geq 0),$$

$$(2) \qquad \operatorname{div} u = \sum_{i=1}^{n} \frac{\partial u_i}{\partial x_i} = 0 \qquad (x \in \mathbb{R}^n, t \geq 0)$$

with initial conditions

$$(3) \qquad u(x,0) = u^{\circ}(x) \qquad (x \in \mathbb{R}^n).$$

Where, $u^o(x)$ is a given, C^∞ divergence-free vector field on R^n, $f_i(x, t)$ are the components of a given, externally applied force (e.g. gravity), ν is a positive coefficient (the viscosity), and

$$\Delta = \sum_{i=1}^{n} \frac{\partial^2}{\partial x_i^2}$$

is the Laplacian in the space variables. For more details, see [NSEP], [FDfP] or read about an implemented version: [GPUGEMS].

2.4 Various Water Phenomena

2.4.1 Specular Lights

Materials having a flat surface (e.g.: leather, glass and water also) present an interesting phenomenon which I have not mentioned yet. There are several different reflection models, some of them make the created image much more realistic while others help to improve the smaller details. One of these in the second group is specular reflection.

Materials like sand have irregular, bumpy surface and this makes incoming light to be reflected in every directions. This is shown on Figure 2-16:

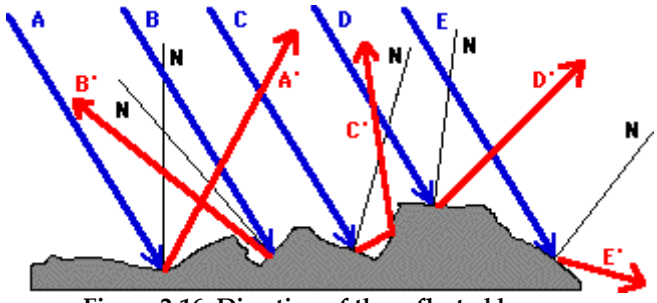


Figure 2-16: Direction of the reflected beams.

But if the material has flat surface, the light waves will be parallel after reflection as well. Because of this property shiny, bright spots will be formed for example on the leather, on different metals or on the water if it is illuminated from a certain angle. The Phong illumination model describes this in the way which is most commonly used in 3D computer graphics. Phong Bui Tong developed his model in 1975 and it is still very popular. According to this model the intensity of point has three components: an ambient, a diffuse and a specular component and the specular highlight is seen when the viewer is close to the direction of

reflection. The intensity of this kind of light falls off sharply when the viewer moves away from the direction of specular reflection. Figure 2-17 shows the vectors for describing specular highlights.

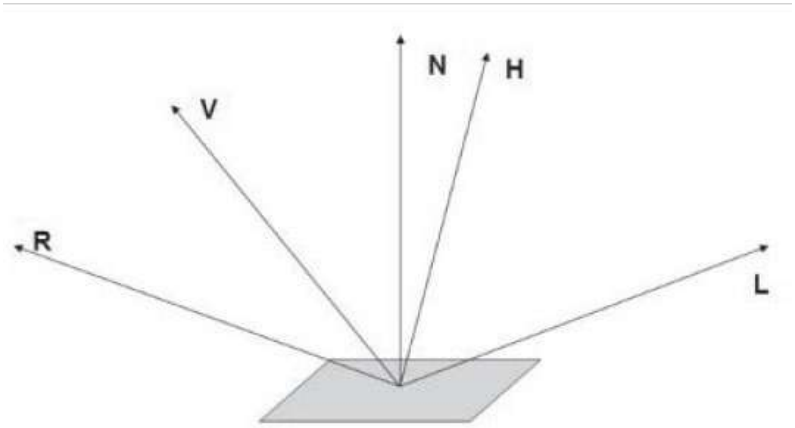


Figure 2-17: The vectors for describing specular highlights. Vector L points towards the light-source, V towards the viewer, N is the normal-vector of the surface, R is the direction of reflection while H halves the angle between L and V.

The approximation of the falloff of the intensity in the Phong model uses the power of the cosine of the angle. The specular part of the original formula looks like this:

$$k_{spec}\ cos^{n}(\beta)$$

where β is the angle between R and V; k_{spec} is the specular coefficient. The exponent n can influence the sharpness of the falloff. A bigger exponent can describe a shinier surface with less gentle falloff. The dot product of two vectors equals the cosine of the angle between them, so the formula can be written in the following form:

$$k_{spec}\ (V \cdot N)^{n}$$

Where "·" is the dot product.

With the diffuse reflection model the Phong illumination model is the following (I means intensity):

$$I = k_{ambient} * I_{ambient} + (Ip/(d))\ [k_{diffuse} * (N \cdot L) + k_{specular} * (V \cdot R)^{n}]$$

Where Is are the different intensities, ks are the ambient, diffuse and specular coefficients, and · is the dot product. Namely, the intensity of a point is equal to the sum of the ambient light-intensity and the sum of the diffuse and specular intensity scaled to the distance of the light-source.

2.4.2 Caustics

A wavy water surface presents caustics: the light beams are refracted in very choppy directions, the coincidence of rays intensifies, leading to very bright regions on every surface under the water as on Figure 2-18:

Figure 2-18: Caustics.

Formation of caustics is illustrated on Figure 2-19:

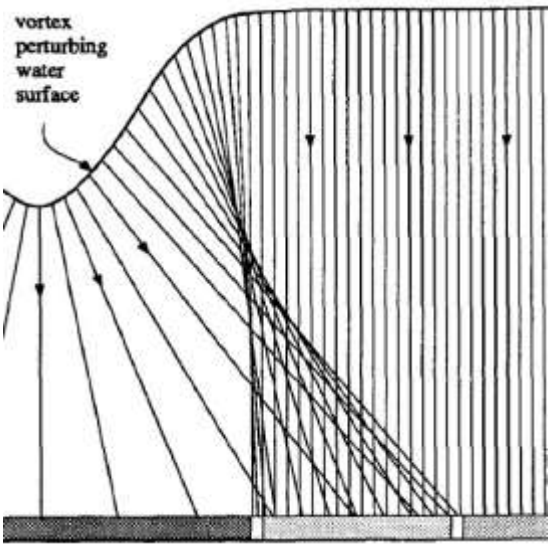


Figure 2-19: Formation of caustics. Caustics are bright regions where the light beams coincident

2.4.3 Godrays

The same physical effects which cause caustics can create Godrays (described in [DWAaR]). The changing water surface focuses and defocuses light rays. The small particles floating in the water can get into these focus points and become visible for a short period. The continuously changing patterns created by these effects are called Godrays which are visible if you are looking from underwater towards the light source. A rendered example is shown on Figure 2-20:

Figure 2-20: Godrays. The image was created by the Typhoon engine. [TYPHOON]

2.4.4 Whitecaps and Foam

Breaking waves produce foam, and the scummy parts of the breaking waves are called whitecaps. According to [RNW], the area of whitecaps depends on the temperature of the water and the air and on water chemistry. They used an empirical formula to approximate the fraction of the water covered by foam that modifies optical properties on the water's surface:

$$f = 1.59 * 10^{-5} U^{2.55} \exp[0.086 * (T_w - T_a)]$$

Where f is the fractional area, U is the wind speed, T_w and T_a are the water and air temperature in degrees Celsius.

2.4.5 The Kelvin Wedge

On open water, moving ships generate waves. These waves cannot be in any reasonable approximation treated as exclusively longitudinal. The phenomenon, the so-called Kelvin wedge was first analyzed by Lord Kelvin. An ideal example is visualized on Figure 2-21:

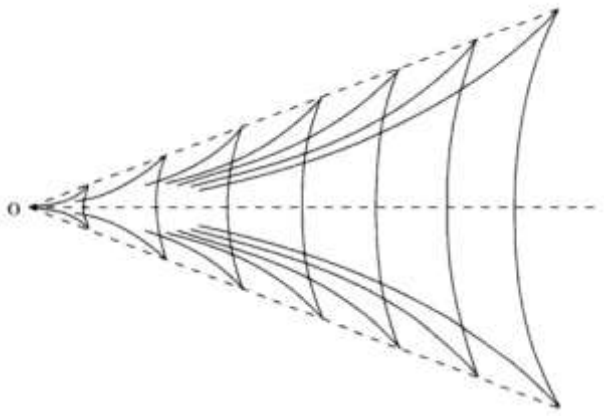


Figure 2-21: Ideal form of the Kelvin wedge.

The complicated wave pattern behind a ship is influenced by the viscosity of the water, by the moving directions, gravity, not to mention non-linear effects which become significant when the amplitude is large. Stern waves and bow waves are superposed on one another, and not infrequently, other wave systems may be discerned originating from somewhere between the bow and stern. The fact, that the angle enclosed by the Kelvin wedge is independent of the speed of the boot can be surprising, but the explanation is out of the scope of this paper. For more details, see [FDfP].

3 SHADERS

3.1 Background

The only thing you need to create an image from a virtual world is the order a color to every pixel on your display. This sounds easy, but for every pixel you need to determine which object is visible (which object is the nearest from the covering ones), what kind of light this surface gets and you need to calculate the real color of this point from all different kind of data, for instance the different angles (viewing angle, place of the light sources, normal-vector of the surface etc.) These calculations have every mathematical background already, you just have to implement them to get surprisingly realistic images very soon. Though, you can have some difficulties if you want to use these so-called ray-tracing techniques for motion pictures instead of still images. The main problem is that you want to order a color to every picture through these steps very fast, but it is impossible to repeat this calculations for every row (e.g.: 900) and every column (e.g.: 1440) of the image 30 times in a second to get a continuous video which needs 900 x 1440 x 30 = 3,888,000 iterations in each second through those time consuming mathematical operations. (The time is not an important factor if you do not need continuous motion picture. For still images you can rely on any picture-producing method.) To accomplish this challenge you need to use quicker alternatives, simplifications and heuristics to make a compromise between faster calculations and more realistic images.

3.2 Tessellation of the Virtual World

Virtual worlds comprise different surfaces and every surface is formed by triangles. The reason for this is that the most geometrical operations modify the base of the polygons, for example they transform a circle into an ellipse, a square into a different parallelogram. Polygons can lose all of their original attributes this way which makes them change during procedures and almost impossible to describe them with simple mathematical elements. (For instance, a circle can be described by a center point and a radius. A projection transforms it into an ellipse, and you cannot describe the ellipse with a center point any more, it needs to use two fixed points: focus.) But if you carry out the same operations on triangles, the result will be triangles and you will not lose the original form of the target. Hence, every geometrical object is tessellated into triangles (also called as triangulation) and all the procedures are carried out on these triangles, whose points are called vertices. An example of polygon triangulation is visible on Figure: 3-1:

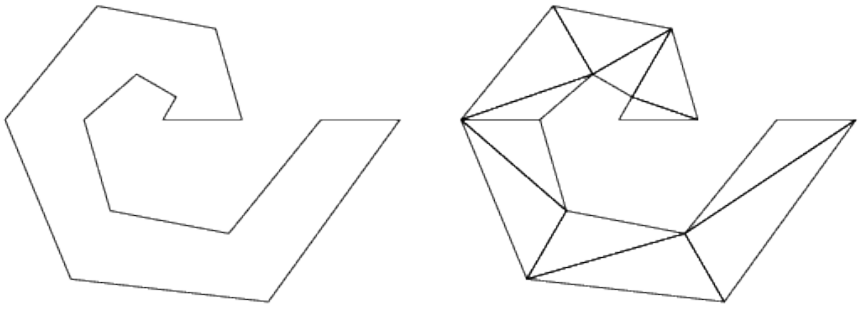


Figure: 3-1. Tesselation example

3.3 Shaders

Before DirectX 8.0 graphical processors were operating only a fixed processing pipeline which limited the arsenal of every kind of rendering technique. The introduced "programmable pipeline" added a new weapon into the hand of the programmers. The essential innovation was the two shaders: the vertex shader and the pixel shader. The vertex shader operates on every vertex of the virtual world while the pixel shader does the same for every pixel.

Vertex shaders are run once for each vertex given to the graphics processor. They transform the 3D coordinates of the vertices to the 2D coordinates of the screen. They manipulate properties such as position, color and texture coordinate, but cannot create new vertices. The output of the vertex shader goes to the next stage of the pipeline.

Pixel shaders calculate the color of every pixel on the screen. The input to this stage comes from the rasterizer, which interpolates the values got from the vertex shader to be able to determine not only the color of vertexes but all the pixels as well. Pixel shaders are typically used for scene lighting and related effects such as bump mapping and color toning.

The newly introduced Geometry shaders can add and remove vertices from a mesh. (Objects in the virtual world are formed by meshes.) Geometry shaders can be used to procedurally generate geometry or to add volumetric detail to existing meshes that would be too expensive to calculate on the CPU for real-time performance. If the geometry shader is present in the pipeline, it is a stage between the vertex shader and the pixel shader. Geometry shaders are beyond the scope of these articles.

Different DirectX versions and supported shader versions are shown in Appendix A.

3.4 The Actual Pipeline

To create an image on the screen, the CPU provides the necessary information of the objects: coordinates, color and alpha channel values, textures etc. From these data the Graphical Processing Unit (GPU) calculates the image through complex operations. The exact architecture may vary by manufacturers and by GPU families as well, but the general ideas are the same. The DirectX 10 pipeline stages to produce an image are visualized on Figure 3-2:

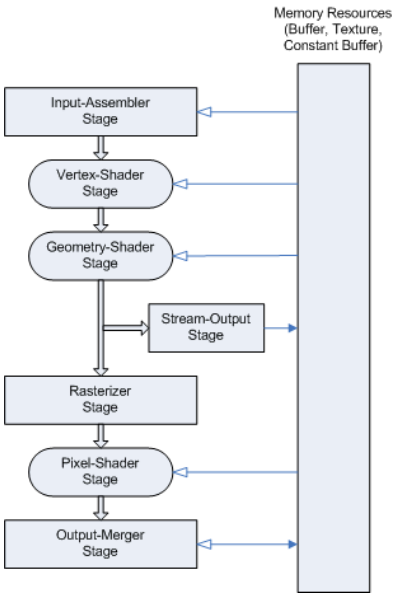


Figure 3-2. The graphical pipeline

1. Input-Assembler Stage - Gets the input data (the vertex information) of the virtual world.
2. Vertex-Shader Stage - Transforms the vertices to camera-space, lighting calculations, optimizations etc.
3. Geometry-Shader Stage - For limited transformation of the vertex-geometry.
4. Stream-Output Stage - Makes data-transport possible to the memory.
5. Rasterizer Stage - The rasterizer is responsible for clipping primitives and preparing primitives for the pixel shader.
6. Pixel-Shader Stage - Generates pixel data by interpolations.
7. Output-Merger Stage - Combines various types of output data (pixel shader values, depth and stencil information) to generate the final pipeline result.

Today's hi-tech graphic cards have only three programmable stages in order to reduce the complexity of the GPUs. The vertex processing stage (Vertex shader) and the pixel processing stage (Pixel shader) will be discussed here, for more details about geometry-shaders see [GS]. The two main application programming interfaces use different terms: pixel processing stage and Pixel shader in DirectX are called fragment processing stage and Fragment shader in OpenGL, respectively.

The Vertex shader engine gets the vertex data as input and, after several operations, writes the results in its output registers. The setup engine relays these data to the pixel shader engine after setting the hardware and interpolation. The pixel shader uses different kinds of registers and textures as input and from this information produces the color of the pixels (fragments) into the output register.

These shaders will be discussed in the next chapters in details.

3.5 The Vertex Shader

The first programmable pipeline stage is the Vertex shader which provides an assembly language to define every necessary operations to bypass the original ones and to be able to create absolutely unique graphical effects. With this freedom it is possible to perform lots of operations including:

- Texture generation
- Rendering particle systems (fog, cloud, fire, water etc.)
- Using procedural geometry (e.g.: cloth simulation)
- Using advanced lighting models
- Using advanced key-frame interpolation

I wrote some words about surfaces in the virtual world in the previous chapters. Because of mathematical reasons every surface is formed into triangles and operations are performed on the points of these triangles, which are called vertices. A vertex is a structure of data: position (coordinates), color and alpha values, normal vector coordinates, texture coordinates etc. but other required information can be stored among these values as well.

Operations of the Vertex shader are executed on each vertex, so the shader program has exactly one vertex as input and one vertex as output. The shader cannot change the number of the vertices: it cannot add or remove any of them, but it can change the coordinates, color or other values to get us closer to the final picture. If the vertex shader is used, the following parts of the fixed pipeline are bypassed:

- Transformation from world space to clipping space
- Normalization
- Lighting and materials
- Texture coordinate generation

Although the other parts of the fixed pipeline of the vertex stage are executed:

- Primitive assembly
- Frustum culling
- Perspective division
- View-port mapping
- Backface culling

I introduce here the key features of Vertex Shader version 1.1. Later versions are backward compatible, which means that everything defined here is correct and works well in newer versions as well.

The Vertex shader ALU performs the real operations (the vertex shader of the Ati X1900 is visualized on Figure 3-3):

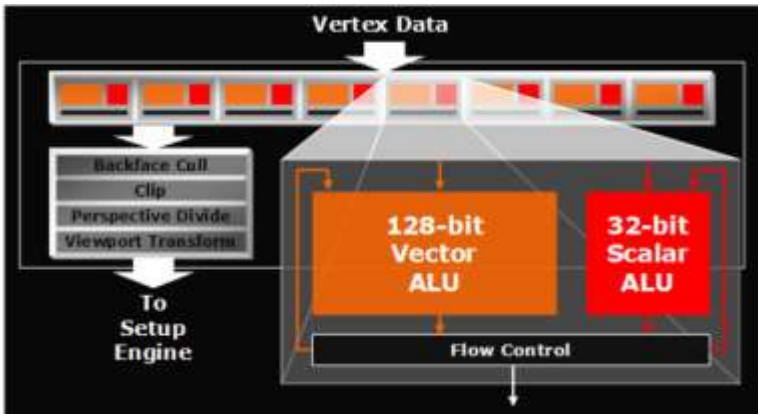


Figure 3-3: The vertex shader ALU.

It gets the values from input and parameter registers and writes the results to output registers with or without using temporary and address registers as shown on Figure 3-4:

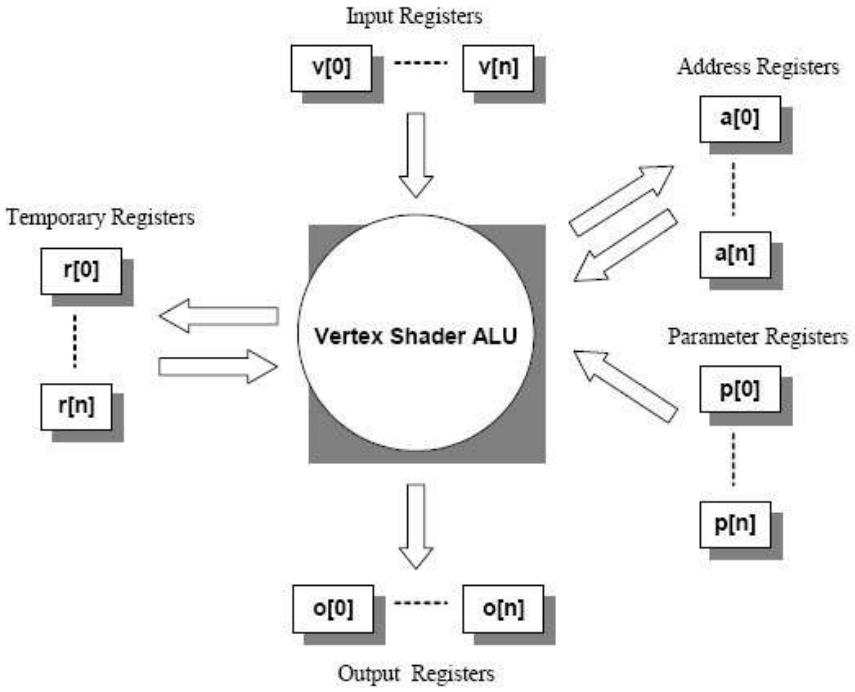


Figure 3-4: Registers of the vertex shader

Each of these registers can store four floating point numbers, for instance every input registers from V[0] to V[n] (this means that there are registers with number 0, 1, 2, …, n altogether: n+1 piece of them) can store four 32 bit-long floating point number. Input and parameter registers are read-only, output registers are write-only while address and temporary registers are both readable and writable by the ALU. Input registers store data about vertices. Parameter registers contain values which are valid for more calculations (shader constants), for example the world-view-projection matrix, light positions or light directions. Indexed relative addressing can be performed using the address registers. The shader works on one instruction but on more data parallel at a time, and generally can operate on whole matrices or quaternions. This data format is very useful because most of the transformation and lighting calculations require these kinds of data (vectors and matrices). The fact, that GPUs are designed to operate on more data in one clock cycle parallel enables them to calculate these special tasks much faster than the CPUs of the computers. The most important operations will be described in the next paragraph. Finally the results will be stored in the write-only output registers and passed on to the next stages of the fixed pipeline.

In DirectX 8, the pipeline was programmed with a combination of assembly instructions, High Level Shading Language (HLSL) instructions and fixed-function statements. With the latest APIs it is possible to use only HLSL; in fact, assembly is no longer used to generate shader code with Direct3D 10, but I introduce the assembly operations first. Vertex shader version 1.1 limits the number of instructions to 128 (later versions increase this number dramatically). Operations are in format:

```
OperationCode DestinationRegister , SourceRegister1 [, SourceRegister2 ] [, SourceRegister3]
```

Namely, first, the code of the operation comes followed by the name of the destination registers, and finally, the source registers, separated by commas. For example, the following operation adds the values stored in constant register c0 and input register v0 and writes the result into temporary register r0:

```
add r0, c0, v0
```

The most important instructions are shown in the next table (vertex shader V1.1):

Name	Description	Instruction slots
add - vs	Add two vectors	1
dp3 - vs	Three-component dot product	1
dst - vs	Calculate the distance vector	1
exp - vs	Full precision 2x	10
log - vs	Full precision log2(x)	10
m3×3 - vs	3×3 multiply	3
m4×4 - vs	4×4 multiply	4
mad - vs	Multiply and add	1
max - vs	Maximum	1
min - vs	Minimum	1
mov - vs	Move	1
mul - vs	Multiply	1
sge - vs	Greater than or equal compare	1
slt - vs	Less than compare	1
sub - vs	Subtract	1

These instructions are enough to make the most of the calculations but there were some which were complicated to perform, for example trigonometric functions needed to be approximated by different Taylor series. Later vertex shader versions made the job of the programmers more comfortable by supporting directly the computing sine and cosine, introducing flow control and new arithmetical instructions. For more details, check the [MSDN].
The output of the vertex shader is a set of vertices, which is passed to the next stages of the graphical pipeline. After interpolation, face culling, viewport mapping, homogeneous division etc. the vertices will arrive to the pixel shader stage.
For more details about vertex shaders, see [LVaPSPwD9], [NGSaR] or [RTSP].

3.6 The Pixel Shader (Fragment Shader)

Incoming vertices determine the general placement of the objects in the virtual world, but in the end every pixel need to be rendered on the output. The pixel shader makes it possible to order a color to every pixel of the image through the programmer defined instructions. Some of the possible effects are:

- Per-pixel lighting
- Advanced bump mapping
- Volumetric rendering
- Procedural textures
- Per-pixel Fresnel term
- Special shading effects

- Anisotropic lighting

The pixel shader operates on pixels before the final stages of the graphic pipeline (alpha, depth and stencil test). Logically, the output of the pixel shader is the corresponding color and depth value for each pixel, interpolated from the input vertices. Using pixel shader bypasses the following parts of the fixed-function pipeline:

- Texture access
- Texture applications
- Blending
- Fog application

The working method of the pixel shader's ALU is similar to the vertex shader's. Using the data in the input and parameter registers, the result will be calculated with or without the help of the temporary registers and texture samplers. It is finally written into the output register as shown on Figure 3-5: :

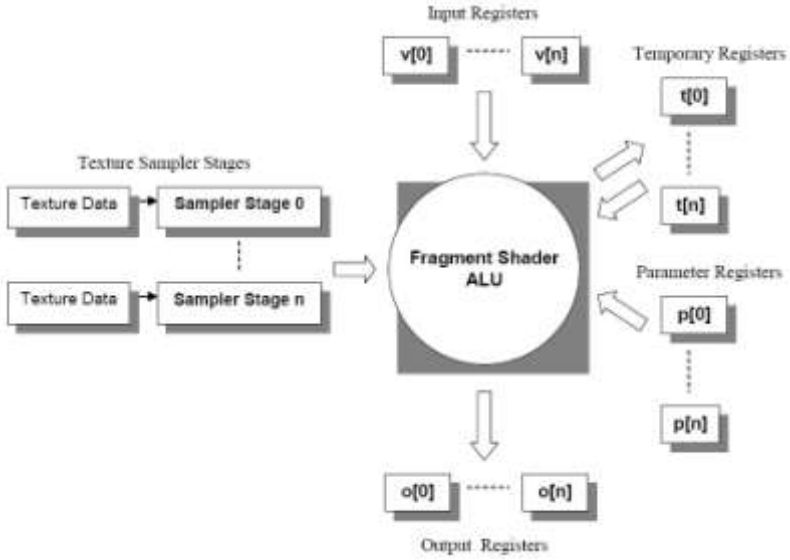


Figure 3-5: Thepixel shader ALU

As you see, the main difference between pixel and vertex shader is the possibility to use texture operations in the pixel shader. Defining a texture as a render-target enables the opportunity to reuse any pre-rendered picture or texture for later calculations as well. There are some other differences also, for example, the order of the different kind of instructions is limited and there are several modifiers and masks to slightly change the input or output reading methods in the pixel shader. For more details see the MSDN library.
Similarly to the vertex shader, the pixel shader supports a programming language to define the shader program. But the difference among instruction set versions is significant. The version 1.1-1.3 had a more complex instruction set, but only the reduced instruction set of the version 1.4 will be discussed here (later versions are with version 1.4 backward compatible). Most of the vertex shader instructions became available in the later pixel shader instruction sets, they are today very similar. The most important operations of pixel shader version 1.4 are visible on the next table:

Arithmetic instructions		Instruction slots
add - ps	Add two vectors	1
bem - ps	Apply a fake bump environment-map transform	2
cmp - ps	Compare source to 0	1

cnd - ps	Compare source to 0.5	1
dp3 - ps	Three-component dot product	1
dp4 - ps	Four-component dot product	1
lrp - ps	Linear interpolate	1
mad - ps	Multiply and add	1
mov - ps	Move	1
mul - ps	Multiply	1
nop - ps	No operation	0
sub - ps	Subtract	1
Texture instructions		
texcrd	Copy texture coordinate data as color data	1
texdepth	Calculate depth values	1
texkill	Cancels rendering of pixels based on a comparison	1
texld	Sample a texture	1

3.7 High Level Shader Language

High Level Shader Language or HLSL is a programming language for GPUs developed by Microsoft for use with the Microsoft Direct3D API, so it works only on Microsoft platforms and on Xbox. Its syntax, expressions and functions are similar to the ones in the programming language C, and with the introduction of Direct3D 10 API, the graphic pipeline is virtually 100% programmable using only HLSL; in fact, assembly is no longer needed to generate shader code with the latest DirectX versions.

HLSL has several advantages compared to using assembly, programmers do not need to think about hardware details, it is much easier to reuse the code, readability has improved a lot as well, and the compiler optimizes the code. For more details about improvements, see [MHLSLR].

I have to mention that Cg shader language is equivalent with HLSL language. HLSL and CG are co-developed by Microsoft and nVidia. They have different names for branding purposes. As a part of DirectX API, HLSL compiles only into DirectX code, while Cg compiles into both DirectX and OpenGL code.

HLSL code is used in the demo applications, so I briefly outline here the basics of the language.

3.7.1 Variable Declaration

Variable definitions are similar to the ones in C:

```
float4×4 view_proj_matrix;
float4×4 texture_matrix0;
```

Here the types are float4×4. This means, 4 x 4 = 12 float numbers are stored in them together, and depending on the operation type, they all participate in the operations. This means, matrix operations can be implemented by them.

3.7.2 Structures

C-like structures can be defined with the keyword *struct*, as in the following example:

```
struct MY_STRUCT
{
        float4 Pos : POSITION;
        float3 Pshade : TEXCOORD0;
};
```

The name of the structure is *MY_STRUCT*, and it has two fields (the names are *Pos* and *Pshade* and the types are *float4* and *float3*). For each field, storing-registers are defined after the colon (*:*). I discussed the possible register types in the chapter 3, although the possible register names vary on different Shader versions. The two types in the example are *float4* and *float3*, which means, they are compounded of more float numbers (tree and four floats) which are handled together.

3.7.3 Functions

Functions can be also familiar after using C:

```
MY_STRUCT main (float4 vPosition : POSITION)
{
        MY_STRUCT Out = (MY_STRUCT) 0;
        // Transform position to clip space
        Out.Pos = mul (view_proj_matrix, vPosition);
        // Transform Pshade
        Out.Pshade = mul (texture_matrix0, vPosition);
        return Out;
}
```

The name of the function is *main*, and its returns *MY_STRUCT* variable. The only input parameter is a *float4* variable called *vPositions*, and it is stored in the *POSITION* register. Two multiplications are also demonstrated in the example (*mul* operation), they are performed on different types: a matrix-vector multiplication is shown in the example. By changing only the variables, it is possible to multiply a vector with another vector, or two matrices with each other as well. A list of possible intrinsic functions can be found in Appendix B.

3.7.4 Variable Components

It is possible to get the components(x, y, z, w) of the compound variables, as vector and matrix components. It is important to mention, that binary variables are performed also per component:

```
float4 c = a * b;
```

Assuming *a* and *b* are both of type *float4*, this is equivalent to:

```
float4 c;
c.x = a.x * b.x;
c.y = a.y * b.y;
```

```
c.z = a.z * b.z;
c.w = a.w * b.w;
```

Note that this is not a dot product between 4D vectors.

3.7.5 Samplers

Samplers are used to get values from textures. For each different texture-map, which you want to use, a *sampler* must be declared.

```
sampler NoiseSampler = sampler_state
{
        Texture = (tVolumeNoise);
        MinFilter = Linear;
        MagFilter = Linear;
        MipFilter = Linear;
        AddressU = Wrap;
        AddressV = Wrap;
        AddressW = Wrap;
        MaxAnisotropy = 16;
};
```

3.7.6 Effects

The Direct3D library helps developers with an encapsulating technique called *effects*. *Effects* are usually stored in a separate text file with *.fx* or *.fxl* extension. They can encapsulate rendering states as well as shaders written in ASM or HLSL.

3.7.7 Techniques

An *.fx* or *.fxl* file can contain multiple versions of an effect which are called *techniques*. For example, it is possible to support various hardware versions by using more *techniques* in a single effect file. A *technique* can include multiple *passes*, and it is defined in each *pass*, which functions are the pixel shader and vertex shader functions:

```
technique my_technique
{
        pass P0
        {
                VertexShader = compile vs_2_0 vertexFunction();
                PixelShader = compile ps_2_0 pixelFunction();
        }
}
```

3.8 Conclusion

The main ideas were shortly introduced about HLSL in the previous paragraphs. Shader programs of the demo applications are written using HLSL. For more detailed information, see the corresponding chapter or one of the mentioned references ([IttD9HLSL] or [MHLSLR]). A useful HLSL tutorial can be found here: [HLSLI]. For a general comparison of shader version 2.0 and 3.0, see [NGSaR].

4 ALTERNATIVE SOLUTIONS

4.1 Introduction

There are various techniques which can function as a core for water surface rendering. Some of the most important methods are Perlin noise, Fast Fourier syntheses and Navier Stokes equations. Different water representations require absolutely different computational power and provide different level of realism. It is also possible to combine various approaches, to get to the best compromise between them. In this chapter, several solutions are introduced as possible approaches for our mission. They are based on the previously discussed mathematical background, and range from simple to extremely complex computations.

The following parts of water rendering will be discussed in this chapter:

- Water representations - we can use several methods to describe our water surface, but in the end, everything needs to be described by vertices to be able to render the result. Grids and particle systems are the most popular ways for this.
- Water simulation approaches - which describe the water waves and gets everything in motion. The different solutions can be useful under different conditions, and complex systems can be built by combining them.
- Reflection rendering techniques.
- Fresnel term approximations.
- Rendering various water phenomena. Effects, for example, splashes, caustics and Kelvin wedge are discussed in this part.

4.2 Water Representations

4.2.1 3D Grids

Representing water by three-dimensional girds make various realistic water behavior simulation possible. The main idea is simple: we determine the physical forces and compute all their effects on the elements of the grid. Although they are easy to describe, the computations can be expensive. Physical simulations must be precise, but for rendering water surfaces, we don't need to be so accurate. If the extreme computation expense is reasonable, it can be used for rendering of small areas of water, and in this case, for example, the Navier-Stokes equations can be nicely applied. The details about physical simulations are out of the scope of this paper, I discuss here the solutions for real-time rendering.

Although 3D grids can represent only small amounts of water in real-time performance, pre-rendering calculations can be computed by them for higher realism. Rendering underwater textures, caustics formation, splashes are just some of the possible effects which can have pre-rendering phases to get higher performance during the real-time animations. Several paper writes about these possibilities (such as [aEMfRUOEUGH] and [ESoLBoWbCTaTDT]), but they are outside the scope of this paper. For our intended use, simpler water representations are needed.

4.2.2 2D Grids - Hightmaps

3D grids are accurate approximations of water volumes, but if we accept some limitations, we can use a simpler solution. To be able to render the water surface, we only need to know the shape of it, in our case, how high the water is at a given (x,y) coordinate. This means that the

volume can be simplified to a height-field using a function of two variables that return the height for a given point in two-dimensional space. This representation for water surfaces has restrictions compared to 3D grids. As a function can return only one value, a height field can represent one height value for a given (x,y) coordinate. This means overlapping waves can be described this way, as shown on Figure 4-1:

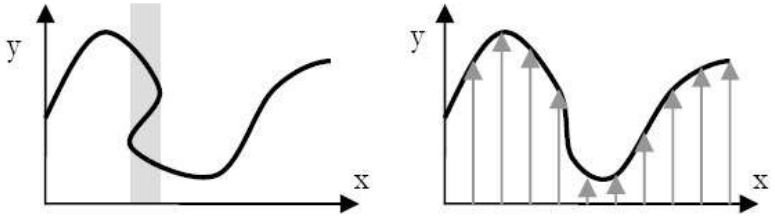


Figure 4-1: Hightfield limitations. As the height field stores only one height value for a coordinate like on the right image, overlapping parts of the surfaces cannot be described as on the left one.

The main advantage of 2D grids compared to 3D grids is that it is easier to use and a much simpler data structure is appropriate to store it. If the height field is stored in a texture, it is usually called height map. Similarly, the rendering process is called displacement-mapping as the original geometry is displaced by the amount stored in the height map.

Different optimization techniques are used to have better real-time performance. For example, if the height-map is defined by a continuous function, it is not needed to render and calculate the entire water surface, rendering the visible part is enough. In other scenarios, for instance, when Navier-Stokes Equations are used, every cell of the height map needs to be updated, even if they are not visible by the camera. Some optimization method is discussed in the following paragraphs.

4.3 Performance Optimization

As graphic hardware processes triangles, there is no way to avoid surface tessellation. Real water surfaces are continuous, but interactive computer graphics need a polygonal representation so the surface must be discretized. More triangles describe more details in the virtual world, although more triangles are made up of more vertices. Every graphics programmer has to find the balance between complexity and performance, this means, between realism and speed.

4.3.1 Classical LOD Algorithms

According to the Level Of Detail (LOD) concept, a complex object can be simplified to different level of details. The smaller is the object on the screen, the less detail is drawn to reduce small, distant, or unimportant geometry and to get better performance through this. Figure 4-2 demonstrates possible level of details:

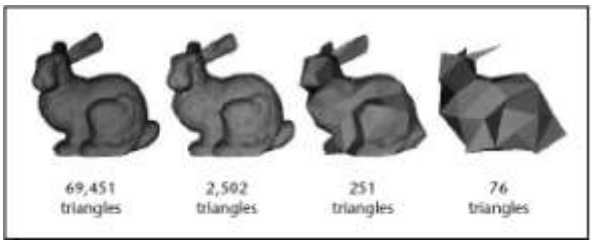


Figure 4-2: LOD levels of a bunny

If the bunny is small or distant, we cannot recognize the visual difference between the more and less detailed versions. To gain performance, if it is not visually distinguishable, only the bunny with fewer triangles is rendered. This idea is visualized on Figure 4-3:

Figure 4-3: LOD example

Other kind of LOD techniques are continuous LODs. Instead of creating different levels before running, it is possible to simplify a detailed object to a desired level in run-time. We store only the most detailed bunny, and the application removes the not necessary polygons to gain performance. This way, the LOD granularity can be much better, not only the previously generated levels can be used, although the application becomes more complex. For more details on general LOD techniques, see [LODf3DG].

4.3.2 LOD Algorithms on Water Surfaces

LOD techniques can be applied for water rendering as well. If the water surface is made up of more triangles, for example, of a triangle strip, this strip can be optimized to result a more realistic surface than by using simple equal size triangles. The main question to answer is that how triangles should be arranged.

The following method is discussed in [BMELAB2]. The 30×30-size grid (triangle strip) simulates the visible part of the water surface, this means that it is always transformed into the front of the camera. The height coordinate can be calculated with the help of continuous functions of the other coordinates ($z = f(x,y)$), so the place of the vertices can be chosen without any limitation to get the most realistic result. If we use equal-size triangles in the triangle strip, the distant ones have only very small visible size.

Optimization can be done by vertex replacement. The distant triangles are too small, they should be transformed to have about equal size visible by the camera. The near and far clipping planes and the width of the triangle strip needs to be taken into account, to have a water surface spread over the entire visible surface by the camera. The more problematic part is the height of the rows in the triangle strip to have equal visible sizes. The following hyperbolic function gives the horizontal place depending on the number of the rows:

$$f(k) = \frac{1}{1 - \frac{k}{k_{max}}}$$

Here k is the number of the row, k_{max} the amount of rows. The first row of triangles ($k = 0$) gets the horizontal distance 0, while the last row ($k = k_{max} - 1$) converges to infinity.
As we do not render objects in the infinity, the coordinates of the previous equation need to be scaled. Before rendering, we surely have a near and a far clipping plane. This means, only triangles between these two planes are rendered. If D_{max} is the far clipping distance, D_{min} the near distance, the next equation replaces and scales the coordinates of the triangle points:

$$f(k) = \frac{D_{max}}{k_{max} - 1}\left(\frac{1}{1 - \frac{k}{k_{max}}} - 1 - D_{min}\right)$$

For example, if the object between the distances 0 and 1000 needs to be rendered, the water surface can be scaled to the same area by setting D_{max} to 1000 and D_{min} to 0.
The far grid rows must be much broader to be equal-sized visible from the camera. This is visualized on Figure 4-4:

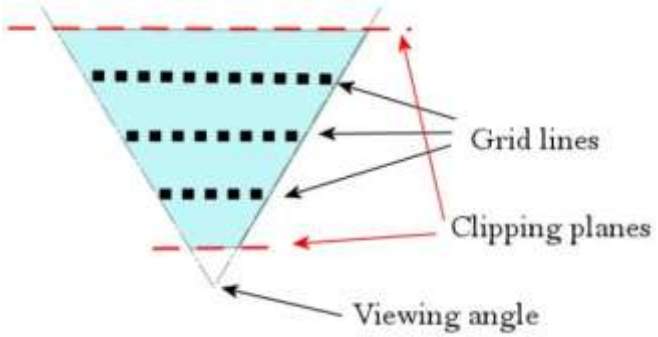


Figure 4-4: Far triangle lines are broader than near ones.

In [BMELAB2], triangle lines are scaled to match the two sides of the viewing angle using the following formula:

$$f(i) = \frac{i - \frac{i_{max} - 1}{2}}{i_{max}} \cdot d \cdot r$$

Where i is the actual column number, i_{max} is the number of the columns, d is the distance of the rows (calculated earlier), and r is the ratio between the sides of the view (for example: 4:3 or 16:9). For a complex review of different paper in the topic LOD, see [TLODRRGA].

4.3.3 Using Projected Grid

LOD algorithms determine where vertices are more frequent and where to skip them to have enough details to render a nice-looking picture real-time. But if the camera moves, the place on the screen can change where vertices are important. Projected grid algorithms ([RTWR]) try to locate vertices smoothly in camera space through the following steps:

1. Create a regular grid in the camera space that is orthogonal towards the camera.
2. Project the grid to the desired plane.
3. Transform the grid back to word-space.
4. Apply displacement, waves etc.
5. Render the grid, which will result a mostly even-spaced gird in camera space.

A real-world analogy can be, for example, if you put a paper with a dotted gird on it in front of the spotlight, the grid is projected onto the surface, as shown on Figure 4-5:

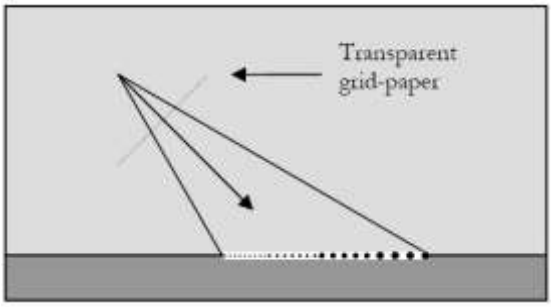


Figure 4-5: Real word analogy to projected grids.

This grid looks regular and smooth from the position of the spotlight, and that is our goal with the vertices of the water surface as well. For more details or for a fancy application which demonstrates this technique, see [RTWR].

In [IAoOW], an adaptive water mesh is used. The motion of the camera induces the shift of the mesh over the ocean surface to make the vertexes have approximately the same projected area on screen. They pay attention for a new problem introduced by this method. The surface normals cannot be estimated using finite differential techniques anymore because of the incorrect surface approximation. To determine the normals, analytical methods need to be used. This is visualized on Figure 4-6:

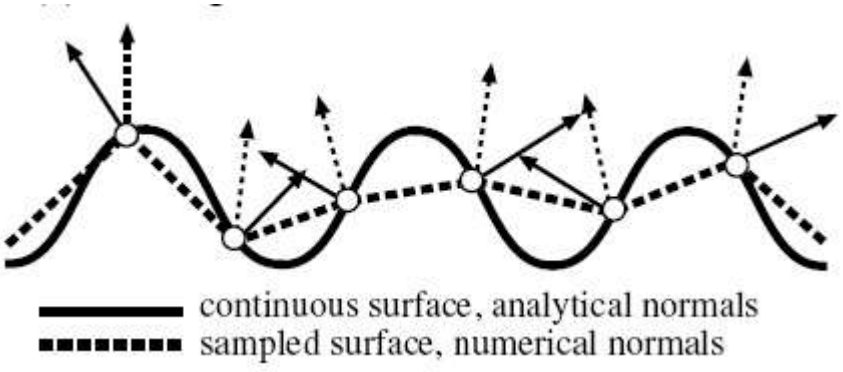


Figure 4-6: Inaccuracy of finite differential methods.

As shown on the figure, depending on how detailed the vertex grid is, analytical methods give much more accurate surface normals, than finite differential approaches.

For all the reasons I mentioned above, projected grids are an efficient technique to optimize water rendering, but they are complex, and application of them needs careful consideration.

4.4 Water Simulation Approaches

4.4.1 Coherent Noise Generation - the Perlin Noise

The water waves can be analytically described, or we can use random-based techniques as water waves are similar to other random natural phenomena. Random noise can be basis for realistic rendering. Ken Perlin published a method which gives continuous noise that is much more similar to random noises in nature than simple random ones. This difference is visualized on Figure 4-7:

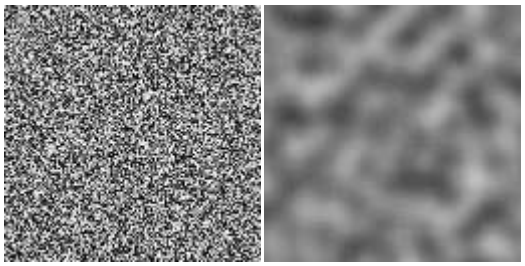


Figure 4-7: The difference between random and Perlin noise. The 2D random noise on the left is generated by a simple random generator. The Perlin noise on the right is much closer to random phenomena in the nature.

Basic Perlin noise does not look very interesting in itself but by layering multiple noise-functions at different frequencies and amplitudes, as shown on Figure 4-8, a more interesting fractal noise can be created.

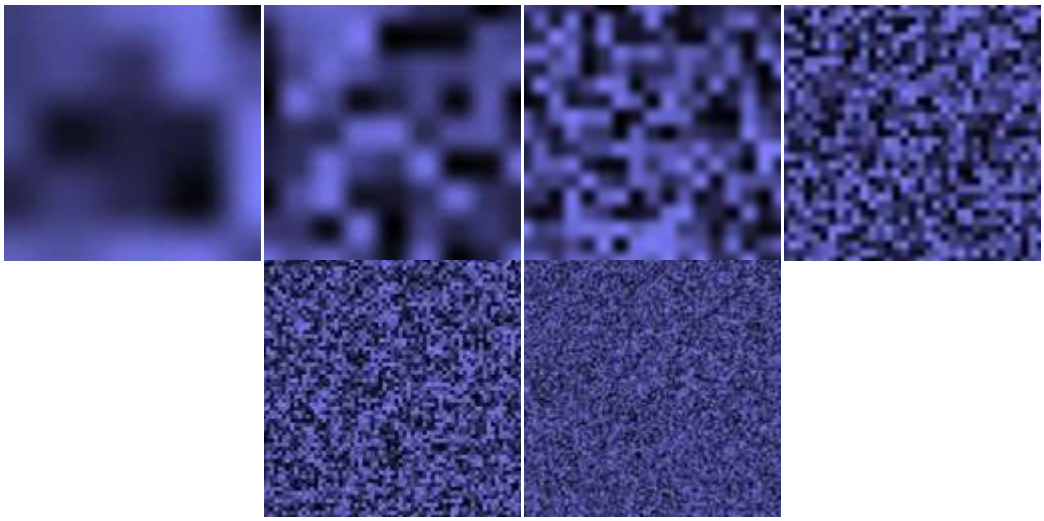


Figure 4-8: Layers of perlin noise with different amplitudes and frequencies.

The sum of them results the compound image on Figure 4-9:

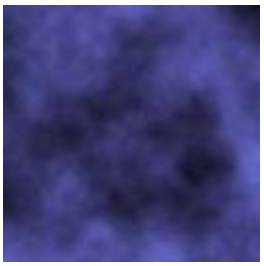


Figure 4-9: The sum of Perlin noises with different amplitudes and frequecies.

The frequency of each layer is the double of the previous one which is why the layers usually are referred to as octaves. By making the noise three-dimensional animated two-dimensional textures can be generated as well. More good explanations and illustrations can be found at [PN2].

A detailed and easy to understand explanation of Perlin noise generation can be found in [PNM]. For complex details, see [SHADERX].

Using Perlin noise as a core for water surfaces needs much less computational power than techniques discussed in the following paragraphs. The main problem with Perlin noise is that it is not controllable accurately, only the wave amplitudes and frequencies are easily changeable. Interaction with external objects is also hard to describe.

4.4.2 Fast Fourier Transformations

While physical simulations can be really resource consuming, random-noise based solutions are not accurate enough for every purpose. As a compromise, observation based statistical models can be used as core of the water surface animation. In this model, the wave height is a variable of position and time (position means horizontal coordinates (X and Y) without height (Z)). The height can be determined through a function, a set of sinus waves with different amplitudes and phases. To quickly get the sum of these amplitudes, inverse Fast Fourier Transformations (FFT) can be used. For a detailed example, check [RT3DEUCADX9]. The resulting surface can be very smooth with round waves at the top. This is not always desirable, various methods exist to add sharpness to the waves, making them look more choppy. For more details about this, see the chapter 4.7.2 - Creating Choppy Waves.

4.4.3 The Navier-Stokes Equations

Navier-Stokes Equations (NSE), as mentioned in chapter 2.3.4, describe the motion of incompressible viscose fluids. The acting forces are gravity, pressure forces and viscose forces. The actual equations are really hard and time consuming to solve, so we need to simplify and discretize them for real-time calculations. An efficient approach can be the simulation of solid volumes of water as a height-field, modeling the flow between adjacent columns of fluid. With this method, waves and other surface artifacts do not need to be explicitly specified because they arise naturally from the physical conditions occurring within the system.

[DSoSF] describes a technique simulating volume transitions through virtual pipes that connect adjacent columns, as shown on Figure 4-10:

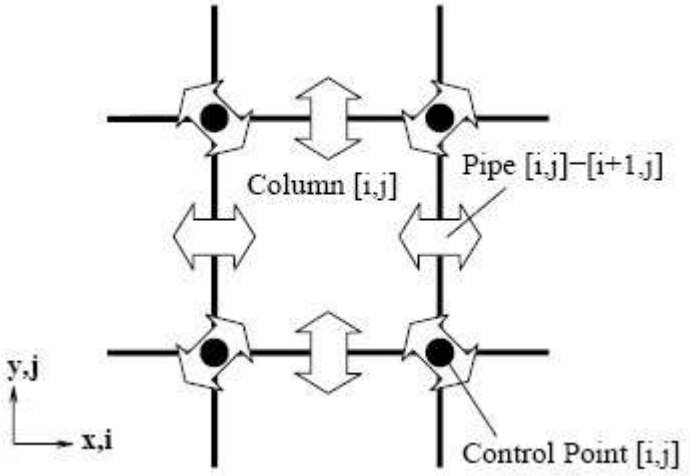


Figure 4-10: Neighboring volume cells are connected to calculate NSE flows.

The vertical columns are connected to their eight neighbors through a set of directional horizontal pipes. These pipes allow the water pressure to distribute over the entire system. The control points of the grid can be sampled to separate adjacent columns. To interact with external objects, the surface can be simulated as a separate subsystem which propagates external pressure to the volume grid (separate subsystem as well), as visualized on Figure 4-11:

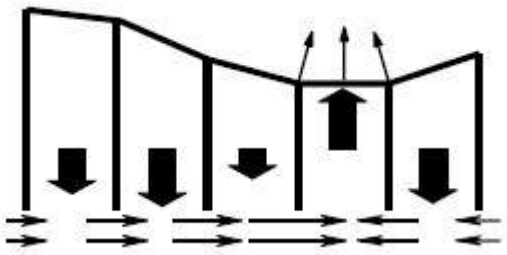


Figure 4-11: Force propagation on the cells in NSE

The bottom arrows indicate the direction of the flow between columns, the vertical arrows show the upward velocity and the thin arrows can indicate the velocity vectors for particle ejection. These different subsystems can interact to form a complex fluid system, as shown on Figure 4-12:

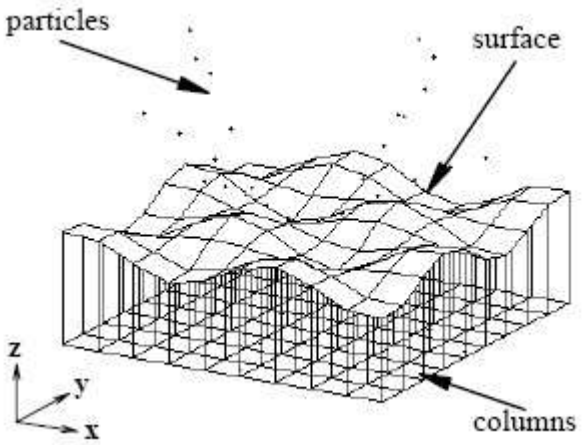


Figure 4-12: Complex water rendering system based on NSE.

Although they are extremely realistic, the NSEs are resource consuming to calculate for every time-step. The grid size must be absolutely limited for real-time simulations even on the latest graphic cards. Not real-time calculations, for example, for the movie Titanic were performed on a 2048 x 2048 grid, but this size cannot be handled real-time yet. NSE can be used for simulating smaller water surfaces like pools or fountains, although implementations combining multiple rendering methods exist also. For instance, a simple vertex displacement technique for distant areas can be combined with NSE for closer interaction with external objects. I have to mention that NSE requires world space grid for the calculations, while other solutions need a different grid-space, like the previously described projected grids.

NSEs are much easier to solve over 2D grids, than on 3D girds. Generally, 2D versions are enough, but they have their drawbacks. As only vertical forces can be inserted to the system, all the external forces must be simulated with vertical approximations. This can influence the result, for example in case of wind forces, which are generally horizontal.

4.4.4 Particle Systems

Physics-based approaches have become very popular recently. Improving hardware performance makes the application of real-time particle systems also possible. There are several ways to compute particle systems; some of them use only the graphics hardware. Depending on the issue, vertex-based and pixel-based solutions can be appropriate as well to make huge amount of independent particles seem alive. Particle system techniques can be combined with other water animation approaches to get a more realistic result.

Particle system approaches need to answer to questions: how do the particles move, and what are the particles as objects. The whole system can have a velocity, as a vector, but this vector does not need to be constant across the entire flow. Figure 4-13 visualizes this:

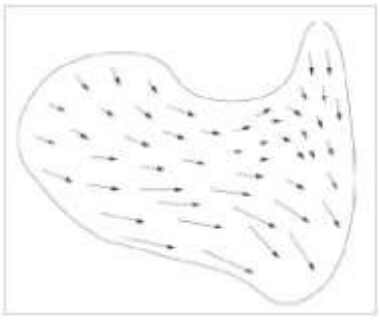


Figure 4-13: Example velocity flow in a particle system.

The answer to the second question is: our particles can be negligible in size and in mass as well. But they can carry further information to make other kind of interaction also possible, for example, color, temperature and pressure, depending on the expected result.

The particles move according to the physical laws, they motion can be calculated in time steps with the help of our previously discussed velocity-vector map. To be able to make these calculations on graphic hardware, a texture must store the place of the particles, so their place is sampled into a texture. These textures are called particle maps.

Based on the previous considerations, the graphics hardware-based method to texture advection is as follows. The velocity-map and the particle-map are stored in separate textures, which have two components. A standard 2D map can be represented this way, the third dimension is added by approximations to gain performance. Offset textures are part of hardware-supported pixel operations, so the move along the velocity-field can be implemented by them. Inflow and outflow (particle generation and removal) is outside the scope of this paper. More detailed explanations and source codes can be found in [SHADERX].

4.5 Rendering Reflections

4.5.1 Static Cube-map Reflections

If the water does not need to reflect everything, it is possible to use a pre-generated cube-map to calculate reflected colors. Cube-maps are a kind of hardware-accelerated texture maps (other approaches are for example *sphere mapping* and *dual paraboloid mapping*). Just imagine a normal cube with six images on its sides. These images are taken as a photo from the center point of the cube, and they show what is visible from the surrounding terrain through the points of the sides. An example is shown on Figure 4-14:

Figure 4-14: Cube map example.

As shown on Figure 4-15, the six sides of the cube are named after the three axle of the coordinate-system: x, y and z in positive and in negative directions:

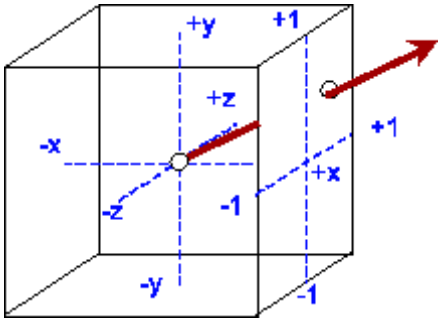


Figure 4-15: Using cube maps.

So we have a cube map and the reflecting surface of the water. We can calculate the vector for each point of the water that points into the direction of the reflected object. By using this 3-dimensional vector (the red one on Figure 4-15), the points of the cube-texture can be addressed from the center of the cube. This vector aims exactly one point of the cube, which has the same color as the reflected object in the original environment. But this calculations are much more efficient and hardware-accelerated to match the real-time requirements, while calculating global illuminations for every reflecting point needs much more time. Using cube-maps has one more advantage: the cube has sides which represent the environment that is not visible by the camera, so even points behind the camera can be reflected. On the other hand, cube-maps needs to be pre-rendered, so it is impossible to reflect changing environment (for instance, with moving objects) if we want to meet the real-time conditions. Using this technique, sky can be easily reflected on the water surface, but a moving boat needs to be handled in another way.

4.5.2 Dynamic Cube-maps

To be able to reflect changing environment the cube-map needs to be updated. Because cube-maps are essentially a collection of six textures on the sides, building a cube-map dynamically requires filling those textures one-by-one. We need to render the scene six times, once for each face of the cube, setting up the camera so that it matches the point of view from that particular cube-map face. Positioning the camera is not too complicated to achieve this, but the Field of View (FOV) needs to be adjusted to get equal-sized, square-shaped pictures, which see the same portion of the scene (90 degree to cover 360 degree together). Because the size of the water surfaces is relatively big compared to the environment, different objects need to be reflected in the same direction from the different point of the water. This means that a single cube-map is not enough to simulate real reflections on the whole water surface. Creating more cube-maps for every frame are extremely expensive, so dynamic cube-maps are generally not real alternatives for water reflection effects on today's graphic cards.

Although they are extremely complex, there exist some very realistic solutions, for example, in the game Half Life 2. They use more dynamic cube maps generated from various points of the water surface, and reflections are got from the stored values through weighted interpolations. To get real-time performance, the cube maps can be regenerated only a few times in every second.

4.5.3 Reflection Rendering to Texture

In chapter Reflection (2.2.1) with the title Reflection, I discussed a method to determine the reflected color for every point of the water surface. One of the most precise solutions for that

is creating a virtual view on the other side of the water plane, and rendering the same scene into a texture, which can be used as a reflection map later. This means, that before rendering the final image, a pre-rendering phase should be added. The place of the camera and the view-vector is mirrored onto the water plane during this phase, and every object of the virtual world which can be reflected by the water on the final image is rendered from this virtual view into a texture. Let me show the idea again on Figure 4-16:

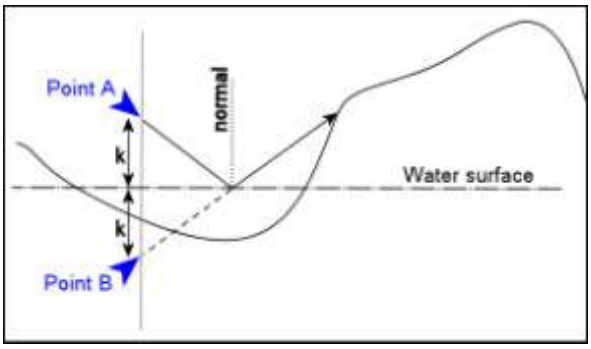


Figure 4-16: Rendering reflections.

To get the expected result, the place of point *B* must be calculated. For this, we have to determine how far the original place of the camera is from the water plane, so we have to determine distance *k*. If the water is horizontal, this distance has to be subtracted from the height of the water plane to find the height coordinate, the other coordinates of the points *A* and *B* are the same. To avoid artifacts, the underwater objects can be removed from the world before the rendering into texture. When the final image is created, this texture can be used as a reflection-map. The reflective color can be sampled by the help of the vector between the camera and the points of the water surface, and the shape of the waves can be taken into account as well. Smaller adjustments can be needed to have better results, for instance, smaller modifications of height of the clipping plane or point B can improve the reality by producing less artifacts.

4.6 Calculating the Fresnel Term

4.6.1 Accurate Approximation of the Fresnel Term

The operations to determine the exact Fresnel value for each pixel of the water are very costly. If the water covers a significant part of the display, calculating the accurate value is unsuitable for real-time conditions, so approximations need to be used. In this case, approximations by linear functions are inadequate due to inaccuracy. In [DWAaR] they used reciprocal of different powers which are surprisingly correct approximations. Some of these are visible on Figure 4-17:

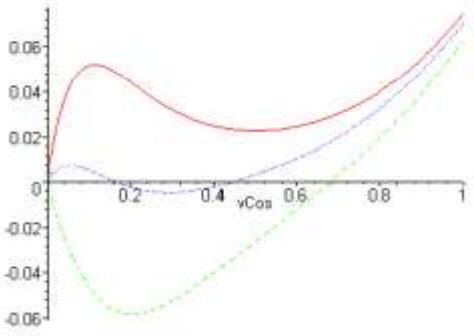


Figure 4-17: Fresnel term approximations. The red solid line shows the power of 8, the blue dashed line is the power of 7 and the green dashed line is the power of six.

The difference between the analytical calculation and the approximation by the power of 8 is visualized on Figure 4-18:

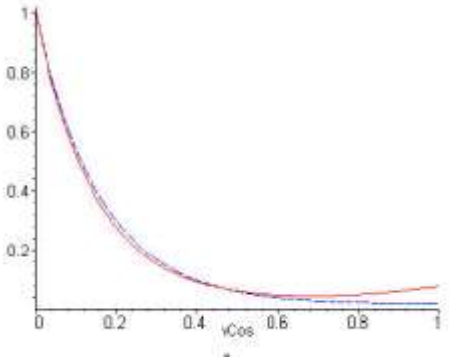


Figure 4-18: The error of the approximation. The dashed blue line is the approximations be the function 1/(1+x)8 and the red line is the analytically calculated accurate value. The values of the X axis show the cosine between the normal and the eye vector.

4.6.2 Simpler Solution

If the angle between the view vector and the normal vector is bigger, the amount of reflection gets higher. For this, [Riemer] used a simple approximation by projecting the eye vector on the normal vector of the water plane as shown on Figure 4-19:

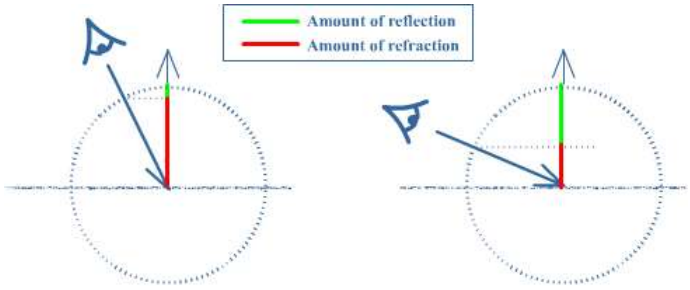


Figure 4-19: Fresnel approximation through projection

The amount of refraction (refraction coefficient) can be easily calculated by the dot product of the eye and the normal vector, and the sum of the two coefficients is always 1.

4.6.3 A Realistic Compromise

The cheap calculations introduced in section *Simpler Solution* do not take the indices of refraction into account and have a stronger divergence from natural effect. This divergence results an unnaturally strong reflection. [SHADERX2] advises a better approximation:

$$R(\alpha) = R(0) + (1 - R(0)) * (1 - \cos(\alpha))^5$$
$$\text{with } R(0) = (n1 - n2)^2 / (n1 + n2)^2$$

Where *n1* and *n2* are the indices of refraction for the involved materials, and α is the angle between the eye-vector and the normal vector of the surface. For air-water boundary, *n1* = 1.000293 and *n2*=1,3333333; this means that *R(0)* = 0.02037f and *1-R(0)* = 0.97963f. Figure 4-20 visualizes the difference between this approximations and a simpler solution also (1-cos(α)):

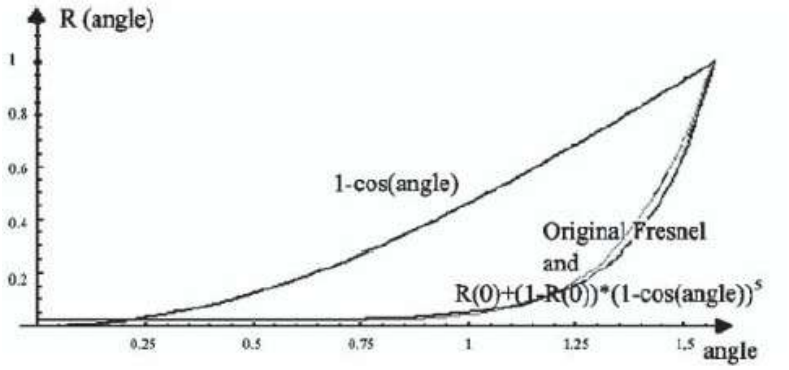


Figure 4-20: Accurate Fresnel term approximation.

4.6.4 Using Texture Lookup

To combine speed and accuracy it is possible to pre-calculate the values of the Fresnel term for different angles and store them in a one-dimensional texture as a look-up table. During rendering, after we calculated the dot product between the normal and reflection vector, we can find the matching Fresnel term value in the look-up table for the dot product. This way the Fresnel term can be determined in a very fast and relatively accurate way. This approach is used in [RTWR].

4.7 Rendering Various Water Phenomena

4.7.1 Generating Spays Using Particle Systems

Particle systems can be good solutions to make real-time interaction between external objects and the water surface. They can efficiently animate moving surface as well, but usually they are applied with other techniques at the same time. Flowing water, water-drops, spay, waterfalls are just some of the possible water-related topics that can be implemented through particle systems.

Sprays are modeled as a separate subsystem in [DSoSF], as mentioned earlier in *The Navier-Stokes Equations* chapter. When an area of the surface has high upward velocity, particles are distributed over that area. Particles don't interact with each other, they only fall back to the water surface because of the gravity, and then they are removed from the system.

[DWAaR] uses a similar particle model to simulate water spray. Simple Newtonian dynamics are taken into account: water-surface's velocity at the spawning position and some turbulence influences their initial velocity. It can then be updated according to gravity, wind and other possible global forces. Rendering is done with mixture of alpha-transparency and additive-alpha sprites. For more details and screenshots, see [DWAaR]. These previously discussed techniques can be really convincing visually for spray-simulation.

4.7.2 Creating Choppy Waves

The general methods discussed in these pages use randomly generated or sinusoidal wave formations. They can be absolutely enough for water scenes with normal conditions, but there are some cases, when choppy waves are needed. For example, stormy weather or shallow water where the so-called "plunging breaker" waves are formed. In the following paragraphs I will briefly introduce some of the approaches to get choppier waves.

4.7.2.1 **Analytical Deformation Model**

[UVTDFRWR] describes an efficient method which disturbs displaced vertex positions analytically in the vertex shader. For example, explosions are important for computer games. To create an explosion effect, they use the following formula:

$$I(r) = \frac{I_0 \sin(kr + \omega t) e^{-bt}}{r^2},$$

where t is the time, r is the distance from the explosion center in the water plane and b is a decimation constant. The values of I_0, w, and k are chosen according to a given explosion and its parameters.

For rendering, they displace the vertex positions according to the previous formula, which results choppy waves to get convincing explosion effects.

4.7.2.2 **Dynamic Displacement Mapping**

[UVTDFRWR] introduces another approach also. The necessary vertex displacement can be rendered in a different pass and later used to combine it with the water height-field. This way, some calculations can be done before running the application to gain performance. Depending on the basis of the water rendering, the displacements can be computed by the above-mentioned analytical model or, for example, by the Navier-Stokes equations as well.

Although these techniques can result realistic water formations, they need huge textures to describe the details. The available texture memory and the shader performance can limit the applications of these approaches.

4.7.2.3 **Direct displacement**

In [DWAaR] they compute the displacement vectors with FFT. Instead of modifying the height-field directly, the vertexes are horizontally displaced using the following equation:

$$X = X + \lambda D(X,t)$$

where λ is a constant controlling the amount of displacement, and D is the displacement vector. D is computed with the following sum:

$$D(X,t) = \sum_{K} -i\frac{K}{k}\tilde{h}(K,t)e^{iK\cdot X}$$

where *K* is the wave direction, *t* is the time, *k* is the magnitude of vector *K* and *h(K,t)* is a complex number representing both amplitude and phase of the wave .

The difference between the original and the displaced waves is visualized on Figure 4-21. The displaced waves on the right are much sharper than the original ones:

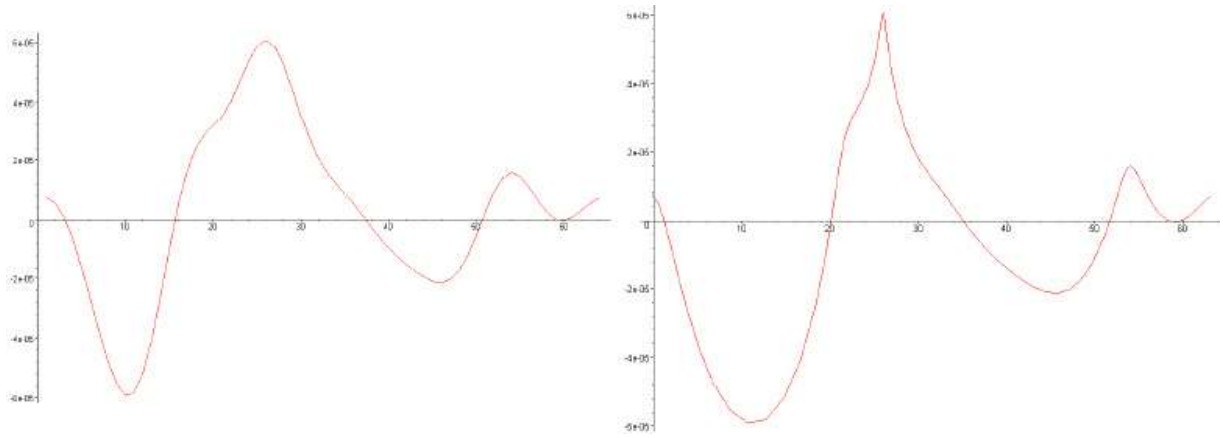


Figure 4-21: Difference between the original and displaced wave forms.

4.7.2.4 Choppy Waves Using Gerstner Waves

If the rendered water surface is defined by the Gerstner equations, our task is easier. Gerstner waves are able to describe choppy wave forms. Amplitudes need to be limited in size, otherwise breaks can look unrealistic. A fine solution to create choppy waves can be the summation of Gerstner waves with different amplitudes and phases. Summation can be carried out through the following sum:

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

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

where k_i is the set of wave vectors, k_i is the set of magnitudes, A_i is the set of wave frequencies, ω_i is the set of phases and N is the number of sine waves.

Sum of 3 Gerstner waves is visualized on Figure 4-22:

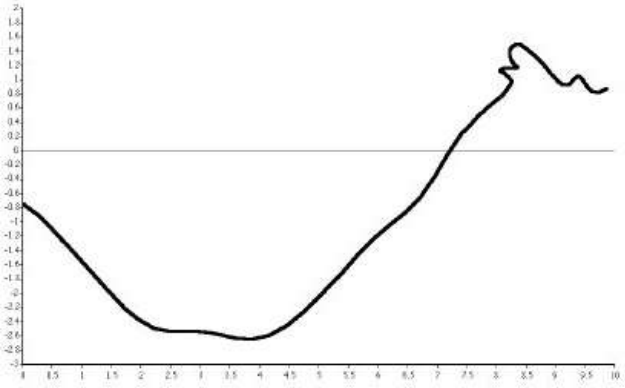


Figure 4-22: Sum of 3 Gerstner waves

4.7.3 Rendering Caustics

4.7.3.1 General Approaches

Some caustics rendering techniques use environment mapping. However it is supported by graphic hardware, it is only good approximation in the case where the reflected/refracted object is small compared to its distance from the environment. This means, environment mapping can be used only when the objects are close to the water surface. Objects under dynamic water surfaces need an often updated environment map, so the usability of environment maps for caustics rendering is limited.

Several approaches render accurate caustics through ray tracing methods, but generally, they are too time-consuming for real-time applications. (See [LWIuBBT]). Other techniques approximate textures of underwater caustics on a plane using wave theory. Although, these moving textures can be rendered onto arbitrary receivers at interactive frame rates, the repeating texture patterns are usually disturbing.

Graphics hardware has made significant progress in performance recently and many hardware-based approaches has been developed for rendering caustics. Real caustics calculation needs intersection tests between the objects and the viewing ray reflected at the water surface. Generally, the illumination distribution of object surfaces needs to be computed, but these are really time-consuming and difficult. Although, backward ray tracing, adaptive radiosity textures and curved reflectors are published methods for creating realistic

images of caustics, they can't be done real time because of the huge computational cost. For more details about these approaches, see [BRT], [ARTfBRT] and [IfCR].

4.7.3.2 Caustic Maps

Caustics-maps show intensifies of caustics. They are generated by projecting the triangles of the water surface onto the objects in the water. The intersecting triangles influence the force of light on the object. The intensity of the caustic triangles is proportional to the area of the water surface triangle divided by the area of the caustic triangle. The more projected triangles intersect each other and the higher their intensity is at a given point, the lighter that point is. In the end, caustics map and the original illumination map is merged as on Figure 4-23:

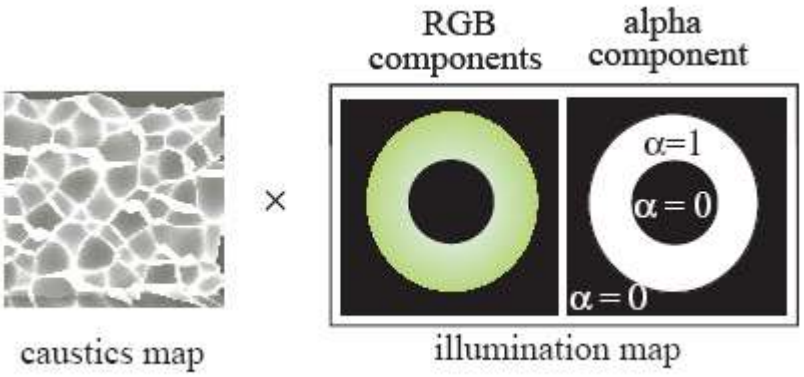


Figure 4-23: Using caustics map.

4.7.3.3 Volume Rendering Based Caustics

[FRMfRaRCDtWS] describes a technique for rendering caustics fast. Their method takes into account three optical effects, reflective caustics, refractive caustics, and reflection/refraction on the water surface. It calculates the illumination distribution on the object surface through an efficient method using the GPU. In their texture based volume rendering technique objects are sliced and stored in two or three-dimensional textures. By rendering the slices in back to front order, the final image is created, and the intensities of caustics are approximated on the slices only, not on the entire object. The method is visualized on Figure 4-24:

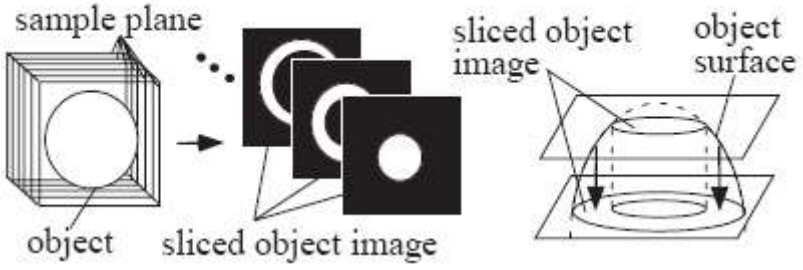


Figure 4-24: Volume rendering based caustics

Although, this reduces computation time, it does not enable real-time caustics rendering. The most expensive part of their technique is refreshing caustics map.

4.7.3.4 Ray-tracing Based Caustics

[IISTfAC] introduces a faster approach for rendering caustics. The method emits particles from the light source and gathers their contributions as viewed from the eye. To gain efficiency, they emit photons in a regular pattern, instead of random paths. The pattern is defined by the image pixels in a rendering from the viewpoint of the light. Or in another way: counting how many times the light-source sees a particular region is equivalent to counting how many particles hit that region. For multiple light sources, multiple rendering passes are required. Several steps are approximated to reduce the required resources, for example, interpolation among neighboring pixels, skipping volumetric scattering effects or restriction to point lights.

4.7.3.5 Bounding Volume Based Caustics

In [IRoCuIWV], a more accurate method is described. In the first pass, the position of receivers is rendered to a texture. In the second pass, a bounding volume is drawn for each caustic volume. For points inside the volume, caustic intensity is computed and accumulated in the frame buffer. They take warped caustic volumes into account also, which is skipped in the other caustics-rendering techniques. Their method can produce real-time performance for general caustic computation, but it is not fast enough for large water areas. For fully dynamic water surfaces with dynamic lighting, their method rendered the image shown on Figure 4-25 at 1280 x 500 pixels with 0.2 fps:

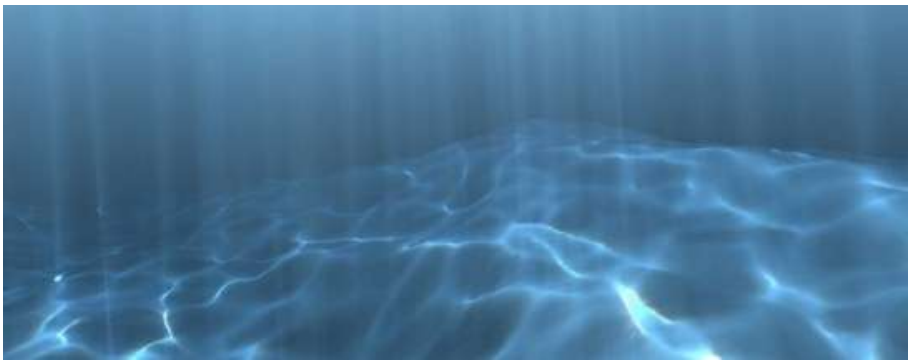


Figure 4-25: Bounding volume based caustics

For more details, see [IRoCuIWV].

4.7.3.6 Optimized, Caustics Map Based Approach

In [DWAaR], they optimize their approach to real-time performance. They consider only first-order rays and assume the receiving surface at a constant depth. Incoming light beams are refracted, and the refracted rays are then intersected against a given plane. Figure 4-26 illustrates the method, it shows the projection of four water surface triangles:

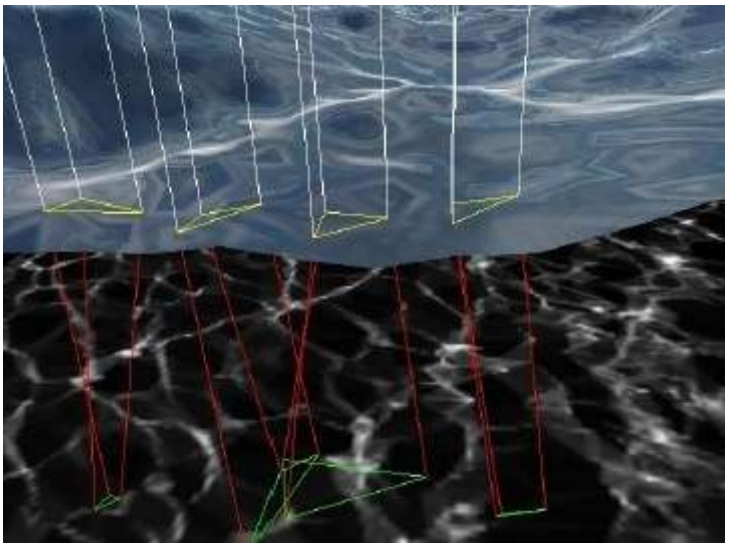


Figure 4-26: Projecting water surface triangles to produce the caustics map.

To reduce the necessary calculations, only a small part of the caustics-map is calculated, and they show a method to tile it for the entire image seamlessly. Finally, the sun's ray direction and the position of the triangles are used to calculate the texture-coordinates by projection. For further discuss on this method, see [DWAaR].

4.7.3.7 Using Pre-generated Textures

The main ideas of caustics rendering were briefly introduced. The accurate methods apply ray tracing techniques, but they cannot produce real-time performance without cheating. The most often used approaches use pre-generated caustic textures and try to avoid the visible repetition. Shifting the texture is necessary, but texture animations and formations can make this solution more realistic. All the other methods can be a source for pre-generating the caustics map, so I encourage the reader to try out the various possibilities. It can be also a good idea to make transitions between more caustic textures to make them more alive. Although, various approaches exist to avoid texture repetition which can help by caustics rendering as well, they are out of the scope of this paper.

4.7.4 Foam generation

To get the most realistic foamy waves particle systems are the best approach. Although they can simulate every property of the foams, only for small water surfaces can they be efficient enough. Other methods need to be taken into consideration.

The main idea for foam generation in the water surface rendering literature is the application of pre-calculated foam-texture. The choppiness of waves is evaluated, and on the places where it exceeds a specific level, foam-texture is blended to the final color. In [UVDfRWR], they use the following formula to calculate the transparency of the foam-texture:

$$Foam.a = \text{saturate}\left(\frac{H - H_0}{H_{max} - H_0}\right),$$

where H_{max} is the height at which the foam is at maximum, H0 is the base height, and H is the current height.

If the foam-texture is animated, it can show the formation and dissipation of the foam also. In [DWAaR] they don't animate the texture, but its transparency is always recalculated. The alpha value decreases continuously, and if the choppiness is high enough, the alpha is increased through some frames to get a good visual result.

The limitations of this technique are the texture repetition and the shortage of motion. The repeating patterns can be noticed because they are the same everywhere. The other problem is that the foam doesn't move on the water surface according to its slope.

4.7.5 The Kelvin wedge

Producing this phenomenon is easier if the basis of the water rendering system is capable of receiving outer forces, like, for example, FFT and Navier-Stokes equations do. In [CBAfAWW], a different approach is used as core of the wave simulation. Their solution uses the motion vector between two picture frames to calculate how the water height-field is to be altered for the following frame. An additive contribution is computed for each swimming object. They got a very realistic result in [CBAfAWW], as shown on Figure 4-27:

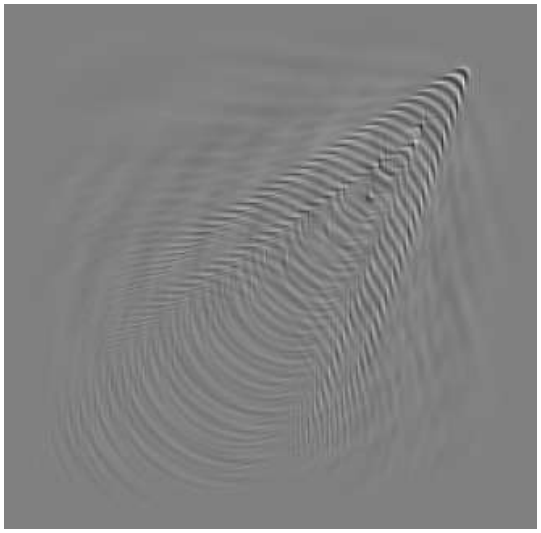


Figure 4-27: Rendering Kelvin wedge.

This idea can be also implemented with FFT based systems as well. Waves behind any moving object can be described and these patterns can be added to the system if necessary. The most important parameters are the speed of the boat and the type and depth of the water.

If a water-rendering system uses Navier-Stokes equations, the Kelvin wedge can be produced by adding external forces to the system. Only experimentation is needed to get realistic results in various cases.

5 LAKE WATER SHADER

There are many different types of water in our world and they range from small water surfaces in a mug to endless areas of oceans. Typically, smaller waters interact with floating or falling objects while bigger surfaces get into reaction with the wind and form waves. This chapter describes an example of middle-sized water area rendering: smaller lakes and rivers with moderate waves.

5.1 Preconditions

In this chapter I discuss creating water effects with the following preconditions:

- Realistic, nice-looking water needed
- Middle-sized, flat water surface
- Moderate interaction with the wind
- No need for breaking waves or foam
- No need for underwater effects (The view point is always over the water surface)
- Real time performance

Using high-fields or triangle-strips can result in very nice-looking effects, but if the waves do not need to be braking and the performance is an important factor than simpler only-shader-driven solutions can be a good compromise. To gain efficiency, the water surface will be approximated only by a single square.

5.2 Before Using the Shader

The water surface will be a square which means that it is represented only by four vertices. This water-plane intersects the virtual world at a certain height, and if the landscape is lower than the height of the water plane, the water is visible. Everywhere else the water will be covered by the landscape. The idea is shown on Figure 5-1:

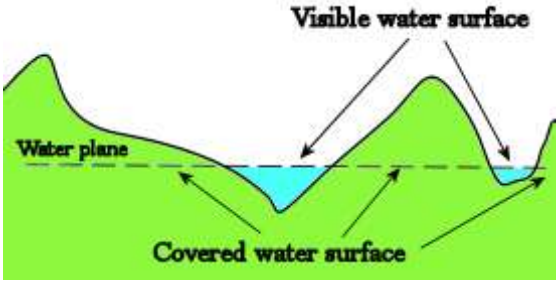


Figure 5-1: The water plane intersects the landscape

Landscape can be created, for example, from a high-map. I discuss a technique for this and for creating sky-dome on the homepage this paper. Those ideas can be the basis for the following water effects. For more details, see [HWS] or the accompanying DVD.

5.3 HLSL Code

In the HLSL code we define the technique to create the water effect. It has only one pass, and both shaders can be set to version 2.0 in it. The definition is the following:

```
technique Water
{
        pass Pass0
        {
                VertexShader = compile vs_2_0 WaterVS();
                PixelShader = compile ps_2_0 WaterPS();
        }
}
```

The definition of the structure returned by the vertex shader describes the variables. At first, we need to determine the sampling positions which are used later in the pixel shader. The sampling positions are returned (the name of the variables shows their use), and *Position3D* is needed to be able to calculate the accurate eye vector in the pixel shader:

```
struct WaterVertexToPixel
{
        float4 Position : POSITION;
        float4 ReflectionMapSamplingPos : TEXCOORD1;
        float2 BumpMapSamplingPos : TEXCOORD2;
        float4 RefractionMapSamplingPos : TEXCOORD3;
        float4 Position3D : TEXCOORD4;
};
```

The structure returned by the pixel shader is not so complicated any more. Using the relayed information, we sample the different textures and calculate the final color of the pixel which is written to the frame buffer. Only this color value is returned:

```
struct WaterPixelToFrame
{
        float4 Color : COLOR0;
};
```

Calculating the sampling coordinates for reflection and refraction maps is possible through creating the necessary matrices. Multiplying the view-matrix and the projection-matrix results the view-projection matrix. This can is multiplied with the world-matrix to get the world-view-projection matrix, and so on and so forth. We can get the sampling positions, for example, with these lines:

```
WaterVertexToPixel WaterVS(float4 inPos : POSITION, float2 inTex: TEXCOORD)
{
        WaterVertexToPixel Output = (WaterVertexToPixel)0;
        float4×4 preViewProjection = mul (xView, xProjection);
        float4×4 preWorldViewProjection = mul (xWorld, preViewProjection);
        float4×4 preReflectionViewProjection = mul (xReflectionView, xProjection);
        float4×4 preWorldReflectionViewProjection = mul (xWorld,
preReflectionViewProjection);
```

```
        Output.Position = mul(inPos, preWorldViewProjection);
        …
        Output.ReflectionMapSamplingPos = mul(inPos, preWorldReflectionViewProjection);
        Output.RefractionMapSamplingPos = mul(inPos, preWorldViewProjection);
        return Output;
}
```

Reflection and refraction maps are sampled using the perturbated positions in the pixel shader. The reason for using perturbations is described in the section 5.7. The perturbated texture coordinates can be calculated as follows:

```
WaterPixelToFrame WaterPS(WaterVertexToPixel PSIn)
        {
        ProjectedTexCoords.x =
PSIn.ReflectionMapSamplingPos.x/PSIn.ReflectionMapSamplingPos.w/2.0f + 0.5f;
        ProjectedTexCoords.y =
PSIn.ReflectionMapSamplingPos.y/PSIn.ReflectionMapSamplingPos.w/2.0f + 0.5f;
        float4 reflectiveColor = tex2D(ReflectionSampler, perturbatedTexCoords);
        …
        ProjectedRefrTexCoords.x =
PSIn.RefractionMapSamplingPos.x/PSIn.RefractionMapSamplingPos.w/2.0f + 0.5f;
        ProjectedRefrTexCoords.y = -
PSIn.RefractionMapSamplingPos.y/PSIn.RefractionMapSamplingPos.w/2.0f + 0.5f;
        float4 refractiveColor = tex2D(RefractionSampler, perturbatedRefrTexCoords);
        return Output;
}
```

5.4 Rendering Reflections

To be able to reflect the objects above the surface as described in chapter *Reflection Rendering to Texture* (4.5.3), we need to have the image of the reflected objects, which shows the reflected color for each pixel of the water. Before creating the final picture, this image can be rendered into a texture as a new render-target and later can be used to the reflection effects.

In the C# code the new texture (new render-target) needs to be defined and initialized first:

```
private RenderTarget2D reflectionRenderTarg;
private Texture2D reflectionMap;
reflectionRenderTarg = new RenderTarget2D(device, 512, 512, 1,
SurfaceFormat.Color);
```

The original view-point and view-direction needs to be mirrored onto the plane of the water, as described in chapter 2.2.1. For more details, see [RIEMER].

We need to create the matrix of the virtual view by mirroring the original one onto the water plane as follows:

```
private Matrix reflectionViewMatrix;
float reflectionCamZCoord = -cameraPosition.Z + 2*waterHeight;
Vector3 reflectionCamPosition = new Vector3(cameraPosition.X,
cameraPosition.Y, reflectionCamZCoord);
```

```
float reflectionTargetZCoord = -targetPos.Z + 2 * waterHeight;
Vector3 reflectionCamTarget = new Vector3(targetPos.X, targetPos.Y,
reflectionTargetZCoord);

Vector3 forwardVector = reflectionCamTarget - reflectionCamPosition;
Vector3 sideVector = Vector3.Transform(new Vector3(1, 0, 0),
cameraRotation); Vector3 reflectionCamUp = Vector3.Cross(sideVector,
forwardVector);

reflectionViewMatrix = Matrix.CreateLookAt(reflectionCamPosition,
reflectionCamTarget, reflectionCamUp);
```

After the entire world (without water) is drawn from the virtual view, this image needs to be rendered onto our temporary render-target. To avoid ghost-reflections and hidden reflected areas, a clipping plane can be used to discard the objects under the plane of the water. This step helps eliminating unnecessary rendering and avoiding possible artifacts.

Clipping planes must be set to remove areas, which cannot be reflected on the water surface, but can hide reflections. The idea is visualized on Figure 5-2:

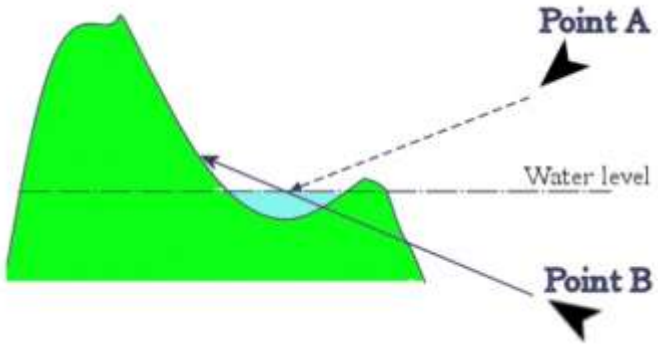


Figure 5-2: Hidden reflections.

If we want to get the possible reflections from point A, we have to render the reflection map from point B. But before rendering, we have to remove every underwater object, because they can hide real reflections, as the underwater terrain does on Figure 5-2. Although the arrow points the reflected point if you look onto the water surface, the first intersection from point B is an underwater part of the scene. After removing everything under the water level with a clipping plane, the first intersection point will be our desired target, which is reflected on the water.

The clipping planes can be set up as follows to remove the underwater objects:

```
Vector3 planeNormalDirection = new Vector3(0, 0, 1);
planeNormalDirection.Normalize();
Vector4 planeCoefficients = new Vector4(planeNormalDirection, -
waterHeight);
Matrix camMatrix = reflectionViewMatrix * projectionMatrix;
Matrix invCamMatrix = Matrix.Invert(camMatrix);
invCamMatrix = Matrix.Transpose(invCamMatrix);
planeCoefficients = Vector4.Transform(planeCoefficients,
invCamMatrix);
```

```
Plane refractionClipPlane = new Plane(planeCoefficients);
device.ClipPlanes[0].Plane = refractionClipPlane;
```

After this, the clipping plane and the view matrix can be used by the *draw* method:

```
private void DrawReflectionMap()
{

Vector3 planeNormalDirection = new Vector3(0, 0, 1);
planeNormalDirection.Normalize();
Vector4 planeCoefficients = new Vector4(planeNormalDirection, -
waterHeight);
Matrix camMatrix = reflectionViewMatrix * projectionMatrix;
Matrix invCamMatrix = Matrix.Invert(camMatrix);
invCamMatrix = Matrix.Transpose(invCamMatrix);
planeCoefficients = Vector4.Transform(planeCoefficients,
invCamMatrix);
Plane reflectionClipPlane = new Plane(planeCoefficients);
device.ClipPlanes[0].Plane = reflectionClipPlane;
device.ClipPlanes[0].IsEnabled = true;
device.SetRenderTarget(0, reflectionRenderTarg);
device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer,
Color.Black, 1.0f, 0);
DrawTerrain(reflectionViewMatrix);
DrawSkyDome(reflectionViewMatrix);
device.ResolveRenderTarget(0);
reflectionMap = reflectionRenderTarg.GetTexture();
device.SetRenderTarget(0, null);
device.ClipPlanes[0].IsEnabled = false;

}
```

I have to notice that, to restore the original state, the clipping plane is set to false at the end of the *draw* method. The terrain and the sky-dome are drawn using the matrix of the virtual view (*reflectionViewMatrix*) because of the reasons discussed earlier. At this point all the reflection data is stored on a texture.

On Figure 5-3, the reflection map is shown on the right which was used to produce the image on the left:

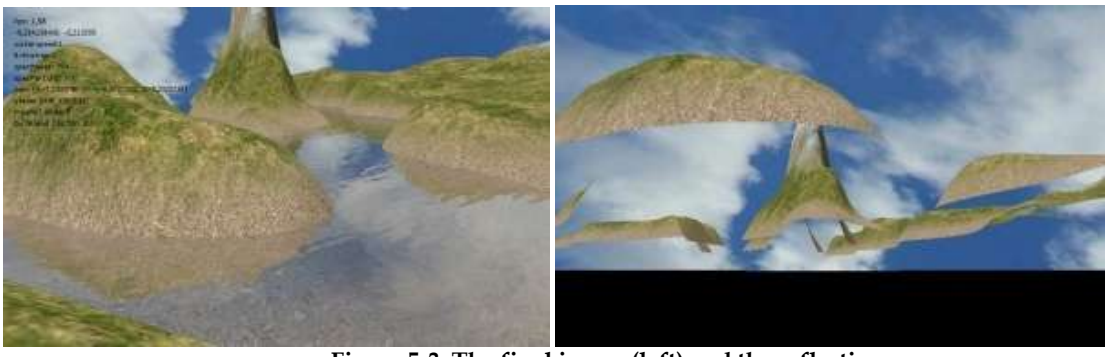


Figure 5-3. The final image (left) and the reflection map (right). The reflection-map is captured from underneath the water level as discussed earlier, and from that point it is possible to see what is "behind the sky". This results the black area on the image.

5.5 Rendering Refractions

The method to produce a refraction map is similar to one of the reflections map. There is no need to change the view point, the virtual and the original view-vectors are the same, but the clipping plane needs to be inverted, as everything is to be rendered below and not over the water level.

```
RenderTarget2D refractionRenderTarg;
Texture2D refractionMap;
refractionRenderTarg = new RenderTarget2D(device, 512, 512, 1,
SurfaceFormat.Color);
```

The clipping plane is applied on the graphic hardware, and therefore, the vertices are in camera space already when they will be compared to the plane. Because of this, the plane needs to be transformed with the inverse of the camera matrix. This can be achieved by the following lines:

```
Vector3 planeNormalDirection = new Vector3(0, 0, -1);
planeNormalDirection.Normalize();
Vector4 planeCoefficients = new Vector4(planeNormalDirection, 5.0f);
Matrix camMatrix = viewMatrix * projectionMatrix;
Matrix invCamMatrix = Matrix.Invert(camMatrix);
invCamMatrix = Matrix.Transpose(invCamMatrix);
planeCoefficients = Vector4.Transform(planeCoefficients,
invCamMatrix);
Plane refractionClipPlane = new Plane(planeCoefficients);
```

In the draw method the clipping plane needs to be created, applied, and finally the original state needs to be restored after drawing onto a texture. The source code for this is the following:

```
private void DrawRefractionMap()
{

Vector3 planeNormalDirection = new Vector3(0, 0, -1);
planeNormalDirection.Normalize();
Vector4 planeCoefficients = new Vector4(planeNormalDirection, 5.0f);
Matrix camMatrix = viewMatrix * projectionMatrix;
Matrix invCamMatrix = Matrix.Invert(camMatrix);
invCamMatrix = Matrix.Transpose(invCamMatrix);
planeCoefficients = Vector4.Transform(planeCoefficients,
invCamMatrix);
Plane refractionClipPlane = new
Plane(planeCoefficients);evice.ClipPlanes[0].Plane =
refractionClipPlane;
device.ClipPlanes[0].IsEnabled = true;
device.SetRenderTarget(0, refractionRenderTarg);

device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer,
Color.Black, 1.0f, 0);

DrawTerrain();
device.ResolveRenderTarget(0);
refractionMap = refractionRenderTarg.GetTexture();
```

```
device.SetRenderTarget(0, null);
device.ClipPlanes[0].IsEnabled = false;

}
```

The refraction data is also stored on a texture at this point of the source code. In the next section they are available to create the final image.

An example is given on Figure 5-4. The refraction map is shown on the right which was used to produce the image on the left:

Figure 5-4: Example refraction map (right) and final image(left).

Notice that the clipping plane is set not exactly to the water level, but a little bit higher to avoid artifacts at the edges. To gain performance, the refraction map is half the size of the original image, just like the reflection map.

5.6 Fresnel Term Approximations

Operations to calculate the Fresnel term correctly are very complex (see chapter 2.2.6). To blend the previously determined reflected and refracted color we need the proper ratio between them. In this demo application I use various solutions to approximate the Fresnel effect.

Reflection and refraction colors need to be blended depending on the cosine of the angle between the eye-vector and the normal-vector. As both of these vectors are one unit long, the cosine of the angle can be determined by the dot product of them.

The first solution is the projection of the eye vector on the normal vector, which approximates the Fresnel term relatively good, but not accurate enough. The projection can be calculated by dot product. This is only adjusted with some terms to get similar result with the approaches described later:

```
if ( fresnelMode == 0 )
{
        fresnelTerm = dot(eyeVector, normalVector);
        //correction
        fresnelTerm = 1-fresnelTerm*1.3f
}
```

The next approach uses the formula of chapter 4.6.3. For more details, see [EMaRoTWoNT].

```
if ( fresnelMode == 1 )
{
	fresnelTerm = 0.02f+0.97f*pow((1-dot(eyeVector, normalVector)),5);
}
```

The third approximation is discussed in [CgTOOLKIT]. It also calculates the dot product between the eye-vector and the normal-vector. After adding 1 to this, to get the result, we divide 1 by the fifth power of this value:

```
if ( fresnelMode == 2 )
{
	float fangle = 1+dot(eyeVector, normalVector);
	fangle = pow(fangle ,5);
	fresnelTerm = 1/fangle;
}
```

To be able to adjust the settings the xDrawMode input variable influences the Fresnel value. It is then reduced between 0 and 1. Finally, the reflection and refraction values are combined:

```
//Hardness factor - user input
fresnelTerm = fresnelTerm * xDrawMode;

//just to be sure that the value is between 0 and 1;
fresnelTerm = fresnelTerm < 0? 0 : fresnelTerm;
fresnelTerm = fresnelTerm > 1? 1 : fresnelTerm;

// creating the combined color
float4 combinedColor = refractiveColor*(1-fresnelTerm) + reflectiveColor*(fresnelTerm);
```

Some screenshots of water with the differently adjusted Fresnel value in the demo application is visualized on Figure 5-5:

Figure 5-5: Screenshot of the demo application to visualize different Fresnel term approximations.

5.7 Creating Waves

To create an efficient wave effect for a bigger area the number of the vertices must be limited. In the lake water shader I used the following optimization techniques:

- Creating the water effect only by pixel shader. In this manner, the water can be made of a very limited number of vertices.
- Wave motion effect created only by bump-map.

The ripples of the water are animated by a moving bump-map. Read [Riemer] for more details about this technique. From an original wave picture it is possible to create the gradient map of the image which shows the perturbations of the surface. A gradient map is the same size as the original picture, and every pixel stores a vector in the RGB components. This vector defines the deviation from the original normal vector at every single point of the image. The original normal vector for an absolutely flat surface is (0;0;1), for more precise calculations the value-range (-1;1) can be scaled to the values of the color components: 1 will be the maximum (256), 0 will be scaled to the half (128), while -1 to the minimum (0). For example the vector (0;0;1) is scaled to (128;128;256). As long as the perturbations are not very significant, every pixel of the image should have some similar values to (128;128;256), and this means, the blue component has always the highest value. This results a predominantly blue gradient map, like the one on Figure 5-6:

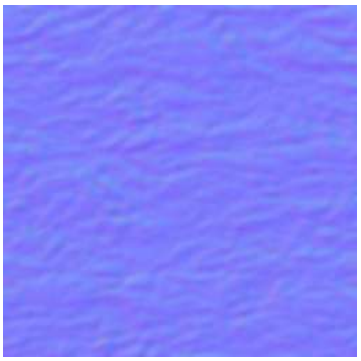


Figure 5-6: Gradiant map.

Gradient maps are usually called bump map in graphic development. In the XNA code we need to load the bump map and pass it as a parameter for the shaders, just like the elapsed time to get the waves in motion:

```
private Texture2D waterBumpMap;
waterBumpMap = content.Load ("waterbump");
effect.Parameters["xWaterBumpMap"].SetValue(waterBumpMap);elapsedTim
e += (float)gameTime.ElapsedGameTime.Milliseconds / 100000.0f;
effect.Parameters["xTime"].SetValue(elapsedTime);
```

In the HLSL code, the input bumpmap and sampler is defined:

```
Texture xWaterBumpMap;Texture xWaterBumpMap;
sampler WaterBumpMapSampler = sampler_state { texture = ; magfilter = LINEAR;
minfilter = LINEAR; mipfilter=LINEAR; AddressU = mirror; AddressV = mirror;};
```

The vertex shader passes the texture coordinates to the pixel shader. Note, that the *inTex* value is divided by the wavelength to make the bump map be stretched over the entire surface. Adding a time-dependant move-vector will move the waves:

```
struct WaterVertexToPixel
{
        float4 Position : POSITION
        float4 ReflectionMapSamplingPos : TEXCOORD1;
```

```
        float2 BumpMapSamplingPos : TEXCOORD2;
        ...
};
...
float2 moveVector = float2(0, 1);
Output.BumpMapSamplingPos = inTex/xWaveLength + xTime*xWindForce*moveVector;
```

At the beginning of the pixel shader code, the bump map is sampled, the values are scaled back and related to the wave height variable. Finally, the perturbation is added to the original coordinates:

```
float4 bumpColor = tex2D(WaterBumpMapSampler, PSIn.BumpMapSamplingPos);
float2 perturbation = xWaveHeight*(bumpColor.rg - 0.5f);
float2 perturbatedTexCoords = ProjectedTexCoords + perturbation;
```

The result can be get by sampling the reflection and refraction maps with the perturbated coordinates:

```
float4 reflectiveColor = tex2D(ReflectionSampler, perturbatedTexCoords);
float4 refractiveColor = tex2D(RefractionSampler, perturbatedRefrTexCoords);
```

To avoid incorrect edges at the border, the clipping pane can be set to higher point:

```
Vector4 planeCoefficients = new Vector4(planeNormalDirection, -waterHeight+1.0f);
```

In the final version of the source code, the wind direction is also a parameter to make the water move along the river and the rotation matrices are generated in the XNA code to gain some performance.

5.8 Adding Dull Color

To get more realistic result, some dark-bluish color is added to the final water color. This can be also adjusted by the user:

```
float4 dullColor = float4(0.1f, 0.1f, 0.2f, 1.0f);
float dullBlendFactor = xdullBlendFactor;
Output.Color = (dullBlendFactor*dullColor + (1-dullBlendFactor)*combinedColor);
```

5.9 Specular Highlights

Specular highlights are approximated by adding some light color to specific areas as the Phong illumination model describes. For computational reasons, the half-vector is used instead of the vector of reflectance. For more details about this, see chapter 2.4.1. The half-vector is also approximated, and some perturbations are added from the values of the bump-map (*specPerturb*). In this demo, I used the following code for this:

```
float4 speccolor;
float3 lightSourceDir = normalize(float3(0.1f,0.6f,0.5f));
float3 halfvec =
normalize(eyeVector+lightSourceDir+float3(perturbation.x*specPerturb,perturbation.y*specPerturb,0)
);
```

The angle between the surface-normal and the half-vector is calculated using the dot product between them. An input variable (*specpower*) adjusts the power, which results the specular highlights only in case of a very little angle between the vectors. Finally, the specular color is added to the original one.

```
float3 temp = 0;
temp.x = pow(dot(halfvec,normalVector),specPower);
speccolor = float4(0.98,0.97,0.7,0.6);
speccolor = speccolor*temp.x;
speccolor = float4(speccolor.x*speccolor.w,speccolor.y*speccolor.w,speccolor.z*speccolor.w,0);
Output.Color = Output.Color + speccolor;
```

On Figure 5-7 a screenshots of lake water specular highlights is visible:

Figure 5-7: Specular highlights of the demo application.

5.10 Summary

Rendering lake water is a huge challenge. I tried to introduce the main steps and ideas, but the number of different solutions is unlimited. The adjustable Fresnel term approximation and other effects can visualize the various possibilities, but to have a visually absolutely convincing result, more specific requirements need to be defined. My demo application can be extended and improved in myriad ways; I only can encourage the reader for experimentation. The entire shader code of the Lake Water Demo application can be found in chapter 10.3 - Appendix C – Shader Code of the Lake Water Demo Application. Some screenshots can be found in Appendix F – Screenshots of the Lake Water Demo Application. For more details and sample videos, check the homepage of the project ([HWS]).

6 OCEAN WATER SHADER

6.1 Introduction

Ocean water has several phenomena which are hard to simulate. There are different types of waves:

- Surging breakers roll onto steep beaches
- Plunging Breakers form tunnels if the beach slope is moderately steep
- Spilling breakers generate foam on gentle sloping beaches

For more details about different types of ocean waves, check [SNTFHP].

In the first part of this chapter, I introduce here briefly the main characteristics of an ocean water shader, published by [KRI]. In the second part, I discuss my demo application which focuses on the most difficult part of the ocean water shaders: wavy beaches. Although, the Navier-Stokes equations can handle all the different wave formations, their computational expense is huge. The other approaches discussed in earlier chapters can't really handle different type of waves, and the height-field representation has several limitations also. For all the reasons I mentioned earlier, I simulate sinusoidal waves moving on a height-field.

6.2 Water Shader in WWII Online: Battleground Europe

[KRI] published a complex water rendering approach, which was used in the game WWII:online as well (see [WW2OL]). The original presentation can be listened on-line as well [KRIPRES] (in Russian). The basis of their water shader is a general illumination equation, like the ones discussed in the chapter 2.4.1. Ambient light, diffuse light and also specular lights are taken into account. The Fresnel term is approximated with the cheapest method discussed in Alternative solutions: Simpler solution. The following screen-shot on Figure 6-1 visualizes the result:

Figure 6-1: Screen shot of the water in WWII:Online.

6.2.1 Optimizations

Approximations and heuristics were applied in their water shader to get higher frame rates. They used the following general ideas:

- Using cheaper calculations instead of most accurate ones.
- Refreshing the reflection map only a few times in a second. (This can result artifacts when the movement is too fast.
- Optimizing the shader code with different compilers.

These optimizations are needed to render visually convincing water surfaces real-time. Although, the list contains general ideas only, they are useful for every computationally complex scene. The details of the approaches need to be adjusted to the required results.

6.2.2 Combining layers

To get a visually convincing scene, several different techniques are combined. There are areas of water and land, but the border between them is much more complex. There is a wavy area around the coast - waves are going towards the beach. Where the water is shallow caustics are also visible. Finally, there is a zone, where the waves run out over the sand. This is visualized on Figure 6-2:

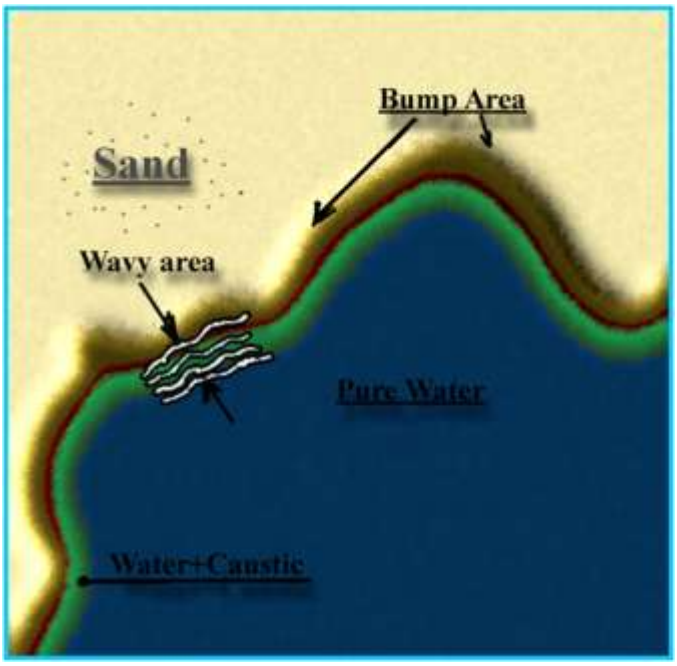


Figure 6-2: Layers of the water rendering technique in [KRI].

For a complex water-shader, the alternatives must be chosen after exhaustive consideration. The deepness of water, the viewing angles, the viewing distances and the length of waves are just some of the factors, which need to be taken into account.

6.2.3 Coastal Wave Formation

In their approach complex mathematical equations are used to determine the wave formations. The waves need to go towards the beach, and their sinusoidal shapes are displaced by

functions depending on horizontal coordinates, time and on an extra variable which influences the shape like weather conditions do. For more detailed explanations and mathematical background, see [KRI]. Figure 6-3 shows their final wave-formation:

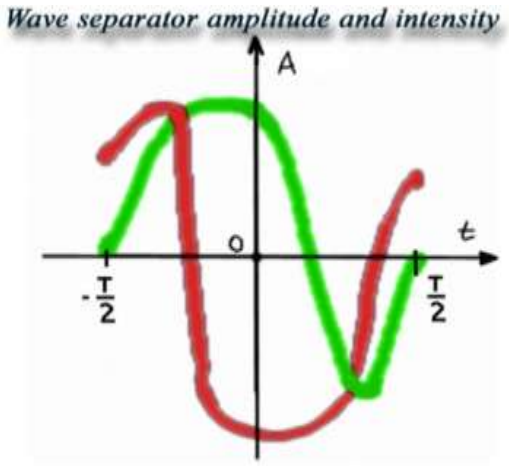


Figure 6-3: Wave form in [KRI].

6.2.4 HLSL code

Their shader code differentiates wave types. After the common variable declaration in the first part, they use 20 GPU instructions to calculate the reflections by a cube map, to compute the Fresnel term and the fog effect in the next section. They also use pre-generated normalvector-map to gain performance. The third part in their shader code calculates the appearance of high quality water. 55 instructions are needed to compute the permutated texture coordinates, determine the normal vectors, weather conditions, and reflected color. Cloud shadow and foam texture is added to the result as well. The amount of foam is computed by a separate function. These parts of their shader code can be found in Appendix D - Ocean Shader Code by [KRI].

6.3 Ocean Shader Demo Application

6.3.1 Introduction

To be able to demonstrate the differences between DirectX SDK and the XNA versions, I chose today's latest DirectX SDK instead of XNA Development Kit for handling managed code in the ocean water demo application. The functions, classes are different and the entire development needs different approaches in these two development framework.

In XNA we use instances of the Model class to handle objects which need to be rendered, while in DirectX SDK the Mesh class is used for the same purpose. The Model class is more complex, various Mesh levels can be handled together, but in the DirectX SDK there exist several functions to create an object and its properties, while in XNA we can only import models or directly determine the attributes of the model one by one. Because of this, it is more comfortable to work with the DirectX SDK now, but XNA will easily replace the support for managed development of the DirectX Development Kit later.

In this demo application I demonstrate the steps to create a beach with waves. The main goals are:

- Creating waves by vertex shader which are going towards the beach
- Applying some other compounds sinusoidal waves to distort the regular wave shapes

- Adjusting the wave speed, colors, daytime, reflectance etc. by user input.
- Adding some foam texture for more realistic result

Different approaches are needed to handle spray, god-rays, exact water color etc, and they are out of the scope of this demo application. For specific purposes, the techniques discussed in chapter 2 and 4 can be added to get more realistic scenes, but they need always be fine tuned not to destroy the other effects.

6.3.2 Vertices of the Water Surface

I created the vertices of the water surface according to the chapter *4.3.2* - LOD Algorithms on Water Surfaces. The grid is always placed in front of the camera to cover only the necessary part of the scene. The performance profit is enormous, I experimented with huge grid sizes, for example in the final version I use 100 x 100 = 10000 vertices. Bigger sizes can be handled real-time as well, but actually I don't render any unnecessary objects, the GPU works only on my water surface. The latest graphic cards are much faster than the one I use for development (Ati Radeon 9600), but generally other elements are also calculated and rendered not only the ones which are important for the water surface. The optimal size of the grid always depends on the target hardware and on the expected realism.

The approach discussed in [BMELAB2] determines the horizontal place of the vertices relative to the camera only once at the beginning. This means that if we change the angles between the camera and the water surface, we can extend the visible area to places which are not covered by the grid. This can be avoided by application of projected girds as the chapter *Using projected girds* describes. The demo application for that technique is also available on-line; I don't need to demonstrate the same approach in my application. I simply don't allow changing the angle between the camera and the grid to handle this problem.

The places of the vertices are determined (also based on the equations described in the chapter *LOD algorithms on water surfaces*) by these lines of code to conform the window size, aspect ratio and the size of the sky-dome as well:

```
float aspectRatio = Camera.AspectRatio*1.55f;
float d = 4000f / (nx - 1.0f) * (1.0f / (1.0f - (float)i / nx) -
1.0f);
Vector3 pos = new Vector3(d, aspectRatio * d * (float)(j - (ny - 1)
/ 2f) / ny, 0);
return (object)new CustomVertex.PositionOnly(pos);
```

The aspect ratio determines the distance between the vertices across the viewing direction, while the variable *d* helps to determine to coordinates along the viewing direction to cover approximately equal-size of the screen by the triangles on the rendered picture.

Finally, to get the expected result, the water is always moved into the front of the camera. The camera directions form a vector which is used to generate an inverse of the camera matrix. This inverse is used to transform the place of the water grid according to the place of the camera:

```
Vector3 pos = new Vector3(Camera.Position.X, Camera.Position.Y, 0);
Vector3 dir = Vector3.TransformCoordinate(
     new Vector3(Camera.Direction.X, Camera.Direction.Y, 0),
Matrix.RotationZ((float)Math.PI / 2));
```

```
Game.Device.Transform.World = Matrix.Invert(Matrix.LookAtLH(pos, pos
+ new Vector3(0, 0, 1), dir));
```

The result can be seen on Figure 6-4:

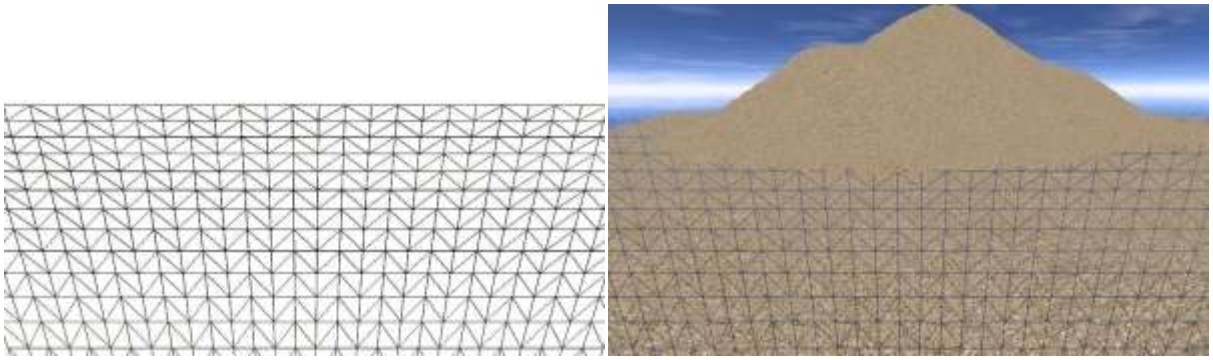


Figure 6-4: The gird of the ocean water shader demo application. The figure on the left shows the alignment on the triangles, which are in use on the right figure.

6.3.3 Waves - Getting Everything in Motion

The vertical place of the grid elements is determined by the vertex shader. It does not modify the horizontal places, only by changing the Z coordinates are the waves formed. The tricky part is the equations of the displacement. Although there are complex frequency-filtering techniques which can be used to generate waves going always towards the beach, they are out of the scope of this paper. I used an easier approach: my island is almost circle shaped, and I generate dominant radial waves moving towards the centre of the beach. This resembles the nature waves, which are generally going towards the dry-land.

The dominant radial waves are going to a center point which changes with the size of the island. The phase of the waves depends on the distance from the center point, and on two other variables influencing the space frequency and the time frequency of the waves.

```
//radial waves
float distance = length(float2(pPos.x-64*xIslandScale,pPos.y-
64*xIslandScale));
float Phase = distance * -rSpaceFreq + Time * 2 * -TimeFreq;
```

The height of the vertices is calculated by certain power of sinusoidal waves, which results steep waves on both sides. To reduce the slope on the back side of the waves, the amplitudes are reduced depending on the wave phase (if the product of the sine and cosine of the phase is less than 0, we are on the back side of the wave). To avoid negative waves the sine function is clamped to have only positive values. A correction term is also modifying the final result to allow the user adjusting the water height:

```
//Calculating waveheight
float Cos,Sin;
int power = 7;
sincos(Phase,Sin,Cos);
float temp2 = 1;
if (Cos*Sin < 0) temp2 = 2;
float WaveHeight = clamp(pow(Sin,power)*rAmplitudes*temp2-
(rAmplitudes*(temp2-1)),0,rAmplitudes) + (xIslandScale*128-
distance)/xWaterSlope/2+xWaterLevel;
```

At this point we have absolutely regular waves going towards the centre point of the island, but we need to add some smaller waves going in different directions to have a more realistic result. I add four of them with different amplitudes and phases going to simulate the random-like waves of natural water surfaces. The four waves can be handled together as a vector (float4). As we always need to perform the same operations on them, it is the same efficient to calculate the effect of one wave as four of them. We can compute the height of the sum of four sinusoidal waves with the following lines (the wave directions and the frequencies are vectors with four elements):

```
//calculating the correction
float4 crrectionPhase = (WaveDirX * pPos.x + WaveDirY * pPos.y) *
SpaceFreq + 10 * Time * TimeFreq;
float4 cCos,cSin;
sincos(crrectionPhase,cSin,cCos);
float correctionHeight = dot(cSin,Amplitudes/2);
```

We need to add this correction term to the wave height and calculate the normal vectors to allow nice visual effects applied in the pixel shader. The screen-shot of the result is visualized on Figure 6-5:

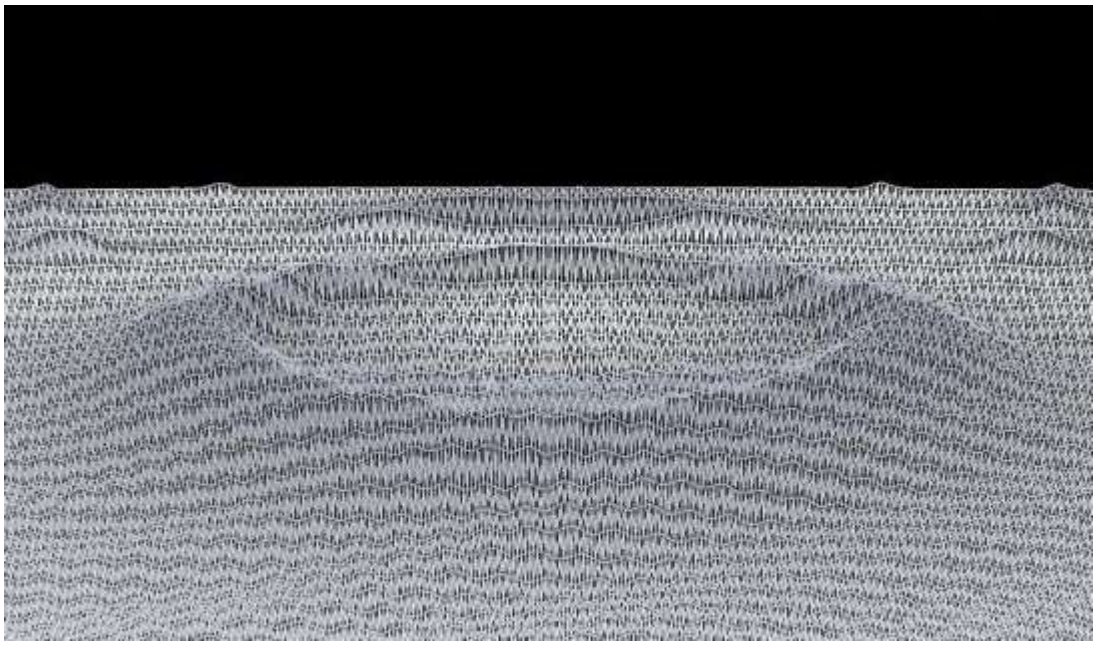


Figure 6-5: Screenshot of the demo application. The image shows the center point of the waves. The radial waves are the most significant; the other waves only influence the locel view – the pattern of the surface from this distance.

6.3.4 Optical Effects

Reflection is one of the most significant optical effects needed to render ocean surfaces as well. I tried two different method, rendering reflection using cube-maps and applying the same technique as in the lake-water shader. Like reflection maps, refraction-maps are pre-generated in a separate rendering phase.

```
// REFLECTIVE AND REFRACTIVE COLORS
float2 RefractionSampleTexCoords;
RefractionSampleTexCoords.x =
IN.RefractionMapSamplingPos.x/IN.RefractionMapSamplingPos.w/2.0f +
0.5f;
RefractionSampleTexCoords.y = -
IN.RefractionMapSamplingPos.y/IN.RefractionMapSamplingPos.w/2.0f +
0.5f;
float4 refractiveColor = tex2D(RefractionTextureSampler,
RefractionSampleTexCoords-newNormal*0.2f);
float4 reflectiveColor = tex2D(ReflectionTextureSampler,
RefractionSampleTexCoords-newNormal*0.2f);
```

The normal vectors of the triangles are distorted by a bump-map to have more random surface normals, like water surfaces have.

```
float3 normal = tex2D(BumpMapSampler,IN.texCoord0.xy)* 2.0 - 1.0;
float3 newNormal = normalize(IN.norm + normal);
```

Using the normal vector, ambient light is calculated. The power of the ambient light is adjustable by an input variable:

```
float3 LightDirection = normalize(float3(-13,-2,4));
float lightingFactor = saturate(saturate(dot(normal,
   LightDirection)) + xAmbient);
```

Addition of the dull color is a bit trickier. I wanted the far part of the ocean be darker, than the closer areas. If you look down into the water standing on a sandy beach, the water does not look dark by itself. But if we add the any amount of dark dull color to the water depending only on the distance from the viewer, the water becomes artificial. The top of the waves need some different color, than the bottom of them. To achieve this, the current phase of the waves influences the final color as well:

```
// ADDING DULL COLOR
float3 dullColor = float3(0.1,0.25,0.5);
     xDullFactor = clamp(xDullFactor* saturate(length(EyePos.xy-
IN.Position3D.xy)/100)+IN.phase*IN.phase/2,0,1);
     finalColor= finalColor * (1.0-xDullFactor) +
dullColor*xDullFactor;
```

Finally, the specular lights are added to the complex water system. The direction of the light and the eye-vector are the basis of the specular light calculation, just like in the Lake Water Shader Demo Application. For more details, see the explanation in chapter 5.9 or check the source code in Appendix E – Shader Code of the Ocean Demo Application. Figure 6-6

visualizes the result. For more screenshots, check Appendix G – Screenshots of the Ocean Demo Application.

Figure 6-6: Screenshot of the application

6.3.5 Summary

The main ideas of wave rendering were introduced in this chapter. Every piece of the jigsaw need to be carefully planned, otherwise they can negatively influence other elements. If you try to handle everything generally, you can have less convincing visual results, like me. Absolutely realistic beaches can be rendered, only the target hardware and performance can limit the possible approaches. The technique discussed here can be basis for several kind of realistic water rendering methods, and can be improved in every direction depending on the expected result. Particle systems, horizontal displacements, wave deformation, animated textures are just some of the ideas for possible improvements. The full shader code can be found in Appendix E – Shader Code of the Ocean Demo Application.

7 CONCLUSION

7.1 General

I tried to summarize the main steps to create various types of water surfaces. Although real-time realistic rendering is a great challenge, there are several possibilities to solve the different problems. The scientific background is complicated; I discussed only the mathematical and physical rules which are really necessary for water rendering.

The main real-time water animation techniques in the literature use Perlin noise, Fast Fourier transformations, Navier-Stokes equations or particle systems as basis of the animation. Optimizations can be done by appropriate selection of the grid points, to avoid unnecessary computations. The main ideas of Fresnel term approximation use the angle between the eye-vector and the surface normal-vector or some other polynomial function. Rendering reflections and refractions is generally achieved by a pre-rendering phase, but cube-maps and other techniques can be used also.

The demo applications demonstrate the numerous approaches. The expected results are achieved; the rendered scenes are realistic. The goals of the lake water shader are much more specific than the aims of the ocean water shader. Therefore, the lake water can seem more alive but the ocean scene can be adjusted to various expectations as well. I did not attempt to optimize the shader code for faster rendering. The attached source code produced 25 fps on an old PC with ATI Radeon 9600 graphics card and AMD Barton 2500 MHz CPU. The latest GPUs are significantly faster than the one used for developing.

Some other topics about water rendering can be found on the website of the project, please feel free to comment the articles on-line also (http://www.habibs.wordpress.com).

7.2 Future Work

Although the demo applications show some promising result, all the used techniques can be improved and extended to a much higher level for better realism, optimizations can be done for faster rendering and new solutions can be introduced or also combined with the existing ones. The possibilities are endless. Breaking waves, physical interaction, splashes and caustics can be also added to my applications, but creating an absolutely convincing water scene is still an extremely difficult task.

8 REFERENCES

[aEMfRUOEUGH]	Kei Iwasaki, Yoshinori Dobashi and Tomoyuki Nishita: An Efficient Method for Rendering Underwater Optical Effects Using Graphics Hardware – COMPUTER GRAPHICS FORUM, 2002.
[ARTfBRT]	P.S. Heckbert, "Adaptive Radiosity Textures for Bidirectional Ray Tracing," Proc. SIGGRAPH
[BMELAB2]	Game development laboratory material 2, BME, AUT, 2007
[BRT]	J. Arvo, "Backward Ray Tracing," ACM SIGGRAPH '86
[CBAfAWW]	J. Loviscach: A Convolution-Based Algorithm for Animated Water Waves, EUROGRAPHICS 2002
[CGaHLSLFAQ]	Fusion Industries: Cg and HLSL FAQ, [ONLINE] - May 2008 (http://www.fusionindustries.com/default.asp?page=cg-hlsl-faq)
[CgTOOLKIT]	nVidia: Cg Toolkit: A developer's Gide to Programmable Graphics – Release 1.4, September 2005. (http://developer.download.nvidia.com/cg/Cg_2.0/beta/AdditionalDocs/CgUsersManual.pdf)
[DSoSF]	James F. O'Brien and Jessica K. Hodgins: Dynamic Simulation of Splashing Fluids - Computer Animation '95
[DWAaR]	Lasse Staff Jensen and Robert Golias: Deep-Water Animation and Rendering – September 26, 2001. (http://www.gamasutra.com/gdce/2001/jensen/jensen_04.htm)
[EMaRoTWoNT]	Nathan Holmberg and Burkhard C. Wünsche: Efficient Modeling and Rendering of Turbulent Water over Natural Terrain, Computer graphics and interactive techniques in Australasia and South East Asia, 2004
[ESoLBoWbCTaTDT]	Geoffrey Irving, Eran Guendelman, Frank Losasso, Ronald Fedkiw: Efficient Simulation of Large Bodies of Water by Coupling Two and Three Dimensional Techniques, ACM SIGGRAPH 2006
[FDfP]	T. E. Faber: Fluid Dynamics for Physicists - Cambridge University Press, 1995
[FRMfRaRCDtWS]	Kei Iwasaki1, Yoshinori Dobashi and Tomoyuki Nishita: A Fast Rendering Method for Refractive and Reflective Caustics Due to Water Surfaces
[GPUGEMS]	Gpu Gems - Chapter 38. : Mark J. Harris: Fast Fluid Dynamics Simulation on the GPU (http://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch38.html)
[GS]	Uni Düsseldorf - Geometry Shaders [ONLINE], May 2008 (http://www.uni-duesseldorf.de/URZ/hardware/parallel/local/xsi/XSI_html/files/mental_ray/manual/ray-4_Geometry_Shaders.html)
[GW]	Jefrey Alcantara: Gerstner waves [ONLINE], May 2008 (http://www-viz.tamu.edu/students/jd/658/T_algorithms.html)
[HLSLI]	Chang Li : HLSL Introduction [ONLINE], May 2008 (http://www.neatware.com/lbstudio/web/hlsl.html)
[HWS]	Habib's Water Shader (http://www.habibs.wordpress.com)
[IAoOW]	Damien Hinsinger, Fabrice Neyret, Marie-Paule Cani: Interactive Animation of Ocean Waves (http://www-evasion.imag.fr/Publications/2002/HNC02/wavesSCA.pdf)

[IfCR]	D. Mitchell, P. Hanrahan, "Illumination from Curved Reflections," Proc. SIGGRAPH
[IISTfAC]	Chris Wyman, Scott Davis: Interactive Image-Space Techniques for Approximating Caustics
[IRoCuIWV]	Manfred Ernst, Tomas Akenine-Möller, Henrik Wann Jensen: Interactive Rendering of Caustics using Interpolated Warped Volumes
[IttD9HLSL]	Craig Peeper, Jason L. Mitchell: Introduction to the DirectX® 9 High Level Shading Language (http://ati.amd.com/developer/ShaderX2_IntroductionToHLSL.pdf)
[KRIPRES]	KRI presentation audio record [ONLINE], May 2008 (http://www.kriconf.ru/2004/rec/KRI-2004.Programming_20.ogg)
[LODf3DG]	Morgan Kaufmann: Level of Detail for 3D Graphics
[LVaPSPwD9]	James C. Leiterman: Learn Vertex and Pixel Shader Programming with DirectX® 9, WORDWARE PUBLISHING, 2004.
[LWIuBBT]	Mark Watt: Light-Water Interaction using Backward Beam Tracing – DIGITAL PICTURES
[MATH]	Jim Van Verth, Lars M. Bishop: Essential Math for Games Programmers [ONLINE], May 2008, (http://www.essentialmath.com/)
[MESHUGGAH]	Carsten Wenzel - Meshuggah Demo and Effect browser [ONLINE], May 2008. (http://meshuggah.4fo.de/)
[MFGD]	Mathematics for Game Developers - Christopher Tremblay - Thomson course technology.
[MHLSLR]	MSDN: Microsoft HLSL Reference, [ONLINE], May 2008 (http://msdn2.microsoft.com/en-us/library/bb509561%28VS.85%29.aspx)
[MSDN]	Microsoft Developer Network, [ONLINE], May 2008 http://msdn.microsoft.com/
[NGSaR]	Bryan Dudash: Next Generation Shading and Rendering. NVIDIA (ftp://download.nvidia.com/developer/presentations/2004/GPU_BBQ/English_Advanced_Shading.pdf)
[NSEP]	CHARLES L. FEFFERMAN: Existence and smoothness of the Navier-Stokes equation (http://www.claymath.org/millennium/Navier-Stokes_Equations/navierstokes.pdf)
[PN]	Wikipedia: Perlin Noise, [ONLINE], May 2008 (http://en.wikipedia.org/wiki/Perlin_noise)
[PN2]	Hugo Elias: Perlin Noise, [ONLINE], May 2008 (http://freespace.virgin.net/hugo.elias/models/m_perlin.htm)
[PNM]	Matt Zucker: The Perlin noise math FAQ, [ONLINE], May 2008 (http://www.cs.cmu.edu/~mzucker/code/perlin-noise-math-faq.html)
[RIEMER]	Riemer Grootjans: C# and XNA Tutorials, [ONLINE], May 2008 (http://www.riemers.net/)
[RNW]	Simon Premoze, Michael Ashikhmin: Rendering Natural Waters (http://www.cs.utah.edu/vissim/papers/water/waterColorPG.pdf)
[RT3DEUCADX9]	Greg Snook: Real-Time 3D Terrain Engines Using C++ and DirectX 9 (CHARES RIVER MEDIA INC.)
[RTSP]	Ron Fosner: Real-Time Shader Programming (Morgan Kaufmann Publishers © 2003)
[RTWR]	Real-time water rendering - introducing the projected grid concept (http://graphics.cs.lth.se/theses/projects/projgrid/)

[SHADERX]	Wolfgang F. Engel: Direct3D ShaderX, WORDWARE, 2002.
[SHADERX2]	Wolfgang F. Engel: ShaderX2: Shader Programming Tips & Tricks with DirectX 9, WORDWARE, 2003.
[SNTFHP]	Science & Technology Focus Homepage, [ONLINE], May 2008 (http://www.onr.navy.mil/Focus/ocean/motion/waves2.htm)
[TEoNIaRT]	International Conference on Computer Graphics and Interactive Techniques: The elements of nature: interactive and realisic techniques - ACM SIGGRAPH 2004 Course Notes.
[TLODRRGA]	Terrain LOD: Runtime Regular-Grid Algorithms, [ONLINE], May 2008 (http://www.vterrain.org/LOD/Papers/index.html)
[TYPHOON]	Stefano Lanza: Typhoon 3D engine, [ONLINE], May 2008 – (http://www.gameprog.it/hosted/typhoon/)
[UVTDFRWR]	Yuri Kryachko: Using Vertex Texture Displacement for Realistic Water Rendering – Addison-Wesley Professional, January 3 2006.
[WW2OL]	WWII Online: Battleground Europe (http://www.wwiionline.com)

9 SOURCE OF THE IMAGES

All the other images in this paper which are not listed above are made by me.

10 APPENDIX

10.1 Appendix A – DirectX and Supported Shader Versions

The different DirectX versions and supported shader versions are shown in the following table:

DirectX Version	Pixel Shader	Vertex Shader
8.0	1.0, 1.1	1.0, 1.1
8.1	1.2, 1.3, 1.4	1.0, 1.1
9.0	2.0	2.0
9.0a	2_A, 2_B	2.0
9.0c	3.0	3.0
10.0	4.0	4.0
10.1	4.1	4.1

10.2 Appendix B – Shader Version Comparison

10.2.1 Pixel Shader Comparison

	PS_2_0	PS_2_a	PS_2_b	PS_3_0	PS_4_0
Dependent texture limit	4	No Limit	4	No Limit	No Limit
Texture instruction limit	32	Unlimited	Unlimited	Unlimited	Unlimited
Position register	No	No	No	Yes	Yes
Instruction slots	32 + 64	512	512	≥ 512	≥ 65536
Executed instructions	32 + 64	512	512	65536	Unlimited
Texture indirections	4	No limit	4	No Limit	No Limit
Interpolated registers	2 + 8	2 + 8	2 + 8	10	32
Instruction predication	No	Yes	No	Yes	No
Index input registers	No	No	No	Yes	Yes
Temp registers	12	22	32	32	4096
Constant registers	32	32	32	224	16×4096
Arbitrary swizzling	No	Yes	No	Yes	Yes
Gradient instructions	No	Yes	No	Yes	Yes
Loop count register	No	No	No	Yes	Yes
Face register (2-sided lighting)	No	No	No	Yes	Yes
Dynamic flow control	No	No	No	24	Yes
Bitwise Operators	No	No	No	No	Yes
Native Integers	No	No	No	No	Yes

PS_2_0 = DirectX 9.0 original Shader Model 2 specification.
PS_2_a = NVIDIA GeForce FX-optimized model.
PS_2_b = ATI Radeon X700, X800, X850 shader model, DirectX 9.0b.
PS_3_0 = Shader Model 3.
PS_4_0 = Shader Model 4.
"32 + 64″ for Executed Instructions means "32 texture instructions and 64 arithmetic instructions."

10.2.2 Vertex Shader Comparison

	VS_2_0	VS_2_a	VS_3_0	VS_4_0
# of instruction slots	256	256	≥ 512	4096
Max # of instructions executed	65536	65536	65536	65536
Instruction Predication	No	Yes	Yes	Yes
Temp Registers	12	13	32	4096
# constant registers	≥ 256	≥ 256	≥ 256	16×4096
Static Flow Control	Yes	Yes	Yes	Yes

Dynamic Flow Control	No	Yes	Yes	Yes
Dynamic Flow Control Depth	No	24	24	Yes
Vertex Texture Fetch	No	No	Yes	Yes
# of texture samplers	N/A	N/A	4	128
Geometry instancing support	No	No	Yes	Yes
Bitwise Operators	No	No	No	Yes
Native Integers	No	No	No	Yes

VS_2_0 = DirectX 9.0 original Shader Model 2 specification.
VS_2_a = NVIDIA GeForce FX-optimized model.
VS_3_0 = Shader Model 3.
VS_4_0 = Shader Model 4.

10.3 Appendix C – Shader Code of the Lake Water Demo Application

```
struct WaterVertexToPixel
{
   float4 Position              : POSITION;
   float4 ReflectionMapSamplingPos    : TEXCOORD1;
   float2 BumpMapSamplingPos       : TEXCOORD2;
   float4 RefractionMapSamplingPos : TEXCOORD3;
   float4 Position3D              : TEXCOORD4;
};

struct WaterPixelToFrame
{
   float4 Color : COLOR0;
};

WaterVertexToPixel WaterVS(float4 inPos : POSITION, float2 inTex: TEXCOORD)
{
   WaterVertexToPixel Output = (WaterVertexToPixel)0;
   float4x4 preViewProjection = mul (xView, xProjection);
   float4x4 preWorldViewProjection = mul (xWorld, preViewProjection);
   float4x4 preReflectionViewProjection = mul (xReflectionView, xProjection);
   float4x4 preWorldReflectionViewProjection = mul (xWorld, preReflectionViewProjection);

   Output.Position = mul(inPos, preWorldViewProjection);
   Output.ReflectionMapSamplingPos = mul(inPos, preWorldReflectionViewProjection);
   Output.RefractionMapSamplingPos = mul(inPos, preWorldViewProjection);
   Output.Position3D = inPos;

    float4 absoluteTexCoords = float4(inTex, 0, 1);
    float4 rotatedTexCoords = mul(absoluteTexCoords, xWindDirection);
    float2 moveVector = float2(0, 1);
    // moving the water
    Output.BumpMapSamplingPos = rotatedTexCoords.xy/xWaveLength +
        xTime*xWindForce*moveVector.xy;
   return Output;
}

WaterPixelToFrame WaterPS(WaterVertexToPixel PSIn)
{
   WaterPixelToFrame Output = (WaterPixelToFrame)0;

   float2 ProjectedTexCoords;
   ProjectedTexCoords.x =
        PSIn.ReflectionMapSamplingPos.x/PSIn.ReflectionMapSamplingPos.w/2.0f + 0.5f;
   ProjectedTexCoords.y = -
        PSIn.ReflectionMapSamplingPos.y/PSIn.ReflectionMapSamplingPos.w/2.0f + 0.5f;

   // sampling the bump map
   float4 bumpColor = tex2D(WaterBumpMapSampler, PSIn.BumpMapSamplingPos);

   // perturbation of the color
   float2 perturbation = xWaveHeight*(bumpColor.rg - 0.5f);
```

```
    // the final texture coordinates
    float2 perturbatedTexCoords = ProjectedTexCoords + perturbation;
    float4 reflectiveColor = tex2D(ReflectionSampler, perturbatedTexCoords);

    float2 ProjectedRefrTexCoords;
    ProjectedRefrTexCoords.x =
        PSIn.RefractionMapSamplingPos.x/PSIn.RefractionMapSamplingPos.w/2.0f + 0.5f;
    ProjectedRefrTexCoords.y = -
        PSIn.RefractionMapSamplingPos.y/PSIn.RefractionMapSamplingPos.w/2.0f + 0.5f;
    float2 perturbatedRefrTexCoords = ProjectedRefrTexCoords + perturbation;
    float4 refractiveColor = tex2D(RefractionSampler, perturbatedRefrTexCoords);

    float3 eyeVector = normalize(xCamPos - PSIn.Position3D);
    float3 normalVector = float3(0,0,1);

////////////////////////////////////////////////
// FRESNEL TERM APPROXIMATION
////////////////////////////////////////////////
        float fresnelTerm = (float)0;

        if ( fresnelMode == 0 )
        {
          fresnelTerm = 1-dot(eyeVector, normalVector)*1.3f;
        } else
        if ( fresnelMode == 1 )
        {
          fresnelTerm = 0.02+0.97f*pow((1-dot(eyeVector, normalVector)),5);
        } else
        if ( fresnelMode == 2 )
        {
          float fangle = 1.0f+dot(eyeVector, normalVector);
          fangle = pow(fangle,5);
          // fresnelTerm = fangle*50;
          fresnelTerm = 1/fangle;
        }

        //    fresnelTerm = (1/pow((fresnelTerm+1.0f),5))+0.2f; //

          //Hardness factor - user input
          fresnelTerm = fresnelTerm * xDrawMode;

          //just to be sure that the value is between 0 and 1;
          fresnelTerm = fresnelTerm < 0? 0 : fresnelTerm;
          fresnelTerm = fresnelTerm > 1? 1 : fresnelTerm;

          // creating the combined color
          float4 combinedColor = refractiveColor*(1-fresnelTerm) +
        reflectiveColor*(fresnelTerm);

        ////////////////////////////////////////////////
        // WATER COLORING
        ////////////////////////////////////////////////
```

```
                float4 dullColor = float4(0.1f, 0.1f, 0.2f, 1.0f);
                float dullBlendFactor = xdullBlendFactor;

                Output.Color = (dullBlendFactor*dullColor + (1-dullBlendFactor)*combinedColor);

        ////////////////////////////////////////////
        // Specular Highlights
        ////////////////////////////////////////////

          float4 speccolor;

              float3 lightSourceDir = normalize(float3(0.1f,0.6f,0.5f));

              float3 halfvec =
              normalize(eyeVector+lightSourceDir+float3(perturbation.x*specPerturb,perturbation.y
              *specPerturb,0));

              float3 temp = 0;
              temp.x = pow(dot(halfvec,normalVector),specPower);
              speccolor = float4(0.98,0.97,0.7,0.6);

              speccolor = speccolor*temp.x;

              speccolor =
              float4(speccolor.x*speccolor.w,speccolor.y*speccolor.w,speccolor.z*speccolor.w,0);

              Output.Color = Output.Color + speccolor;

          return Output;
        }

technique Water
{
   pass Pass0
   {
      VertexShader = compile vs_2_0 WaterVS();
      PixelShader = compile ps_2_0 WaterPS();
   }
}
```

10.4 Appendix D – Ocean Shader Code by [KRI]

10.4.1 STARTUP PART

```
arbfp1 half4 main {
        uniform half4 cc[6],
        in half4 Col0       : COLOR0,
        in half4 Col1       : COLOR1,

        in float2 NVec      : TEXCOORD0,

        in float2 NVec1     : TEXCOORD1,
        in float2 NVec2     : TEXCOORD3,
        in float3 NCubM     : TEXCOORD2,

        in float3 EyeVec    :TEXCOORD4,
        in float4 ScrPrj    :TEXCOORD5,
        in float2 LMCrd     :TEXCOORD6,
        in float2 RMCrd     :TEXCOORD7,

        uniform sampler2D       NMap,
        uniform sampler2D       NMap1,
        uniform samplerCube     NormCMap,
        uniform sampler2D       NMap2,
        uniform samplerCube     EnvCMap,
        uniform sampler2D       WRefl,
        uniform sampler2D       LMap,
        uniform sampler2D       RMap
} : COLOR
```

10.4.2 Fastest Water shader – 20 instructions using only cub map, Fresnel term and fog computation

```
{
        fixed3 N = 2*tex2D(NMap1,NVec1).xyz-1;

        fixed3 Eye = -texCUE(NormCMap, EyeVec.xyz).xyz * 0.5;

        half dotEye_N = dot(Eye,N);
        fixed3 reflectVec = 2*N*dotEye_N – Eye /**dot(N,N)*/;

        half3 reflectColor = texCUBE(EnvCMap, reflectVec).xyz * 8;

        fixed3 diffuseL = saturate(dot(N, c2.xyz));
        fixed3 diffuse = diffuseL * c0.xyz;
        half Fresnel = pow((1-max(dotEye_N,c3.b)),5)c3.g + c3.r;

        half3 WaterColor = reflectColor*Fresnel+diffuse;
        float RevFa = 1.0-Col0.a;
        half4 Out; Out.xyz = WaterColor * RefFa + Col1.rgb;
        Out.w = Fresnel * RevFa + col0.a;
        return Out;

}
```

10.4.3 High quality water shader with 8 textures and 55 instructions

```
        half Wavy = tex2D(RMap, RMCrd).w*c3.w;              // 1→ 1.5, 0.5→ 0.75

        fixed3 N0 = tex2D(NMap, NVec).xyz;
        fixed3 N1 = tex2D(NMap1, NVec1).xyz;
        fixed3 N2 = tex2D(NMap2, NVec2).xyz;

        fixed3 N21 = lerp(N2,N1, c4.x);
        half3 N = lerp(N21, N0, c4.y) – 0.5;

        N.z *= Wavy; N = normalize(N);
        half4 Scale_dsdt = half4(0.1*0.7,0.05*0.7,0.0,0.0);
        half4 TexCrd = ScrPrj*pow(ScrPrj.w, -1.25);
        TexCrd = TexCrd + Schale_dsdt * N.xyzz; // permutate original texture coordinates

        fixed3 CloudsShadow = tex2D(LMap,LMCrd).xyz * c1.x + c1.y;
        fixed3 PlaneRefl = tex2Dproj(WRefl, TexCrd).xyz;
        half3 Eye = -texCUBE(NormCMap, EyeVec.xyz).xyz*0.5;
        Eye = normalize(Eye);

        half dotEye_N = dot (Eye, N);
        half3 reflectVec = 2*N*dotEye_N – Eye /**dot(N,N*/;

        half3 reflectColor = texCUBE(EnvCMap, reflectVec).xyz * 8;
        fixed3 diffuseL = saturate(dot(N, c2.xyz));
        fixed3 diffuse = diffuseL*c0.xyz;

        reflectColor = lerp(reflectColor, PlaneRefl, c4.z);
        half Fresnel = pow((1-max(dotEye_N, c3.b)), 5) * c3.g + c3.r;
        refectColor  * = Fresnel;
        half3 WatercColor = reflectColor + diffuse;

        half3 F3 = ((N21-0.5) * half3(3.15,3.15,0))/Wavy; half Foam = dot(F3, F3);
        Foam = saturate(pow(Foam,4))*0.75;
        half3 FoamColor = diffuseL*c5.xyz + c5.w;

        WaterColor = lerp(WaterColor , FoamColor, Foam) * CloudsShadow;

        float RevFa = (1.0-Col0.a);
        half4 Out; Out.xyz = WaterColor*RevFa+Col1.rgb;
        Out.w = Fresnel*RevFa+Col0.a;
        return Out;

}
```

10.4.4 FOAM SHADER: 4 WAVES, WITH DRY COAST, 4 TEXTURES. 28 INSTRUCTIONS ALTOGETHER

```
half4 computeFoam (uniform sampler2D sCoast,
                   uniform sampler2D sFoam0,
                   uniform sampler2D sFoam1,
                   uniform sampler2D sSin,
                   uniform sampler2D sNoise,
                   uniform sampler2D Lmap
        ) : COLOR
{
        fixex4 Coast = tex2D(sCoast, vCoast);
        fixex4 Foam0 = tex2D(sFoam0, vFoam0);
        fixex4 Foam1 = tex2D(sFoam1, vFoam1);
        fixex4 SinA01 = tex2D(sSin, vSin);
        fixex4 SinB01 = tex2D(ssin, vsinB);

        half4 FoamC = Foam1;

        fixed4 Dissolve = SinA01.zwzw; Dissolve.zw = SinB01.zw;
        fixed4 SinA = SinA01; SinaA.zw = SinB01.xy;

        fixed dA = Coast.a + c4.y + FoamC.r +0.5;
        half4 A = SinA*c4.x+dA;

        fixed SeaFactor = Col0.g ; Dissolve*=SeaFactor;

        A = saturate(A*c0+c1)*saturate(A*c2+c3);
        half4 Lighting = c4.z;
        Lighting.a = saturate(dot(A,Dissolve));

        FoamC = Lighting;
        half4 T = FoamC.aaaa; T.a = -1;
        FoamC = FoamC*T + half4(0,0,0,1);

        //Fog
        FoamC.rgb = FoamC.rgb*Col1.r+Col1.rgb*(1-FoamC.a);
        return FoamC;
}
```

10.5 Appendix E – Shader Code of the Ocean Demo Application

```
////// STRUCTURES

struct vs_in
{
		float4 position : POSITION;
};

struct vs_out
{
		float4 position                                    : POSITION;
		float2 texCoord0                                   : TEXCOORD0;
		float3 norm                                        : TEXCOORD1;
		float4 RefractionMapSamplingPos : TEXCOORD2;
		float4 phase                                       : TEXCOORD3;
		float3 eyeVector                                   : TEXCOORD4;
		float3 view                                        : TEXCOORD5;
		float4 Position3D                                  : TEXCOORD6;
};

/*********************************************
Vertex SHADER
*********************************************/

vs_out WaterVS(vs_in IN)
{
		// variables
		vs_out OUT;
		float4 pPos = mul(IN.position, World);

		//calculating the correction
		float4 crrectionPhase = (WaveDirX * pPos.x   + WaveDirY * pPos.y) * SpaceFreq + 10
		* Time * TimeFreq;
		float4 cCos,cSin;
		sincos(crrectionPhase,cSin,cCos);
		float correctionHeight = dot(cSin,Amplitudes/2);

		//radial waves
		float distance = length(float2(pPos.x-64*xIslandScale,pPos.y-64*xIslandScale));
		float Phase = distance * -rSpaceFreq + Time * 2 * -TimeFreq;


		//Calculating waveheight
		float Cos,Sin;
		int power = 7;
		sincos(Phase,Sin,Cos);
		float temp2 = 1;
		if (Cos*Sin < 0) temp2 = 2;
		float WaveHeight = clamp(pow(Sin,power)*rAmplitudes*temp2-
		(rAmplitudes*(temp2-1)),0,rAmplitudes) + (xIslandScale*128-
		distance)/xWaterSlope/2+xWaterLevel + correctionHeight;
```

```
                    float4 newPos = IN.position + float4(0,0,WaveHeight,0);

////////////////////////////////////////

                    OUT.position = mul(newPos,WorldViewProj);

                    float3 worldPos = mul(IN.position, World);
                    OUT.texCoord0 = float2(worldPos.x/10, worldPos.y/10)+float2(-0.01f,0.04f)*Time;
                    OUT.view = normalize(worldPos - EyePos.xyz);

                    float temp = 0;
                    if (Sin > 0) temp = power*Cos*pow(Sin,power-1);

                    float4 CosWaveHeights = (temp+dot(cSin,Amplitudes/2)) * Amplitudes * SpaceFreq;

                    OUT.norm = normalize(cross( float3(0,1,0), float3(1,0,0)));
                    OUT.RefractionMapSamplingPos = mul(newPos, WorldViewProj);
                    OUT.phase =Sin*Sin/2+0.5;
                    OUT.eyeVector = normalize(EyePos-newPos);
                    OUT.Position3D = pPos;
                    return OUT;
}

/*******************************************
PIXEL SHADER
*******************************************/

float4 WaterPS(vs_out IN) : COLOR
{
                    float3 normal = tex2D(BumpMapSampler,IN.texCoord0.xy)* 2.0 - 1.0;
                    float3 newNormal = normalize(normalize(IN.norm) + normal);

                    //lighting factor computation
                    float3 LightDirection = normalize(float3(-13,-2,4));
                    float lightingFactor = saturate(saturate(dot(normal, LightDirection)) + xAmbient);
                    //newNormal volt

/*******************************************
// REFLECTIVE AND REFRACTIVE COLORS
*******************************************/

   float2 RefractionSampleTexCoords;
   RefractionSampleTexCoords.x =
                    IN.RefractionMapSamplingPos.x/IN.RefractionMapSamplingPos.w/2.0f + 0.5f;
   RefractionSampleTexCoords.y = -
                    IN.RefractionMapSamplingPos.y/IN.RefractionMapSamplingPos.w/2.0f + 0.5f;

   float4 refractiveColor = tex2D(RefractionTextureSampler, RefractionSampleTexCoords-
                    newNormal/40);
   float4 reflectiveColor = tex2D(ReflectionTextureSampler,
                    RefractionSampleTexCoords+newNormal/40);
```

```
                float phase = (rAmplitudes-clamp(IN.phase,0,rAmplitudes));

                float fresnelTerm = saturate(length(EyePos.xy-
                IN.Position3D.xy)/xFresnelDistance)+0.0000001;

                float3 finalColor = reflectiveColor*fresnelTerm+ refractiveColor * (1-fresnelTerm);

/*******************************************
Adding dull color
*******************************************/
                float3 dullColor = float3(0.1,0.25,0.5);

                xDullFactor = clamp(xDullFactor* saturate(length(EyePos.xy-
                IN.Position3D.xy)/100)+IN.phase*IN.phase/2,0,1);
                finalColor= finalColor * (1.0-xDullFactor) + dullColor*xDullFactor;

/*******************************************
Specular lights
*******************************************/

                float3 lightDir = normalize(float3(10.84,-12.99,3));
                float3 eyeVector = normalize(EyePos.xyz - IN.Position3D.xyz);
                float3 halfVector = normalize(lightDir + eyeVector);

                float temp = 0;
                temp = pow(dot(halfVector,normalize(IN.norm+normal/1.5)),200);
                float3 specColor = float3(0.98,0.97,0.7)*temp;

                finalColor = finalColor*lightingFactor+specColor;

                return float4(finalColor, 1.0f);
}

////////////////// TECHNIQUE DEFINITION

technique Water
{
                pass P0
                {
                        VertexShader = compile vs_2_0 WaterVS();
                        PixelShader = compile ps_2_0 WaterPS();
                }
}
```

10.6 Appendix F – Screenshots of the Lake Water Demo Application

Figure 10-1: Lake water shader screenshot

Figure 10-2: Specular reflection

10.7 Appendix G – Screenshots of the Ocean Demo Application

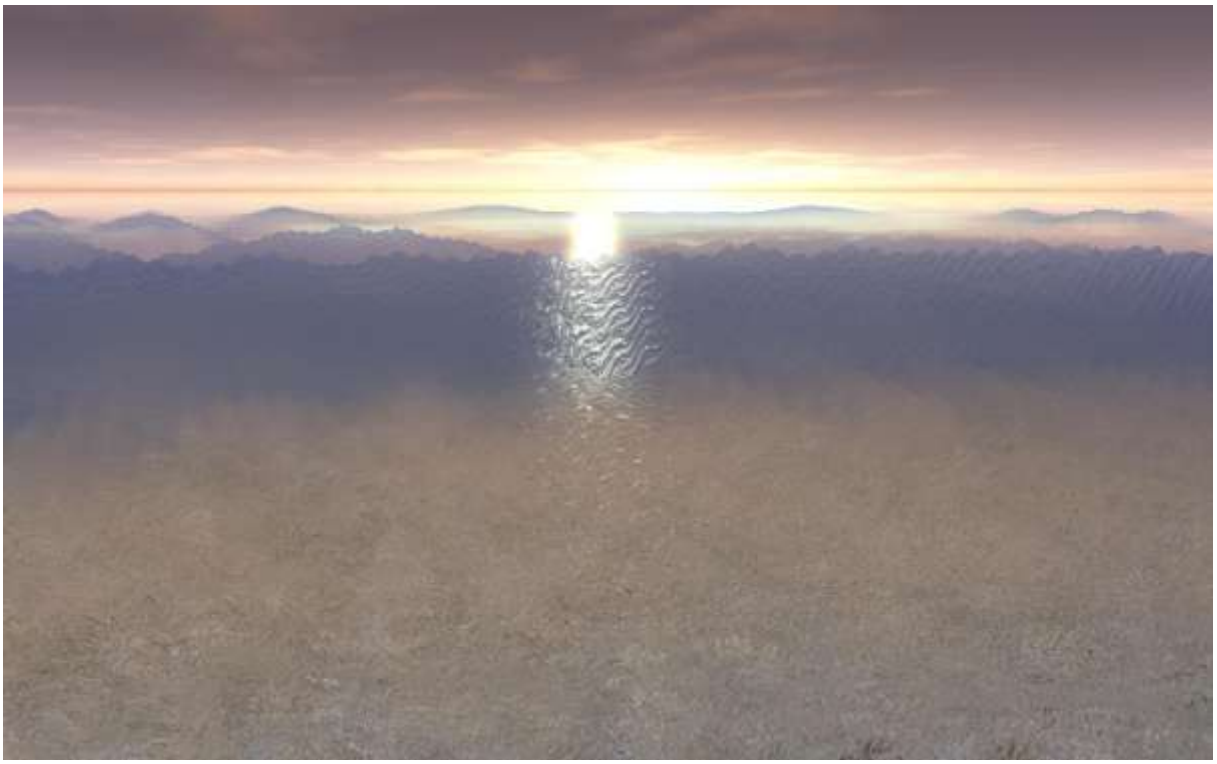
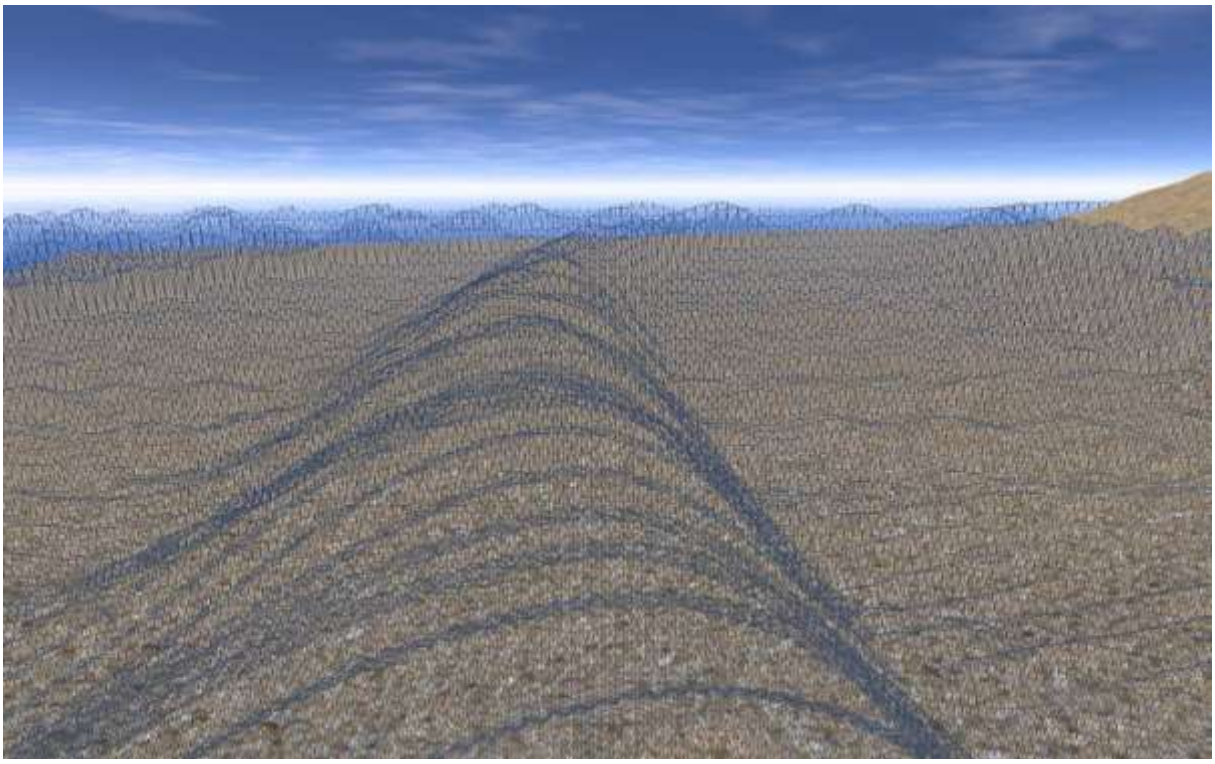


Figure 10-3 – View from the beach



Rigure 10-4: The wave form - rendered in wireframe mode