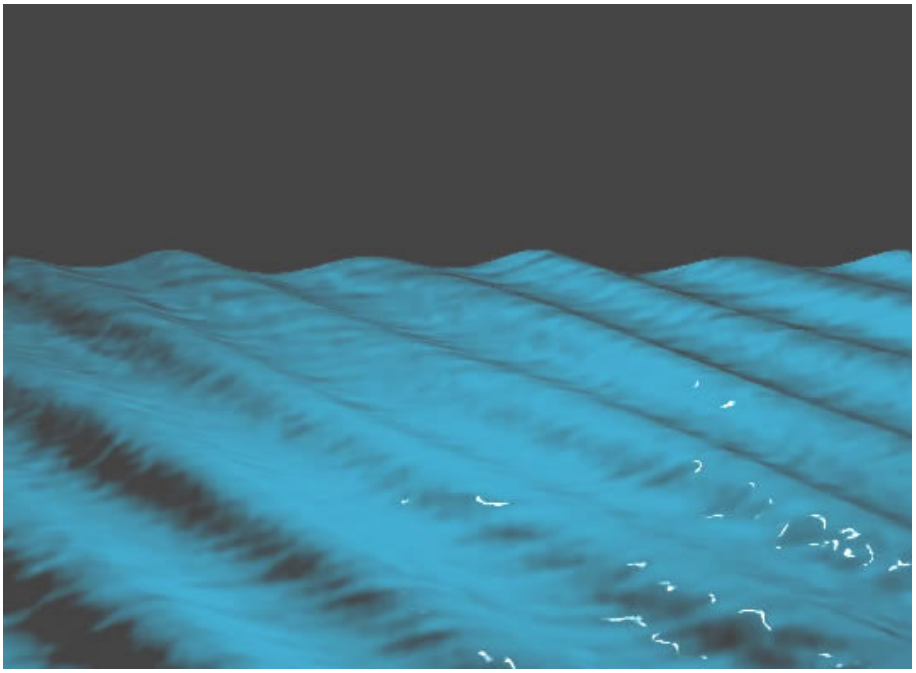
Water simulation with Cg

Håvard Homb - [280982-hah1]
Peder Johansen - [211081-pej1]
Supervisor: Kim Steenstrup Pedersen



This report is the result of a twelve week project cluster in computer graphics at the IT University of Copenhagen, Media Technology and Games program. The project was concluded May 2006.

Abstract

Water simulation is used in different fields like industrial research and computer games. Water simulation in games is done mainly by manipulating the way a surface reflects specular and diffuse light. Useful techniques for creating water simulation are bump mapping, normal mapping and displacement mapping. This report describes the implementation and theory of several water simulation techniques. The implementation focuses on using Cg to create the water simulation directly on the graphics card. Cg is a high level shading language developed by Nvidia, to program per-pixel lighting using vertex and pixel shaders. The implementation described also uses OpenGL and glut. The report also briefly covers water interaction.

Contents

1 Introduction **1**

1.1 Background 1

1.1.1 A short history of graphics and water effects in 3D games 1

1.1.2 Industry water simulation 2

1.2 Aim of This Project 2

1.3 Overview of the Report 3

2 Theory **3**

2.1 Cg 3

2.1.1 Vertex programs 4

2.1.2 Fragment programs 4

2.2 Waves 4

2.2.1 Power waves 6

2.2.2 Gerstner waves 6

2.3 Lighting 7

2.3.1 Phong illumination model 7

2.3.2 Ambient color 7

2.3.3 Diffuse color 8

2.3.4 Specular color 8

3 Analysis **8**

3.1 Initial specification 10

3.2 Ripples 10

3.3 Texturing 11

3.4 Normal mapping 12

3.5 Phong illumination model 13

3.6 Interaction with the water 13

3.7 Revisions 14

3.7.1 Sinusoid waves 14

3.7.2 Gerstner waves 14

3.7.3 Water interaction/ Ripples 14

4 Implementation **15**

4.1 Limitations of Cg . . . 15
4.2 OpenGL . . . 15
4.2.1 Main program . . . 16
4.2.2 Surface mesh . . . 16
4.2.3 Waves in OpenGL . . . 17
4.3 Water interaction . . . 18
4.4 Cg programs . . . 19
4.4.1 CgSineWave . . . 19
4.4.2 CgGertsnerWave . . . 19
4.4.3 CgRippleWave . . . 19
4.4.4 CgPhongNorm and CgPhongTexture . . . 19

5 Results **20**
5.1 Sine waves . . . 20
5.2 Gerstner waves . . . 20
5.3 Ripples . . . 22
5.4 Displacement VS normal rotation . . . 22
5.5 Performance . . . 23

6 Conclusion **26**

7 Future Work **26**
7.1 Improved Graphics card . . . 26
7.1.1 More waves . . . 26
7.1.2 Dragging an object though the water . . . 26
7.1.3 Animating Normal Maps . . . 26
7.2 Transparency and reflection . . . 27
7.3 Better Interactivity . . . 27
7.4 Approximation of procedural waves . . . 27

8 Refrences **28**

A Cg Code **30**
A.1 CgSineWave . . . 30
A.2 CgGerstnerWave . . . 30
A.3 CgRippleWave . . . 30

A.4 CgPhongNormal 30
A.5 CgPhongTexture 30

1 Introduction

1.1 Background

Cg is a high level language used to program graphics directly on the graphics hardware in modern computers. Cg stands for "C for graphics"[1], and is inspired by the ANSI C programming language standard. Cg was developed by Nvidia, which is one of the two biggest suppliers of graphics cards on a world basis (The other one being ATI). It was developed to make it easier for programmers to program the graphics cards, since they had to program the graphics processors by assembly earlier. The functionality in Cg is very hardware dependent, in the sense that the newest shaders often requires the latest graphics cards to run. Cg is made to work with Nvidia graphics cards, but also works well with other brands, by using compilers provided by the specific hardware manufacturers.

1.1.1 A short history of graphics and water effects in 3D games

High level programming languages for graphic cards like Cg and HLSL (High Level Shading Language [2]) has been developed as a result of a fast growing computer game industry in the recent years. Water simulation in games however, where literally nonexistent up to 2001, when Nvidia released the GeForce3 chip. The GeForce3 had programmable vertex and pixel processors that made it possible to calculate per-pixel reflections and lighting on uneven surfaces. This technology is the basis for real time water simulation.
The first 3D games, like Doom and Wolfenstein 3D (id Software, 1993 and 1992), were not built up by actual 3D worlds. They where built using sprites in a way that gave the impression of a 3D environment. The first real 3D game using a raster game engine and polygonal models was Descent, released in 1995. An OpenGL version of Quake (GLQuake) was released early 1997 by id Software, and this was the first game using a engine with OpenGL support. GLQuake experimented with simple reflections and semitransparent surfaces, and was the first software to show the capabilities of the 3Dfx "Voodoo" chipset. Still, water was simulated using simple textures as illustrated in figure 1, which is a screenshot from Duke Nukem 3D (3D Realms, 1996).
Towards the end of the decade, games got better and better graphics with higher level of detail. Water, however, was still implemented only using semitransparent textures, and reflection were often approximated by drawing them statically in the textures. Some nice looking waves could be made by using animated textures.
The first realistic water simulation were introduced when game developers started to use bump mapping and normal mapping in games. Normal mapping is explained in a section later in the report. Techniques like normal mapping and bump mapping requires the per-pixel lighting capabilities of a GeForce3 chip or newer. One of the first games using pixel shaders for water simulation was Elder Scrolls III: Morrowind (Bethesda, 2002). Doom 3 (id Software, 2004) took full advantage of the new GeForce3 technology, and showed the game industry the possibilities with normal and bump mapping.

Figure 1: Water in Duke Nukem 3D.

The latest and most realistic real-time simulations of water, as we know of, can be seen in the upcoming game "FarCry Instincts: Predator" for the Xbox 360 (Crytek, 2006). These water simulations are made by combining normal mapping, texture mapping and displacement of vertices in the water surface. The information for this section was gathered from Wikipedia [3].

1.1.2 Industry water simulation

While computer graphics hardware development has been driven by a growing game industry, a lot of theory for describing complex waves have been developed by scientists. Industry working with fluid dynamics are often dependent of simulating waves in fluids on computers. This way they can build better equipment without wasting money on experimenting on expensive prototypes that might not work.
The level of realism in industry simulations can probably not be compared to water simulations in games, since games only try to make something that looks real to the player. Still, it is clear that scientific approaches has inspired effects in games. For example the Gerstner waves that we use in this project, was first described by FrantiŽek Josef Gerstner (1756-1832), a Czech physicist and engineer.

1.2 Aim of This Project

Since this project is a project cluster in computer graphics, the main goal is to learn general computer graphics theory and implementation. Through the project, we want to acquire the same skills and knowledge that we would have acquired by following the course Computer graphics at ITU.

Furthermore, we want to learn how to use Cg [1] to create realistic, high-detail, real-time animations.
To get some practical experience in computer graphics and Cg, we will implement a simple water simulation software. The software will be written in C++, using OpenGL, glut and Cg. Water effects should be simulated by displacing vertices in a mesh, based on sinusoidal functions, as explained in [4]. Some texture and bump/normal mapping could be added for some extra realism.
If we get the time, we would like to implement one or more other features as well. This would be features like: Water caustics, interaction with the water surface, real-time adjustable parameters like wind and so on, and finally it would be very rewarding for us if we managed to create the tool as a library in a typical game-engine. This last feature, however, requires a very careful program design to fit with the standards of the engine in question, and it might be to much work for a project cluster like this.

1.3 Overview of the Report

This report consists of 5 major sections; "Theory", "Analysis", "Implementation", "Results", and "Conclusion".
The theory section explains theory that will be used later in the project. This is mostly theory about water simulation, but also a bit about the Phong illumination model, and a brief section about Cg in general. The second section, the analysis chapter is an analysis of how we plan to implement the parts that cannot be directly implemented from the theory. This section covers areas from creating ripples in the water, to texturing and normal mapping the water surface. The implementation chapter explains what we managed to implement, and how we implemented it. This section runs through every part of our program and explains each part individually. Following the implementation chapter, is the result chapter. This chapter contains images that are taken from our different programs, and explanations of them. The last major chapter is the conclusion of our project, followed by a section about future work, where we break down further improvements into specific sections.

2 Theory

2.1 Cg

All newer graphics cards has a Graphical Processing Unit (GPU) that has two programmable processors. These processors can be programmed to lighten the workload of the CPU in computer games, and other graphical programs. The first of the processors is the vertex processor, which is programmed by writing vertex-programs in the Cg language. The other processor is called the fragment processor, also called the pixel processor. Figure 2 shows how the data flow goes through the GPU.

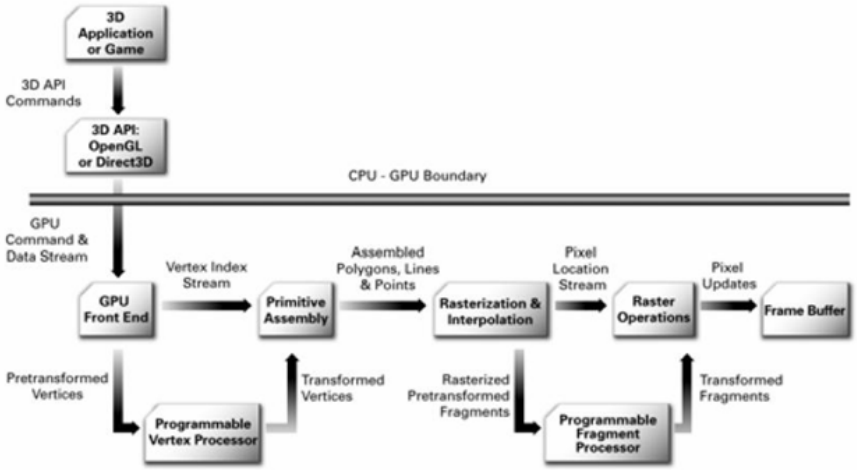


Figure 2: The pipeline of Cg [5]

2.1.1 Vertex programs

Vertex programs are pieces of code that run through every vertex on the screen, and processes them. It can mainly do two operations on the vertices, and that is transforming them and set their colour. The vertex program also can pass on parameters to the fragment program, such as vertex normals and vertex positions.

2.1.2 Fragment programs

Fragment programs are programs that works with every pixel on the screen. The only operation it can do on the them, is to set their color. As mentioned the vertex program can pass on parameters to a fragment program, and when it receives these parameters it will interpolate the normals and positions between the closest vertices to find an approximate value of the pixel.

2.2 Waves

There are many ways to describe waves, but most of them have a common feature; they are periodical. Periodical waves can be described in mathematics using sinusoidal functions. A simple sinusoidal wave can then easily be described in terms of frequency (w), amplitude multiplier (A) and time (t).

$$f(x,t) = A\sin(wx - t) \tag{1}$$

A sum of simple cosine and sine waves like this can be used to create any periodical wave.
If we extend equation 1 to two dimensions by adding a direction ($\vec{D}$) and a

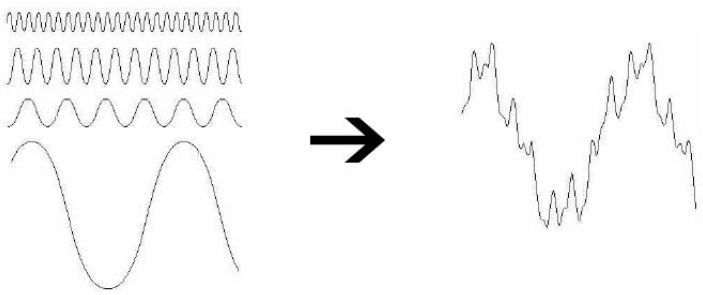


Figure 3: A complex wave created as a sum of simple sine waves. (Figure taken from [6])

speed multiplier (S), we get

$$W(x, y, t) = A\sin(\vec{D}\cdot(x, y)w + t\phi), \tag{2}$$

where the speed is expressed as a phase-constant $\phi = Sw$.

A wavy surface containing several sine waves is on the form:

$$H(x, y, t) = \sum(A\sin(\vec{D}\cdot(x, y)w + t\phi)) \tag{3}$$

Here $H(x, y, t)$ describes the height of the surface in a given x and y coordinate, and the point $\mathbf{P}(\mathbf{x}, \mathbf{y}, \mathbf{t})$ on the surface at a given time t, can be found by:

$$\mathbf{P}(\mathbf{x}, \mathbf{y}, \mathbf{t}) = (\mathbf{x}, \mathbf{y}, \mathbf{H}(\mathbf{x}, \mathbf{y}, \mathbf{t})) \tag{4}$$

The normal vector ($\vec{N}(x, y, t)$) for the point ($\mathbf{P}(\mathbf{x}, \mathbf{y}, \mathbf{t})$), is the cross product of the binormal and the tangent of the points. The binormal is the partial derivative in the x direction:

$$\vec{B}(x, y, t) = (\frac{dx}{dx}, \frac{dy}{dx}, \frac{d}{dx}H(x, y, t)) \tag{5}$$

$$\vec{B}(x, y, t) = (1, 0, \frac{d}{dx}H(x, y, t)) \tag{6}$$

while the tangent is the partial derivative in the y direction:

$$\vec{T}(x, y, t) = (\frac{dx}{dy}, \frac{dy}{dy}, \frac{d}{dy}H(x, y, t)) \tag{7}$$

$$\vec{T}(x, y, t) = (0, 1, \frac{d}{dy}H(x, y, t)) \tag{8}$$

The cross product of these two vector yields the normal at the coordinate x, y at the given time t:

$$\vec{N}(x, y, t) = (-\frac{d}{dx}(H(x, y, t)), -\frac{d}{dy}(H(x, y, t)), 1) \tag{9}$$

Using this kind of waves to simulate water is called The Sum of Sines Approximation. To get more water-like waves with sharper tops, power-waves and Gerstner waves are commonly used.

2.2.1 Power waves

Power wave is the name we use on waves that are to the power of a number, i.e

$$f(x) = \frac{1}{2}\sin(x)^4 + \frac{1}{3}\sin(x)^2 \tag{10}$$

These waves has a different shape than normal sine waves, and we can use this to our advantage. By adding together power waves that are to the power of a product of 2, we can create complex waves like saw-tooth looking waves (Figure 6). An example of a power wave can be seen in figure 4. We don't want to use waves to the power of an odd number, since these waves gives saddle points in $f = 0$ when the wave alternates between positive and negative values for the wave. We believe that these waves looks unnatural as water.

2.2.2 Gerstner waves

The difference between Gerstner waves and normal sinus waves is that the Gerstner waves does not only compute a z-value for a given x, y and t value, it also transforms the x and y coordinate of the wave. Intuitively the Gerstner wave can be seen as a sine wave where discreet points on the top of the surface are pushed closer together, while points in the valleys are separated. A point P from the point x, y on the surface is calculated by this formula at the given time t:

$$\mathbf{P}(x,y,t) = \begin{bmatrix} x + \sum(Q_i\, A_i\, \vec{D}_i.x\, \cos(w_i\, \vec{D}_i \cdot (x,y) + \phi_i t)), \\ y + \sum(Q_i\, A_i\, \vec{D}_i.y\, \cos(w_i\, \vec{D}_i \cdot (x,y) + \phi_i t)), \\ \sum(A_i\, \sin(w_i\, \vec{D} \cdot (x,y) + \phi_i t)) \end{bmatrix} \tag{11}$$

Here a point **P** is returned, this is the new placement of the vertex, and in contrast to the normal sine waves, the x and y are also moved. The Q_i parameter is the sharpness of the wave, which decides how much the vertices should be moved toward the crest of the wave. The parameter should be maximum $Q/(wA \cdot \text{numWaves})$, if it was bigger, the points would have been moved past the mid point in the wave. The $\vec{D}$ parameter is the direction of the wave, and the w is the frequency of the wave. A is the amplitude of the wave, and the ϕ is the speed of the wave. The last parameter t is the time.
The normal of the point **P** is calculated in the same way as it was calculated for the normal sine waves, by taking the cross product of the binormal and the tangent for that point. The binormal $\vec{B}$ and the tangent $\vec{T}$ are partial derivatives in x and y direction.

$$\vec{B}(x,y,t) = \begin{bmatrix} 1 - \sum(Q_i\,\vec{D}_i.x^2\,w_i\,A_i\,\sin(w_i\,\vec{D}_i\cdot\mathbf{P}(\mathbf{x},\mathbf{y},\mathbf{t}) + \phi_\mathbf{i}\mathbf{t})), \\ -\sum(Q_i\,\vec{D}_i.x\,\vec{D}_i.y\,w_i\,A_i\,\sin(w_i\,\vec{D}_i\cdot\mathbf{P}(\mathbf{x},\mathbf{y},\mathbf{t}) + \phi_\mathbf{i}\mathbf{t})), \\ -\sum(\vec{D}_i.x\,w_i\,A_i\,\cos(w_i\,\vec{D}_i\cdot\mathbf{P}(\mathbf{x},\mathbf{y},\mathbf{t}) + \phi_\mathbf{i}\mathbf{t})) \end{bmatrix} \quad (12)$$

$$\vec{T}(x,y,t) = \begin{bmatrix} 1 - \sum(Q_i\,\vec{D}_i.x^2\,w_i\,A_i\,\sin(w_i\,\vec{D}_i\cdot\mathbf{P}(\mathbf{x},\mathbf{y},\mathbf{t}) + \phi_\mathbf{i}\mathbf{t})), \\ -\sum(Q_i\,\vec{D}_i.x\,\vec{D}_i.y\,w_i\,A_i\,\sin(w_i\,\vec{D}_i\cdot\mathbf{P}(\mathbf{x},\mathbf{y},\mathbf{t}) + \phi_\mathbf{i}\mathbf{t})), \\ -\sum(\vec{D}_i.x\,w_i\,A_i\,\cos(w_i\,\vec{D}_i\cdot\mathbf{P}(\mathbf{x},\mathbf{y},\mathbf{t}) + \phi_\mathbf{i}\mathbf{t})) \end{bmatrix} \quad (13)$$

$$\vec{N}(x,y,t) = \begin{bmatrix} -\sum(Q_i\,\vec{D}_i.x\,\vec{D}_i.y\,w_i\,A_i\,\sin(w_i\,\vec{D}_i\cdot\mathbf{P}(\mathbf{x},\mathbf{y},\mathbf{t}) + \phi_\mathbf{i}\mathbf{t})), \\ 1 - \sum(Q_i\,\vec{D}_i.y^2\,w_i\,A_i\,\sin(w_i\,\vec{D}_i\cdot\mathbf{P}(\mathbf{x},\mathbf{y},\mathbf{t}) + \phi_\mathbf{i}\mathbf{t})), \\ -\sum(\vec{D}_i.y\,w_i\,A_i\,\cos(w_i\,\vec{D}_i\cdot\mathbf{P}(\mathbf{x},\mathbf{y},\mathbf{t}) + \phi_\mathbf{i}\mathbf{t})) \end{bmatrix} \quad (14)$$

2.3 Lighting

To give the scene more realism we want to apply a better illumination model than the Gouraud illumination model that is used in OpenGL. We want to use the Phong illumination model, which is explained in the next section.

2.3.1 Phong illumination model

The Phong model calculates two separate contributing factors from each light-source, and adds contribution from a global light as well. The global factor is the ambient color and the other two are the diffuse and the specular color.
The general equation for Phong shading is[7]:

$$I = I_{ambient} + \sum_{lights} I_{diffuse} + I_{specular} \quad (15)$$

2.3.2 Ambient color

The ambient light is used to simulate indirect lighting. It is not affected by the position of the light sources, nor the viewer. Ambient light makes everything brighter, without adding shading. We made the ambient color the same as the diffuse. Changing the intensity of the ambient light gives all objects more diffuse color, even where there is no lighting.

$$I_{ambient} = k_a g_a \quad (16)$$

The ambient light intensity ($I_{ambient}$) at a surface point, is simply the product of the global ambient intensity (g_a), and the ambient coefficient (k_a) for that object's material.

2.3.3 Diffuse color

The diffuse color is the actual color of the object. The diffuse color will be visible everywhere on a surface that is lit by a light source. The intensity of the light will be highest when the angle between the surface normal and the direction of the light is zero and fade to black as the angle approaches 90 degrees.

$$I_{l,diffuse} = \begin{cases} k_d I_l(\mathbf{N} \cdot \mathbf{L}), & \text{if} \quad \mathbf{N} \cdot \mathbf{L} > 0 \\ 0, & \text{if} \quad \mathbf{N} \cdot \mathbf{L} \leq 0 \end{cases} \tag{17}$$

Here $I_{l,diffuse}$ is the diffuse color as RGB values. $\mathbf{N}$ is the normal vector at the point on the surface, while $\mathbf{L}$ is a vector pointing to the light from the surface.

2.3.4 Specular color

Specularity gives highlights to the surface of shiny objects. The specular model we use, consists of a set of RGB color values, and the specular reflection exponent. The RGB values determine which colors should be reflected, and the intensity of the reflection. The specular reflection exponent determines the size of the highlight.

$$I_{l,specular} = \begin{cases} k_s I_l(\mathbf{V} \cdot \mathbf{R})^{n_s}, & \text{if} \quad \mathbf{V} \cdot \mathbf{R} > 0 \quad \text{and} \quad \mathbf{N} \cdot \mathbf{L} > 0 \\ 0, & \text{if} \quad \mathbf{V} \cdot \mathbf{R} < 0 \quad \text{or} \quad \mathbf{N} \cdot \mathbf{L} \leq 0 \end{cases} \tag{18}$$

Here $I_{l,specular}$ is the reflected specular color and intensity, in our case in the form of a set of RGB values. $\mathbf{V}$ is the viewing vector, a unit vector pointing towards the camera or another point we want to calculate the reflected light at. $\mathbf{R}$ is the direction of the light, after it is reflected on the surface. The light-vector $\mathbf{L}$ is a unit vector pointing at the light source, while $\mathbf{N}$ is the normal vector at the point on the surface.

3 Analysis

We want to make a flexible system that can dynamically create different waves. The simplest way to implement this is to use a set of predefined waves that are added together, while the parameters in the equation for each wave can be changed dynamically. The waves are then blended together by changing the amplitude multipliers for each wave.

We also want to use some more interesting waves than just standard sinusoids. Waves that bends sharper on the top than in the bottom. A way to do this is to take the power of the sines (Fig. 4). We should however only use the exponents that are dividable by 2. This is because odd exponents gives points close to saddle points that would be considered unnatural for water.
By using a big exponent, we get a big distance between waves where the function is almost zero. To fill out the empty gaps between the tops, we can add

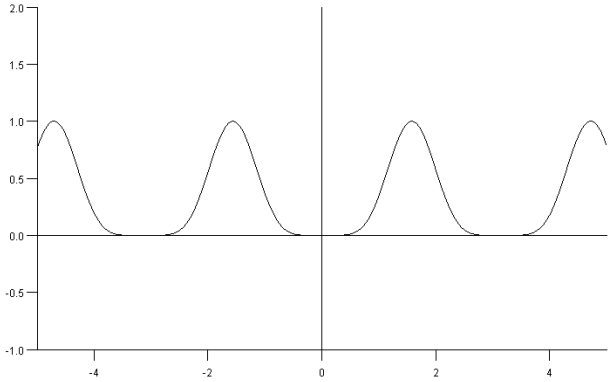


Figure 4: A power wave created by taking the exponent of a sine wave.

a power wave that is a sinusoid to the power of two. This wave will also have only positive values, but no gaps where the function is zero. Another advantage with this design is that we don't have to move the vertices sideways, like we do with Gerstner waves, which makes the power waves easier to work with. The result can be seen in figure 5, where the top pictures shows 2 different power waves, and the bottom one shows the two power waves added together.

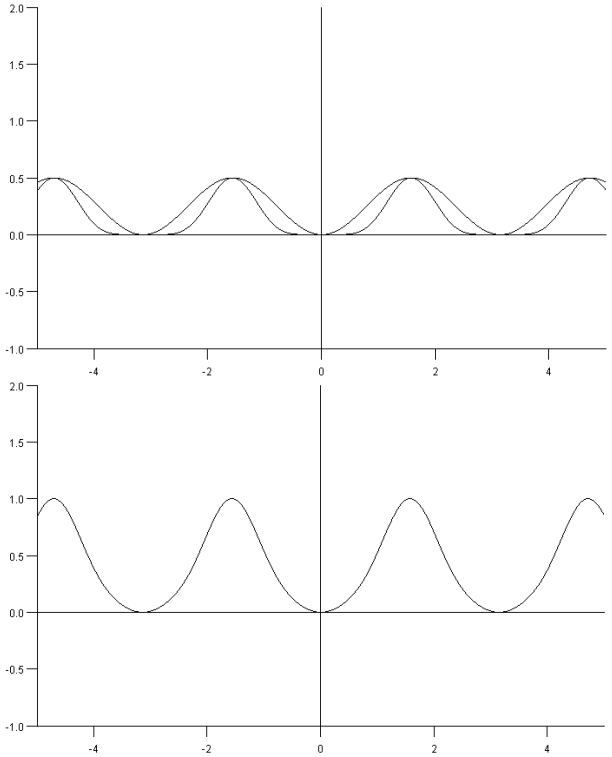


Figure 5: Top: $f(x) = \frac{1}{2}\sin(x)^6$ and $g(x) = \frac{1}{2}\sin(x)^2$. Bottom: $f(x) = \frac{1}{2}\sin(x)^6 + \frac{1}{2}\sin(x)^2$

The power waves with the gaps between the tops can also be used to our advantage, without smoothing them. The positive part that is non-zero can be viewed as a small wave component. We can use many small waves like that as building blocks to build up more complex waves. Using this technique, we can build up very interesting waves quite easily (Fig. 6). The wave in that figure is built from 5 different power waves. Figure 4 to figure 6 are all made with an online function plotter [8].

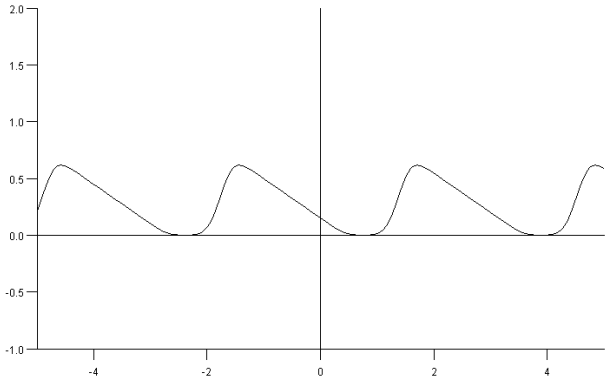


Figure 6: $f(x) = \frac{1}{10}\sum_{i=0}^{4}(5-i)\sin(x-0.4i)^{20}$

3.1 Initial specification

The first revision of the system design consists of 10 predefined waves. Two sine waves, and two cosine waves in each of the x and y direction. The last two waves are ripples, or ring waves. All the sine waves and cosine waves are supposed to be in power of a given number, with the default exponent set to 1.

3.2 Ripples

We define a ripple as a sine wave moving outward in all directions from a point xy. This can easily be described as a function of the radius, which is the distance from the centre of the ripple to any point on the surface. We define the radius using Pythagoras:

$$r = \sqrt{x^2 + y^2}, \tag{19}$$

where x and y are the components of the vector from the centre of the ripple, to a random point on the surface. By inserting equation 19 into the standard sine wave equation (1), we get the ripple function:

$$W(x, y, t) = A\sin(w\sqrt{x^2 + y^2} - t), \tag{20}$$

where A is the amplitude, w is the frequency and t is the time. To make sure the ripple does not stretch out indefinitely, we also need to add some attenuation to the function. The ripples will die out nicely in the distance if we multiply

the ripple with the function,

$$att = e^{-(\frac{x^2+y^2}{k})} \tag{21}$$

When creating ripples at run time, it is also nice if the ripples slowly dies out, and disappears. We can do this by adding a time dependant attenuation function. The function should start as one, and decrease towards zero with increasing rate. One such function is $att = \frac{1}{1+kt^2}$, here k is a constant that controls how fast the wave should die, and we use t^2 to make the wave change at a non linearly rate. If we add a time parameter to equation 21 as well, we can create waves that starts as a big sharp splash in the middle, and then evens out in amplitude over a bigger area. By combining our two attenuation function, we get nice ripples that spreads out and dies naturally:

$$W(x,y,t) = \frac{1}{1+\alpha t^2} e^{-(\frac{x^2+y^2}{\phi t})} A \sin(w\sqrt{x^2+y^2} - t) \tag{22}$$

The normal vector is calculated according to equation 9 as follows:

$$\frac{d}{dx}H(x,y,t) = \frac{Axe^{-(\frac{x^2+y^2}{\phi t})}}{1+\alpha t^2}\left(\frac{w\cos(w\sqrt{x^2+y^2}-t)}{\sqrt{x^2+y^2}} - \frac{2\sin(w\sqrt{x^2+y^2}-t)}{\phi t}\right)$$

$$\frac{d}{dy}H(x,y,t) = \frac{Aye^{-(\frac{x^2+y^2}{\phi t})}}{1+\alpha t^2}\left(\frac{w\cos(w\sqrt{x^2+y^2}-t)}{\sqrt{x^2+y^2}} - \frac{2\sin(w\sqrt{x^2+y^2}-t)}{\phi t}\right)$$

$$N(x,y,t) = [-\frac{d}{dx}H(x,y,t), -\frac{d}{dy}H(x,y,t), 1] \tag{23}$$

3.3 Texturing

There are several ways to use texture to make the waves look more realistic. One of them is to put a layer of foam texture on the top of the waves to make it look like they are breaking on the top. Our plan is to put on a gradient foam texture, where there is 100% foam on the top and decrease the value of the foam as the height of the wave decreases. This may not look totally realistic, but we want to test it out and see what it will look like since we haven't seen it before. This could be done by interpolating the diffuse color of the waves from the color on the water and the color of the foam texture. On the top it would then would take 100% of its color from the texture and nothing from the watercolour, in the middle of the gradient it would take 50% of its color from the texture and 50% from the watercolour and in the bottom it would have 100% from the original watercolour and nothing from the texture. Another way to use texture to make it look more realistic is to texture the bottom of the sea, and make the waves refract the light in the same way that real water does. This is probably not a subject we will touch since we have seen it in many other wave applications, and it also may take too much time to make.

3.4 Normal mapping

If the water looks totally smooth, as it is going to look if we just apply sinus waves of any form on the mesh, it would not look very realistic. We want to give the water a more realistic and imperfect surface by using normal mapping. This technique is a fast way to make surfaces look bumpy. This is done by using a texture where each RGB value is considered a coordinate in a vector, such that $x = 2R - 1$, $y = 2G - 1$ and $z = B$. The vector extracted from the normal mapping is the new surface normal when the normal map's coordinate system is rotated to fit the surface. If the original surface has a normal pointing in the z-direction, the coordinate system of the normal map matches the surface normal and the surface normal will simply have to be exchanged with the map normal. If the surface does not lie flat in the xy-plane (normal pointing in the z-direction), we will have to rotate the map normal according to the surfaces orientation. This process is illustrated in figure 7.
To rotate the map normal, we find the axis of rotation by taking the cross product of the original surface normal and the map normal. Then angle of which the map normal should be rotated is the angle between the original surface normal and the unit z-vector.

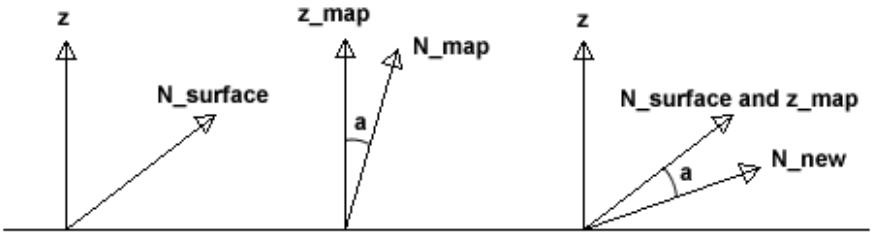


Figure 7: Left: Original surface normal. Center: Map normal. Right: New surface normal after rotating the map normal's coordinate system.

By doing this we will get changing diffuse and specular illumination on the surface, and it give the illusion of being bumpy. There are a couple of ways to make the normal mapping more interesting. The way to make the water most realistic would perhaps be to generate the normal maps procedural so that we could dynamically change the granularity of the normal map at run time, and we wouldn't have to use the same map on all waves. This however, would probably take too much processor time. What we would like to do, is to take the average of two normal maps, that are moving in different directions on the surface. This way the normal maps will produce interference and we will get a very nice moving surface.

Figure 8: An example of normal map used to create water. [9]

3.5 Phong illumination model

The way we plan to implement the Phong model is by using a fragment program that goes over every pixel of the geometries in the scene to check what color it should have, using the Phong illumination model. The theory behind the Phong illumination model is explained in the theory chapter.

To calculate the lighting at each point we need some information from the scene. This is the camera position, light position and properties, and these must be sent in as uniform parameters to the Cg program. This is done by creating in-parameters in the Cg method, and then bind those parameters in the OpenGL code by their name.

3.6 Interaction with the water

The first option of interaction is the possibility to move a simple object around in the water. This would create a trail of disturbance/waves in the water behind the moving object. This could be done by changing a small part of the mesh in OpenGL before it is sent to the Cg program. More specifically this could be done by moving a mask over the mesh where the movements are. The mask could consist of several predefined animations so that the best one is added to the mesh by adding the z-values. Another option is to describe the trail mathematically. This would be an extremely complicated function, so it would be a really good idea only to calculate the z-values for a small virtual patch or mask,

and then add it to the mesh.
A way to approximate this effect would be to have a trail of overlapping ripples following the mouse pointer. This could be implemented by generating a new ripple at the pointers position on the surface every single frame, or often enough to make a smooth trail. To avoid making huge accumulated ripples when the pointer is immobile, the new ripples amplitudes should be multiplied with the movement distance of the pointer since the last frame. To create a believable trail using this technique, we would need maybe 50 simultaneous ripples, depending on the frame rate. This would be far too much for Cg to handle, at least for the ARB vertex profile. Perhaps it could have run on a more powerful GPU with a newer Cg profile.

The second option of interaction is to drop objects in the water, and when it hit the water it creates ring waves outwards from that point. This could be pretty easily implemented since a ring wave like this does only need a start point and the necessary parameters to describe the wave. It would not be dependent on mouse movements, and therefore a lot less complicated than the animations we would have to generate to move objects in the water.

3.7 Revisions

Debugging revealed a problem with the power function implemented on the graphics card on the computer we are using. Since we are using an ATI card that is a few years old, we are not able to use the more advanced Cg profiles. We tried making our own simple power function, which resulted in an error since the Cg profile we are using does not support for-loops if the number of iterations is unknown. This could explain the erroneous results from the built in power function.

Because of these limitations, we decided to implement three different water simulation demos, illustrating different techniques. By dividing it like this we hope to show the different effects we had planned while avoiding the limitations of our Cg profile.

3.7.1 Sinusoid waves

We will make simple water simulations by combining up to 5 sinusoidal waves.

3.7.2 Gerstner waves

We will make water simulations by combining 2 Gerstner waves.

3.7.3 Water interaction/ Ripples

We will let the user interact with the water surface by clicking on it with the mouse. This action will cause ring waves, or ripples, to appear on the water

surface. We want to implement three simultaneous ripples, if the Cg profile can handle it.

4 Implementation

The implementation is done in two separate parts, the OpenGL and the Cg part. The interaction between these two is that OpenGL uses the vertex program and fragment program to change the scene after it is set up in OpenGL. This is explained in the theory part about Cg. The Cg programs work on several parts of the graphics that are made in OpenGL. The vertex programs we use modify all the vertices of the surface mesh, while the fragment/pixel program modifies the colour of every pixel on the surface of the surface mesh.

4.1 Limitations of Cg

Since we used OpenGL, and a couple years old ATI card, we had to use the simplest vertex and fragment profile available (CG_PROFILE_ARBVP1 and CG_PROFILE_ARBFP1). These are the first profiles that where made, and therefore have some limitations since the GPUs was not as strong back then as they are now. The limitations we "plunged into" was:

Maximum number of instructions:
In the ARB profiles, there can be a maximum of 128 assembler instructions when the vertex/fragment program is compiled. This was a big problem for us and limited the project very much. When we wrote our first vertex program, which was where we tried to add 10 different sinusoid waves, it just wouldn't work. We went through a considerable amount of troubleshooting to find the problem, but just could not find an error in the code. All the parts worked individually, but when we put them together it didn't work.

No variable number of loops:
The ARB profiles does not support a variable number of for or while loops. This is because all the loops are completely unrolled in the assembly code it is compiled into. This is because the assembly version used by this Cg profile does not support jump/goto operations. This problem could be solved by giving every loop a defined number of loops, and use as many of them as we need, but since every loop is unrolled at compilation, we do not benefit by using loops, since they use the same amount of instructions as sequential code.

4.2 OpenGL

For our OpenGL code, we rely heavily on Glut. We use OpenGL in our main programs that we use to test our Cg files. These are rather simple main programs, without any strong object oriented design. The only part of the program that is object oriented, is our surface mesh object.

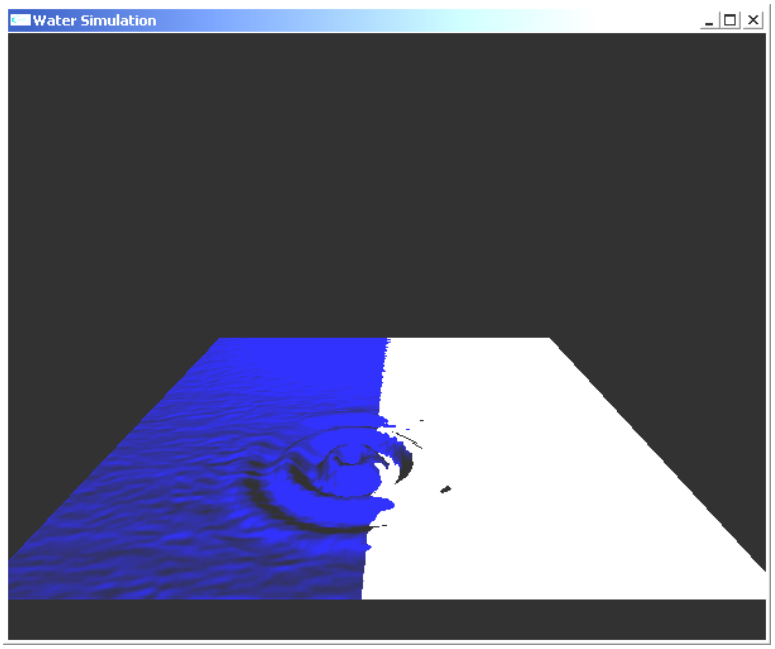


Figure 9: This is what happens when we push the number of instructions to the limit.

4.2.1 Main program

The WaterSimulation class takes care of all OpenGL initialization. It contains the main method that sets up viewport to draw all the graphics in. It also sets up the connection with Cg by making a fragment program and a vertex program, and then sets up a link with each uniform parameter in the Cg programs. The main method also runs the init method that instanciates the SurfaceMesh.

The main class also has methods to handle key presses, mousebutton clicks, texture loading and it has the display method that draws the surface mesh on the screen .

4.2.2 Surface mesh

The SurfaceMesh class is a general purpose mesh class that consists of a two dimensional vector of 3D point objects. Since the points have both x and y components in addition to z, it is possible to move the points sideways, based on their indices in the mesh. The alternative would be to only move them in the z direction, based on x and y position. Since all the points have their own three dimensional coordinates, it is also possible to make more complex surfaces, i.e. a sphere. This is however not recommended since the constructor generates a flat, squared surface, and all the points would have to be moved individually to create more complex objects.
When manipulating the vertices on the surface in Cg, it is not necessary to have functions for moving the points in the mesh. We still implemented functions

for moving points in the mesh in OpenGL. These functions simply respecify the points position so that the point gets a different position the next time it is drawn by OpenGL. Having this kind of general purpose mesh is still useful because it enables us to experiment with waves in OpenGL, without using Cg. This is a bit slower, but gives better possibilities for debugging.
The SurfaceMesh object has two draw functions that both uses OpenGL commands to draw the surface. The first one draws a solid surface using triangles. The other draw function creates a wire frame surface using lines. The default surface specified in the constructor has x and y values from 0 to 1, and a z value of 0. The size and position can then be changed by moving each of the points in the mesh (hard), or simply by calling glScale and glTranslate before evoking the draw function. The dimension of the mesh (number of vertices) is specified when calling the constructor.

4.2.3 Waves in OpenGL

We have created a pure OpenGL implementation of a ring wave, or ripple. This wave is based on a simple sine function. The vertices in the mesh are moved by assigning new z values to the points in the SurfaceMesh object, before the mesh is drawn. This approach worked well, except for two things: It was terribly slow, and we had big problems with the shading.

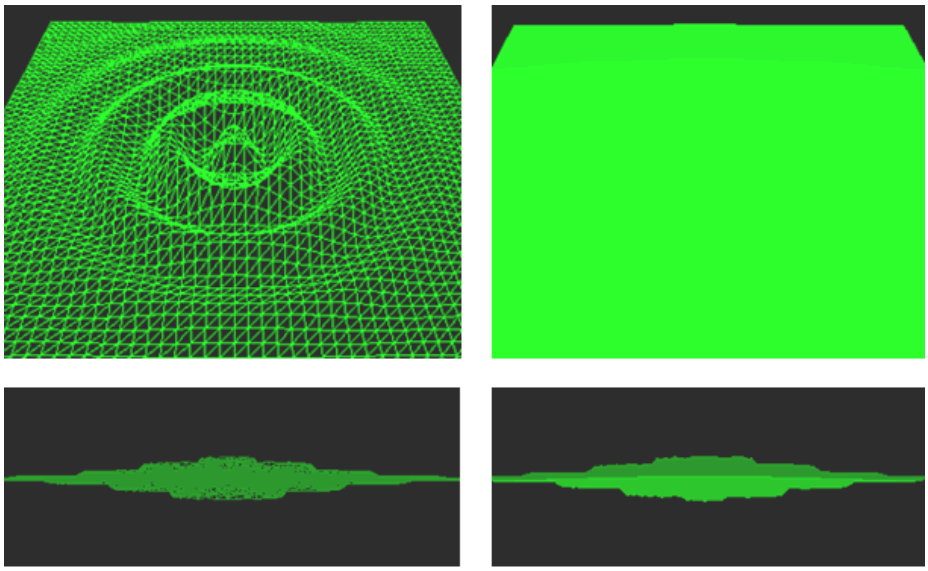


Figure 10: The figure illustrates the illumination problems in wireframe and solid rendering.

When we assigned a material to the surface, and added a light source, we got almost only ambient lighting. Since the ambient lighting is uniform, we were not able to see any movement on the mesh unless it was drawn in wire frame. One obvious explanation is that we don't calculate new normals when the surface is animated, and that the normals are always pointing in the z direction. This is

however not the whole problem, since we are not even getting specular highlights when there are no waves on the surface. The problem also seems to affect the shading of other objects in the scene. We spent a lot of time investigating this problem, without finding an answer. In the end, we just gave it up. We where supposed to do the project in Cg anyway, and there where no problems with the mesh itself when it was illuminated in the fragment program. We never tried to manually calculate the normals in OpenGL either. This is because we thought such a calculation would have made the OpenGL program so slow that we would not have been able to run any animations. At this point we where using the debug build in Visual Studio. However, testing towards the end of the project revealed that the release version is several times faster, which means calculating the normals would not be a problem. Still, we are not sure how much it would help, since the illumination problem seemed to go deeper than that.

4.3 Water interaction

Because of time pressure, we did not attempt our first option for water interaction, namely dragging objects through the water. The type of reaction such an action would cause on the water would be difficult to describe mathematically as one or just a few functions.

Since the first option was out of the question for us, we decided to implement the second option about creating ripples by dropping objects in the water. Unfortunately, we had to simplify the final implementation to just clicking on the surface instead of dropping objects. The reason for this is the same illumination problem we experienced in the pure OpenGL implementation of the ripple (mentioned in the previous section). Since the Cg program processes all vertices without caring about which objects they belong to, we decided to disable it while processing the falling object. If we had rendered falling objects the same way as the mesh, the Cg program would have created waves on the surface of objects. The problem with disabling Cg is that the falling object, in our case a sphere, is drawn by OpenGL alone, and thereby gets its illumination from OpenGL. As mentioned in the section above, the SurfaceMesh creates illumination problems for all the objects in the scene that get the illumination from OpenGL. In this case the problem was that the sphere became completely black, and invisible unless superimposed on the mesh. Because we had already used a lot of time investigating the illumination problems caused by the SurfaceMesh in OpenGL, we decided to just remove the falling objects. In retrospect we see that we could easily rendered the spheres in Cg without waves. This could be done by setting a flag in Cg that determines whether waves should be generated or not.

We now create ripples simply by moving predefined ripples to new locations on the surface and resetting the animation by setting the time parameter to zero. To get the interaction with the user, we register mouse click events using glut. When we have registered a mouse click, it is easy to extract the position of the mouse pointer in screen coordinates. We then use the function gluUnProject to get the intersection with the geometry in world coordinates. The new ripple position is then simply the x and y coordinates of the intersection

point. The actual ripples are crated in the vertex program describes in section "CgRippleWave".

4.4 Cg programs

We use 5 different Cg programs in this project. One vertex program that makes waves out of several sine waves, one vertex program that makes Gerstner waves, one vertex program that makes ripple waves / ring waves, and two fragment / pixel programs that takes care of the lighting and texturing.

4.4.1 CgSineWave

This program is the simplest of our 3 vertexprograms. It takes in parameters about each wave, and a timer parameter that counts from 0 to 2π to make the waves move. What the program does is to take every vertex and add a certain value to the $z - axis$. This value is computed by adding several sine waves, where each sine wave is calculated together from its parameters, such as the amplitude of the wave.

4.4.2 CgGertsnerWave

This program generates more complicated waves than the added sine waves. Therefore it is only possible to have two of them on the Vertex profile that is used. The vertices are moved as explained in the theory section about Gerstner waves.

4.4.3 CgRippleWave

This program contains three predefined ripples. The ripples are animated by individual "wave" parameters that acts as the time in the equations. The program also contains attenuation for the ripples, so that they spread out and eventually die out. The attenuation is also based on the wave parameters, and the speed of which it takes affect is dependent on two attenuation multipliers. The program takes in parameters for frequency, amplitude, position, time (wave) and attenuation multipliers for all three waves. The new z position is calculated using equations 20 and 22. The new surface normal is calculated using equation 23.

4.4.4 CgPhongNorm and CgPhongTexture

The fragment programs is the cg programs that decides which color should be assigned to each pixel. This is done by using the Phong illumination model on each pixel. To calculate the color of the pixels we need some information from the program, and these are sent from the OpenGL code as CgParameters. The parameters which are sent over are:

- GlobalAmbient and Ka: The ambient light and ambient color in the scene. This is light that lights up all objects event without any lightsources.

- Lightcolor and Lightposition: The color and the position of the light source. Our pixelshader supports only one lightsource as this isn't high priority.
- Eyeposition: The position of the camera.
- Kd: The diffuse color of the object.
- Ks and Shininess: Ks is the specular color of the object and the shininess is the specular exponent that determine how shiny the object is

The CgPhongNormal program has an extra parameter which is the normal map it should use, this is a sampler2d parameter, which is a lookup table that takes in the position of the mesh, and returns the normal this point should have. Then the normal that is gotten out of the normal map, is rotated as much as the original one is, and the original is swapped with the new normal from the normal map.

The CgPhongTexture program also has an extra parameter. This parameter is the foam texture that should be on the top of the waves. What the program does is to get the texture colour for the given point on the mesh though the sampler2d parameter of the texture. Then the colour of the texture is interpolated with the diffuse colour of the waves to make a gradient change of colour, from the colour of the water in the bottom of the waves, to the colour of the foam texture on the top of the waves.

5 Results

5.1 Sine waves

We made a program that generated 5 sine waves that could either move in the x axis or the y axis. This is the simplest form of waves, and do not represent the reality very much.

5.2 Gerstner waves

We made two different gerstner wave applications. One of them is normal mapped gerstner waves, as seen in Figure 12. This application runs the CGGerstner vertex program to do the displacement of the pixels, and the CGPhongNormal fragment program to take care of the normal mapping of the waves. The other application is textured gerstner waves. This program uses a foam texture on the top of the waves. The texture is gradually put on, meaning that on the top of the wave, the texture is very strong, and further down it fades away. This application runs the CGPhongTexture fragment program to take care of the texturing of the waves.

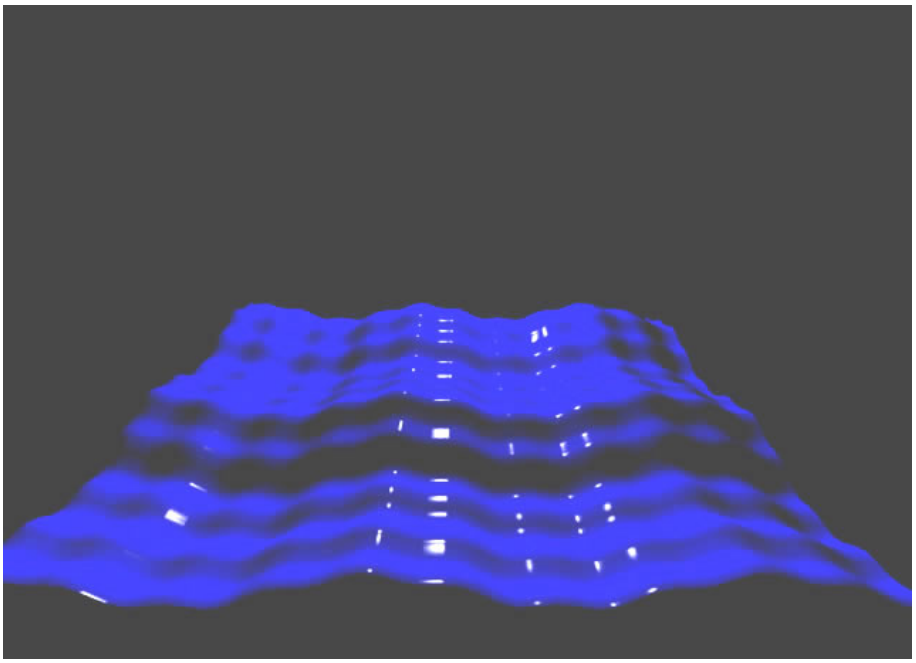


Figure 11: Four sinusoidal waves. Two in each of the x and y direction.

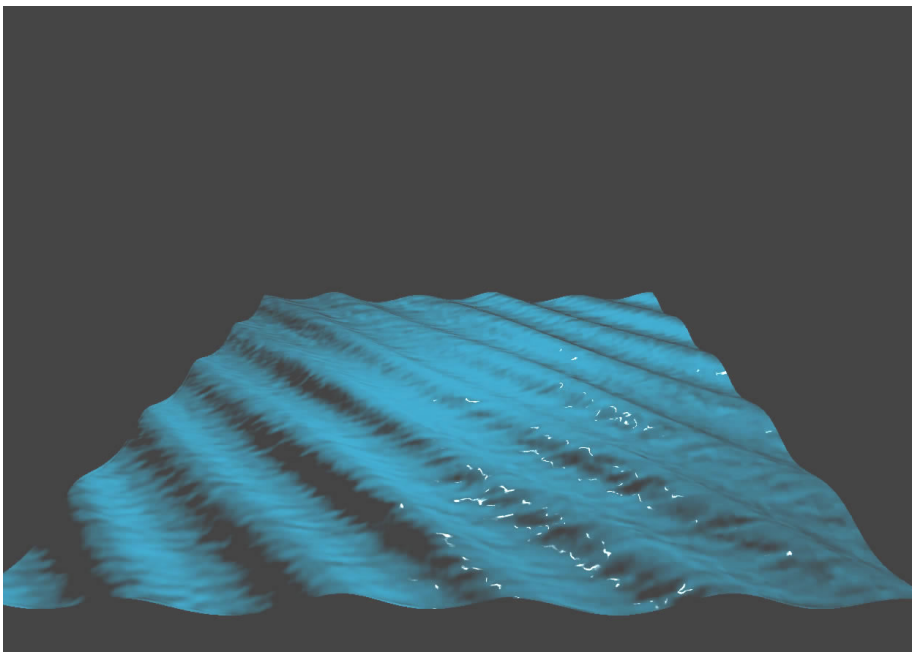


Figure 12: Two meeting Gerstner waves with normal mapping.

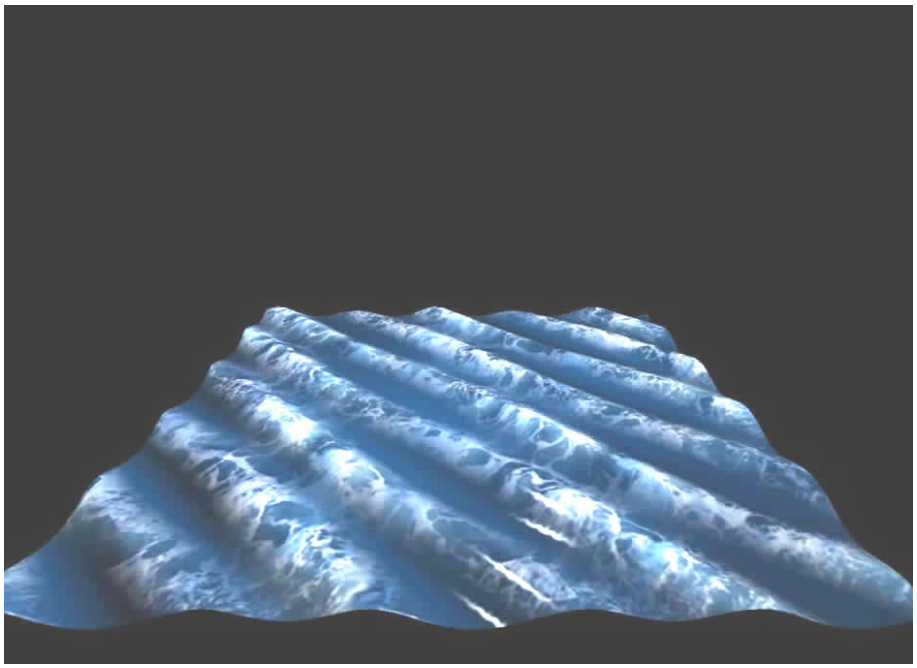


Figure 13: Two meeting Gerstner waves with texture mapping.

5.3 Ripples

The ripple application is the only one that has interaction with the user. This is done by clicking on the surface, which causes ripples to occur. There can be only three waves on the screen simultaneously, as this vertex profile cannot handle more at the same time. Figure 14 shows one ripple with normal mapping of the surface, figure 15 shows how three waves interfering with one another, and figure 16 shows how the attenuation of the waves works. This application runs the CGRipple vertex program to do the displacement of the vertices, and the CGPhongNormal fragment program to take care of the normal mapping of the waves

5.4 Displacement VS normal rotation

One test we wanted to do with our application was to check how realistic we could make the waves by just using lighting. This because it is a big difference in workload for the GPU, and much easier to implement just lighting compared to do both displacement and lighting. As seen from the two next figures, where figure 17 is with displacement, and figure 18 is without, there is a big difference when the waves are that big. On the other side, when the waves are much smaller, the phong shaded waves with transformed vertices looks no different than a smooth surface with normal mapping.

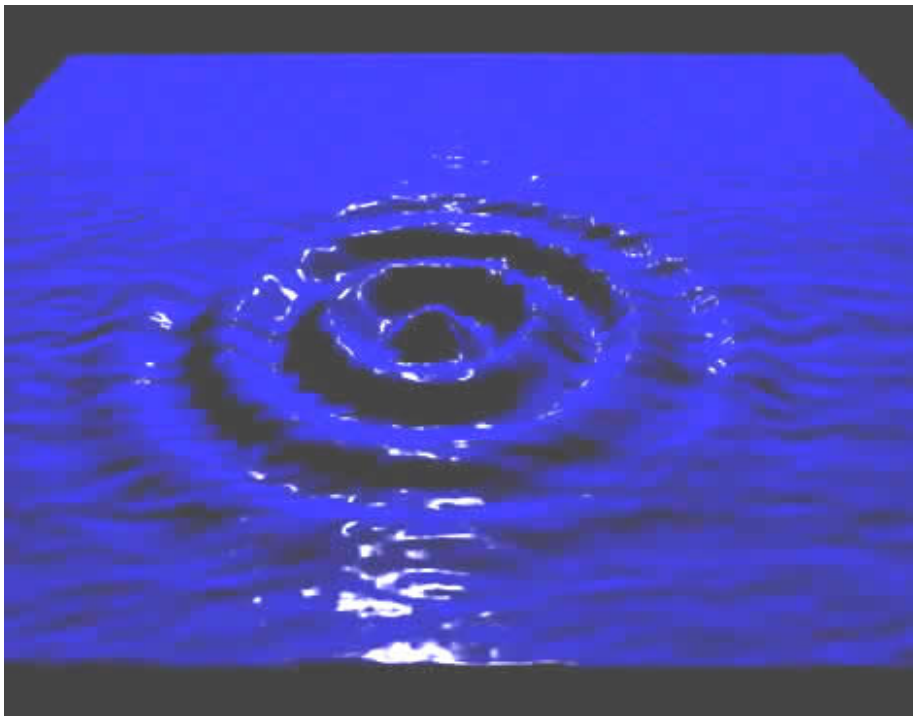


Figure 14: A Ripple created by clicking on the surface. The surface also has normal mapping.

5.5 Performance

This is a performance test to check how fast the graphics card can render a specific amount of polygons. A 10x10 mesh yields 100 vertices, a 100x100 yields 10000 vertices and so fourth. Gerstner NM means normal mapped gerstner waves, Gerstner T means textured Gerstner waves and Ripples NM means normal mapped ripples.

	Gerstner NM	Gerstner T	Ripples NM
10x10 mesh	120 fps	250 fps	120 fps
50x50 mesh	85 fps	200 fps	90 fps
100x100 mesh	68 fps	71 fps	70 fps
150x150 mesh	33 fps	53 fps	45 fps
200x200 mesh	25 fps	29 fps	24 fps
250x250 mesh	20 fps	20 fps	17 fps

From the benchmark we can see that all the applications manages to animate a 250x250 mesh (62500 polygons) in around 20 frames per second. Compared to our GL implementation, that could animate this mesh in around 15 fps, it is pretty impressive. This because the Cg implementation is much more complex than the GL implementation since it uses both normal mapping and the Phong illumination model instead of Gouraud shading. We also made a stripped down

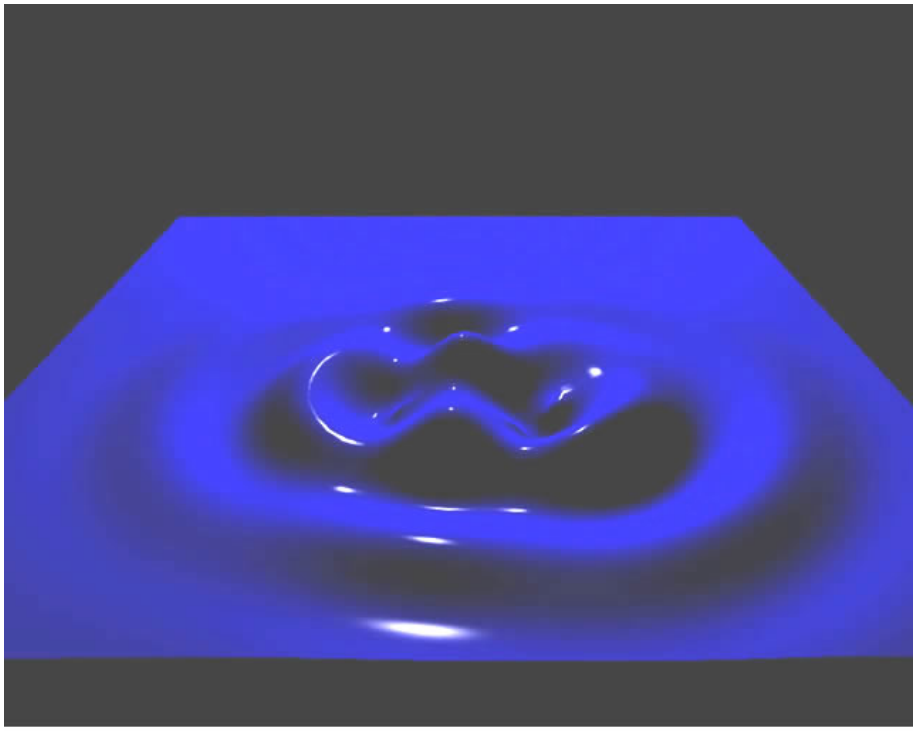


Figure 15: Interference between three ripples.

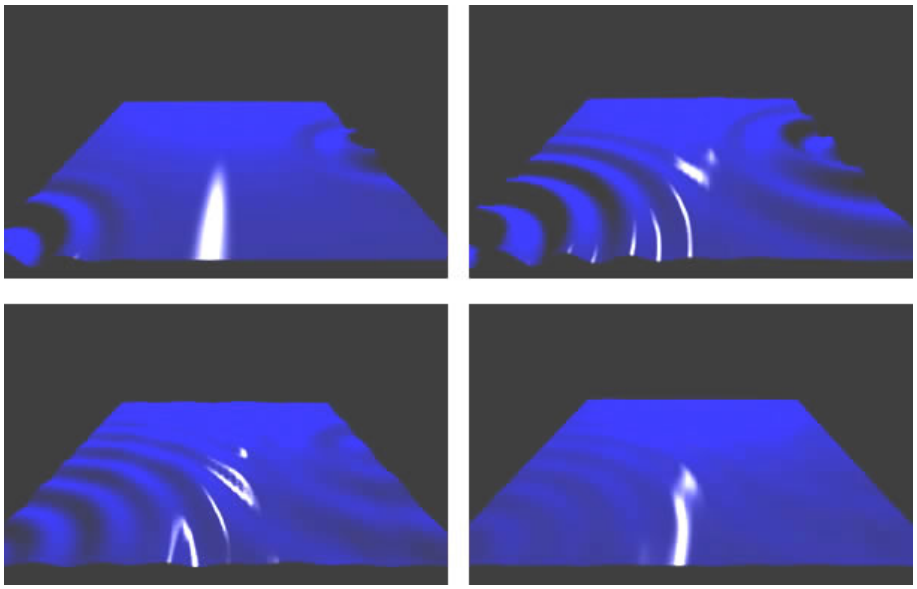


Figure 16: The images show how the attenuation function makes the waves disperse, and die out.

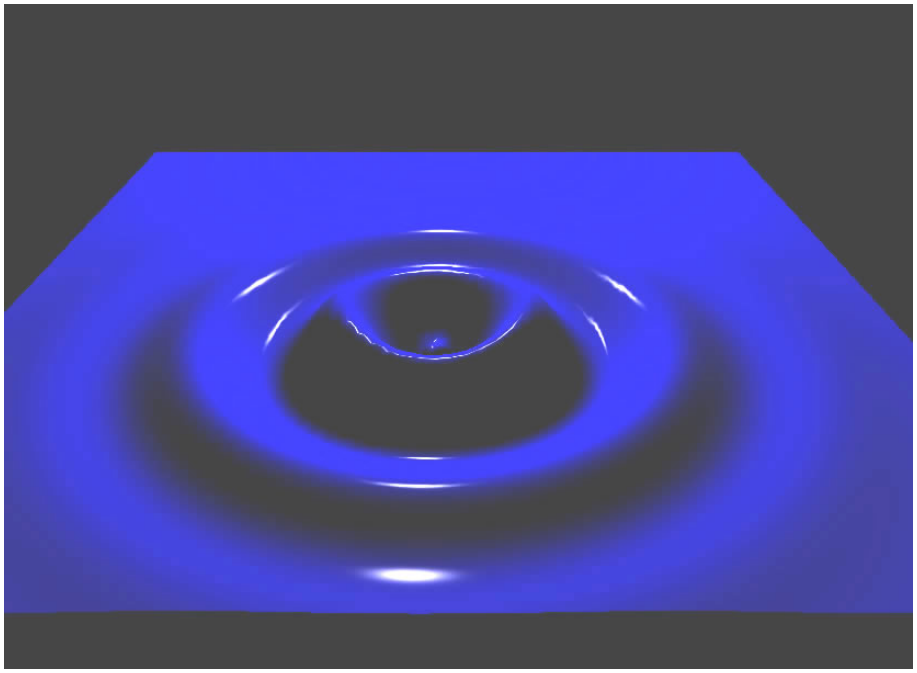


Figure 17: A ripple created by moving vertices and rotating normals.

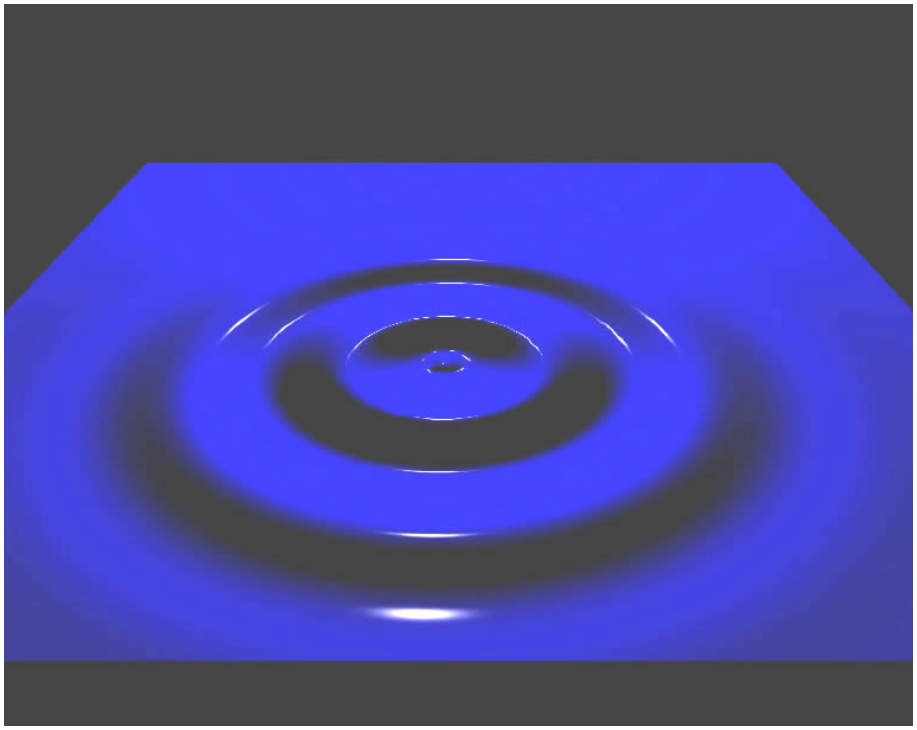


Figure 18: A ripple created by rotating normals, but without moving vertices.

Cg program that did the same as the GL program, and it ran at 55 frames per second, that is almost 4 times as fast as the GL implementation.

6 Conclusion

During this project, we have implemented a simple water simulation program in OpenGL. We use this program to create different water effects using different Cg shaders. Together this forms a package that can make directional and circular waves, normal and texture mapping, and lets the user interact with the surface. Looking beyond the main goals of this project, we did not implement all the features that we wanted. This is mainly because of strong limitations on the graphics hardware we were using, but also because we simply did not have the time.
Still, we are very happy with the features that we have implemented. We also feel that we have learnt a lot about general computer graphics and OpenGL, and especially about programming with Cg.

7 Future Work

7.1 Improved Graphics card

As mentioned in the conclusion we had problems with the maximum limit of operations on the used profile. Without this limit it could be possible to do a lot more improvement on our applications. We do not know if all these are possible to run, due to their complexity, but they should be fairly easy to implement.

7.1.1 More waves

The first point is as simple as to add more waves. Since we had the limit of instructions, we could not implement more. This should be easy to implement, since it would be reuse of code that is already written. By using a loop to go though all the waves it would not be many lines of code we had to add to the existing code.

7.1.2 Dragging an object though the water

It would have been really interesting to implement the possibility to drag an object though the water. This could be done as explained in the first paragraph in the *Interaction with the water* section in the analysis part of the report. Since this would need a lot of waves we need a newer graphics card to implement it.

7.1.3 Animating Normal Maps

One problem with the way that we do normal mapping is that it is very static. It would be a lot better if the small ripples in the water actually was moving.

This could be done by having a parameter that went from 0 to the length of the normal map, and added that parameter to the x axis on the normal map lookup in the fragment shader. This would make the small ripples in the water to roll over the surface in a periodic manner. This might look a little awkward, so therefore two normal maps moving in different directions would be preferable. This would make it look a little more random in the movement of the ripples.

7.2 Transparency and reflection

Transparency and reflections was two aspects we chose to not implement in our project. These features would have been nice to implement to make the water look more realistic.

7.3 Better Interactivity

If we had more time we also would have liked to add more interactivity with the waves. This could be to manage where the wind should come from and how strong it should be, and these parameters should determine how many waves there should be, the amplitude of the waves and the speed of the waves.

7.4 Approximation of procedural waves

Instead of calculation the height of every surface point real time, a lot of processing time can be saved by using a lookup table. This way, common amplitude values with matching coordinates and time variables can be calculated before running the program, and stored in arrays/matrixes. To make this integrate easily with Cg, we would store the precompiled values in an RGB image that can be sent to the fragment program as a texture.
If we assume that the waves travel along one known direction, we can map the wave in a 2D coordinate system. For a given distance from the origin of the wave, we must store the amplitude and two components of the normal vector (in 2D space). Using this, we can create a vector containing RGB sets for each distance along the wave. If make additional vectors for different timestamps, we can store animations using a whole 2D image to store the values. Figure 19 illustrates how the waves could be coded in colour.

An RGB image also has an alpha channel that could be used as a black and white texture, specular map, or some kind of culling filter. Conversion between the RGB values and the wave parameters are done accordingly:

$R = \frac{A+1}{2}$ $\quad A = 2R - 1$
$G = N_y$ $\quad N_y = G$
$B = \frac{N_x+1}{2}$ $\quad N_x = 2B - 1$

This approach is well suited to create ripples, since all you need is the distance from the centre of the ripple. Directional waves are created by applying a world to local transformation (rotation) on the coordinates before the lookup.

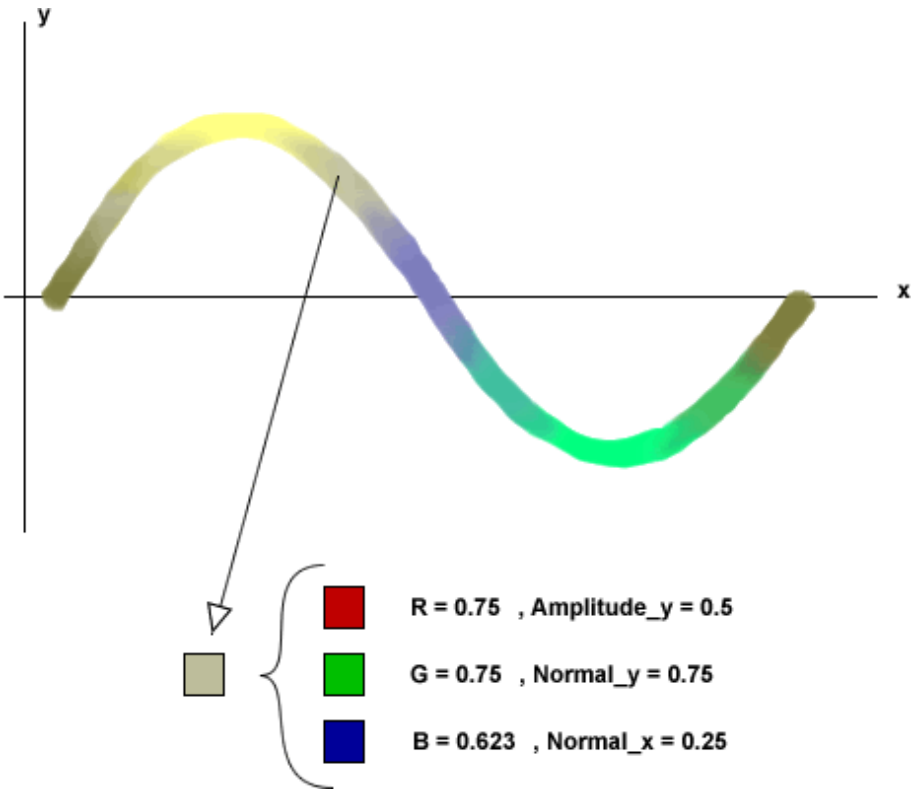


Figure 19: A 2D wave with colour values representing the amplitude and the x, and y components of the normal vector.

8 Refrences

References

[1] Nvidia. Cg homepage. http://developer.nvidia.com/page/cg_main.html.

[2] Neatware. HLSL introduction. http://www.neatware.com/lbstudio/web/hlsl.html, 2004.

[3] Jimmy Wales and independent authors. Wikipedia - the free encyclopedia. http://en.wikipedia.org, 2006.

[4] Mark Finch. Effective water simulation from physical models. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, 2004.

[5] Nvidia. Cg toolkit user's manual, 2004.

[6] Gonzales and Woods. *Digital Image Processing.* Prentice Hall, 2. edition, 2002.

[7] Hearn Baker. *Computer Graphics, with OpenGl (3rd edition).* Pearson Prentice Hall, 2004.

[8] David Austin. Function plotter, 1996. http://www.sunsite.ubc.ca/LivingMathematics/V001N01/UBCExamples/Plot/calc.html

[9] Håvard Homb Peder Johansen and Páll Ragnar Pállson. Java raytracer. 2005.

A Cg Code

A.1 CgSineWave

A.2 CgGerstnerWave

A.3 CgRippleWave

A.4 CgPhongNormal

A.5 CgPhongTexture