Foreword

This master thesis where written during the time-period from spring 2004 until summer 2005, under the teaching supervision of Professor Morten Dæhlen, University of Oslo, and Scientific Programmer Thomas Sevaldrud, Simula Research Laboratory.

The intent of the thesis is to examine how to best implement existing methods for water simulation into GeoGFX, a real-time terrain triangulation engine.

The diagrams presented in this thesis may resemble UML diagrams, but they are merely illustrations and do not follow the UML standard.

I want to thank my supervisors, Morten and Thomas, of being great help during the development of this thesis, Simula Research Laboratory of being so supportive with the necessary hardware and software, and of course my girlfriend Kristine of being so patient with me the last 1.5 years!

Fredrik Danielsen

Table of contents

1 Introduction

This thesis will examine a method for efficient rendering of realistic water surfaces in a 3D terrain visualization engine, using modern graphics hardware and state-of-the-arts rendering techniques. The method produces a water visualization model that will be implemented in a real-time terrain visualization engine named *GeoGFX* [12].

The demand for realism in real-time visualization terrain engines is increasing rapidly with the development of faster and better computers. To meet this demand modern computer games have added more realistic elements to their simulated worlds, providing a greatly enhanced user experience. One important detail is the appearance of water. In older games water was often treated as planar surfaces with an artist generated texture applied on them, ignoring important properties that give water its characteristic look. With the introduction of new and better hardware, developers are starting to pay more attention to these properties, and many games have impressive, realistic looking water effects. Having realistic water effects often tend to impress the user, and significantly raise the overall impression of the simulated world. However, even if hardware is improving rapidly and the introduction of programmable GPUs have made graphical processing a lot faster, the physics and optics of water is immensely complex and needs to be simplified in real-time visualization. A common way of simplifying water visualization is by using high-resolution height fields which provides a non-planar approximation of the water surface. This technique, combined with a "Level-of-Detail" (LOD) method for spatial scalability and a vertex disturbance algorithm for simulating waves, produces a realistic, but costly polygonal representation of the water surface.

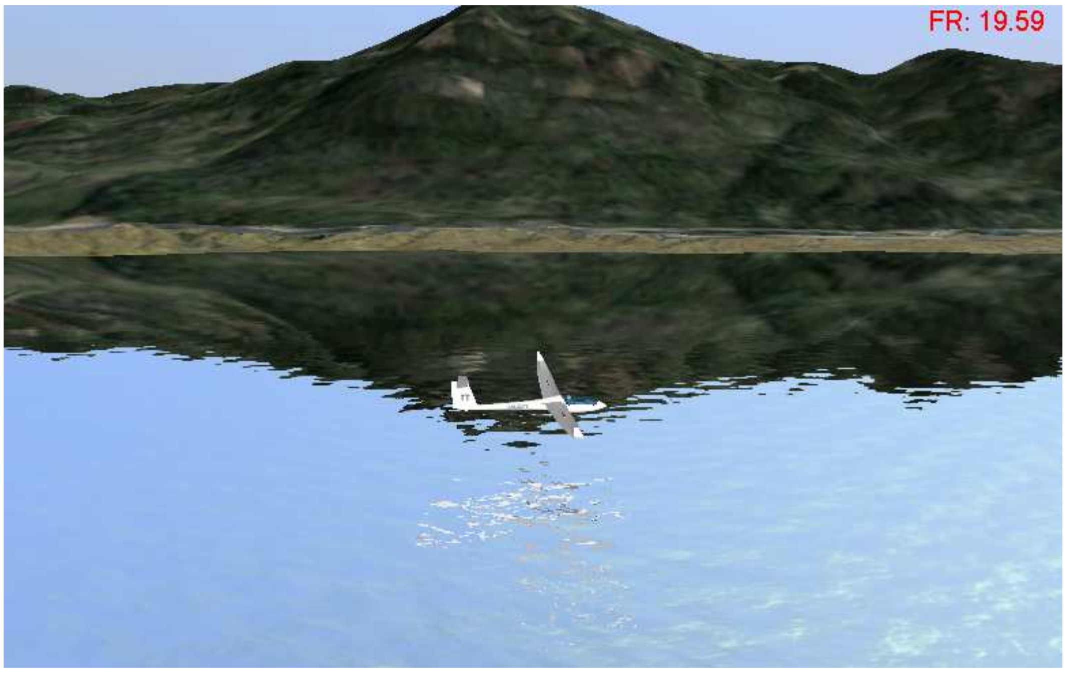


Figure 1.1 – Realistic 3D water surfaces.

Figure 1.1 displays a realistic water surface implemented into GeoGFX. The surface possesses the characteristic water features such as reflections and wave rippling, and greatly increases the realism of the scene.

In this thesis we will present an alternative water visualization technique called "texture Level-Of-Detail with bump mapping". The intent of this technique is to combine advanced, formerly developed, texture rendering methods in a hierarchical LOD arrangement to gain a result that simulates realistic water surfaces without the cost of detailed polygonal representations. "Bump mapping" contributes to this technique by using textures to simulate high-resolution height fields composed as a raster of plane normals. Disturbing the raster with an algorithm provides a wave-model without polygonal representation. This wave-model, combined with a Level-Of-Detail management system and other water features, is proposed as a method for scalable, dynamic, real-time rendering of a large number of realistic looking water surfaces in GeoGFX. *Figure 1.1* presents an outline of the components utilized in the *texture LOD with bump mapping* technique.

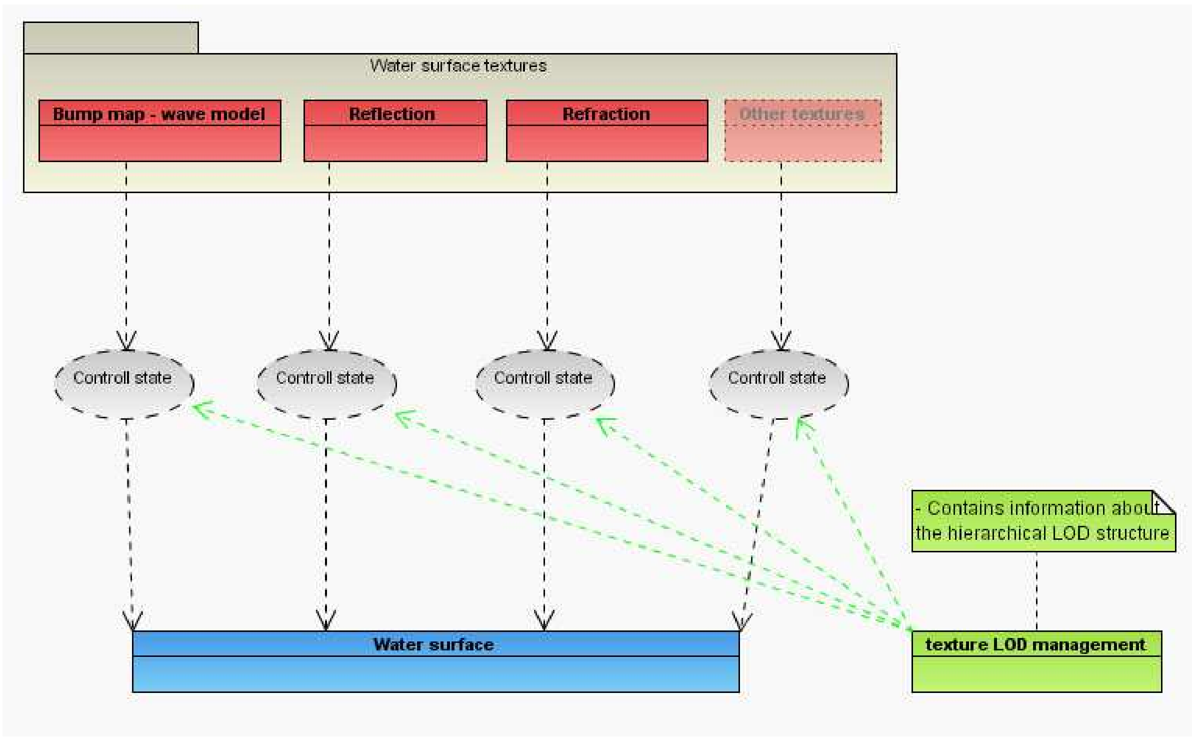


Figure 1.2 – texture LOD with bump mapping

The **texture LOD management** component invokes different states on **water surface textures** such as animation/no animation on the **wave model,** or reflection/no reflection on the **water surface**. These states are depending on parameters fed to the component by the application or the end-user. The **LOD management** component can be adjusted to behave differently depending on user defined parameters such as highest and lowest desired detail level and hardware defined parameters like supported OpenGL extensions, video graphic memory, etc. In *chapter 6* we will present different performance and visual results gained from adjusting parameters in this component.

Hopefully, by introducing texture based, low polygonal count water model to GeoGFX, we are able to maintain acceptable frame-rates in the real-time visualization while keeping a high detail level on the visualized scenes.

2 Terrain engine - GeoGFX

This chapter introduces the basic concepts of the GeoGFX terrain visualization engine. It explains the construction and functionality of the system, along with its intended use.

2.1 General

GeoGFX is object-oriented terrain-visualization tool built on top of a graphics engine named "GraphicsNGine". GraphicsNGine utilizes OpenGL as means for visualization of 3D objects. Both GeoGFX and GraphicsNGine make use of utility components for purposes such as vector operations, matrix transform, text handling, importing 3D models etc. *Figure 2.1* shows a simple diagram illustrating the collaboration of GeoGFX (Gg) components, GraphicsNGine (Gng) components and the utility components. These components are composed of several packages containing classes programmed in C++.

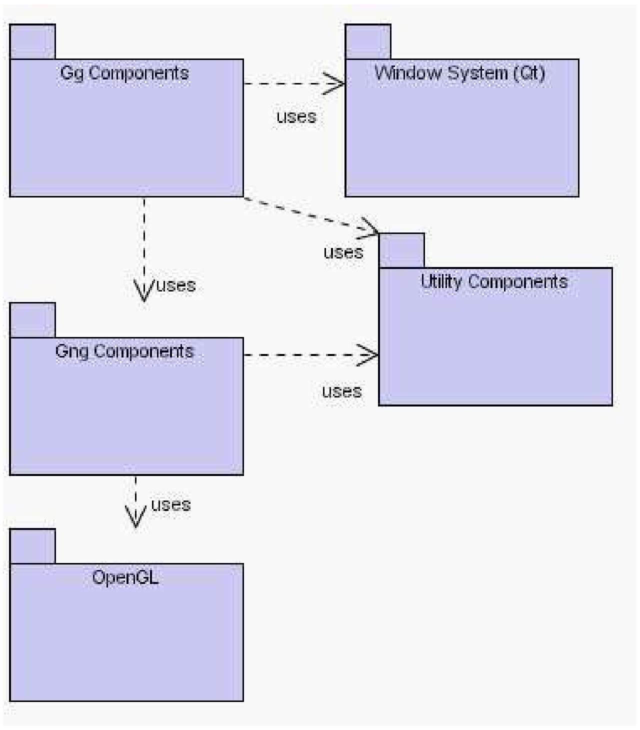


Figure 2.1 – GeoGFX component collaboration

Gg components employ, as *figure 2.1* illustrate, a window handling system for visualizing the terrain model, (any system capable of visualizing OpenGL).

2.2 Gg components

Gg components main tasks are to build 3D objects and deploy them as nodes in a scene graph (see *section 2.3.1*) using suitable classes from the Gng components. 3D objects references the triangulated terrain (described in *section 2.5*), the sky, light sources (sun) and other objects (planes, houses, etc.). The Gg components are also responsible for handling user input, drawing the scene graph on a window system, and updating scene graph transforms. *Figure 2.2* shows the packages forming the Gg components.

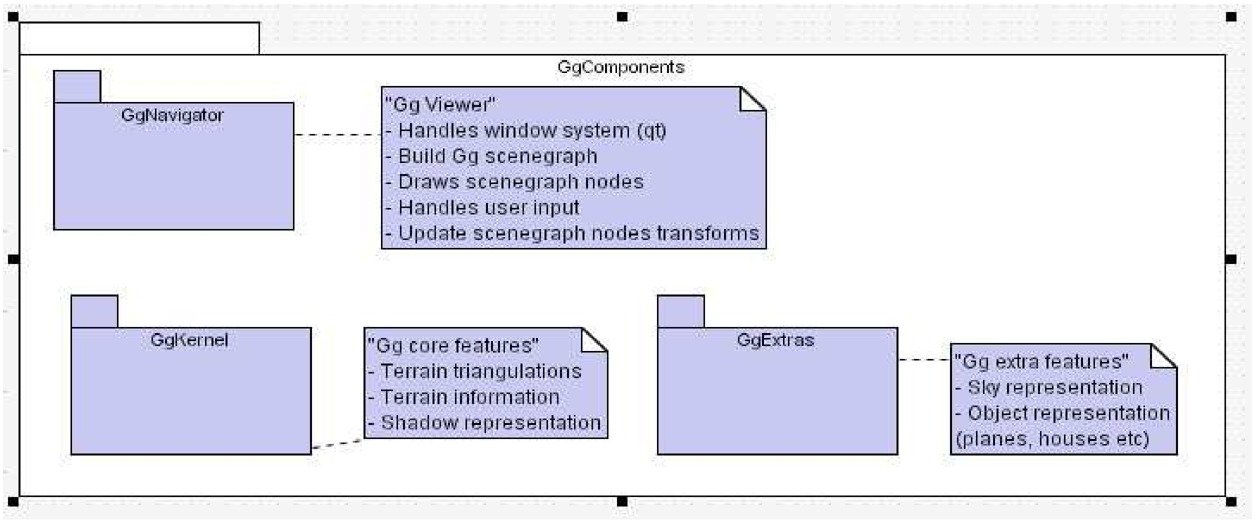


Figure 2.2 - The packages forming the Gg components

2.3 Gng components

The *Gng components* consist of four packages. Two of which are designed to handle basic OpenGL features such as extension support, states, different types of arrays (vertexes, indexes, normals, etc.), light and materials, blending, textures etc. Another package handles image loading and manipulation, and one package provides a hierarchic mean of visualizing and traversing 3D objects called a *scene graph*. These packages constitute a graphics engine capable of managing 3D objects with OpenGL. *Figure 2.3* shows the packages forming the Gng components.

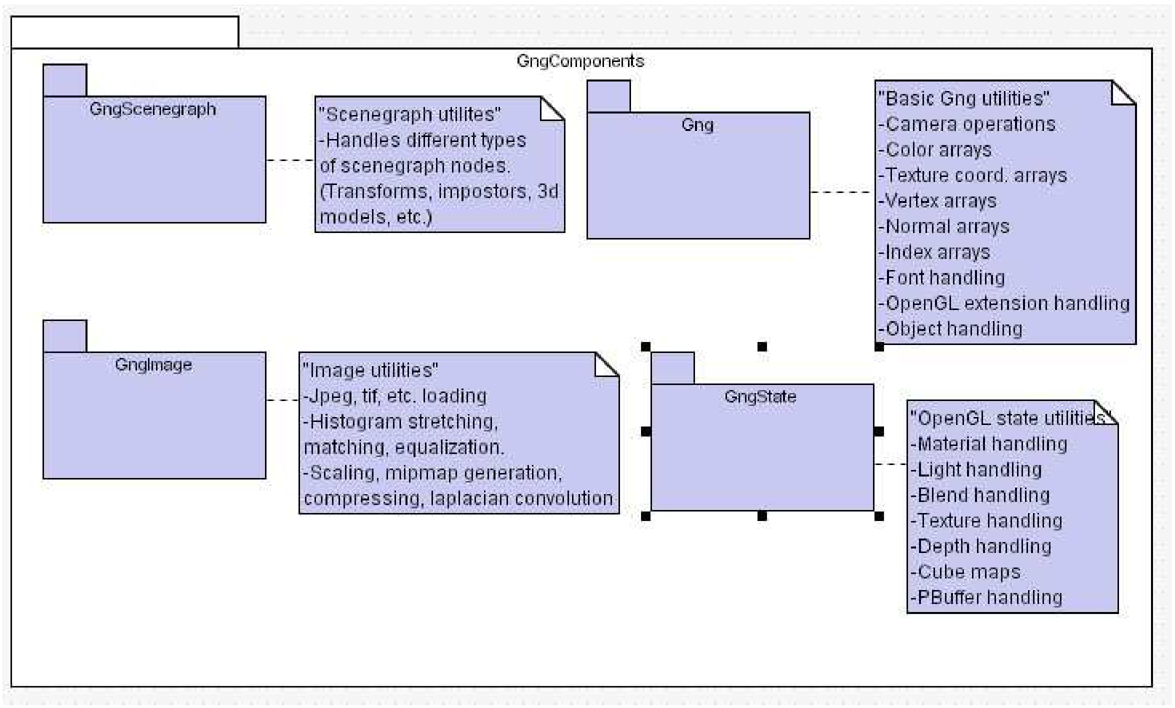


Figure 2.3 - The packages forming the Gng components

2.3.1 Scene graph

A *scene graph* is a hierarchical tree structure, consisting of nodes bound together in a parent/child relationship. The characteristic of a scene graph is that if a state or transform is applied to a node, this state or transform also pass for all its child nodes (unless a child node is applied some other state or transform). The tree is parsed top to bottom. *Figure 2.4* illustrates a basic scene graph where a transform has been applied to the top nodes right child.

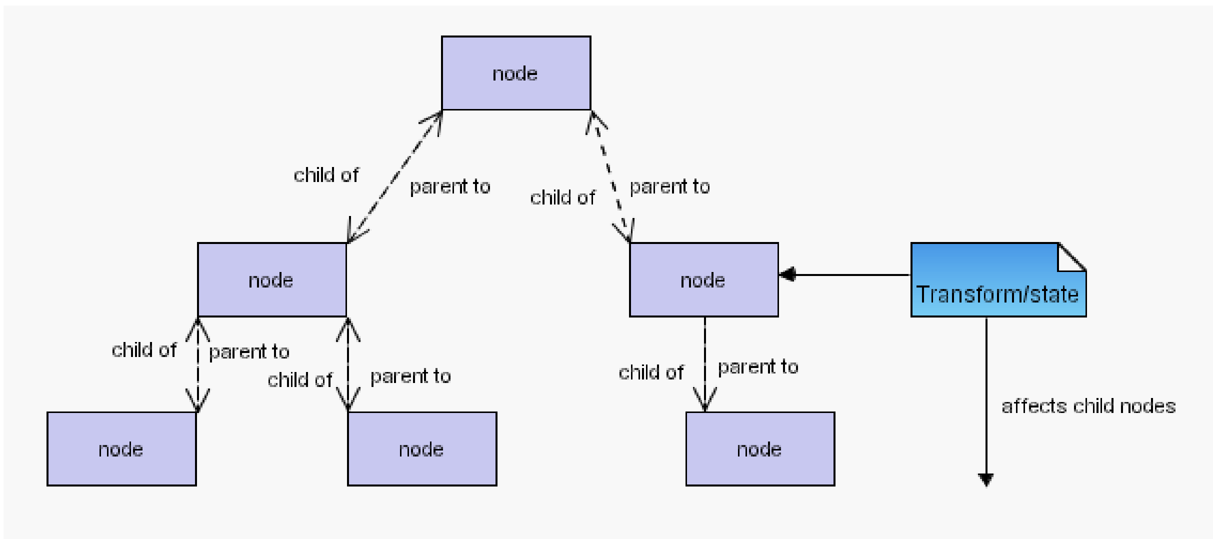


Figure 2.4 - A scene graph with a transform/state applied on the roots right child.

2.3.2 GeoGFX scene graph

The *GeoGFX scene graph* uses **GngTransform** classes as the nodes responsible for transformations. *Figure 2.5* illustrates a basic GeoGFX scene graph consisting of three **GngTransform** nodes; *surface transform*, *sky transform* and *geocentric transform*.
Transformations set on these nodes apply to all their child nodes. These children may either be other **GngTransform**'s or **GngRenderNode** objects. *Sun*, *Sky* and *Terrain* are such objects, which inherit from the **GngRenderNode** class. By rotating the *surface transform* all objects in the scene graph is rotated since *surface transform* is the top node, while rotating *geocentric transform* rotates only the *Terrain* object.

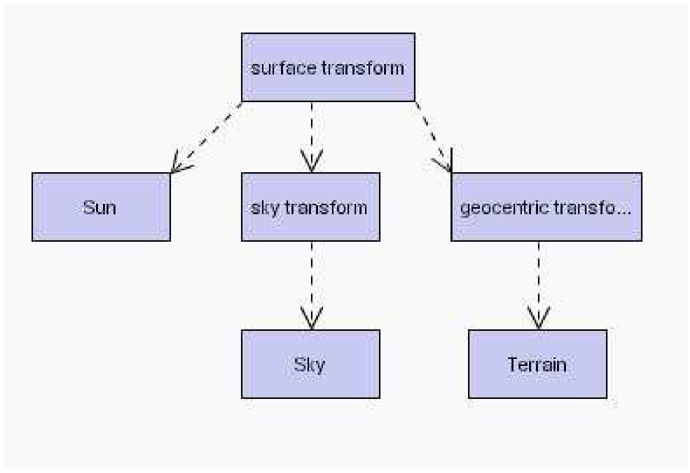


Figure 2.5 - Basic GeoGFX scene graph

Basic GngRenderNode objects
The three **GngRenderNode** objects *Sun, Sky* and *Terrain* are the basic nodes in GeoGFX scene graph.

- *Sun* is a **GngLightSource** object that is not actually rendered; it only contains basic information about the direction and color of the sun.

- *Sky* is a **GgSky** object that is responsible for visualizing the sky including the sun. **GgSky** needs to be told which longitude and latitude together with what time of day the visualization pass for. Based on that information it draws a sky dome relative to the view camera in such way that the dome covers the horizon with the view camera always inside. Only the sky colors are drawn, other sky features like clouds and weather behavior are left out. **GgSky** also hands information about the sun position to the *Sun* node.

- *Terrain* is a **GgTerrain** object that is responsible for visualizing the ground. This object manages the triangulation described in *section 2.5* together with scaling of textures such that they fit relative to the triangulation. **GgTerrain** also includes a function for checking the ground elevation on a given longitude/latitude, a function for creating ground shadows, a function for setting triangulation detail level, etc.

2.4 Utility components

The *utility components* provide the Gng components and the Gg components with tools to simplify and reuse certain operations commonly needed. This includes vector operations, matrix transforms, file reading, font reading, etc.

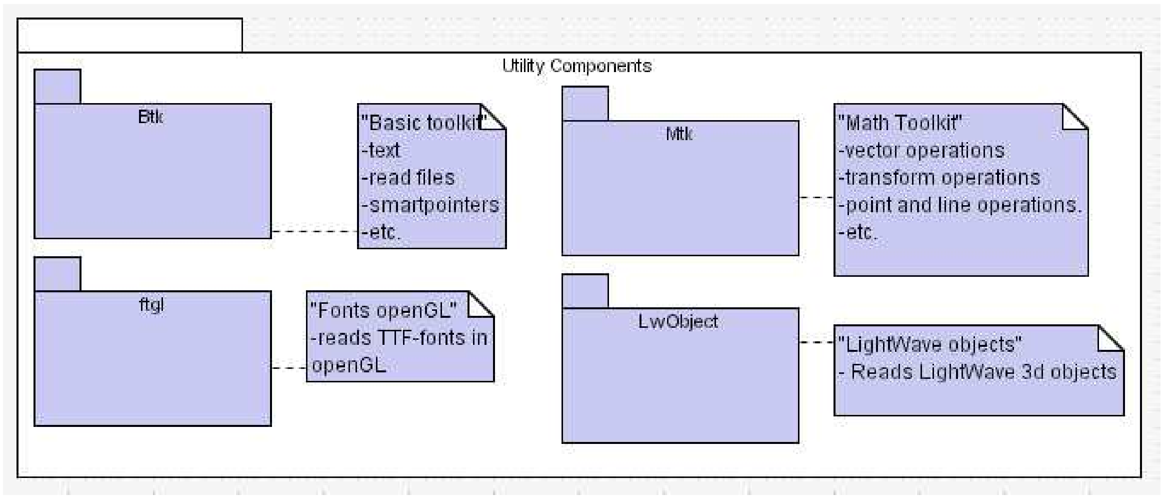


Figure 2.6 - The packages forming Utility components

2.5 The triangulation structure

The terrain triangulation structure in GeoGFX is based on a hierarchical (or “Level-Of-Detail”) representation of height-fields [12]. The purpose of this structure is to render as few triangles as possible, with as high accuracy as possible, for each frame in the fly-through sequence. Without going into too many details, the basis of the hierarchical structure is that the coarsest triangulation is the root node, and the further down the branches in the hierarchical tree we traverse, the more fine-grained the triangulations become. To be able to visualize large terrain areas we split the full triangulation into rectangular triangulated subsets (tiles) and assign a hierarchical structure to each subset. (The subsets are rectangular because

it makes it easier to add texture on the surface). The number of levels in the tree depends on the size of the entire triangulation area (larger areas → more levels).
In GeoGFX the coarsest triangulation levels are basic regular triangulations, while the fine grain triangulation levels are precompiled (constrained) Delaunay. These triangulations are also known as TIN's (Triangular Irregular Networks). The rendered level of each subset is determined by the camera's distance from the tile and the topography of the tile. Thus, neighboring tiles can be at different levels, which may lead to glitches and artifacts in the triangulation. This problem is dealt with by precompiled transition skirts (triangulations), connecting neighboring tiles at different levels to ensure that the entire triangulation is valid.

The tiles in the triangulation are also evaluated with a method called "frustum culling". Frustum culling ensures that only tiles inside the cameras frustum (view area) are rendered. This efficiently saves a lot of graphical computations since most of the terrain is normally outside the frustum.

2.6 The coordinate system

The GeoGFX engine is designed with the purpose to visualize huge real-world terrain models in 3D. The model data fed to the engine are samples of real terrain measures given in world coordinates (longitude, latitude and elevation). Translating the world coordinates into OpenGL coordinates yields huge numbers, which may cause numerical instability when projecting to screen. Hence, GeoGFX contains methods to convert the world coordinate system to a local coordinate system to ensure numerical stability.

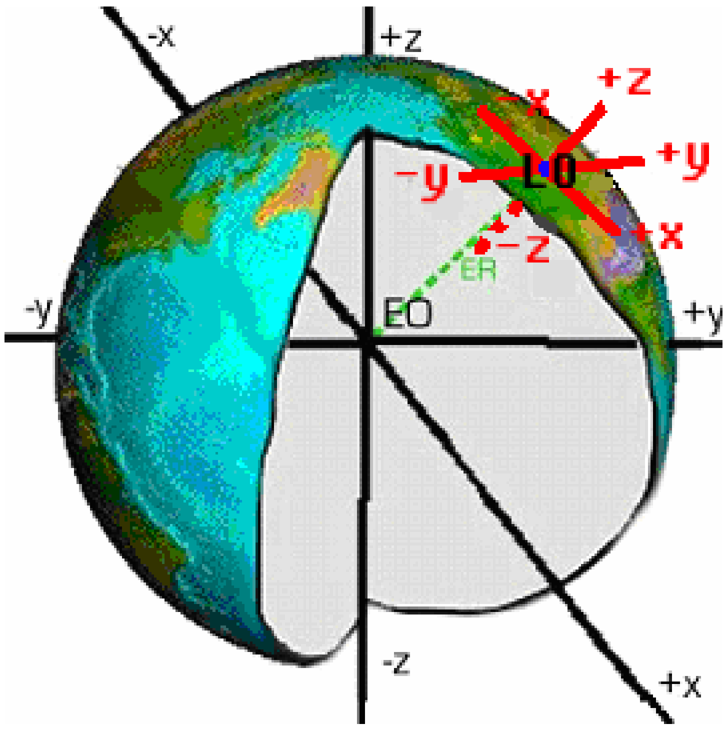


Figure 2.7 - World and local coordinate system.

Figure 2.7 illustrates the conversion from a world coordinate system to a local coordinate system. *EO* is the earth center and the origin of the world coordinate system. The z-axis intersects with the Poles and is positive with latitudes north of the equator and negative with latitudes south of the equator. The x-axis is positive with longitudes less than 90 degrees west and 90 degrees east and negative with longitudes greater than 90 degrees west and 90 degrees east. The x-axis intersects the earth surface on longitude 0 (and 180) when y=0. The y-axis intersects the surface at longitude 90 west and east when x=0, and is positive between longitudes 0 and 180 degrees west and negative between 0 and 180 degrees east. The local coordinate system has its origin, *LO*, measured as *EO* + the earth radius, *ER*. The axes in this coordinate system are rotated relative to the surface point of *LO* with the z-axis always pointing upwards, the x-axis always in the longitude direction and the y-axis in the latitude direction.

The fact that the earth is slightly elliptic causing the world radius to differ at the Poles is overlooked since it does not affect the visual presentation of the terrain model.

The **GgTools** class contains functions for translating coordinates between latitude/longitude/altitude and XYZ coordinates, and between world and local coordinate systems.

2.7 A GeoGFX application

GeoGFX is designed to be a foundation for basically any type of 3D application that uses a real world model as the base of its visualization. However, because of its ability to visualize huge terrain models it is especially capable for use in flight simulators. This has led to a computer game named "Silent Wings" [13], which is a very realistic sailplane simulator with an authentic physics and weather model.

Together with the basic GeoGFX nodes described in *section 2.3.2*, Silent Wings consists of different features (nodes) such as LightWave models of sailplanes and pilots, sky elements such as clouds, a weather model simulating wind and lifts, and a flight physics model simulating the behavior of the sailplanes.

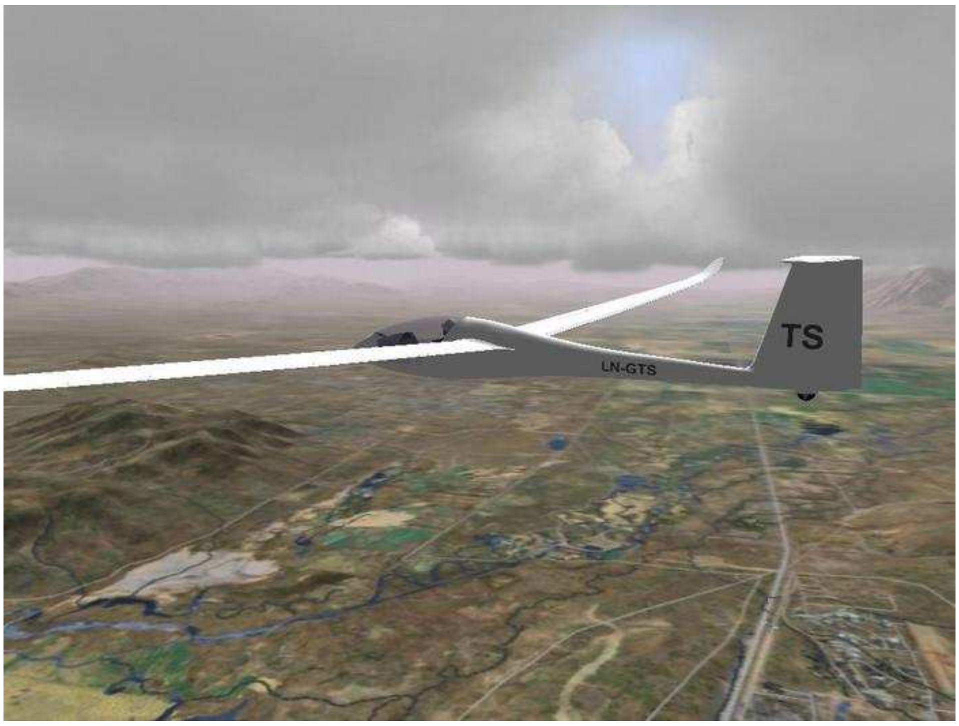


Figure 2.8 – Screenshot from Silent Wings

As stated in *chapter 1* the present version of GeoGFX does not have realistic visualization of water. Water surfaces are integrated as part of terrain triangulation and are visualized with bluish textures that are either computer generated or derived from satellite/aerial photos. The water surfaces possess none of the realistic physics of water described in *section 3.1* and the contribution of these surfaces actually lowers the realism of the visualization as *figure 2.9* illustrates.

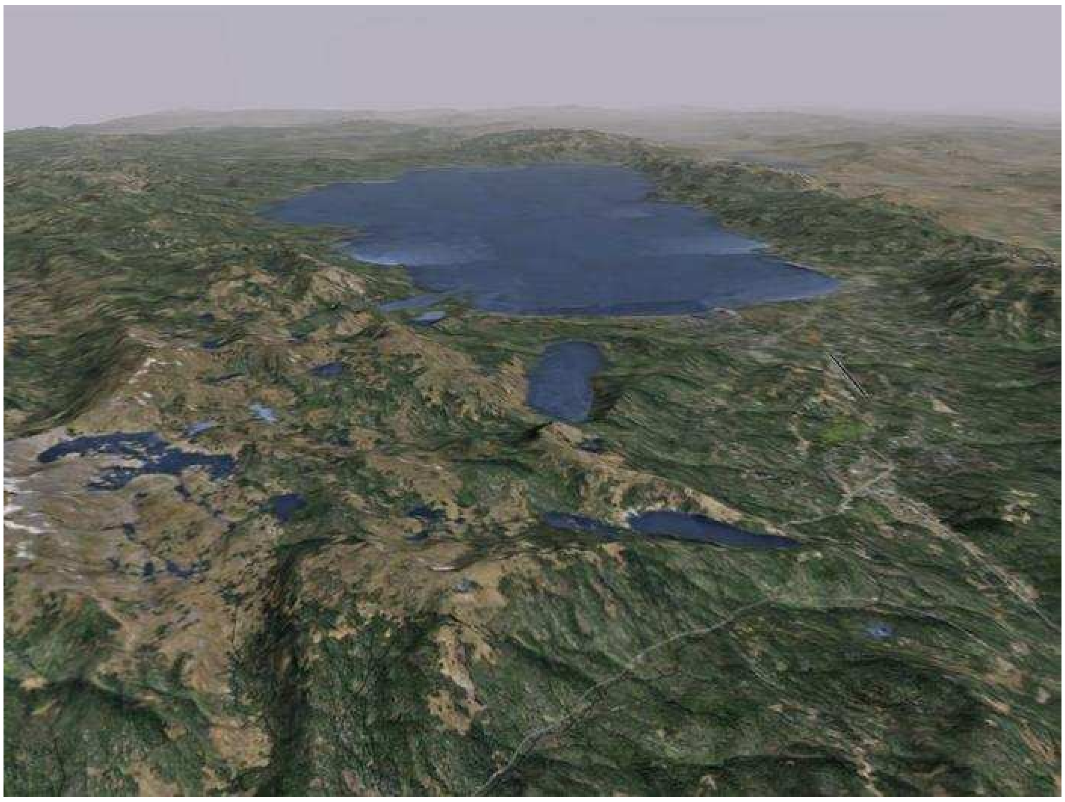


Figure 2.9 Water surface in GeoGFX – Lake Tahoe

3 Water visualization

This chapter describes the theory and tools we are using to create the visual characteristics of the water surfaces. *Section 3.1* and *3.2* introduces the basic theory for creating **water surface textures** described in *figure 1.1*, while *section 3.3* describes the tool used to apply these textures on the surface.

3.1 Theory of water

According to Premoze and Ashikhmin [1] "*creating and rendering realistic water is one of the most daunting tasks in computer graphics*". In order to accurately simulate the realistic physics of water we need large mathematical and numerical models. These models are based on several parameters such as sunlight and skylight illumination, wind speed and direction, light transport within the water body, etc. The appearance of water can vary significantly according to the compound of these factors. Because of the size and complexity of the mathematical and numerical models, the required time to calculate and visualize realistic water models is heavily dependent of software and computing power used in the rendering process. The process of rendering water is described in *figure 3.1.* Computing a fully realistic model may take an immensely long time, which is totally unacceptable in a real-time rendering environment such as GeoGFX. In order to obtain a water model that is applicable in a real-time environment we need to do some large simplifications of the mathematical and numerical model.

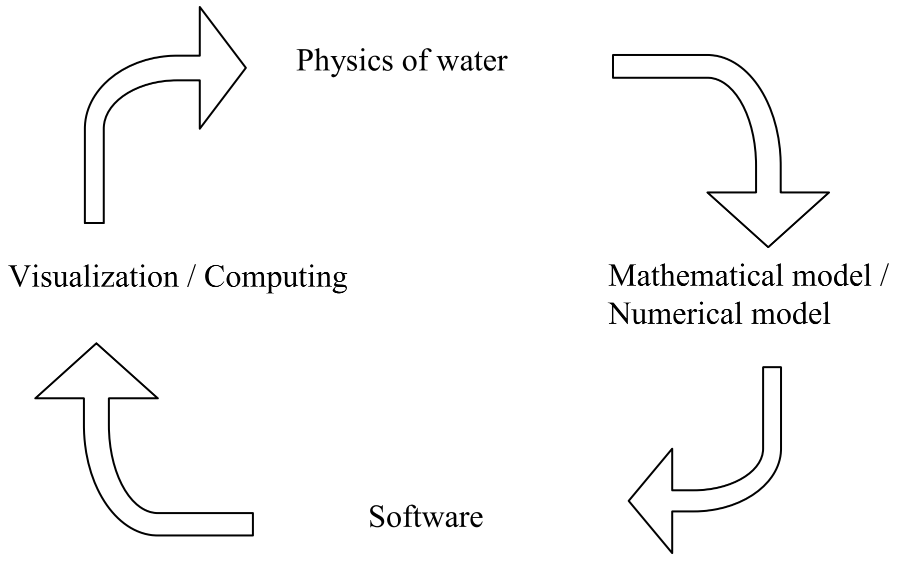

Physics of water
Mathematical model / Numerical model
Software
Visualization / Computing

Figure 3.1 – the process of rendering simulated water

To create realistic water models Premoze and Ashikhmin [1] addresses three main components: Atmospheric conditions**,** wave generation and light transport. In the next sections of this chapter we will present a simplification of these three components based on previous work on real-time visualization of water. This simplification will be used in *chapter 5* as the model for implementing the visual presentation of the water surfaces in GeoGFX.

3.1.1 Wave generation

The appearance and shape of waves on water surfaces are based on several factors. They emerge from the winds influence on the surface, by its direction and speed. However, they also emerge and are shaped from other factors such as their impact on each other, the influence of natural water flows and the characteristics of their surroundings, typically the shallowness of the water. Waves may often appear to be randomly scattered, but research has proven that they in fact are behaving according to a pattern and that the motion can be defined and described with a set of differential equations. A well-known mathematical model for simulating wave patterns is the Navier-Stokes Equations (NSE) [2]. The NSE is the cornerstone in the field of Fluid Mechanics and describes the motion of incompressible fluids. In their basic form the NSE are immensely complex and require a lot of computation, but they can be simplified for use in less computational water models. However, even though they are extremely realistic they do not suite our intended use. In [2] NSE is suggested implemented as a rectangular grid of columns representing the water body. For every column, a set of virtual pipes is used to describe the flow of fluids between itself and the adjacent columns as shown in *figure 3.2*.

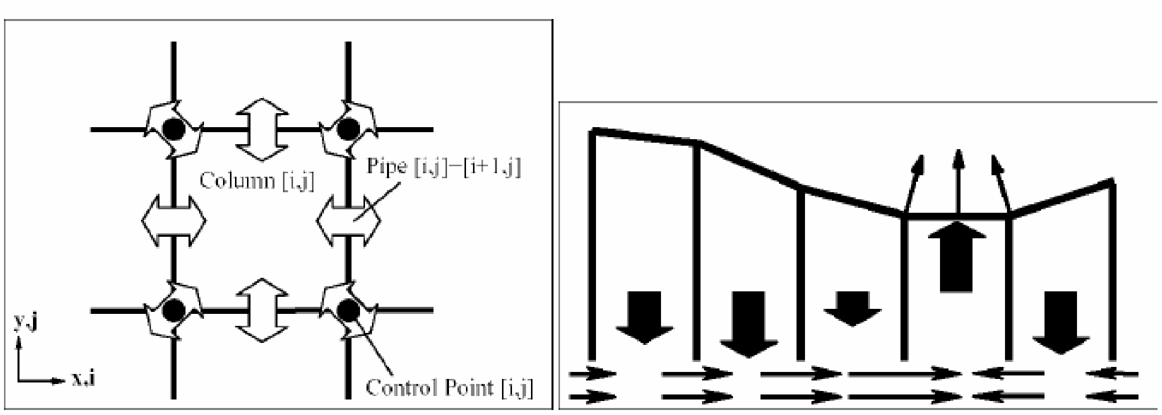


Figure 3.2 – Approximation of the NSE. The images are copied from [2]

The NSE also requires the previous state of the surface to be known. In our model we want to use a simulated rectangular grid, so called normal map textures (see *section 3.2*). Using NSE to create normal map textures would be a very complex and difficult task to handle, since the textures would have to be updated every frame. It would also require very large textures, covering the whole surface since it would be practically impossible to tile the textures on the surface. Using textures on moving surfaces also requires mipmapping to avoid aliasing (*section 3.2 – mipmapping*), which creates scalability problems when applying the NSE on the textures. As suggested in [2] NSE can be used in combination with other wave models, applying it only on limited areas of water surface. This way the NSE can handle water response to objects intersecting it.

Another approach for wave generation suggested in [2] is a statistical model rather than simulating the entire process of the waves being built up. Oceanographers have developed models that describe the wave spectrum in frequency domain depending on the weather conditions. By employing a Fourier transform we can use these spectrums to filter a block of 2D-noise. This method can be a computationally efficient way to generate a two-dimensional height-map, which can be transformed into a normal map texture using a normal map generation algorithm [3].

A common and simplified way to generate simulated wave patterns is by using a technique called Perlin Noise, named after Ken Perlin which invented a way to generate continuous noise. The Perlin Noise technique is described more detailed in [2], [8] and [9]. In short Perlin Noise is an algorithm for creating seemingly random noise. However, given the same input to the algorithm, the same noise is produced each time. By varying the Perlin Noise frequency and amplitude we get different octaves of the noise, high amplitude values give rougher looking noise, while low values makes the noise look smoother. High frequencies yield dense noise, while low frequencies yield less noise. By layering the octaves of noise we get fractal noise, as shown in *figure 3.3*.

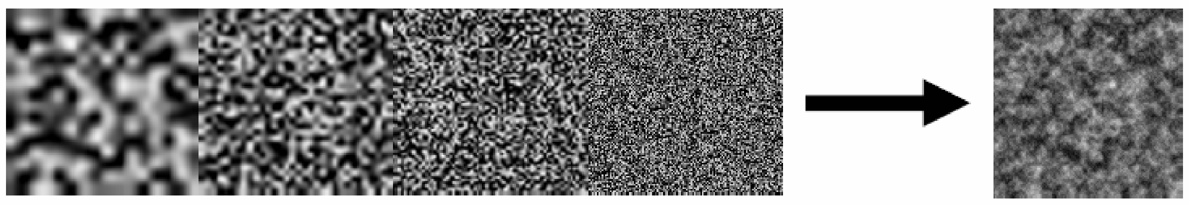


Figure 3.3 – Different octaves of Perlin Noise summed up to a fractal noise. Images are copied from [2].

The noise can be 1 dimensional, producing noisy line textures, 2 dimensional, making static noise textures, or 3 dimensional or higher, making dynamic, animated noise textures. By filtering the noise through a normal map algorithm [3] we get interesting wave-patterned normal maps (see *section 3.2*) for use on the water surfaces in our model.

The Perlin Noise technique does not provide visually correct waves. There are no way of applying parameters like the wind speed and direction to the generation of the noise. However, for use in a real-time visualization the technique provides a pretty satisfactorily result as shown in *figure 3.4*.

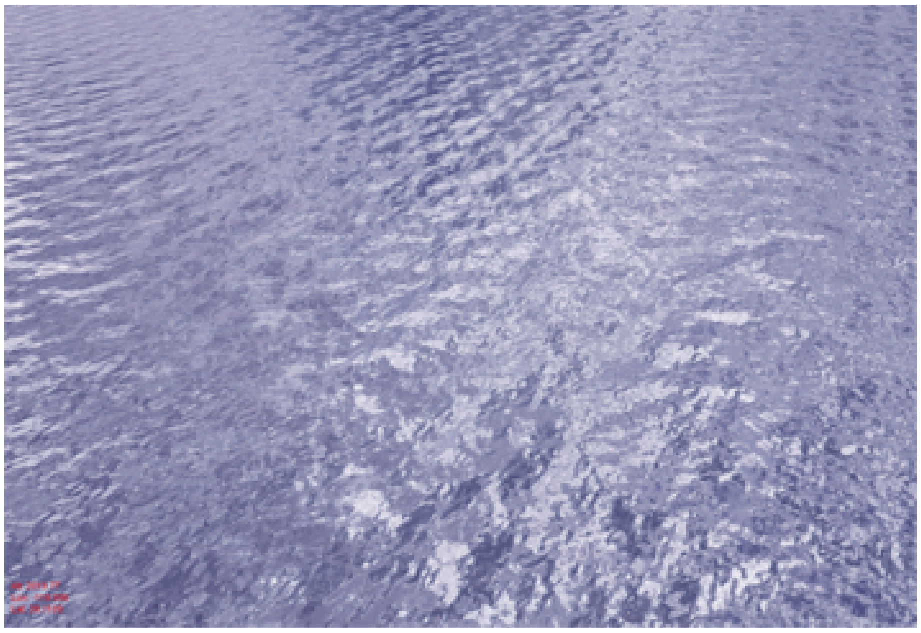


Figure 3.4 Screenshot of waves on a water surface generated with Perlin Noise and Per-Pixel Lighting bump mapping technique.

3.1.2 Water optics

One of the most important properties of water is its ability to act as reflector. The reflectivity of water varies between five and one hundred percent [1] dependent on the angle between the view vector and the surface normal. For angles where the reflectivity is high the refraction will be low, meaning that most of the incoming light will be reflected of the surface and it will act much like a mirror reflecting its surroundings with little loss of intensity. For angles where the reflectivity is low the refraction of the water will be high, and most of the incoming light will be transmitted through the surface. On areas where the refraction is high the light coming from below is visible to the viewer. This light can be reflected from the water bottom if the water is shallow, or from the water volume itself. The impurities of the water determine the scattering of the light and its color, thus the brown color of muddy waters, greenish color of tropical waters and dark, almost black color of deep ocean water.

Fresnel reflection

The property of water that causes it to reflect light at certain angles and refract it at other angles is described in [4] and called Fresnel reflection. Fresnel reflection occurs commonly in nature and is most visible in semi-transparent materials such as water and glass. But the effect also occurs when viewing opaque materials such as paper and metal. Fresnel reflection is expressed by a formula which describes how much light reflects at the material boundary, and is called a Fresnel factor denoted R(θ). *Figure 3.5* illustrates a case of Fresnel reflection where a ray of light from material *i* incident at the surface of material *t*. The materials *i* and *t* have a given index of refraction, which is the probability that a photon of light will be transmitted into the material. The index of refraction for air is close to 1 (vacuum is 1), while the index of refraction for water is approximately 1.33.

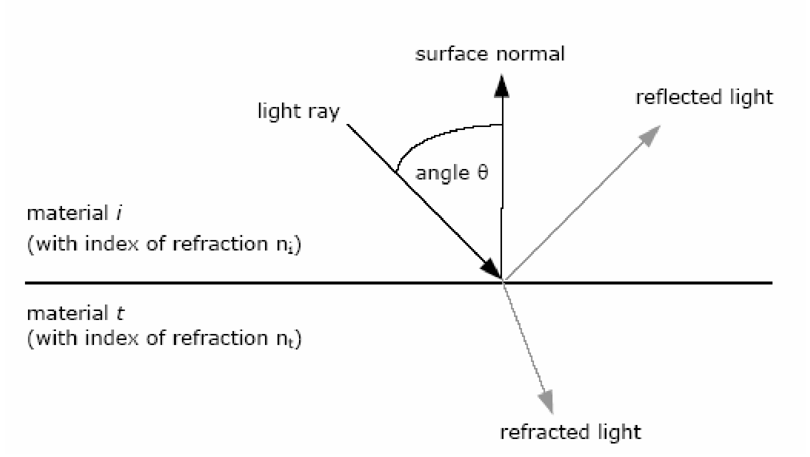


Figure 3.5 – Ray of light traveling through material *i*, striking a denser material *t*. Images copied from [4].

The amount of reflection depends on the angle of incident θ, the polarization of the light, the ratio of the indices of refraction n_t/n_i and the lights wavelength. *Figure 3.6* shows the amount of reflection, R(θ), on a water surface dependent on the angle θ.

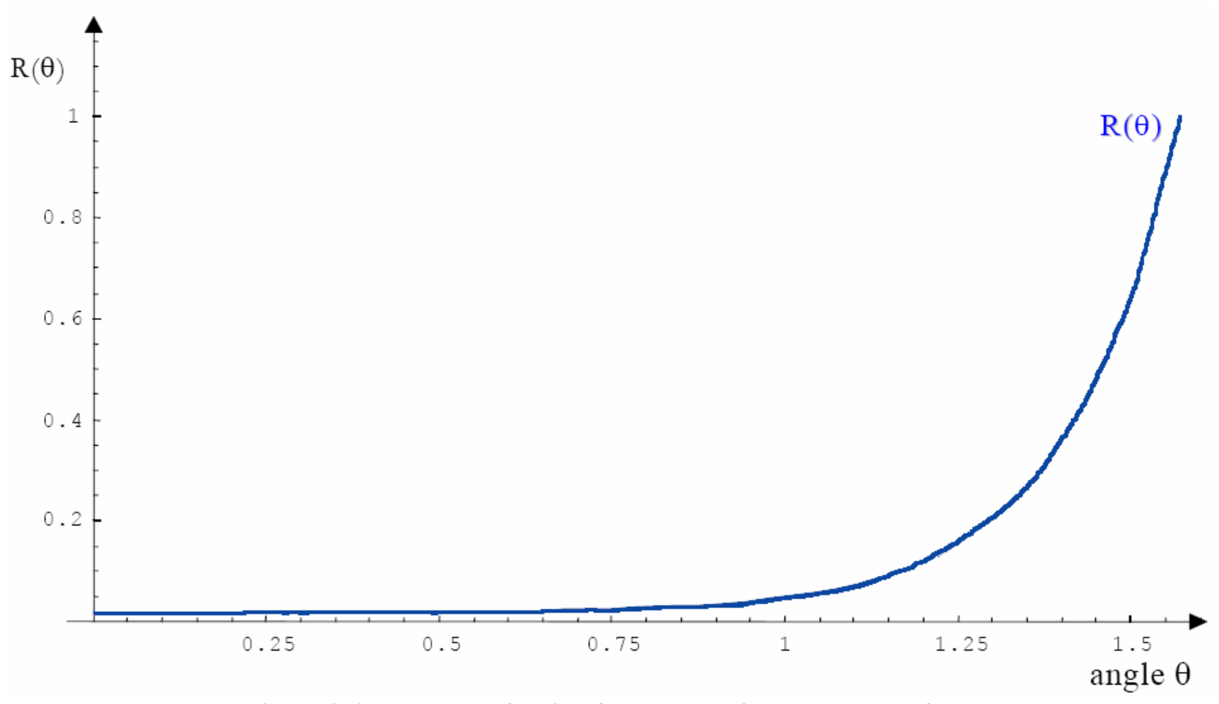


Figure 3.6 – Fresnel reflection for an angle θ on a water surface.

The Fresnel factor equations:

Equation 1.1 $R(\theta) = \frac{1}{2}((g-c)/(g+c))^2(1+[(c(g+c)-(n_i/n_t)^2)/(c(g-c)+(n_i/n_t)^2)]^2)$

Equation 1.2 $c = \cos(\theta)\ n_i/n_t$

Equation 1.3 $g = \mathrm{sqrt}(1 + c^2 - (n_i/n_t)^2)$

Computing the exact Fresnel factor is not very efficient due to the required number of instructions as shown in *equations 1.1* through *1.3*. However, we can approximate the factor to yield a pretty close result with a lot less computation. The most simplistic approach is 1-cos(θ) which is compared to R(θ) for water in *figure 3.7*. As you can see, this approximation gives a curve that is a little to steep measured up to R(θ).

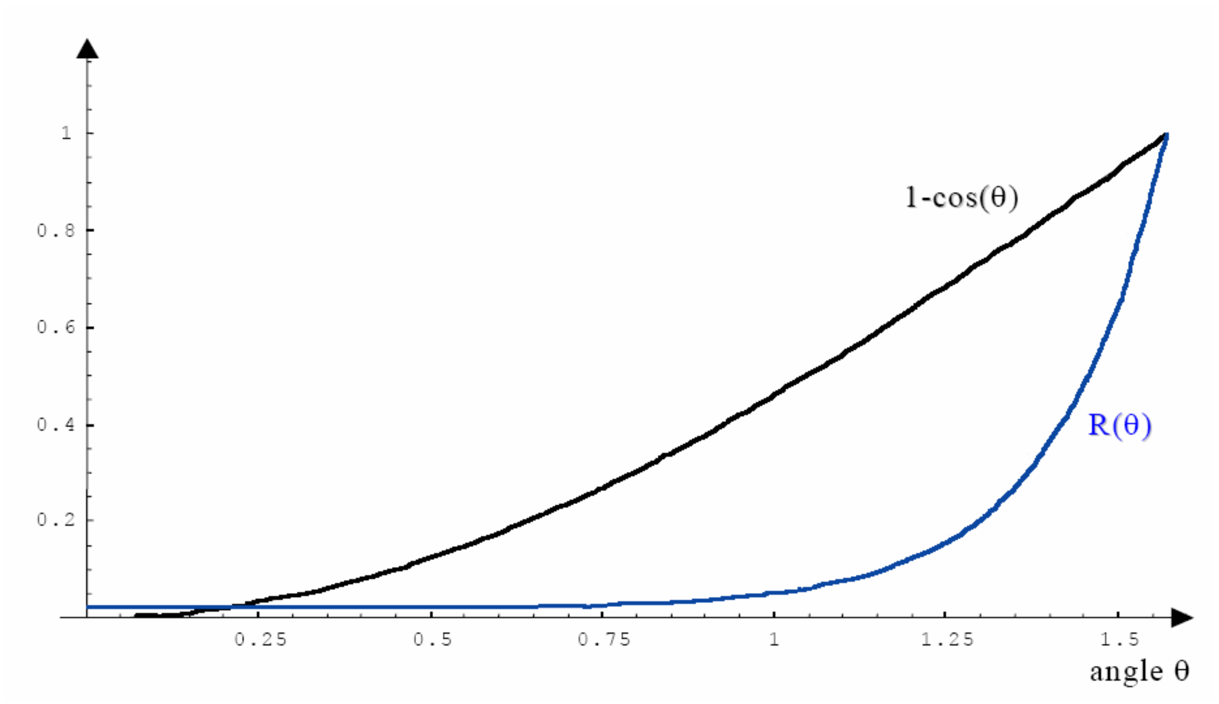


Figure 3.7 – 1-cos(θ) approximation of R(θ).

A more accurate approximation is shown in *figure 3.8* as the red line. This approximation yields a curve that very close to R(θ) and is denoted $R_a(θ)$.

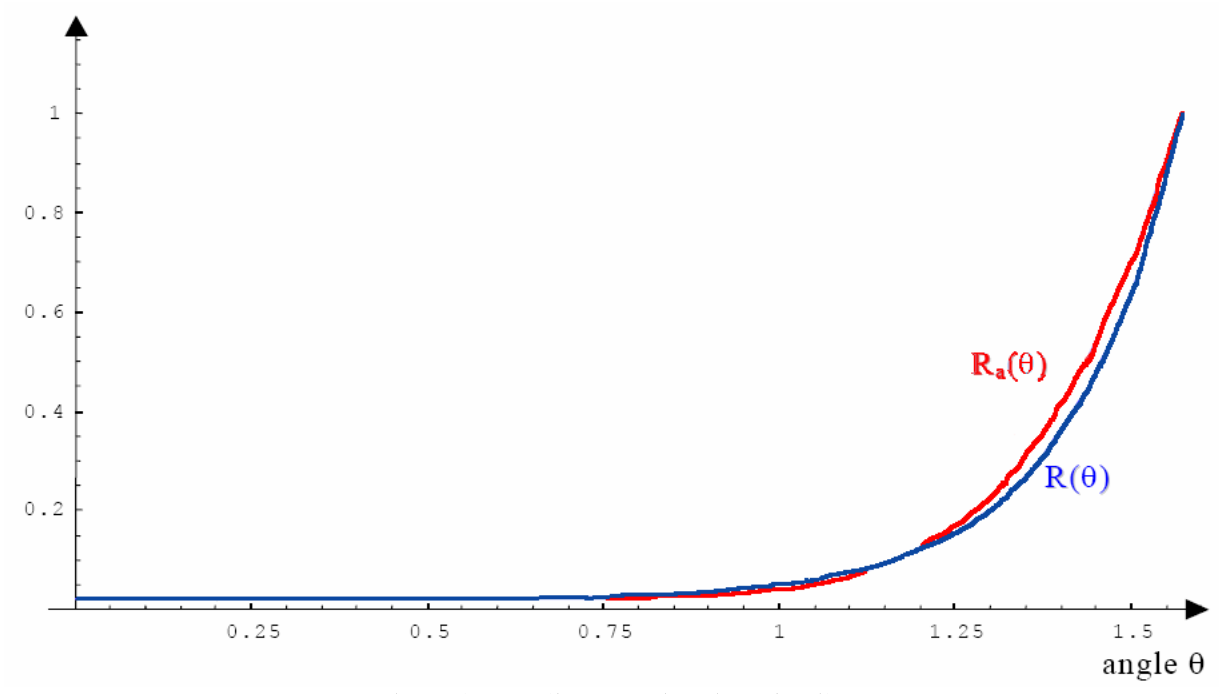


Figure 3.8 - R_a(θ) approximation of R(θ).

Equation 1.4 $R(\theta) \approx R_a(\theta) = R(0) + (1-R(0))(1-\cos(\theta))^5$

Equation 1.5 $R(0) = (1.0-n_t)^2/(1.0+n_t)^2$

Equation 1.4 shows the simplification of R(θ) into R_a(θ). R(0) is a constant that denotes the reflectivity percent of material at a given angle θ [8]. For water this constant is ≈ 2%.

However, by choosing this value the surface might look less reflective than real water because the refractive index is essentially a complex value entity [8]. By adjusting R(0), other effects like dispersing minerals or other particles may be simulated.

The Fresnel factor can either be computed directly per pixel or vertex using *equation 1.4* and *1.*5, or obtained by doing a lookup on a pre-generated 1D texture using the dot product between the *reflection vector* and the *surface normal*. The lookup returns a shade of color which equals `R(θ)` if the texture is generated using *equation 1.4*.

Figure 3.9 shows computer-generated water with and without a Fresnel factor.

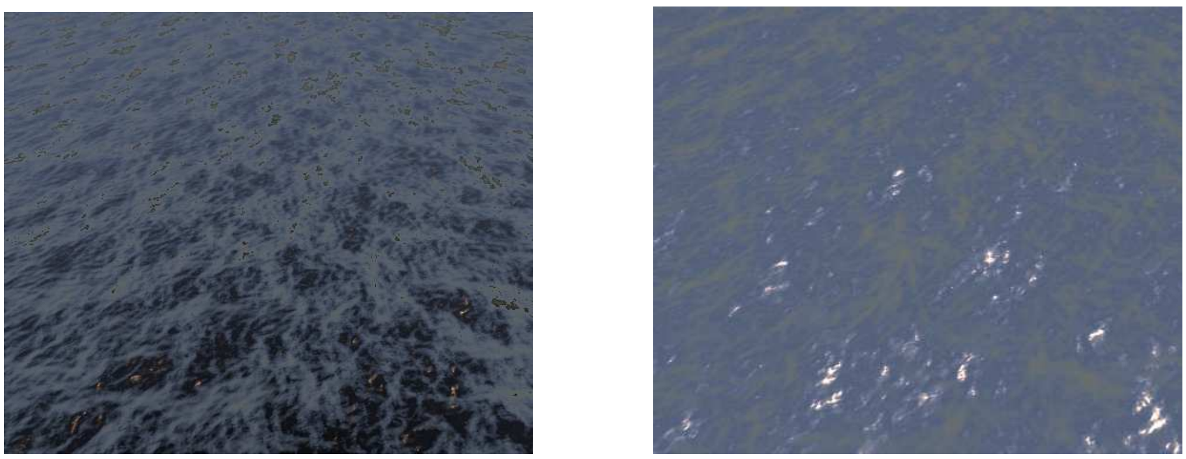


Figure 3.9 – Left: Computer generated water with Fresnel factor. Right: Without Fresnel factor

Reflections

Reflection is probably the effect that gives water its most distinct appearance and is a subject that is described in a lot articles written about realistic water rendering. The most correct and realistic way of creating water reflections is by using ray tracing. However, this is virtually impossible to do real-time with today's hardware. Consequently, a lot of simplified ways to create water reflections has been proposed. In this thesis we will focus on techniques described by Yann Lombard [5] and Claes Johanson [2]. They suggest that water reflections are divided in to 3 separate types of reflections, *global*, *local* and *sunlight reflections*.

- *Global reflections* are the reflection of objects that "infinitely" far away. An example of such an object is the sky, even though it is not actually infinitely far away, we can consider it to be. What part of the sky that is reflected of the surface is only dependent on the reflection vector and the position of the reflecting surface has no influence. These types of reflections are usually implemented as cube maps [6].

- *Local reflections* are reflections of objects that are not infinitely far away. This is typically objects that are part of the environment such as mountains, trees, houses, planes, etc. The difference between global and local reflections is that the location of the reflecting surface does matter with local reflections, and not just the angle and direction of the reflection vector. Local reflections should be done with ray tracing in order to be visually correct. However, the common way to treat local reflections is by using the mirroring concept [5]. If we consider water surfaces to be flat planes they will act just like mirrors. By flipping the local objects along the desired plane, and cut away everything that is on the backside of this plane, we get a mirrored scene of the environment. Subsequently, we render this scene to a texture, and apply the texture to the surface using projective texturing (see *section 5.4.1*). This approach yields a pretty good approximation of the local reflection as shown in *figure 3.10*. By using the textures alpha channel we can mark of the sections of the local reflection that we want to replace with global reflections.

Since a water surface is not really a flat plane the local reflections needs to be disturbed to obtain a water-ripple looking effect. By offsetting the texture coordinates with the surface normals, which in our case are the wave-patterned normal maps, we can approximate this effect. *Figure 3.10* illustrates approximation of local reflection ripples with different offset variables.

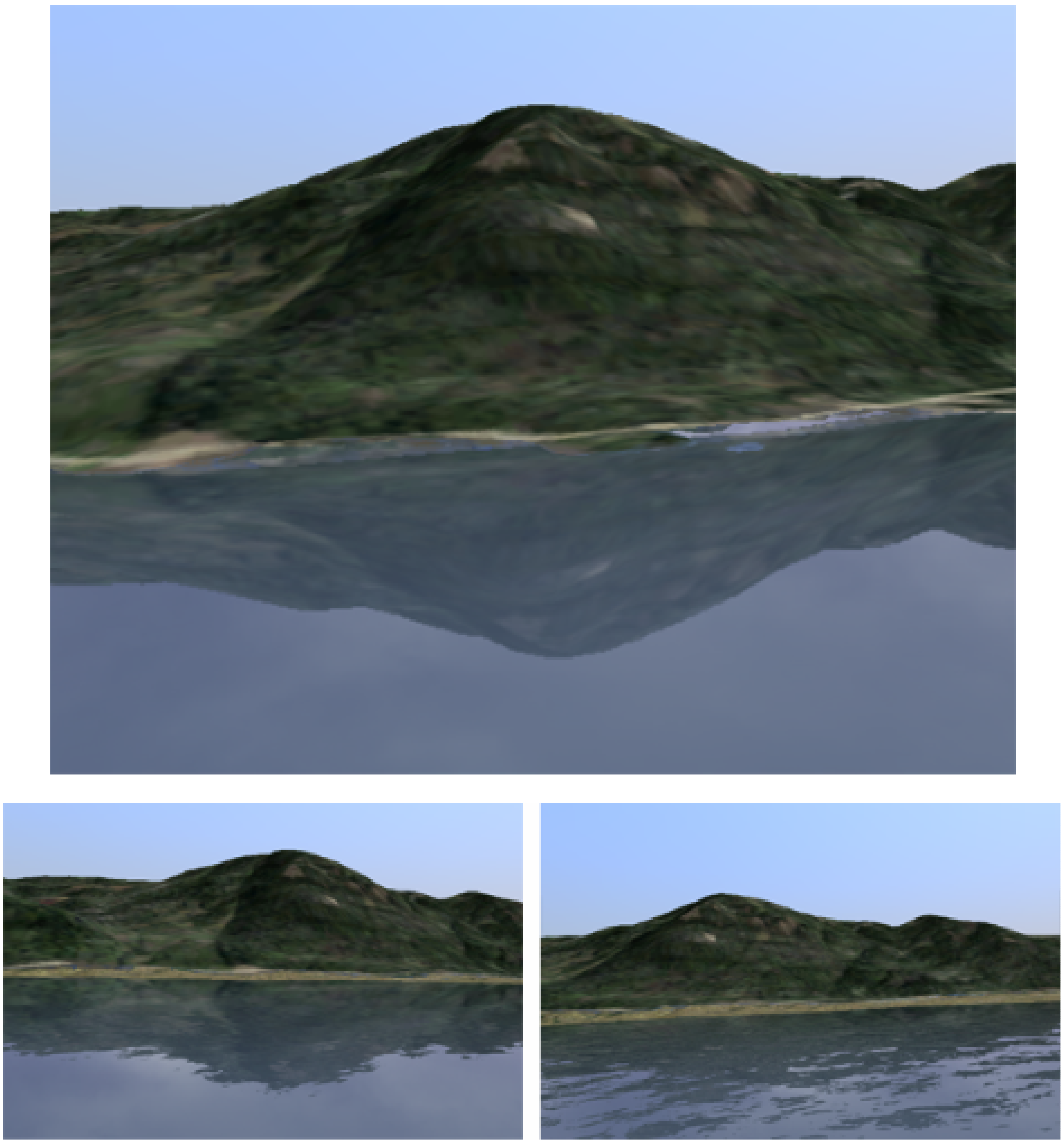


Figure 3.10 – Local reflections using the mirroring concept.
Top: No offset variable.
Left: small offset variable. Right: large offset variable.

- *Sunlight reflections* are the direct reflections of the sun. Since the sun is also “infinitely” far away it could be part of the cube map in the global reflections. But this could make the sun reflection look bleak and washed out, since the amplitude of the sun has to be saturated in order to fit the dynamic range of the texture. It is instead usually better to use classic Phong or Blinn lightning to add specular highlights.

Refractions

Water refraction is the effect gained from Fresnel reflection described earlier. Claes Johanson [2] proposes an approximation technique that is pretty similar to the local reflection effect. Instead of flipping the environment, the scene is scaled by 1/1.33 in the height direction due to the difference of index-of-refraction across the boundary, making the water look shallower when seen from above the surface. Subsequently everything above the surface is removed, and the scene is rendered to texture. The texture is then applied to the surface with projective texturing, as the refractions seen on steep view angles. The refraction is rippled the same way as the local reflection to make it look more realistic. If we do not want refractions, or if the water is very deep, we just set the refraction to be a constant color.

3.2 Bump mapping

In a real world environment we often recognize the shape of an object based on its large-scale geometry, even if its small-scale geometry might differ considerably from this shape. Walls, tables, paper, golf balls, etc. are examples of such objects that we may think of as *round* or *flat*, but in most cases their surface contain small bumps and irregularities. The eye percept these irregularities because of the variation in the light reflections, and we subconsciously register that the objects surface is not completely flat or rounded. However, since these irregularities normally are quite small relative to the objects overall geometry, we recognize the object according to its large-scale shape.

The significance of these irregularities, or lack of them, becomes quite obvious when modeling objects with computer graphics. If we where to model a brick wall, and just rendered a rectangular polygon with a brick texture attached to it, we would get a rather flat and unrealistic looking object. Instead, we could render the entire perturbed geometry of the wall with lit polygons. However, this would be a very computationally expensive and data intensive operation. A more efficient approach to model surface irregularities is by using a *bump mapping* technique. Bump mapping was invented in 1978 by Blinn, and decouples texture-based description of small-scale irregularities per-pixel, from the per-vertex description of the large-scale geometry. Mark J. Kilgard [3] describes bump mapping as *“…a normal-perturbation rendering technique for simulating lighting effects caused by patterned irregularities on otherwise locally smooth surfaces.”* In other words; bump mapping is a way of fooling the eye to believe that a smooth surface is actually bumpy.

Per-pixel lighting

Bump mapping is not the name of a single technique. There are several methods that formerly have been used to bump map surfaces such as the “Offset Vector Bump maps” technique [3] and “Emboss bump mapping” technique [3]. The most common bump mapping method in today’s computer graphic is the “Normal mapping” technique [3], also known as “Per-Pixel Lighting”, which currently also is the most advanced technique. This is the technique we are using in this thesis.

As the name implies, *Per-Pixel Lighting* is a technique where the surface lighting intensity is computed per-pixel, instead of the standard per-vertex lighting commonly used in 3D modeling.

Equation 1.6 – Blinn light model for a single point light source per-vertex

$$I = Ambient + Diffuse*\max(0, L\cdot N) + Specular*\max(0, H\cdot N)^{shininess}$$

Equation 1.7 – Phong light model for a single point light source per-vertex

$$I = Ambient + Diffuse*\max(0, L\cdot N) + Specular*\max(0, L\cdot R)^{shininess}$$

Equation 1.6 and *1.7* describes two single point light source models commonly used in 3D graphics. **L** is the normalized light vector per-vertex, **N** is the normalized plane vector per-vertex, **H** is the normalized half-angle vector and **R** is the normalized reflection vector. By computing the light intensity (**I**) at each vertex, and interpolate the result across the surface, we get an object lit by a single point light source. Per-pixel lighting uses the exact same equations, but with the vectors given per-pixel instead of per-vertex. This yields the modified light models described in *equation 1.8* and *1.9*.

Equation 1.8 – Blinn light model for a single point light source per-pixel

$$I = Ambient + Diffuse*\max(0, L'\cdot N') + Specular*\max(0, H'\cdot N')^{shininess}$$

Equation 1.9 – Phong light model for a single point light source per-pixel

$$I = Ambient + Diffuse*\max(0, L'\cdot N') + Specular*\max(0, L'\cdot R')^{shininess}$$

L', **N'**, **H'** and **R'** are the light, normal, half-angle and reflection vector given per-pixel.

There are three typical ways of computing per-pixel lighting, by using the *Dot3ARB technique* [11], by using the *Register Combiners technique* [3] or by using *Vertex and Pixel shaders* [9]. What is common for all three methods is that they describe the surface's small scale geometry with as raster of perturbed normals composed as a texture map, thus the name "Normal mapping". Each texture texels RGB value corresponds to a XYZ value, and describes a directional normal vector of a single pixel. Since direction vectors has a XYZ range of -1 to 1 they have to be remapped to 0 to 1 in order to be stored as RGB values (we cannot have negative RGB values). The remapping is done by adding 1 and dividing by 2 on each XYZ value. Since normal vectors with direction straight up tends to be the most common vector on a surface, normal maps usually has a distinct bluish color, as shown in *figure 3.11*. This is because the up direction vector [0,0,1], remapped to RGB, is [0.5, 0.5, 1.0].

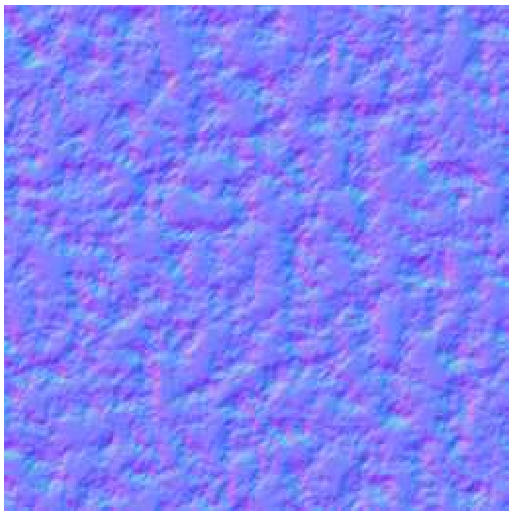


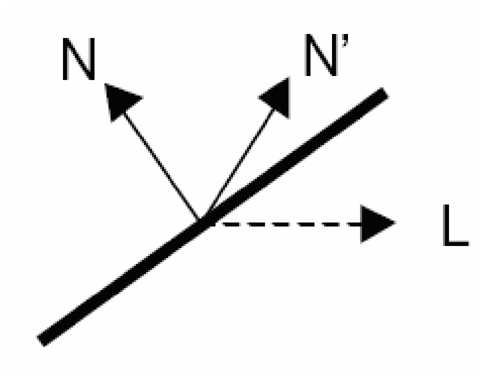
Figure 3.11 – Normal map showing the distinct bluish color

Another common factor for the *Dot3ARB technique* and the *Register Combiners technique* (but not *Vertex and Pixel shaders*) is the need for a normalization cube map [3]. A normalization cube map returns a texture composed of normalized vectors based on the direction of the vector handed to the cube. The *Dot3ARB technique* and the *Register Combiners technique* use the normalization cube map to normalize the **L'** and the **H'** vector or the **L'** and the **R'** vector. The normal map and the cube map normalized vectors are then combined according to either *equation 1.8* or *equations 1.9* in order to obtain per-pixel lit surfaces. *Vertex and Pixel shaders* do not need a normalization cube map because the normalization can be done directly in the pixel shader (see *section 3.3*).

A great advantage of *Vertex and Pixel shaders* is that the Per-Pixel Lighting can be done in one rendering pass. The other two techniques have to perform a rendering pass for each light contribution, ambient, diffuse and specular, and blend the passes together. An additional advantage of *Vertex and Pixel shaders* is that they do not have the limitation on the **shininess** component in *equation 1.8* and *1.9* as the two other techniques have. The implementation of the *Register Combiners technique* is normally limited to have a shininess component of maximum 8, while the *Dot3ARB technique* generally limits the component to be maximum 2 (but this might be increased by more advanced programming). The only obvious advantage of the *Register Combiners* and *Dot3ARB* bump mapping techniques, are that older graphic cards that do not support Vertex and Pixel shaders usually support them.

Since Per-Pixel Lighting is a technique where the small scale geometry is described with textures that are decoupled from large scale geometry, the vectors used in *equation 1.8* and *1.9* has to be rotated into texture space before applied to the equations. This is done by first computing the **L'**, **H'** and **R'** vectors in object space and then multiply them with a matrix consisting of the surface's tangent, binormal and normal. Another possible solution is to instead rotate the normal map vectors into object space. Either way, the vectors used in *equation 1.8* and *1.9* have to be in the same coordinate system.

A problem with Per-Pixel Lighting is the occurrence of so called self-shadowing. This state is shown in *figure 3.12,* and occurs when a pixel should be lit according to the perturbed normal, but shadowed according to the surface normal. This problem is handled by adding a self shadowing term, s_{self} , to *equation 1.8* and *1.9* as shown in *equation 1.11*. The value of s_{self} is specified in *equation 1.10*.



$L \cdot N' > 0$
$L \cdot N < 0$

Self-shadowed due to N but not N'

Figure 3.12 – Occurrence of self-shadowing. Image copied from [3]

Equation 1.10 – Self-shadowing term.

$$
\begin{aligned}
S_{self} &= 1 && \rightarrow && L \cdot N > c \\
S_{self} &= 1/c\,(L \cdot N) && \rightarrow && 0 < L \cdot N <= c \\
S_{self} &= 0 && \rightarrow && L \cdot N <= 0
\end{aligned}
$$

Equation 1.11 – Self-shadowing term added to equation 1.8 and 1.9

$$I = Ambient + Diffuse*S_{self}*max(0, L'\cdot N') + Specular*Self*max(0, H'\cdot N')^{shininess}$$

$$I = Ambient + Diffuse*S_{self}*max(0, L'\cdot N') + Specular*Self*max(0, L'\cdot R')^{shininess}$$

Mipmapping

In order to avoid aliasing and artifacts on textured objects in a dynamic scene, OpenGL built-in mipmapping-functions [10] may be used. Aliasing and artifacts on textured objects appear when an object is moving away the viewpoint. This is because the pixel-count of this object is decreasing. If a high-resolution texture is applied to the moving object the texture may seem to change abruptly at certain transition points. To avoid this problem, down-sampled versions of the texture are stored in memory, and the texture with the appropriate size according to the object is determined and applied to the object by OpenGL. However, using mipmapping with normal maps requires the down-sampled textures to be renormalized [3]. Consequently, the built-in mipmapping-functions do not work with these textures. Kilgard [3] suggests a filtering algorithm used for mipmapping normal maps.

3.3 Shading languages

Pixel and vertex shading languages, or just shaders, is a relatively new generic term referring to high-level programming languages that are used to send instructions directly to the Graphical Processing Unit, the GPU. By sending the instructions directly to the GPU the CPU get offloaded, which yields much faster processing of graphical rendering. The introduction of shaders have made it possible to produce real-time, high-detailed graphics that until now only where seen in pre-rendered animations. The most common shaders are the OpenGL Shading Language (GLSL [9]), the High Level Shading Language (HLSL) and C for Graphics (Cg), but a lot of other languages exist as well. GLSL is, as the name implies, a shader designed especially for OpenGL, HLSL work only with DirectX, while Cg is a portable language that work with both OpenGL and DirectX.

GLSL

Since GeoGFX is based upon OpenGL we have chosen to utilize GLSL [9] as the shader language used to implement rendering of the water effects described in this thesis.

GLSL is a C/C++ like language, which as of OpenGL version 2.0 is supported directly through the extensions *GL_ARB_vertex_program*, *GL_ARB_fragment_program* and *GL_ARB_shader_objects*. The language includes support for scalar, vector and matrix types, structures and arrays, texture lookup through sampler types, data type qualifiers, constructors and type conversion, and operators and flow control statements. Variables can be passed from OpenGL as uniforms (which do not change during the execution of the code), or as attributes that can vary on each vertex. When a GLSL program is enabled it fully controls the OpenGL processing pipeline meaning that the program affects every vertex drawn until the program is disabled. For each vertex a vertex program is executed, which is responsible for transforming and rotating the vertex. The final vertex position is set by the built-in variable *gl_Position*. A vertex program may also be used to interpolate vertex attributes and pass them to the pixel program, an example is the vertex normal which can be obtained with the built in variable *gl_Normal*. These attributes are sent to the pixel program as *varying* variables. A pixel program cannot exist without a vertex program (but the other way around is okay). For every execution of a vertex program, a pixel program is executed once per pixel. A pixel program is used to set the pixels final color. The color is set to the built-in variable *gl_FragColor*. GLSL code is written in a separate file, but sent to OpenGL as a string and OpenGL is responsible for compiling the code during runtime.

4 Model building

This chapter presents an outline of the implementation of the *texture Level-Of-Detail with bump mapping* technique in GeoGFX. The actual implementation of the model is discussed in *chapter 5*. *Section 4.1* describes how to acquire the **Water Surfaces** from *figure 1.1*. *Section 4.2* describes two different models for implementing the **LOD management** component. *Section 4.3* outlines an object model based on the *texture Level-Of-Detail with bump mapping* technique.

4.1 Acquiring water surfaces

4.1.1 Concept

As stated earlier GeoGFX is designed to visualize huge real world models fed to the engine as samples of real terrain measures. These samples contain information about specified ground areas. In addition to ground elevation on a given longitude/latitude the information often includes data like ground conditions (rock, soil, water, etc.), vegetation, power lines, matching satellite/aerial photos, etc. A separate GeoGFX application is able to read many different file formats containing this type of sample information. The application filters out the desired information from the samples, and stores it in a file format that can be read and triangulated by GeoGFX.

A simple method for acquiring water surfaces can be done with a similar approach. Information about water location and elevation (lakes can have elevation different from 0) is read from the samples by a separate application, and stored in a file which can be read by GeoGFX. Subsequently, the application reads the terrain triangulation sample file and removes the points that coincide with the water surfaces. The new ground triangulation now contains irregular holes, so called boundaries. The triangulation technique described in *section 2.5* is capable of handling these boundaries.

4.1.2 The water surfaces file

In its simplest form we can think of water as a rectangular, planar surface. If we do so, we only need to know the longitude and latitude of each corner, plus the elevation above ocean water level. Another approach is to use a lot more point samples from the water surface, especially around the shoreline, treating the water surfaces as complex, polygonal non-planar surfaces (water surfaces are non-planar because of the curvature of the earth). The latter approach creates a much larger and complex water surface sample file, and also requires more sophisticated and demanding tools for visualization. We have chosen to use the simple approach, which can be "faked" to look like a complex polygon without high-detailed triangulations (see *section 5.5*). But there are also some limitations that arise using this method, which will be discussed later in this thesis.

4.2 Surfaces in scene graph

As described in *section 2.3.2*, objects rendered with GeoGFX are added as nodes in a scene graph and we can consider water surfaces to be separate objects. However, water surfaces are essentially part of the terrain, and it might make sense to let the *Terrain* node act as the water render control node (**LOD management component**). This is illustrated in *figure 4.1*.

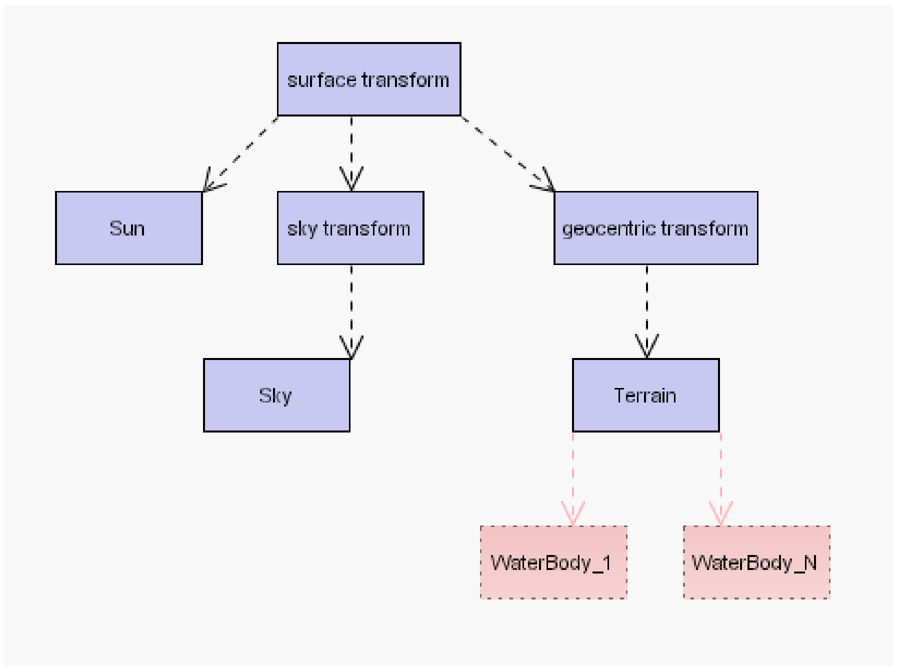


Figure 4.1 – Terrain water surface render node

However, we have chosen not to use this solution because *Terrain* (**GgTerrain**) is already a large class file composed of a huge amount of code. Adding even more code to this class would make further work in GeoGFX a lot more complex.

Instead we are considering water as separate objects, treating them like nodes in the scene graph. We have looked at of two sensible implementations of this scene graph:

- *Model 1* - add each water surface as an object node in the scene graph.
- *Model 2* - add one node in the tree that controls the rendering of the water surfaces.

With both model 1 and 2 we attach the nodes under the *geocentric transform* in the scene graph. This makes sense because even if we think of water as a separate object, they are still part of the terrain and needs to do all the same rotations and translations as the *Terrain* node. The *Terrain* node is attached to the *geocentric transform*, meaning that all actions applied to this transform also will affect all other nodes attached to it.

4.2.1 Model 1 – surfaces as nodes

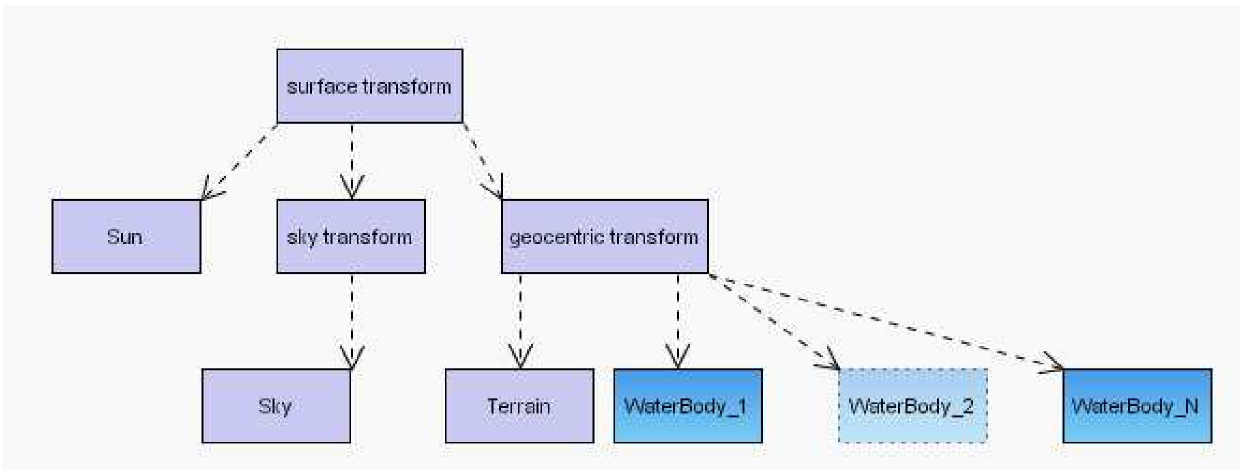


Figure 4.2 - Model 1. Water surface objects as nodes

Model 1 looks correct compared to reality if we think of water surfaces as entirely separate objects with different behavior. Behavior denoting features like wave models and colors. Each object is responsible for handling its own features like textures and frustum culling. A drawback with this model is that the water surface objects do not know anything about each other, and are rendered only according to their own properties. Implementing the **Level-Of-Detail management** will require some sort of separate controlling object, and the final code could end up being quite complex. Instead of implementing this model in GeoGFX, we have chosen model 2 as the basis for the *texture Level-Of-Detail with bump mapping* technique.

4.2.2 Model 2 – control object as node

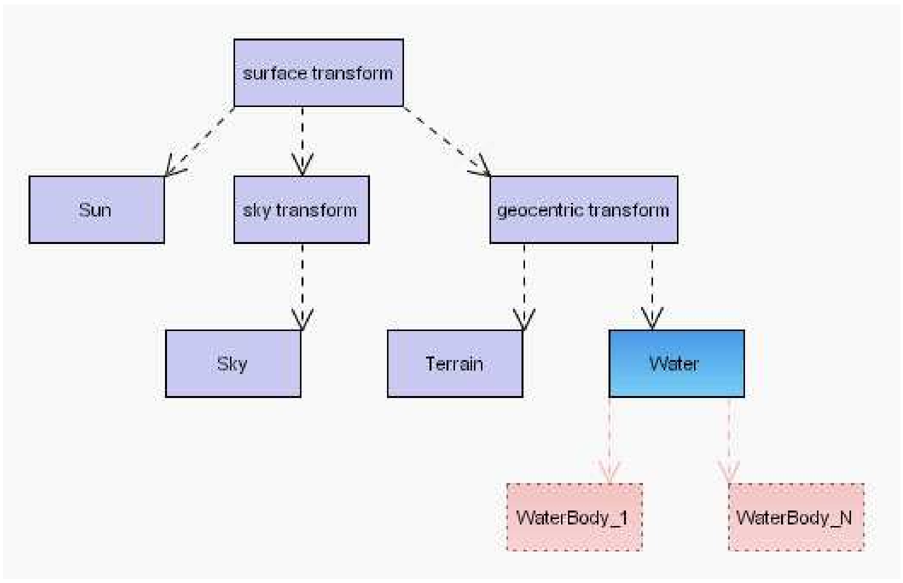


Figure 4.3 - Model 2. Water control object as node

In model 2 the *Water* control object is the only node added to the scene graph. This node is responsible for rendering all water surfaces, and acts as the **LOD management** component. The advantage of having a dedicated scene graph node that handles rendering of all water surfaces, is that this node can act as a tool for handling optimization and controlling of the rendering process. This tool may check properties such as each surfaces distance from the camera, and invoke different states from the "Level-Of-Detail" hierarchy. In addition the object may handle the texture loading and distribution, etc. This way we separate all handling of water surfaces from higher levels in the scene graph, or from separate objects. Further work in this thesis will be based on this model.

4.3 Object Model

Section 4.2 gave an overview on the integration of water surface nodes in the GeoGFX scene graph. *Figure 4.4* shows an outline of the object model. This model will be the basis of the implementation of the code in the final work. The structure of the object model is based on *model 2* (*section 4.2.2*).

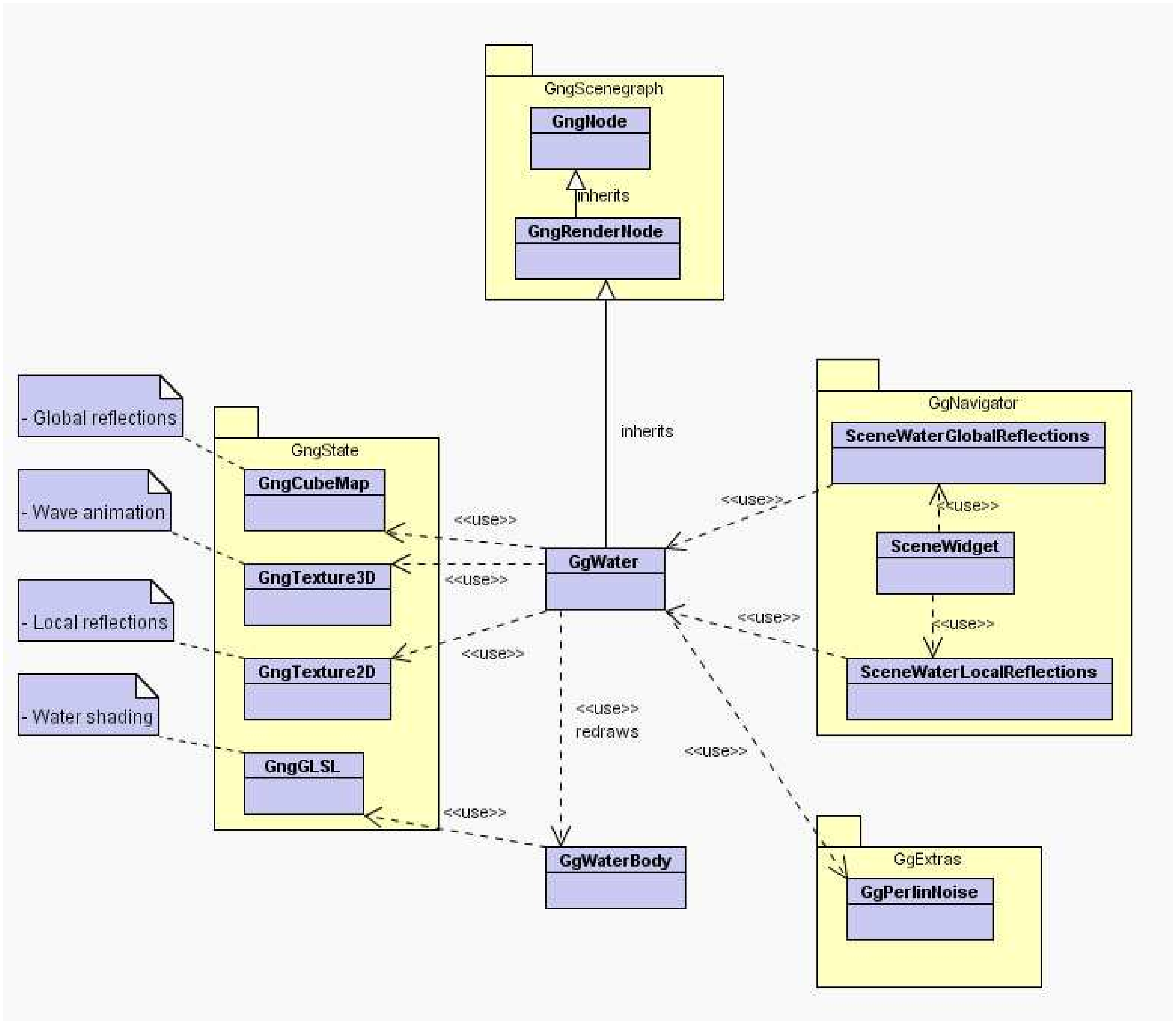


Figure 4.4 - Object model outline

The following list gives a brief description of the classes used in the object model.

GgWater

GgWater can be linked back to model 2, described in *section 4.2.2*, as the render control node added to the scene graph. This class handles the rendering of the water surfaces

(**GgWaterBody** objects) and manages the LOD controlling. Since scene graph nodes are **GngRenderNode** objects, GgWater inherits from this class.

GgWaterBody

GgWaterBody are the water surface objects. This class handles the actual vertex rendering and applies the textures handed from GgWater on the surface. A GgWaterBody object knows nothing about the other GgWaterBody surfaces.

GngCubeMap

GngCubeMap is a class that already exists in GeoGFX but need to be modified to support Pbuffer texture rendering. This class is used to handle the global reflection textures acquired with the **SceneWaterGlobalReflections**.

GngTexture2D

GngTexture2D is also a class that already exists in GeoGFX but needs to be modified to support Pbuffer texture rendering and normal map filtering. GngTexture2D is used to handle the local reflections acquired with **SceneWaterLocalReflections**, and the static wave-pattern textures.

GngTexture3D

GngTexture3D is used to handle the animated wave-pattern textures. This class does also already exist in GeoGFX but needs modification in order to support filtering of 3D normal maps.

GngGLSL

GngGLSL is the implementation of the binding between the OpenGL Shading Language and GeoGFX.

GgPerlinNoise

GgPerlinNoise implements an algorithm for generation of Perlin Noise raster.

SceneWaterGlobalReflections

SceneWaterGlobalReflections handles the creation of dynamic global reflection **GngCubeMap** textures.

SceneWaterLocalReflections

SceneWaterLocalReflections creates the local reflections used on the **GgWaterBody** objects.

SceneWidget

SceneWidget is responsible for invoking **SceneWaterGlobalReflections** and **SceneWaterLocalReflections**. This class already exists in GeoGFX.

5 Water visualization in GeoGFX

This chapter explains the techniques used for the implementation of the water model described in *chapter 4*. The actual code implemented in GeoGFX can be viewed in *Appendix B*.

5.1 Implementing shaders

A binding between GeoGFX and the OpenGL Shading Language, GLSL, is implemented through the **GngGLSL** class as part of the **GngState** package. The implementation includes simplified loading of GLSL code files and easy handling of uniform, attribute and texture variable exchange between GeoGFX and GLSL.

GLSL vertex and pixel programs can either be loaded simultaneously with the function `setShaders(pixel_file, vertex_file)` or separately with the functions `setFragmentShader(pixel_file)` and `setVertexShader(vertex_file)`.

Uniform and attribute variables can be handed to the GLSL code with the functions `setUniform[1-4](name, [xyzw])` and `setAttrib[1-4](name, [xyzw])`. These variables can have 1, 2, 3 or 4 parameters depending on what type they are (vector, single variables, etc.). The name parameter in the functions is simply the name of the variable in the GLSL code. Uniform and attribute matrices can similarly be exchanged with the `setUniformMatrix[1-4]()` and `setAttribMatrix[1-4]()` where *[1-4]* implies whether it is a 1x1, 2x2, etc. matrix. Attribute arrays can be set with the `setAttribArray()` function, and enabled and disabled with the `enableAttribArray()` and `disableAttribArray()` functions. Textures can be passed with the `setTexture(name, number)` function. The texture handed to *name* is the texture stored in the current state *number*. **GngGLSL** objects are enabled and disabled with the `enable()` and `disable()` functions.

5.2 Simulating wave patterns

Section 3.2.2 describes three different methods for simulating wave-patterns, The Navier-Stokes equations, Fourier Transforms and the Perlin Noise algorithm. As stated, Perlin Noise is the less accurate method, but is fairly easy to compute and yields a pretty good visual result. It is commonly used in real-time rendering of water and is the method we have chosen to simulate wave patterns in GeoGFX. Since the wave pattern is used as a normal map with Per-Pixel Lighting, the Perlin Noise texture (which is grey-scale) has to be run through a normal map filter. We have implemented two methods of acquiring Perlin Noise normal maps in GeoGFX.

The first method is by using a pre-generated grey-scale Perlin Noise texture map, either a 2D map for static wave-patterns or a 3D map for animated wave-patterns. An Adobe Photoshop Nvidia normal map filter [14] is applied on the texture as shown in *figure 5.1*, and the texture is loaded into standard **GngTexture2D** or **GngTexture3D** state objects in GeoGFX.

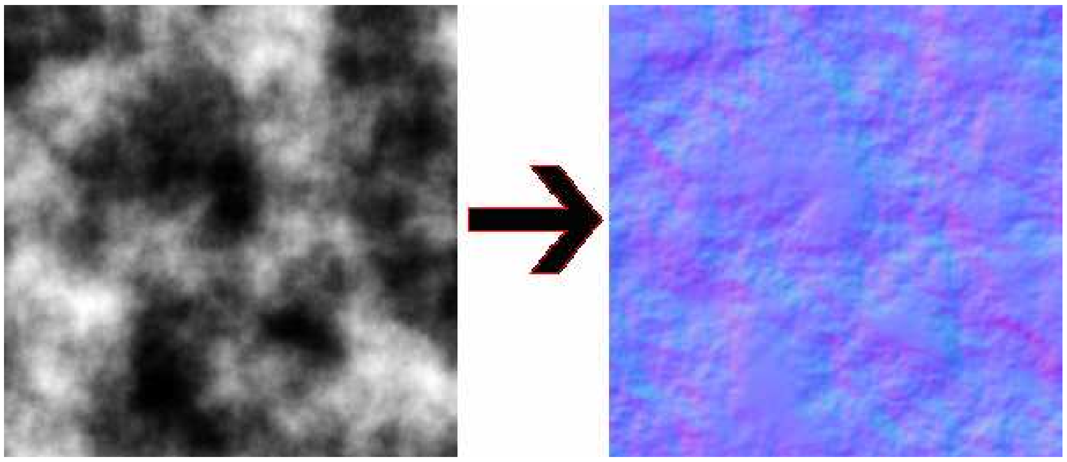


Figure 5.1 – Left: Perlin Noise texture. Right: Texture filtered with Nvidia Normal Map filter

The other method is by using an algorithm to create a Perlin Noise raster during runtime, and filter it through a normal map creation algorithm. Creating the Perlin Noise raster is based on the algorithm described in [9] and implemented in the **GgPerlinNoise** class. The normal map filtering algorithm is based on the algorithm described in [3] and implemented in the **GngTexture2D** and **GngTexture3D** classes. In **GngTexture3D** the algorithm is modified to support filtering of 3D textures instead of 2D textures.

The Perlin Noise algorithm [9] produces as 32-bit RGBA image where each color channel consists of a Perlin Noise raster at a given frequency. Since the normal map algorithm is designed to filter 8 bit mono images, we must either use one of the channels as the normal map wave pattern, or add all the color channels together to form a normal map that is the sum of four Perlin Noise frequencies. Exploiting the **GngImage** class in GeoGFX can obtain the color channels of an RGBA image. By generating the Perlin Noise raster as a **GngImage** object, we get access to the function `decompose()` which splits a 32 bit RGBA **GngImage** into four 8 bit mono **GngImage** objects. Either one of these **GngImage**'s can be filtered through the normal map algorithm and be used as a wave pattern. However, using a single frequency Perlin Noise texture yields a visual result that does not look very realistic. Instead the four channels should be added together to form an 8-bit mono image which creates a more random feel to the wave pattern. The function `composeSUM(r,g,b,a)`, which performs this operation, has been implemented into **GngImage**. Additionally this function may manipulate the final look of the wave pattern by performing different computations on the channels. *Figure 5.2* illustrate the difference between using one Perlin Noise frequency, and using the sum of four frequencies as the wave pattern.

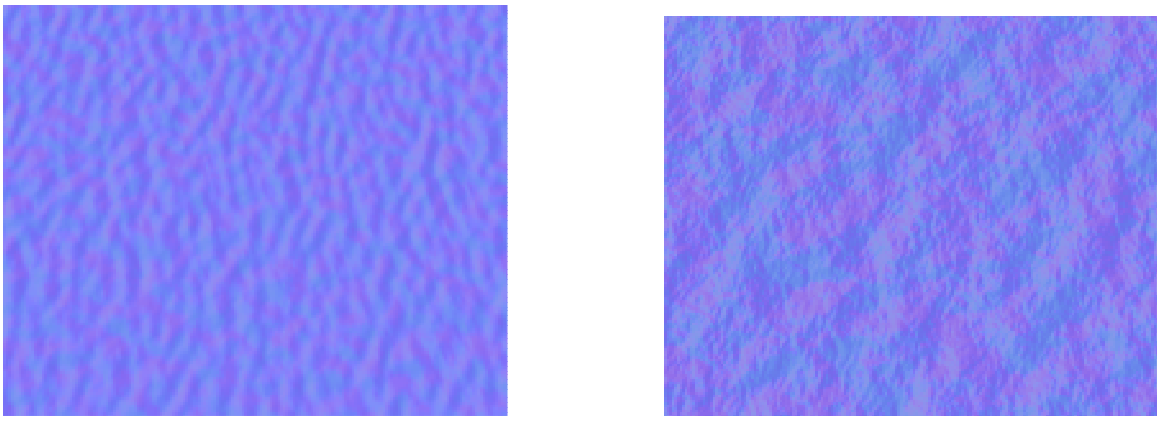


Figure 5.2 – Difference between one frequency wave pattern and sum of four frequencies
Left: 1 frequency. Right: four frequencies

An essential advantage of the latter method compared to the first is its ability to set the size of the texture at runtime. This gives us the possibility to create textures with size relative to the present amount of memory on the computer graphic card. Since 3D textures are nothing but an animated sequence of textures, and the textures need to be tiled to cover a surface. Long sequences and large sized texture tiles will look more realistic that short sequences and small texture tiles. However, these large textures also consume more texture memory.

Another advantage of runtime-generated textures is, as stated earlier, the possibility to perform computation on the different Perlin Noise frequency layers. This can be used to achieve more realistic appearances on the wave pattern. Unfortunately runtime generated textures are quite expensive to compute, so we have to generate them when loading the application. Consequently we are not able to alter the textures during runtime.

Using Per-Pixel Lighting to simulate wave-patterns looks convincing as long as the viewer is further away then a few meters from the surface. The fact that the waves are nothing but a flat texture is revealed when the viewer is to close to the surface (*figure 5.3)*. This makes the technique best suited for flight-simulators such as Silent-Wings [13] (*section 2.7.1*).

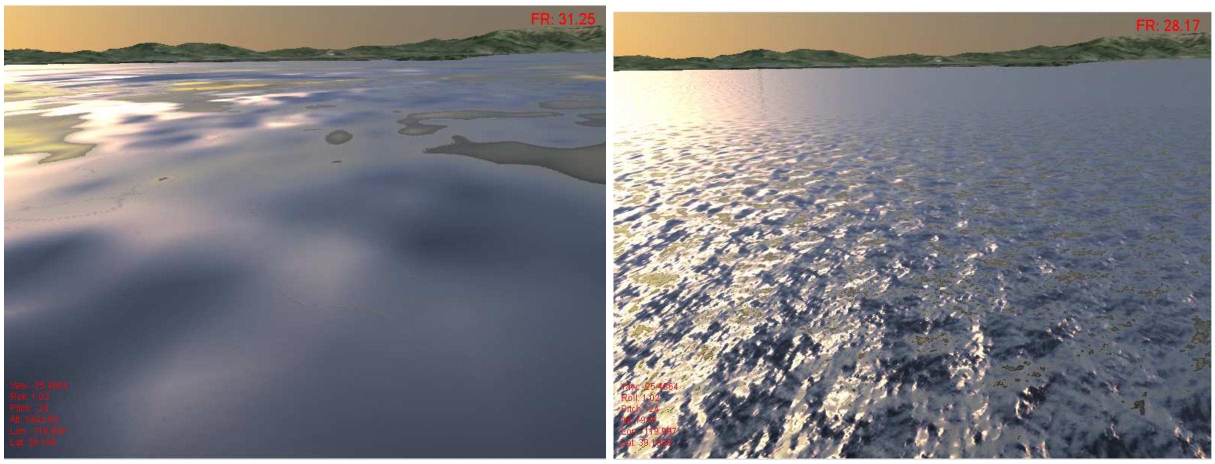


Figure 5.3 – Left: Too close to the surface ruins the illusion. Right: The illusion looks convincing when viewed from a distance

5.3 Visualization vectors

In order to implement the simplified water visualization described in *3.2* we need to acquire various fundamental vectors from GeoGFX. The two most important vectors are the view vector and the light vector (the sun vector). By combining these two vectors and the surface normal (either per pixel or per vertex), we are able to create the different effects we need to simulate water looking surfaces. It is essential that all vectors are expressed in the same coordinate system in order to get a correct result. For reasons explained in the *section 5.4.2* we have chosen to express the vectors in a local coordinate system, also referred to as object space.

5.3.1 The view vector

The view vector is usually obtained by acquiring the view position from the inverted modelview matrix, and subtracts it from the vertex position. However, because of the GeoGFX structure and coordinate system, there is a better way to obtain the vector. As described in *section 2.3*, GeoGFX has component named **GgNavigator**. This component has a class called **SceneWidget** which is responsible for drawing all 3D objects, and handle user input such as mouse movement. **SceneWidget** passes this input to a class named **Observer** whose main task is to update the rotation and transformation transforms in the scene graph as described in *section 2.3.2*. The **Observer** class has functions named `getLatitude()`, `getLongitude()` and `getAltitude()`. These functions can be used to acquire to position of the view camera in a global coordinate system. Converting the coordinates to a XYZ coordinate system can then be done with the class **GgTools'** function `geoToXYZ()`. The coordinates are then passed on to the **GgWater** class described in *section 4.4*, and subtracted from the vertices in each **GgWaterBody** object. The procedure is illustrated in *figure 5.4*.

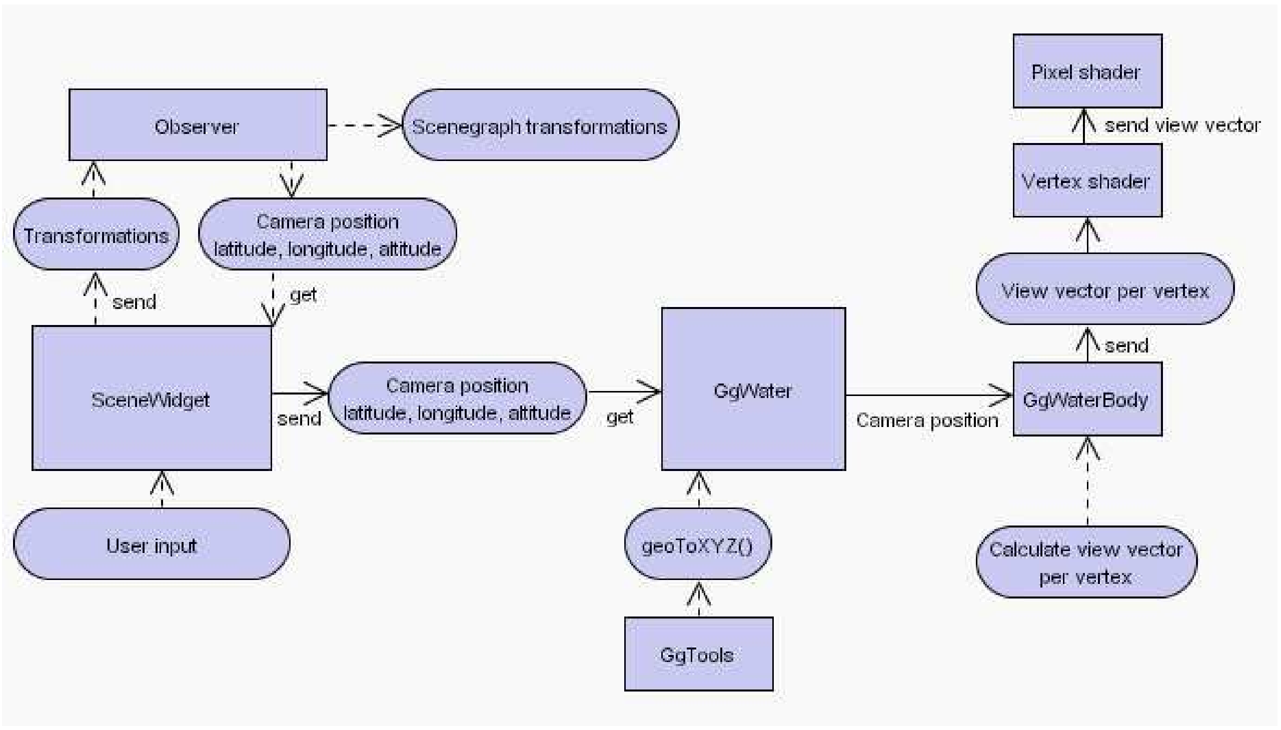


Figure 5.4 – The process of acquiring the view vector

The vertices in **GgWaterBody** are accumulated in a vertex array handling class named **GngVertexArray3f**. They are stored as coordinates in the same XYZ coordinate system as the camera position, but the coordinates are the original position (denoted *vertex_position)* minus the origin of the water surface (denoted *water_origin)*. This is because the surfaces are drawn relative to the origin of the terrain. Thus, to get the view vector per vertex we subtract the sum of *vertex_position* and *water_origin* from the camera position as described in *equation 1.12*.

Equation 1.12
```
view_vector = camera_position – (vertex_position + water_origin)
```

As stated, the view vector is expressed in global coordinates and need to be transformed into local coordinates. This is done with the `earth_to_local` transform obtained with the **GgTools** function `calcEarthToLocal()` as shown in *code 1*.

Code 1 – transforming global view vector to local view vector

```
//Get local transform relative to lat/lon
MtkTransform3f earth_to_local;
GgTools::calcEarthToLocal(centerGEO_.x(), centerGEO_.y(), earth_to_local);
//Transform from local to global direction
view_vector_local = earth_to_local.transform(view_vector);
```

The view vector also needs to be normalized, but this is done in the pixel shader since we are doing per-pixel operations.

5.3.2 The light vector

The light vector is used for the sun contribution on the water surfaces. The **GgSky** class is, as described in *section 2.3.2*, responsible for visualizing the sky dome. The sun is drawn as part of the dome relative to the parameters longitude, latitude and time of day handed to the function `setTimePos()`. Its position can be gained with the function `getSunPosition()` which returns an **MtkVector** from the utility components. This vector is added to a **GngLightSource** class, which is appended to the GeoGFX scene graph as explained in *section 2.3.2*. By handing the **GngLightSource** to **GgWater**, the light vector can easily be acquired by using the **GngLightSource** function `getDirection()`, which returns a normalized direction vector for a light object. The whole process is described in *figure 5.5*.

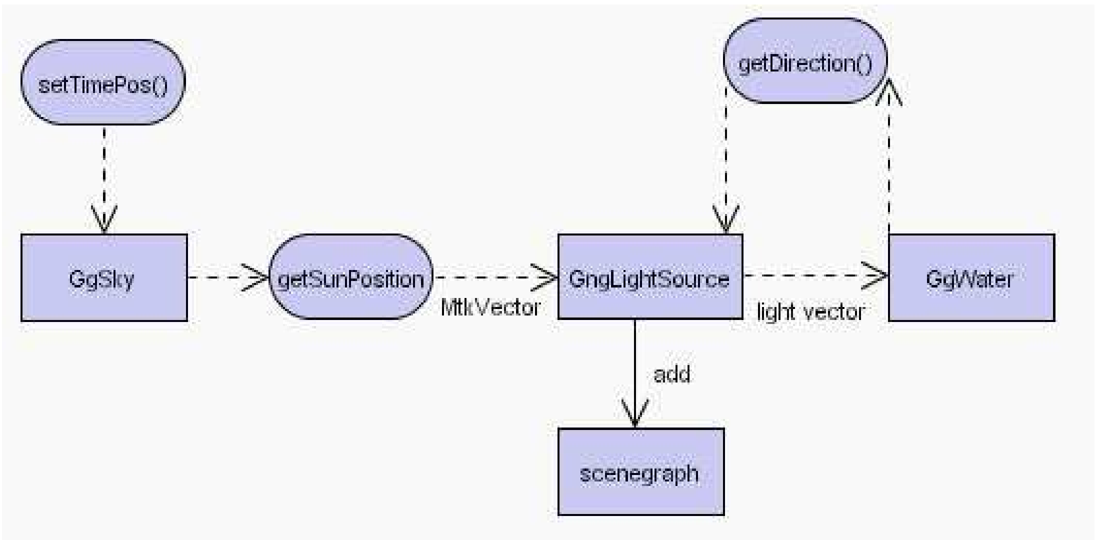


Figure 5.5 – The process of acquiring the light vector

Since the sun can be considered to be "infinitely" far away, the light vectors will be parallel when striking the surface and the angle between the vector and the normal will be the same at each vertex if the surface is planar. This is one reason why we in *section 4.2.2* chose the simple representation of the water surfaces. The light vector is expressed in the local coordinate system relative to the surfaces, and do not need to be transformed.

5.3.3 The reflection vector

The reflection vector is the view vector mirrored against the plane defined by the surface normal (either per-vertex or per-pixel). It tells us from which direction the light that is reflected of the surface originates. The reflection vector can be used for reflection cube map lookups, Phong shading and Fresnel texture lookups. *Equation 1.13* describes the vector.

Equation 1.13 `R = V – 2.0 * (N·V) * N`

R is the reflection vector expressed in the same coordinate system as **V** and **N**, **V** is the view vector and **N** is the surface normal. It is important that **V** and **N** is oriented in the same coordinate system

5.3.4 The surface normal vector

The surface normals are either the vertex normals or the perturbed normal map normals described in *section 3.2*. We have chosen to simulate the wave pattern using normal maps (*section 5.2*). These normals are expressed in a texture coordinate system (usually referred to as tangent space) and need to be rotated into the local coordinate system. To do so we need to multiply the normals per pixel with a suitable transformation matrix. Such a matrix can be expressed as the inverse of a local space to tangent space transformation matrix, shown as the right matrix in *equation 1.14*. This is a 3x3 matrix where the first row is the **tangent vector** of a vertex, the second row is the vertex **binormal vector**, and the third row is the vertex **normal vector**. The **tangent vector** is a vector that is parallel to the texture coordinates on the given vertex. Since our surfaces are plane this vector equals the vector from origin of the surface to the opposite corner The **normal vector** is the standard vector perpendicular to the plane, and the **binormal** is the vector perpendicular to the normal and the tangent vector (expressed as the cross product of the tangent and the normal vector). All these vectors must of course be expressed in a local coordinate system, subsequently the normal vector and tangent vector in GeoGFX has to be transformed the same way as the view vector shown in *code 1*.

$$\begin{vmatrix} Tx & Bx & Nx \\ Ty & By & Ny \\ Tz & Bz & Nz \end{vmatrix} \qquad \begin{vmatrix} Tx & Ty & Tz \\ Bx & By & Bz \\ Nx & Ny & Nz \end{vmatrix}$$

Equation 1.14 Left: Object to tangent space matrix. Right: Inverse matrix: Tangent to object space

5.4 Creating water optics

Reflections and refractions where described in *section 3.1.2,* and the implementation of the reflection handling classes **SceneWaterGlobalReflections** and **SceneWaterLocal-Reflections** where outlined in *chapter 4*. This section describes the actual implementation of these classes into GeoGFX.

5.4.1 Local reflections

Section 3.1.2 describes local reflections as the reflection of objects that are not infinitely far away from the reflecting surface. These types of reflection should ultimately be visualized using ray tracing, but in today's hardware that is not possible to do in real-time. Instead Yann Lombard [5] proposes a so-called mirroring technique where the local objects are flipped around the planar surface, everything behind the surface is clipped and the scene is captured to a texture. The texture is then attached to the reflecting surface using projective texturing and the texture coordinates are offset by the surface normals to create a rippling effect on the reflections.

In this thesis we are using this technique with some modifications to create local reflections on the water surfaces in GeoGFX. The class **SceneWaterLocalReflections** is responsible for setting up the mirrored scene, capture it to a texture and pass the texture to **GgWater** class. **GgWater** then passes the texture on to the correct **GgWaterBody** object and renders the surface with the reflection texture using projective texturing. *Figure 5.6* illustrates the collaboration between the different classes involved in the process of acquiring the reflective textures, while the *code 2* describes with pseudo code the process done in **SceneWaterLocalReflections** to capture a mirrored scene to a texture.

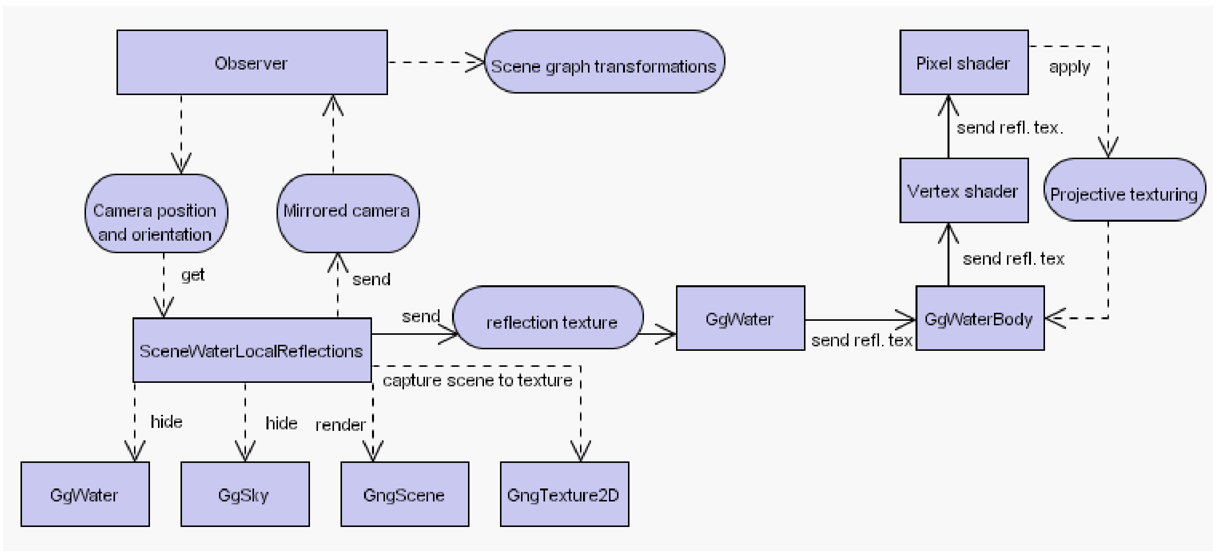


Figure 5.6 – Outline of local reflection process

Code 2 – Create reflection texture in SceneWaterReflections

```
Step 1      Mirror scene
Step 2      Remove objects behind the reflective water surface
Step 3      Set up camera matrices
Step 4      Hide water surfaces
Step 5      Hide sky cone
Step 6      Render mirrored scene
Step 7      Capture mirrored scene to texture
Step 8      Hand reflection texture to GgWater
Step 9      Attach  reflection  texture  to  GgWaterBody  with  projective
            texturing
```

The 9 steps of *code 2* are explained in detail in the following sections.

Step 1 – Mirroring a scene in GeoGFX

Yann [5] proposes a transformation matrix for flipping the local objects. This matrix is described in *equation 1.15* and is called the reflection matrix.

Equation 1.15 – Reflection matrix

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 2h \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

When applying the matrix to local objects they are transformed by flipping them around the z-axis. If the reflecting surface does not cross the origin, the height of the surface above the origin has to be applied to matrix as the *h* parameter.

In theory this matrix could also be used to flip local objects in GeoGFX, but because of the rendering techniques used in this engine we encounter some problems when trying to apply this matrix directly. The main problem is the LOD technique described in *section 2.5*, which used on the ground visualization in GeoGFX. When using the reflection matrix to flip the environment, strange artifacts like glitches in the terrain and unmatched textures appear. This forces us to find some other way to mirror the scene. An approach is instead to mirror the view camera around the surface. Exploiting some of the already existing classes in GeoGFX can do this fairly easy.

The first thing we need to do is mirror the position of the camera around the surface. This means finding the viewpoint (the camera) coordinates in longitude, latitude and height above ocean water level, altitude. As described in *section 5.4.1* these positions are stored in the **Observer** class and can be acquired with the functions `getLongitude(), getLatitude()` and `getAltitude()`. If the rendered surface where ocean water, the only thing we have to do was mirroring the cameras altitude by setting it to *–altitude* since the all the water surfaces would have elevation equal to zero. However, lakes have elevation different from zero, which has to be considered when mirroring the viewpoint. This introduces another problem if we take into account the earths curvature. If so, we have to mirror the viewpoint relative to the position of its closest vertex, meaning a lot more computation than if the whole surface is in the same plane. This is another reason why we in *section 4.2.2* chose to store the water surfaces data as simple files with equal elevation for the whole surface. The mirroring of the viewpoint around a water surface is illustrated in *figure 5.7*. The position of the mirrored viewpoint is described with *equation 1.16*. Only the altitude of the viewpoint is mirrored, the longitude and latitude stays unaltered.

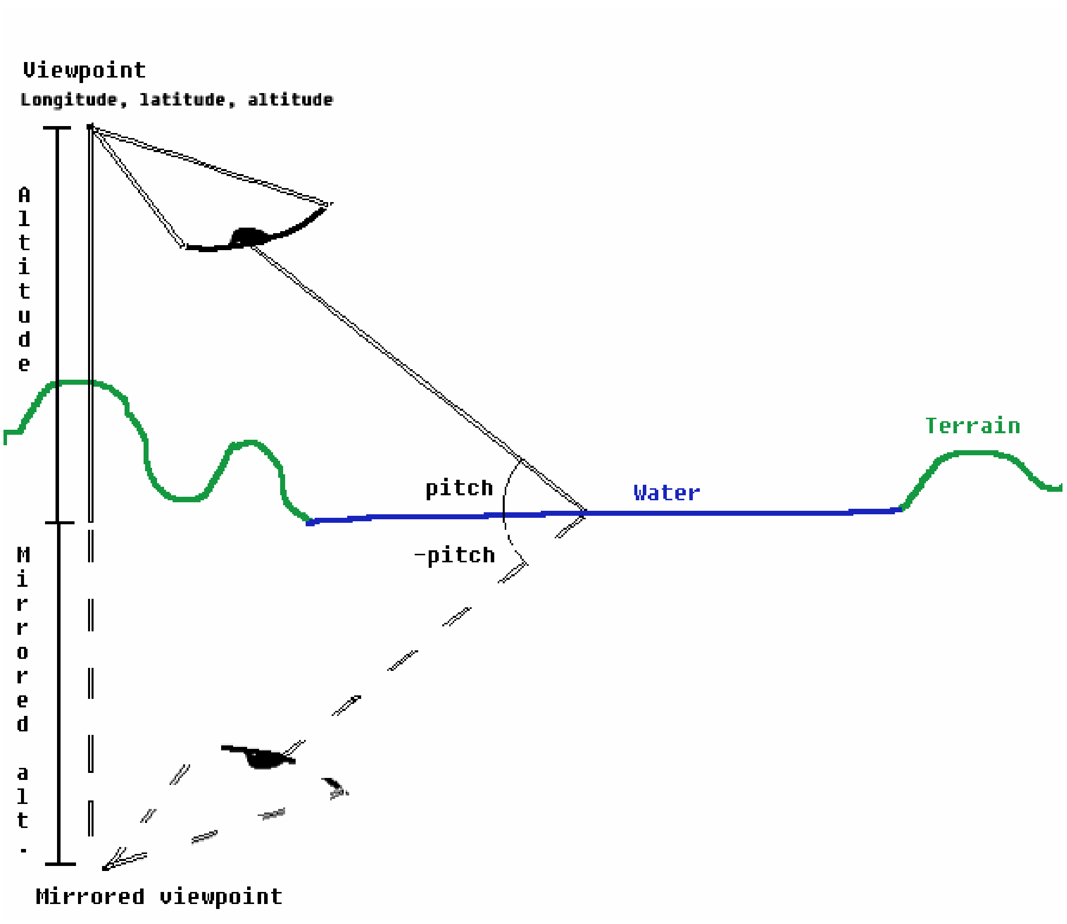


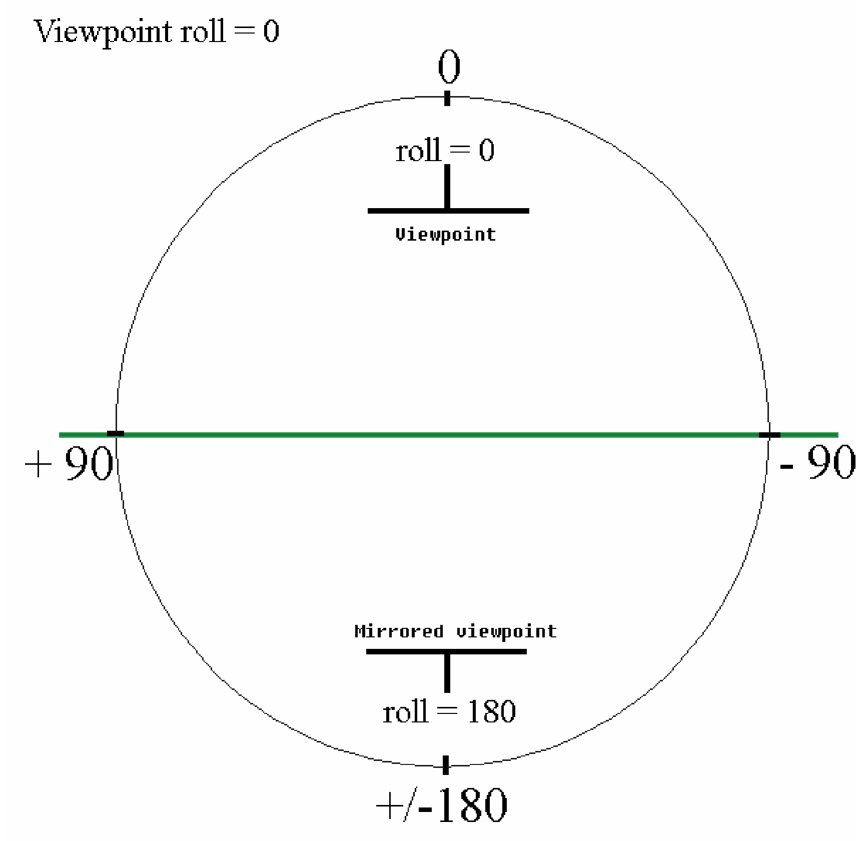
Figure 5.7 – Mirroring of viewpoint position and pitch.

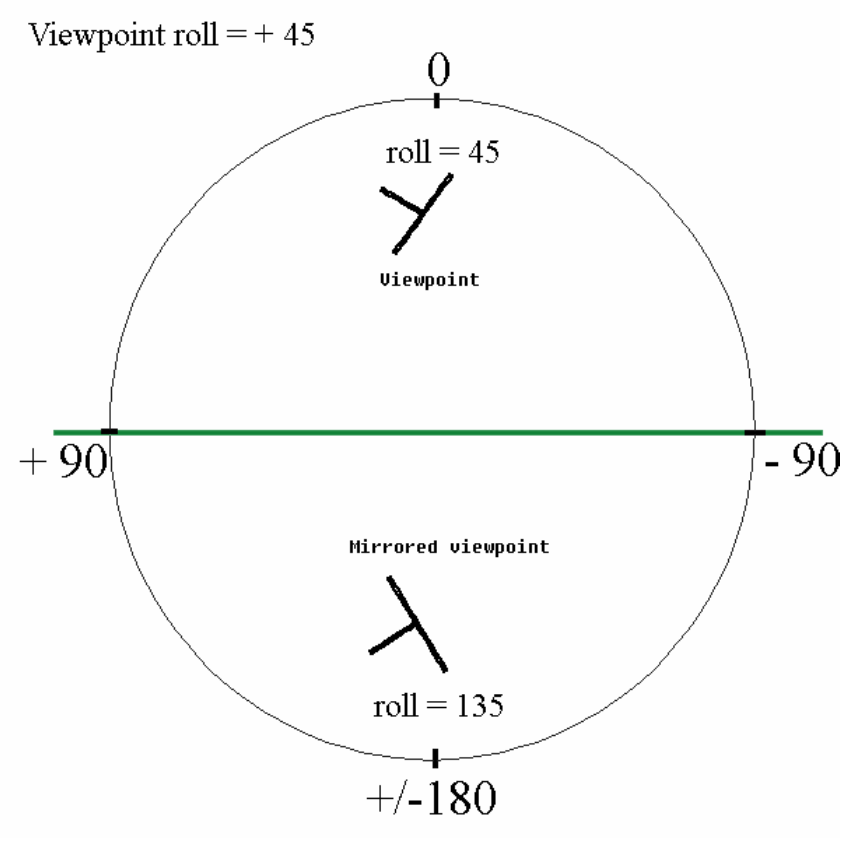
Equation 1.16
```
mirrored_alt = surface_elevation – (viewpoint_alt- surface_elevation)
```

Equation 1.17
```
mirrored_pitch = -viewpoint_pitch
```

Equation 1.18
```
if (viewpoint_roll = 0)
      mirrored_roll = 180
if (viewpoint_roll > 0)
      mirrored_roll = (180 – viewpoint_roll)
if (viewpoint_roll < 0)
      mirrored_roll = -180 + abs(viewpoint_roll)
```

The next thing we need to do is mirror the viewpoints orientation which is the *yaw*, *pitch* and *roll*. These variables can be obtained with the **Observer** functions `getYaw()`, `getPitch()` and `getRoll()`. We don not want to change the direction of the view vector so the yaw of the viewpoint is left unchanged. The pitch of the viewpoint as illustrated in *figure 5.7* defines the angle of the view vector that intersects the surface. Mirroring the pitch is as simple as flipping the viewpoint around the z-axis setting the pitch to *–pitch* as described in *equation 1.17*. The roll of the viewpoint defines its orientation relative to the xy-axis. This orientation has to be mirrored when working with objects in 3D. Flipping it upside down relative to its initial roll does mirroring of the viewpoint roll. Since "no roll" is defined as 0 degrees, the mirrored roll of no roll is 180 degrees. A right roll is defined as -1 degree to -180 degrees, and a left roll is defined as +1 degree to +180 degrees. The mirroring of viewpoint roll is illustrated in *figure 5.8,* and described in *equation 1.18*.





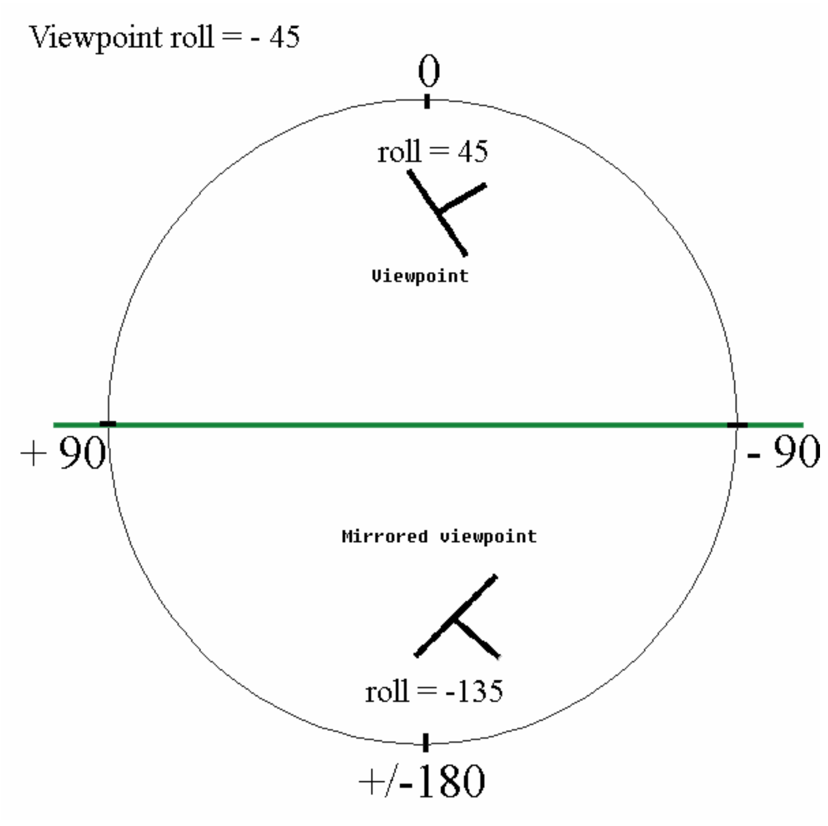


Figure 5.8 – Mirroring viewpoint roll

The mirrored viewpoint position is then set by the **Observer** function `setPosition(lon,lat,alt)`, and the mirrored orientation of the viewpoint is set with the functions `setPitch()` and `setRoll()`. To update scene graph transforms we use the **Observer** function `updateTransforms()`.

Step 2 – Remove objects behind the reflective water surface

Testing has shown that using clip planes to remove objects behind the reflective surfaces in GeoGFX are unnecessary as long as no objects exist underneath the water surfaces. GeoGFX has, as stated in *section 5.4.4*, no seafloor triangulation, and the parts of the triangulated terrain that have elevation lower than the mirrored surface inflicts no visible errors on the mirrored image. Subsequently we have decided to omit clip planes in this thesis.

Step 3 – Setting the camera matrices

After the mirrored transformation matrices has been set up we need to create camera matrices (modelview and projection matrices), which has the proper values for rendering the scene to texture and for using projective texturing to apply it on the surface. GeoGFX has a class **GngCamera** that handles these matrices. In order to get a reflection texture that is correct for projective texturing, we need to set the viewport of the camera to be the same size as the size of the reflection texture we are creating. The perspective of the mirrored camera, the field-of-view and the aspect ratio, has to be the same as the perspective of the camera we are viewing the final scene with.

Step 4, 5 and 6 – hiding water objects, sky cone and render scene

Before we draw the mirrored scene we need to hide the **GgWaterBody** surfaces since they are not part of the reflected environment. **GgWater** is a scene graph node which inherits the **GngRenderNode** properties and can be set invisible by handing a *true* Boolean variable to the `setInvisible(bool)` function. The same property applies to the sky dome node, which needs to be hid since its part of the global reflections, and not the local reflections. Instead of rendering the sky as part of the local reflection texture, we want to replace it with a global reflection texture. Since the sky dome is always the object furthest from camera, the only thing visible when hiding it becomes the background color. By setting the alpha value parameter of the `glClearColor()` to 0.0 the reflection texture gets parts where its alpha value is 0.0. These parts can be replaced in the final rendering stage with the global reflection texture. When the water surfaces and sky dome is hid, the mirrored scene is drawn with the function scene graph function *`redraw(GngCamera)`*, where the **GngCamera** parameter is the camera set up in *step 2*.

Step 7 – Capturing the mirrored scene as a texture

As of today there are 3 ways of capturing an OpenGL scene to a texture, one slow way and two faster ways. The slow way is by using `glPixelRead()`, this function reads the screen pixels one by one, and stores them in a byte array. The array can be used to create the local reflection textures. However, this method is far from optimal. A better way is to use the `glCopyTexImage2D()` function that captures the frame-buffer directly to texture. The **GngTexture2D** class in GeoGFX has implemented direct support for this functionality, which also automatically creates mipmaps of the generated texture. Another way of capturing OpenGL scenes as textures is to render the scene to an off-screen frame-buffer, a *Pbuffer*, and then transfer the content of the Pbuffer to texture. Pbuffer rendering is in theory supposed to be the fastest way of capturing scenes to texture, but there are some of drawbacks with this method:

- Switching between off-screen and on-screen frame-buffers are a demanding process, which may cause us to lose the advantage of the Pbuffer to texture transfer. The frame-rate difference between `glCopyTexImage2D()` and PBuffer rendering will be tested in *chapter 6*.

- The implementation of Pbuffers seems to be unstable; on some graphic cards we got errors with Pbuffer code that worked on other graphic cards. This can be a problem with the Pbuffer implementation in the graphic driver.

- Rendering Pbuffers to texture is in this thesis done by using the OpenGL extensions `WGL_ARB_pbuffer` and `WGL_ARB_render_texture`. (Use of `glCopyImage()` from the PBuffer is possible, but this is less efficient). WGL extensions are Windows platform specific meaning that the pbuffer-to-texture method will only work on computers with the Windows platform. Fortunately, a new extension, `EXT_frame_buffer_object`, which is supposed to handle Pbuffers on all platforms, has just been released. However, the code produced in this thesis might have to be changed in order to support this extension.

GeoGFX has implemented support for Pbuffers with the class **GngPBuffer,** however, this class needs to be modified in order to gain support for rendering the Pbuffer directly to texture.

Step 8 – Transferring the texture to GgWater

Since the water surfaces all have different elevation, the reflection scene has to be rendered to texture once for each surface. This is a very demanding task, which has to be optimized. This optimization process is described in *section 5.6,* and is an essential factor in achieving acceptable frame-rates. In short, the LOD management object **GgWater** described in *section 4.3* passes information to the **SceneWaterLocalReflections** object about which

GgWaterBody surface that needs a reflection texture, and **SceneWaterLocalReflections** passes the texture back to **GgWater**.

Step 9 – Attach reflection texture with Projective texturing

Projective texturing is a texture coordinate generation technique where the coordinates are projected on an object relative to a projection camera. The projection camera can be position virtually anywhere in a scene. In GeoGFX the technique is already used with object shadow casting. With planar reflections the use of projective texturing is a pretty straightforward and simplified case. The projection camera used is the same as the viewpoint camera, meaning that the viewpoint projection and modelview matrices can be reused with the projective texturing. Yann [5] describes a common way of creating the projected texture coordinates. The vertex position is multiplied by the projection and modelview matrices, transforming it into clip space (eye space). A clip space has a defined range of [-1 < c < +1] in the x and y direction and has to be remapped to the 0 to 1 range, since the transformed vertex position is used as texture lookup coordinates. The remapping can be done with a 4x4 remapping matrix *Mr* described in *equation 1.19*.

Equation 1.19 – Remapping matrix Mr

$$\begin{vmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Equation 1.20 describes the matrix (*Mprojtex*) that is multiplied with the vertex position in order to get the projective texture lookup coordinates.

Equation 1.20 `Mprojtex = Mr * Mp * Mv`

Mp is the projection matrix and **Mv** is the modelview matrix. The resulting conventional texture lookup coordinates *[s,t]* is in projective texturing divided by the homogenous coordinate *q* in order to get the projective texture access coordinates *[s/q, t/q]*.

5.4.2 Global reflections

Global reflections where described in *section 3.1.2* as the reflection of objects that are "infinitely" far away. The technique suggested is by using reflection cube map texturing. A reflection cube map is in short a cube where each side has a texture that together form a 360-degree view of an environment from the perspective of the reflecting surface. By handing the reflection vector described in *section 5.3.3* to the cube map, the proper reflection texture is automatically picked for display. This texture then replaces the parts of the local reflection texture where the alpha value equals 0.0 as explained in *section 5.4.1, step 4, 5 and 6.*

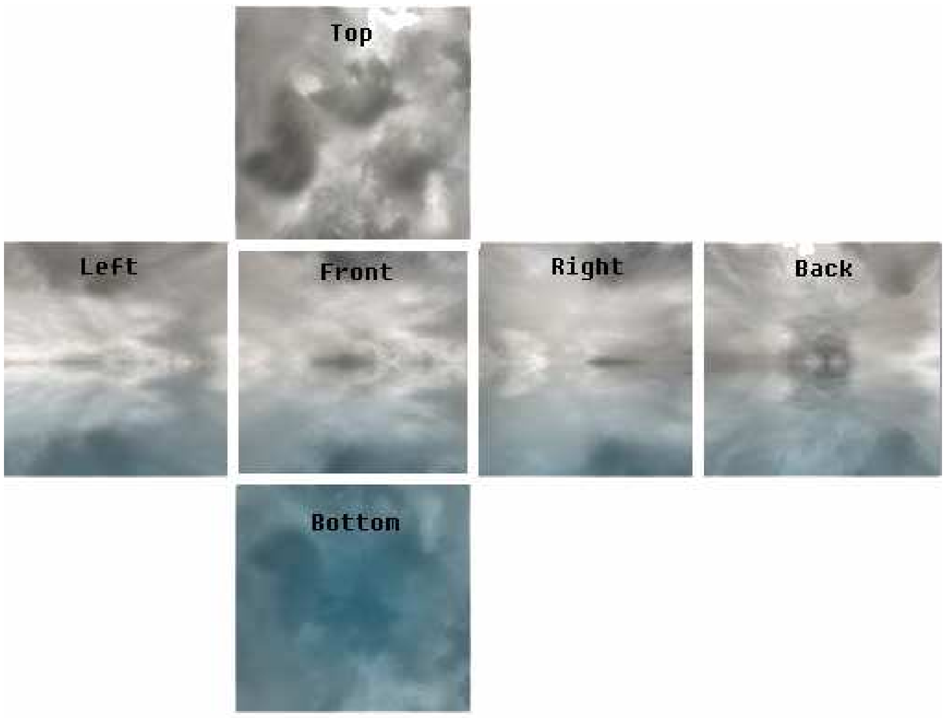


Figure 5.9 – Textures in a reflection cube map

Figure 5.9 illustrates the six textures that constitute a cube map, top, bottom, left, right, front and back. As we can see the lower part of this cube map represents the refractive part of the environment and the upper part represents the reflective part. Since we are only interested in the reflections we need to force the reflection vector to the upper hemisphere by remapping on the vector component that fetches the upper part of the cube map. In GeoGFX this is the z-component since the z-axis is pointing upwards. The remapping is described in *equation 1.21*.

Equation 1.21 $R.z = 0.5 * R.z + 0.5$

The most realistic way of creating cube map reflections is by using dynamic cube maps where all or some of the textures are updated regularly. This is a method that is very computationally demanding. A less demanding technique is to instead use a static cube map where the textures never are updated. The latter technique will not yield a correct visual display because of the continuous change in the global reflections, but is a way to gain better frame-rates. The optimization of global reflections is described in *section 5.6.* The following section proposes a technique to acquire global reflection textures in GeoGFX, for use in a dynamic reflection cube map. The acquisition and creation of global reflection cube maps is described in *code 3* and is handled by the **SceneWaterGlobalReflections** class.

Code 3 – Acquiring reflection cube map textures in SceneWaterGlobalReflections

```
Step 1      Set viewpoint position to the water surface center
Step 2      Set the camera matrices
Step 3      Hide all local objects
Step 4      Set the camera direction in on of the 6 major axis directions
Step 5      Render the global reflection scene
Step 6      Capture the rendered scene to one of the cube map textures
Step 7      Go to step 4 and repeat step 5 and 6 until all 6 directions has
            been captured to a cube map texture
Step 8      Hand reflection cube map texture to GgWater
```

Code 3 has to be repeated once for each **GgWaterBody** surface. **GgWater** is responsible for handing **SceneWaterGlobalReflections** information about which surface to render global reflections for.

Step 1 – Set viewpoint position to water surface center

The **Observer** function `setPosition()` is used to set the viewpoint at the water surface center. The center of the surface is pre-calculated in **GgWater** and handed to **SceneWaterGlobalReflections**.

The center coordinate x of a **GgWaterBody** surface is calculated by adding the minimum and maximum latitude of the surface corners, and divide by 2. The center coordinate y is calculated by adding the minimum and maximum longitude and divide by 2. The z coordinate is just set to the elevation of the surface since it is the same for the entire surface as described in *section 4.1.2*. This is another example of why choosing a simple representation of the surface samples can be more practical than a complex representation.

Step 2 – set the camera matrices

Since the cube map is a 360 degree representation of the environment we need to capture 6 scenes to textures that are displayed with a 90 degree field-of-view. The scenes also have to be rendered in a resolution that is equal to the wanted texture size. Setting up a **GngCamera** with the correct modelview and projection matrices does this. The scene resolution is set by using the function `setViewport(x,y,w,h)` where *x,y* is the upper left corner and *w,h* is the resolution of the camera, and the 90 degree field-of-view is set by the function `setFOV(fov)`.

Step 3 – hide all local objects

In order to only capture the global objects of the scene to texture we need to hide all local objects before rendering the scene. Since all local objects are attached to the *Geocentric transform* scene graph node described in *section 2.4.2*, the only thing we need to do is hand a *true* Boolean variable to the transforms function `setInvisible(bool)`. This causes all the attached nodes to become invisible. When now rendering the scene the only visible objects are the one attached to the *sky transform* which are the global reflection objects.

Step 4 – set camera direction

The viewpoint of each the 6 different scenes rendered around the surface center has to have direction along the one of the major axis. In GeoGFX we control this direction with the **Observer** functions setYaw() and `setPitch()`. *Figure 5.10* illustrates the relation between the x, y and z axes and yaw and pitch.

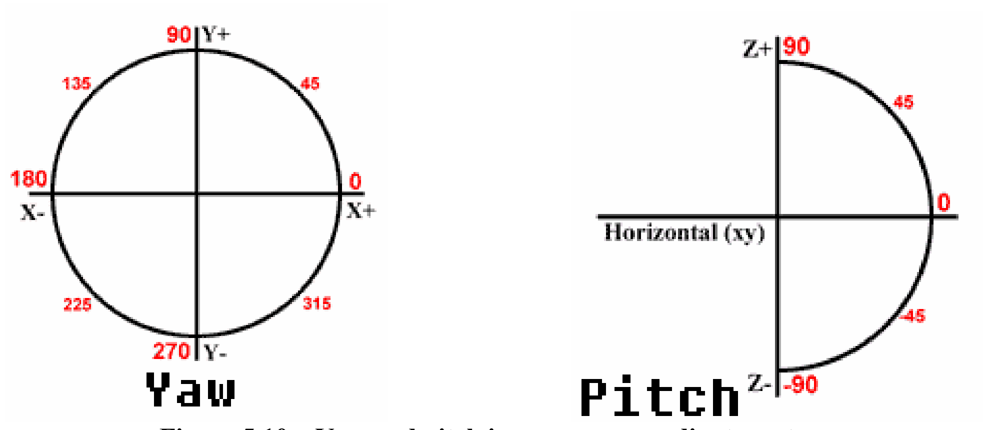


Figure 5.10 – Yaw and pitch in an x, y, z coordinate system

Step 5 – render the global reflection scene

The scene is rendered from the given viewpoint with the camera set up in *step 2* as parameter to the `redraw()` function.

Step 6 and 7 – capture scenes to cube map textures

Rendering the scene to a Pbuffer is the only way to create dynamic cube maps with the current OpenGL implementation. The `glCopyTexImage2D()` function described in *5.5.1* does not support copying of the frame buffer directly to a cube map. This also means that in order to create dynamic cube maps we get the drawbacks with Pbuffers as explained in *5.5.1*.

The way a dynamic cube map is created is that a Pbuffer that renders directly to a texture is activated and step 4 and 5 is repeated 6 times, once for each axis direction. Before each rendering pass the Pbuffer function `wglSetPbufferAttribARB()` with the cube map side as parameter is invoked. When all 6 sides are rendered the Pbuffer is deactivated and the cube map is created.

The **GngPBuffer** class in GeoGFX needs modification to support Pbuffer to cube map rendering.

Step 8 – transferring the cube map to GgWater

As with local reflections **GgWater** is the class that controls which **GgWaterBody** surface needs the global reflection cube map from **SceneWaterGlobalReflections**. **GgWater** hands over the necessary information about the surface, **SceneWaterGlobalReflections** passes back the cube map.

The cube map created with this technique is oriented in a local coordinate system since the **Observer** functions `setYaw()` and `setPitch()` rotates the view camera in local coordinates. In order to gain a visual correct result the reflection vector used for lookups also has to be oriented in local coordinates. This is the reason why we chose to orient all vectors from *section 5.3* into local coordinates.

5.4.3 Sunlight reflections

In *section 3.2.3* it is suggested to render sunlight reflections with either Blinn or Phong lighting instead of encoding the sunlight into to global reflection textures. This is because textures have to be mapped to the range 0 to 1, and the sun might seem bleak and washed out. The Blinn [3] and Phong [3] equations are described in *equation 1.22* and *1.23*.

Equation 1.22 – Blinn specular lighting
$$\texttt{Sun\_specular = sun\_color * sun\_strength * max(0, H} \cdot \texttt{N)}^{\text{shininess}}$$

Equation 1.23 – Phong specular lighting
$$\texttt{Sun\_specular = sun\_color * sun\_strength * max(0, L} \cdot \texttt{R)}^{\text{shininess}}$$

These equations are the specular contribution of the single point lighting equations described in *equation 1.6* and *1.7*. **N** is the plane's normal, either per-vertex or per-pixel. In our case, we are using normal maps to compute the surface wave-patterns, so the normals are provided per-pixel. As discussed in *section 3.3* the specular contribution can computed with different per-pixel lighting techniques such as vertex and pixel shaders or Register Combiners/Dot3ARB with normalization cube maps. However, we are using shaders since they are a faster and more flexible than the other methods. Blinn specular lighting requires less computation that Phong since **H** is the half angle vector defined as **L+V** where **L** is the light vector defined in *section 5.3.2* and **V** is the view vector described in *section 5.3.1*. However, we are using the reflection vector **R** described in *equation 1.13* for global cube map reflection lookups, so it is more reasonable to reuse this vector to compute Phong sunlight contribution, rather than calculating both the **R** and **H** vector. As displayed in *figure 5.11* Blinn yields a longer, skinnier specular contribution then Phong, which yields a more bulb-shaped specular highlight when the sun is low on the horizon.

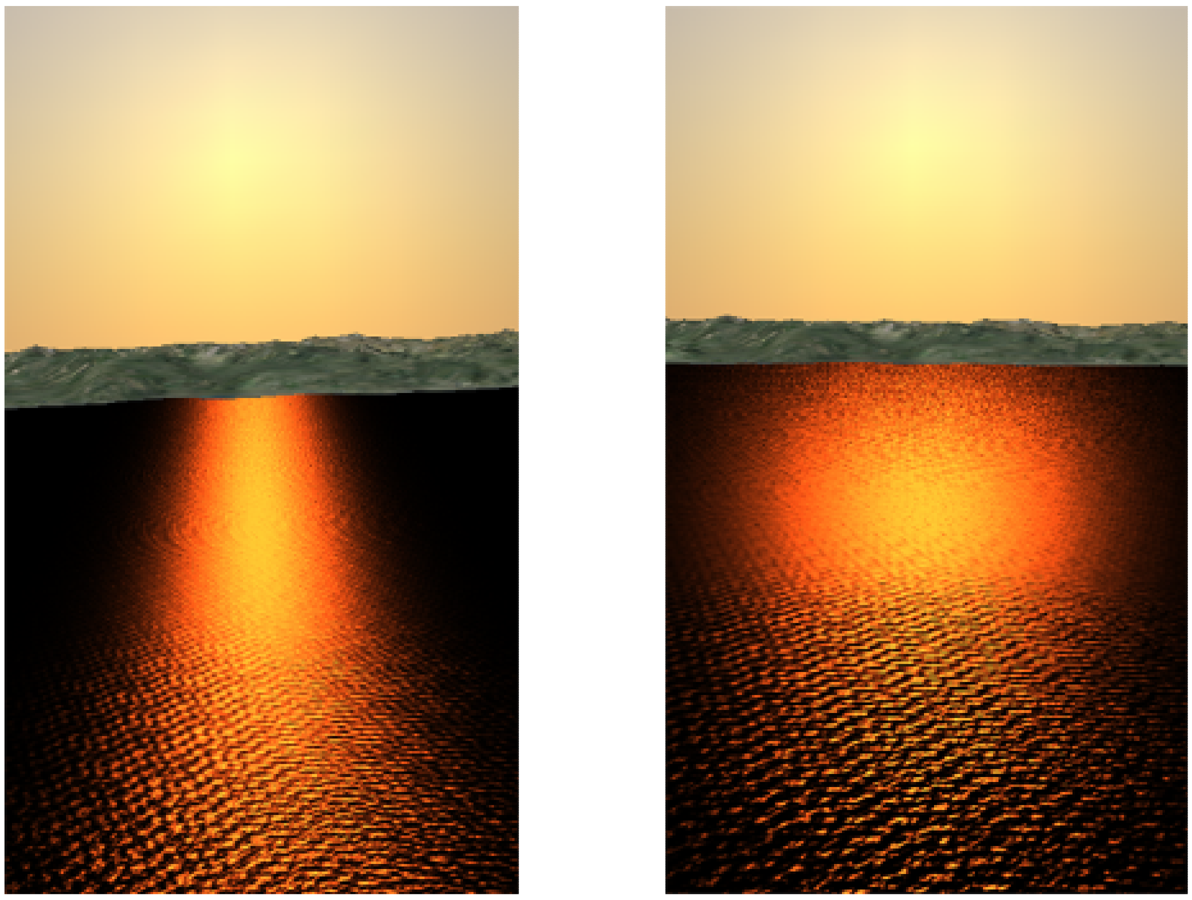


Figure 5.11 – Screenshot of Blinn (left) and Phong (right) sunlight contribution

The light model does not take into account the light refractions of clouds, haze and other environmental conditions in GeoGFX. Using different values for the variables **sun_color**, **sun_strength** and **shininess** may approximate these factors.

5.4.4 Refractions

Because of the limited access on actual data about ground conditions underneath water surfaces, GeoGFX does not have rendering of these areas built in to the engine. This means that removing the water regions described in *section 2.7* leaves nothing but holes in the triangulated surface. If we want to use the refraction model explained in *section 3.1.2*, we have to approximate water floors by rendering a textured plane a given distance underneath the water areas. These planes can be simple polygons with the same size as water surfaces, since they only are visible, according to the Fresnel reflection model, when looking close to straight down at the water surface. This is illustrated in *figure 5.12*.

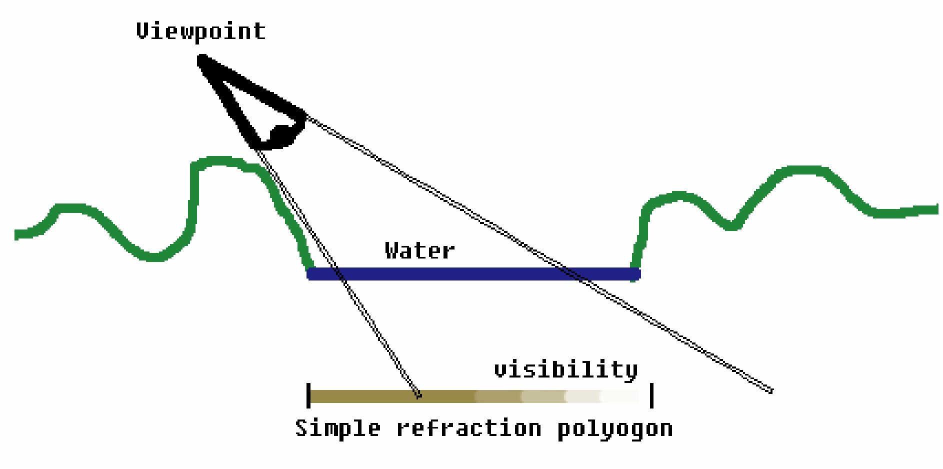


Figure 5.12 – Rendering simple refractions

In order to get the refractions to look right [2] we need to make another rendering pass, removing everything above the surface, capture the scene to a texture and apply the texture with projective texturing as with local reflections. Since creating refractions this way are just approximations, we have instead chosen to set them to a single color assuming that most lakes are so deep that we do not see the bottom. This saves a lot of computation since we do not need the additional rendering pass and the projective texturing lookup.

5.4.5 Combining reflections and refractions

The final visual result of the reflective surfaces is obtained by interpolating between reflections and refractions using the Fresnel value, and between the local and global reflections using the alpha value as suggested in [5]. The operation is done per-pixel and the final value is described in *equation 1.24*.

Equation 1.24 – Computing the final pixel value for reflective surfaces

```
Result = (1-fresnel)*refraction + fresnel*(refl_local.rgb*refl_local.a) + (refl_local.a)*(refl_global + refl_sun))
```

5.5 Rendering surfaces

5.5.1 GgWater

GgWater is the class responsible for drawing the water surfaces, the **GgWaterBody** objects. The **GgWaterBody** objects are drawn with `render()` in **GgWater**'s `redraw()` function. `Redraw()` also invokes the GngCamera function `frustumTest()` to check whether the surfaces are inside the frustum of the view camera. Only surfaces inside the frustum are drawn on screen.

Another "trick" performed **GgWater** is making the water surfaces look like complex polygons even if they are just rectangular, simple polygons as described in *section 4.1.2*. This is done making the triangulated surfaces a little bigger than the holes which come into being when the water samples are removed from the terrain triangulation, as described in *section 4.1.1*. Then, before the **GgWaterBody** objects are drawn, the z-buffer is disabled by invoking `glDepthMask()` with the parameter `GL_FALSE` and the objects are drawn on screen without writing values to the depth buffer. When all the visible water surfaces are drawn, the z-buffer is enabled and the terrain triangulation is drawn on top of water surfaces. Since the z-buffer contains no values when the terrain triangulation is rendered, the water surface edges will intersect with, and stay underneath the terrain without creating any glitches or artifacts (so-called z-buffer fighting). The water surfaces now look like they have a complex shape, equal to the hole in the terrain triangulation. *Figure 5.13* illustrates how rendering the water surfaces with depth buffers turned off are used to "fake" the complex polygonal shape of the water boundaries.

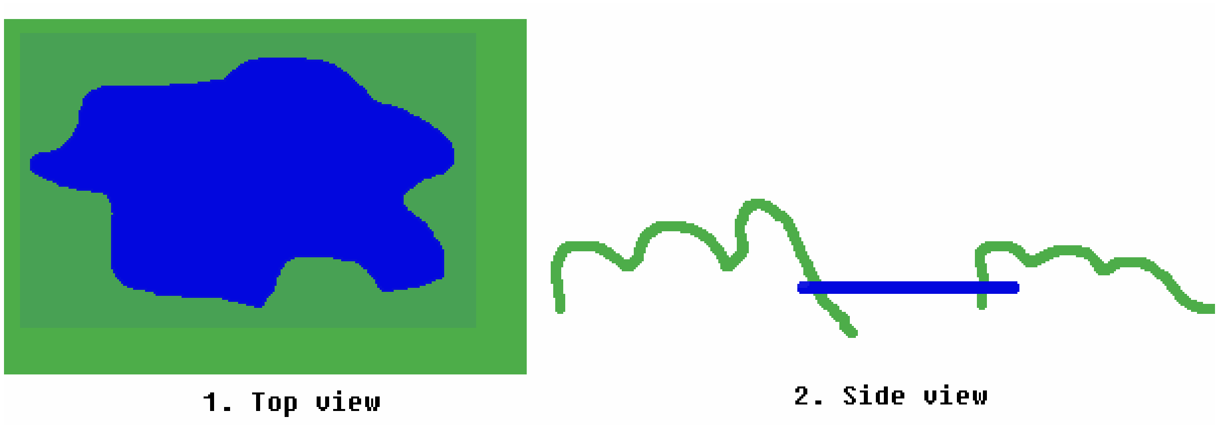


Figure 5.13 – Faking complex polygons with depth buffers. 1. show how the water surface looks from a top view, with the contours of the whole polygon. 2. show how the surface look from a side view, cutting through the terrain triangulation.

The drawback with using this method to obtain complex shapes on the water surfaces is that finding the points on the surface that intersects with the terrain is very difficult. Thus, adding extra features on the water such as foam where the waves hit the shoreline are a quite challenging and are therefore left out of the thesis due to the workload.

Code 4 explains the process which is done in the `redraw()` function.

Code 4 – The GgWater redraw() function

```
Step 1      Disable depth buffers
Step 2      For all GgWaterBody objects:
Step 3            Compute surface frustum sphere
Step 4            If sphere is inside camera frustum:
Step 5                  Draw GgWaterBody objects
Step 6      Enable depth buffers
```

5.5.2 GgWaterBody

GgWaterBody is the class that actually draws the water surface vertices on screen. Objects of this class know nothing about other **GgWaterBody** objects and is just responsible for drawing its own vertices on screen when the `render()` function is invoked. The `render()` function translates the starting point of the drawing operation to the water surface origin relative to the terrain origin, computes the view vector, applies the different OpenGL states, enables the shaders and draws the surface vertices. The states applied depend on Level-Of-Detail parameters passed from **GgWater**. *Code 5* explains the process of the `render()` function.

Code 5 – The GgWaterBody render() function

```
Step 1      Translate drawing start point to surface origin relative to
            terrain triangulation origin
Step 2      Calculate view vector
Step 3      Update timer variable used to animate waves
Step 4      Enable shaders
Step 5      Pass uniforms and attributes to shader (view-vector, sun-
            vector, timer variable, surface colors, etc.)
Step 6      Set states (reflection/no reflection, etc)
Step 7      Draw vertices, set client normal and texture arrays
Step 8      Remove states
Step 9      Disable shaders
```

5.6 Optimizing

As stated in *chapter 3* rendering fully realistic water surfaces is an extremely computationally demanding task. The techniques described in the previous sections are formerly developed methods for simplifying the task, in order to gain surfaces with realistic looking water optics, which can be rendered real-time. In addition to implementing these techniques in GeoGFX, it is necessary to optimize them in order to maintain acceptable frame rates in a real-time rendering sequence. This chapter proposes methods for further optimization of the rendering process. *Chapter 6* presents a comparison between a collection of the suggested methods based on visual quality and frame-rate performance. The result of this comparison is the foundation for selecting the Level-Of-Detail parameters in the **texture LOD management** component described in *figure 1.1*.

Optimizing wave generation

As described in *section 5.2* the wave-pattern used in this water model are Perlin Noise normal map textures that are tiled on the surfaces. In order to get a pattern that does not look to repetitive we should use as large textures as possible. 3D Perlin Noise textures consist of several layers of images at different octaves. This means that a 3D texture width same height and width as a 2D texture, consumes texture memory equal to the 2D texture amount times its number of layers. Thus, using 3D textures limits the possible width and height of the texture considerably. *Chapter 6* compares the visual result of using large 2D textures to the visual result of using smaller, animated 3D textures.

Optimizing local reflections

Creating local reflections is the most computationally expensive operation of the water-render process. Even with the mirroring technique described in *sections 3.1.2* and *5.4.1*, this task is the one that steals the most frame-rate because the triangulated terrain has to be rendered an additional time per visible surface, per frame. In order to maintain an acceptable frame-rate in GeoGFX we have to cut back on some of the visual details on the local reflections.

The optimizing technique that without a doubt buys the most frame-rate is by only showing local reflections on the surface closest to the viewpoint. We can accomplish this by measuring and comparing the surface's distance from the camera, and render only the local reflections relative to the position of the closest surface. This ensures that the triangulated terrain is only rendered one additional time per frame and not one time per visible surface. As long as the reflective surfaces do not lie to close to each other the viewer will not perceive that only the closest surface that shows local reflections.

Because of the high complexity of the terrain triangulation in GeoGFX, utilization of this optimization technique is necessary to obtain acceptable frame-rates. Consequently, every test done in *chapter 6* is executed with this method as basis.

An implementation of this technique has been created through the **GgWater** function `closest()`, which returns **GgWaterBody** object closest to the camera. The **SceneWaterLocalReflections** class uses this function in order to provide only the closest **GgWaterBody** object with a local reflection texture. *Code 6* explains the process of acquiring closest **GgWaterBody** object with the `closest()` function.

Code 6 – Get closest GgWaterBody object in GeoGFX

```
Step 1      Send camera position as MtkPoint3f variable to GgWaterBody
Step 2      Get distance between camera position and all vertex positions
            with the MtkPoint3f dist() function
Step 3      Compare all distances from step 2 and return the shortest
            distance between camera and vertices
Step 4      Go to step 1 and repeat step 1 to 3 for all GgWaterBody objects
Step 5      Compare all distances returned from step 3
Step 6      Return the GgWaterBody object with the shortest distance
            returned in step 3
```

Since the alpha value of local reflection texture is used to replace the parts of the texture where we want to display global reflections, the local reflection texture on the surface that is not closest to the camera has to be replaced with a texture that has alpha value equal to 0 on all texels.

However, just rendering local reflections for the closest surface may not be enough to maintain an acceptable frame-rate. In addition we have to look at possible methods for optimizing the actual rendering process. There are several optimizing methods that may increase the frame-rate, but most of them lower the detail level of the reflections, and perhaps also the realism of the scene. *Chapter 6* compares the different optimization methods on frame-rate gain, versus loss in visual appearance.

One possible way of optimizing the local reflections is by creating small reflection textures. When we are generating reflections the captured scene has to be rendered with a viewport size identical to the size of the reflection texture. This implies that high resolution textures with size 256x256, 512x512 or greater, require rendering of high-resolution scenes which is very computationally demanding. Instead we could create small reflection textures with size 64x64 or 128x128. This way rendering the scenes would be a lot less demanding and the frame-rate may be increased. The downside with using low-resolution textures is that the reflection becomes very pixelated, especially when viewing the surface from a distance since the texture then has to be stretched to cover a larger area. Howeveer, the water rippling effect added to the texture (see *section 3.1.2*) may hide most of this pixelation.

Two GeoGFX functions for optimizing the process of rendering the terrain triangulation are `setPixelScale(scale)` and `setTexelScale(scale)` from the **GgTerrain** class. The `setPixelScale()` functions uses the *scale* parameter to indicate the size of the projected pixel in the terrain, given the screen resolution and camera parameter. A large pixel covers more error in the terrain, so a large *scale* value means higher error tolerance and less detail, while a small *scale* value means lower error tolerance and more detail. Setting the *scale* parameter to a large value causes the terrain triangulation to “pop” which is an undesired effect where polygons suddenly appear on the screen. In order to keep the “pops” down to a minimum the *scale* parameter should never be greater than 2. However, for use with our reflection textures a higher number of “pops” on the triangulation could be acceptable to increase the frame-rate, especially since they may not be as obvious when applying the rippling effect on the textures.

The `setTexelScale()` function uses the *scale* parameter to adjust the detail level of textures covering the triangulated areas. The parameter has to be in the range [0, 1.0], with 1.0 being the value where the textures have the highest possible detail level, and 0 being the value with the lowest texture detail level. Adjusting *scale* to values lower than 1.0 is not recommended for the main triangulation because the realism of scene diminishes rather quickly, but for the local reflections lowering this parameter may increase the frame-rate without very obvious loss in visual quality.

Another possible optimization technique is adjusting the camera clip planes. By setting the far clip plane of the local reflection camera to a much shorter distance than the main view camera we limit the size of the triangulation to a much smaller area. This decreases the range on where we can see the reflections, and has to be adjusted so that it looks realistic.

An additional method for possible optimization of local reflection rendering is with use of Pbuffers instead of the built in function `glCopyTexImage2d()`. This method is described in *section 5.4* and is supposed to be the fastest way of capturing screen buffers to texture. However, switching between the standard screen buffer and Pbuffers are a demanding process, which may cancel out the screen capture frame-rate gain. Whether it actually yields any better frame-rate or not is tested in *chapter 6*.

Optimizing global reflections

Section 5.4.2 describes a technique for creating global reflections with dynamic cube maps. Generating these cube maps is an extremely computationally demanding task, since they require six sky render passes per visible surface, per frame. Rendering just the sky dome for global reflections may be achievable because it is not a high-resolution triangulation. But the contribution of clouds in the scene may make it virtually impossible to maintain an acceptable frame-rate while updating the cube map.

One possible way of optimizing the generation of dynamic cube maps is by lowering the texture size. As with local reflections, the viewport of the camera rendering the global reflection scene has to be the same size as the size of the texture. This implies that a high-resolution texture has to be rendered with a large camera viewport, which is a more costly process than rendering with a small viewport. By downsizing the viewport and texture resolution we might be able to gain some frame-rate, but it also lowers the visual quality of the scene since the textures has to be stretched over a larger area.

Another possible optimization technique, in addition to the one previously mentioned, is to only update the cube map on the surface closest to the view camera. The other surfaces further from the camera uses their old updates for display, while the closest surface’s cube map is updated every frame. This way only six addition passes is required per frame no matter how many surfaces are visible. To acquire the closest surface we can reuse the `closest()` function described in *code 4*.

However, rendering six additional passes per frame could still be a too demanding task for most computers. In order to maintain an acceptable frame-rate we may have to use some supplementary optimization techniques to the generation of global reflections. First, we do not have to update the bottom of the cube since the reflection vector used for lookup never will come from underneath the surface; this side needs only to be a static texture. Thus we only need five additional rendering passes per frame. Second, the cube maps may not need to be updated every frame. Since the appearance and disappearance of clouds is relatively slow process, and the sky does not change color dynamically in GeoGFX, the cube maps might not need updates more frequently than every few seconds or less without being noticeable to the viewer.

If dynamic cube maps are too heavy for the computer to handle, we could instead use static cube maps. Either generated at the first frame with Pbuffer rendering, or simply with use of pre-generated cloud textures, fitted to match the sky color and an approximated cloud pattern in GeoGFX. In order to save some texture memory, these textures can be shared among, and applied to all surfaces. This approximation will be most apparent with a calm surface since the global reflections will be wrong, but with a rough wave pattern the reflections will not very visible, and may be used to increase the frame-rate without to much loss in visual quality.

5.7 The mipmapping problem

As we explained in *section 3.2*, mipmapping [10] is necessary to avoid texture artifacts such as aliasing on objects. Aliasing occurs when trying to apply textures with high resolution to an object with lower resolution than the texture. OpenGL automatically downsizes the resolution on objects relative to their distance from the camera, which means that aliasing will occur when moving away from an object if the attached texture has high resolution. If mipmapping is enabled OpenGL instead applies a pre-generated, downsized texture to the object with equal or less resolution than the object, and aliasing is avoided. With regular texturing, built-in OpenGL mipmapping functions can be used to create the downsized textures, but normal maps has to be renormalized when downsized and therefore needs a special downsizing algorithm explained in [3].

However, with our water simulation technique mipmapping creates another problem: Filtering the normal map downsizes the perturbed normals since the smallest texture with resolution 1x1 only has a single normal pointing straight upward. This means the wave pattern disappears from the surface in the distance, making the surface looking perfectly calm. In real life you do not see the waves after a certain distance, but the water still does not look perfectly calm. *Figure 5.14* displays two images, one with mipmapping and one without mipmapping. Notice the odd pattern in the waves on the left image. This pattern is called a “Moire pattern”. The problem is more apparent when actually moving the camera through the application.

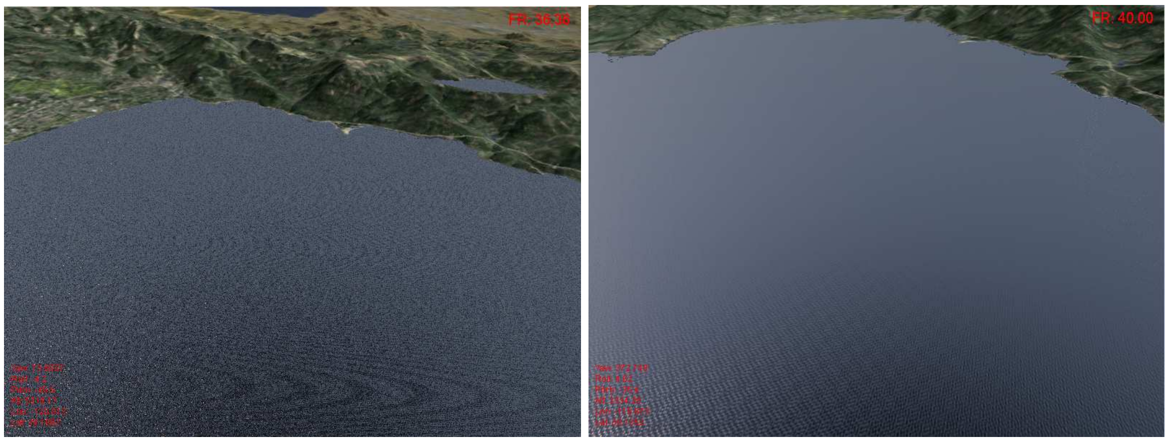


Figure 5.14 – Left: Without mipmapping. Right: With mipmapping

To attempt to solve this problem we have tried different solutions. One approach was to stretch the wave pattern textures over larger areas. This surely decreases the need for mipmapping, but the result is that the waves become too wide relative to the other objects as *figure 5.15* shows. Notice that the waves are a lot bigger than the sailplane located in the middle of the picture.

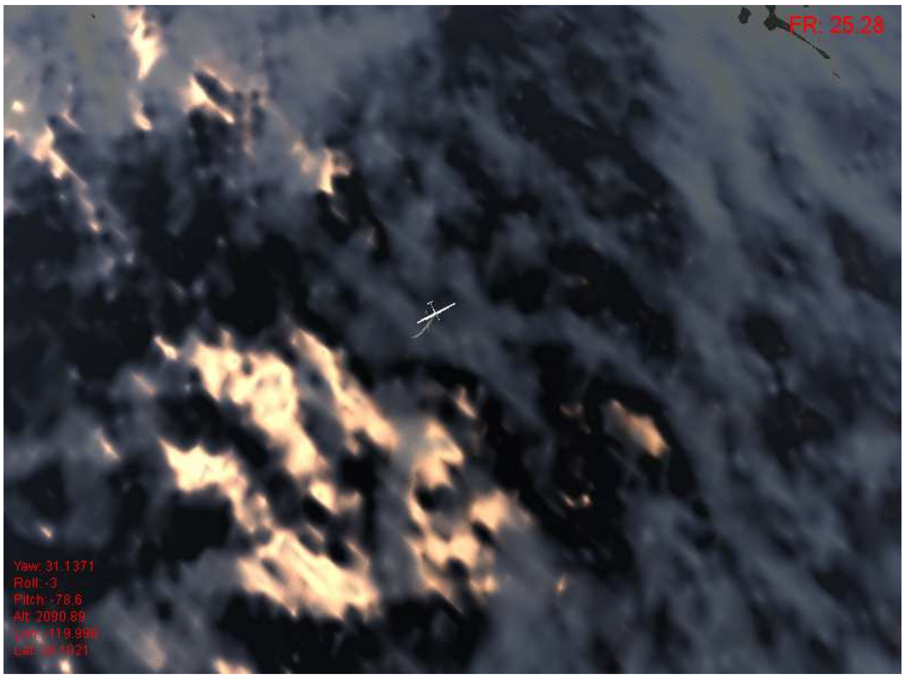


Figure 5.15 – To wide waves

Another approach was to decrease the height on the waves. This also decreases the need for mipmapping, but the waves become less apparent as shown in *figure 5.16*. On large water areas the waves has to be very small in order to avoid aliasing, which is not very ideal.

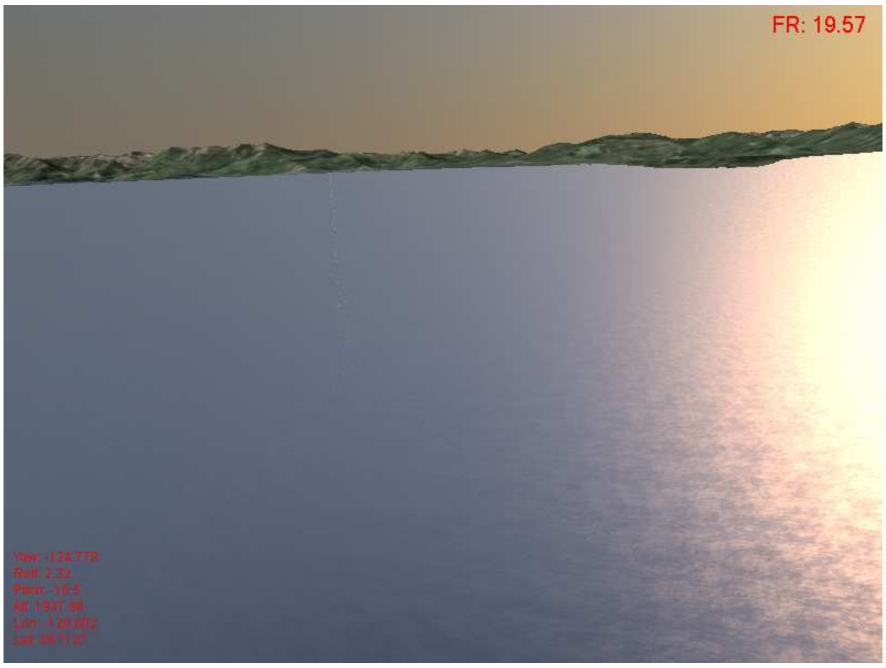


Figure 5.16 – Small waves

The last approach we tried was to control the highest level of the mipmapping. By handing the parameter `GL_TEXTURE_MAX_LEVEL` together with highest level texture to the OpenGL function `glTexParameteri()`, the mipmapping functionality is forced to never use any textures with lower resolution then the highest level texture. Thus, the wave pattern will never be perfectly calm in the distance (if the full size wave-pattern texture is not totally flat). This technique allows us to use higher waves, tiled closer together, but we still need to be careful with the wave size in order to avoid aliasing.

6 Results

In this chapter we present a comparison of the optimization techniques described in *section 5.6*. These options are tested in a fly-through sequence and a still-photo sequence. Based on frame-rate and visual quality they are compared in order to conclude which optimization techniques are most suited to use with the water surface rendering in GeoGFX.

The fly-through sequence is animated with linear interpolation between key-frames. These key-frames are the centers of all water surfaces in our test-data, a total of 53 surfaces. Using the centers as key-frames ensures that all surfaces are visited and included in the visual comparison. During the animation the highest, lowest and average frame-rate is recorded. Each sequence is run three times and the average of the frame-rates is computed and used in the comparison. Before the animation is started we move the viewpoint to the first frame in the sequence, ensuring that the recorded frame-rates are representative for the animated path, and the frame-rate is stabilized. One fly-through sequence lasts approximately 2.5 minutes.

The still-photo sequence is a frame-rate and visual test executed on five selected locations in the test-data. The viewpoint is moved to the different locations, and the upper- and lower-bound frame-rate is recorded. This test is run 3 times with each option, and the average frame-rate at all location is computed and used in the comparison. Screenshots of the five locations are shown in *chapter 8, appendix B*.

Before testing begins both the fly-through sequence and the still photo-sequence are run once to make sure the GeoGFX engine has stabilized, and that no texture or vertex loading-problem occurs.

The test is run on a computer with the following specifications:

Processor: Pentium 3 – 866 MHz
RAM: 512 Mb
Hard disk: 30 Gigabyte, 7200 RPM
Video card: Asus V9999, GeForce 6800 GT, 128 Mb DDR

Notice the relatively weak CPU which probably causes some overhead that steals frame-rate. With a better CPU the following tests would most likely yield greater frame-rate differences.

The computer is rebooted before each test in order to run all tests on the same premises.

GeoGFX is run with 800x600 pixel resolution during the tests. The pixel scale of the terrain triangulation is set to 2.0, and the texel scale is set to 1.0. Pixel scale and texel scale is explained in *section* 5.6. Local reflections are only visible on the surface closest to the camera, and dynamic global reflections are only updated at the closest surface.

Because of wide variety in composition of the optimization options, the specifications of the tests have to be limited to a certain number of important factors. We have selected the most significant factors to be:

Global reflections:
- Dynamic or static cube maps.
- Dynamic cube map update rate.

- Texture resolution.

Local reflections:
- Texture resolution.
- Terrain pixel scale
- Pbuffer vs. `glCopyTexImage2D()` capture of local reflections to texture

Wave pattern:
- Animated or static wave pattern
- Texture resolution

Other factors that are not varied in the test are:

Local reflections:
- Terrain texel scale is set to 1.0
- Far clip-plane is 5 km

Global reflections:
- Only the sky dome is rendered, no sky cloud pattern

Refraction:
- Refractions are set to a constant color

Sunlight reflections:
- Surfaces are lit with the Phong specular shading equation
- The shininess component is set to 32.

Wave pattern:
- Mipmapping is controlled, with highest level of 8x8x(thick/(width/8))

Fresnel factor:
- Fresnel factor is approximated using a 1D texture lookup.

Test 1, 2 and 3 compares frame-rate versus texture resolution. Test 4, 5, 6, 7, 8, 9 and 10 uses the same texture resolution as *test 3* and the frame-rate is compared relative to this test. Test 10 omits local reflections and is the only test that does not include all the water features: global and local reflections, and wave pattern. This is done to test the impact on the frame-rate when creating local reflections.

All ten tests have screenshots to display the visual quality, but only test 1, 2, 3, *5* and 10 shows any visible differences.

All frame-rates are measured in frames per second (fps).

Test 1 – High texture resolution

Specifications		
Global reflections:	*Cube map type:*	Dynamic
	Updated:	Each frame
	Texture resolution:	256x256
Local reflections:	*Texture resolution:*	1024x1024
	Terrain pixel scale:	1.0
	Rendering type:	PBuffer
Wave pattern:	*Type:*	3D animated
	Texture resolution:	256x256x128

Fly-through frame-rates

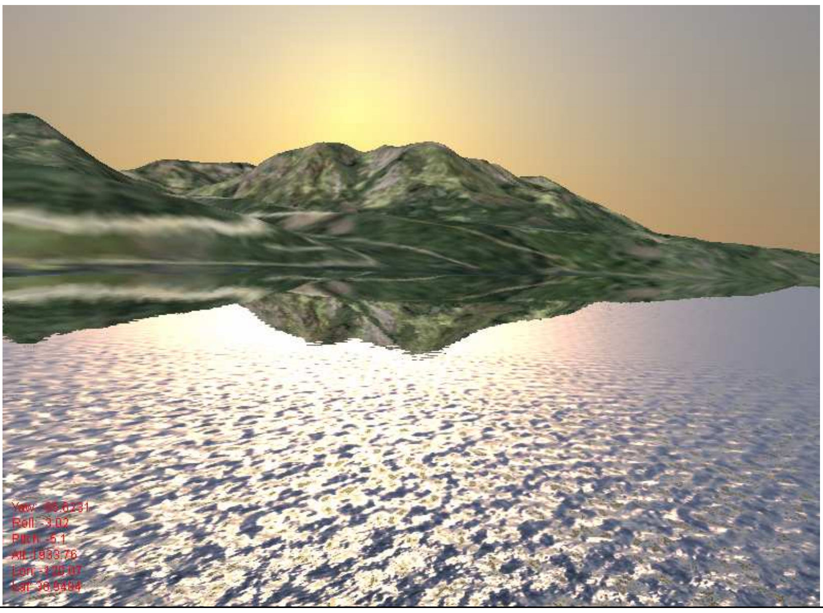
	Min	Max	Average
1. Run	16.9	42.5	25.5
2. Run	17.7	43.4	25.5
3. Run	16.8	44.4	25.5
Average	**17.1**	**43.4**	**25.5**

Still-photo frame-rates

	Location 1	Location 2	Location 3	Location 4	Location 5
1. Run	(18.8 - 33.3)	(24.9 - 32.8)	(18.7 - 20.0)	(19.8 - 20.6)	(25.3 - 32.7)
2. Run	(19.8 - 20.2)	(24.9 - 33.3)	(18.0 - 20.0)	(19.8 - 20.6)	(25.3 - 31.7)
3. Run	(19.8 - 20.6)	(24.7 - 31.3)	(18.8 - 20.4)	(17.7 - 20.6)	(25.0 - 30.7)
Average	**19.5 – 24.7**	**24.8 – 32.5**	**18.5 – 20.1**	**19.1 – 20.6**	**25.2 – 31.7**

Visual quality

- The high resolution on the local reflection textures make the reflections look good even when viewed from a distance.
- The high resolution on wave texture the makes the wave pattern look relatively random
- Since the terrain pixel scale is set to 1.0 there is no visible popping on the reflection texture.



Test 2 – Low texture resolution

Specifications		
Global reflections:	*Cube map type:*	Dynamic
	Updated:	Each frame
	Texture resolution:	64x64
Local reflections:	*Texture resolution:*	128x128
	Terrain pixel scale:	1.0
	Rendering type:	PBuffer
Wave pattern:	*Type:*	3D animated
	Texture resolution:	128x128x32

Fly-through frame-rates

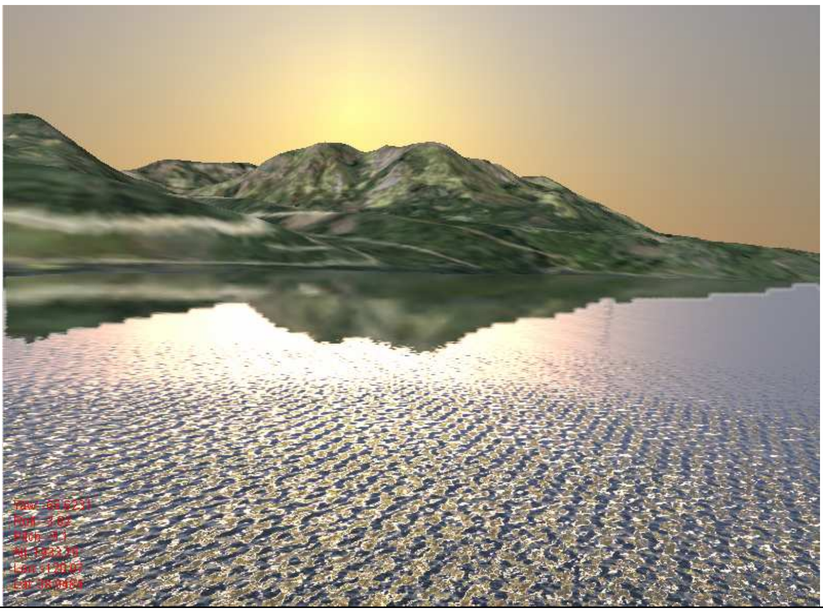
	Min	Max	Average
1. Run	16.4	44.3	26.5
2. Run	16.0	43.5	27.1
3. Run	16.9	45.4	26.9
Average	**16.4**	**44.4**	**26.8**

Still-photo frame-rates

	Location 1	Location 2	Location 3	Location 4	Location 5
1. Run	(14.9 - 43.4)	(25.3 - 33.3)	(19.0 - 20.4)	(22.7 - 27.0)	(38.5 - 45.5)
2. Run	(19.8 - 20.6)	(25.0 - 31.2)	(19.2 - 20.6)	(24.1 - 28.9)	(39.9 - 43.5)
3. Run	(19.8 - 20.6)	(25.0 - 32.7)	(19.4 - 20.6)	(24.0 - 28.1)	(40.0 - 42.6)
Average	**18.2 – 28.2**	**25.1 – 32.4**	**19.2 – 20.5**	**23.6 – 28.0**	**39.5 – 43.9**

Visual quality

- The low resolution on the local reflection textures makes the reflections look *very* pixelated, especially on a distance when the texture has to cover more of the surface.
- The low resolution on the wave texture reveals that the wave-pattern is repetitive, and looks unrealistic.
- The low resolution on the global reflections does not affect the visual quality, at least when there are no clouds in the sky.



Test 3 – Medium texture resolution

Specifications		
Global reflections:	*Cube map type:*	Dynamic
	Updated:	Each frame
	Texture resolution:	128x128
Local reflections:	*Texture resolution:*	512x512
	Terrain pixel scale:	1.0
	Rendering type:	PBuffer
Wave pattern:	*Type:*	3D animated
	Texture resolution:	256x256x64

Fly-through frame-rates

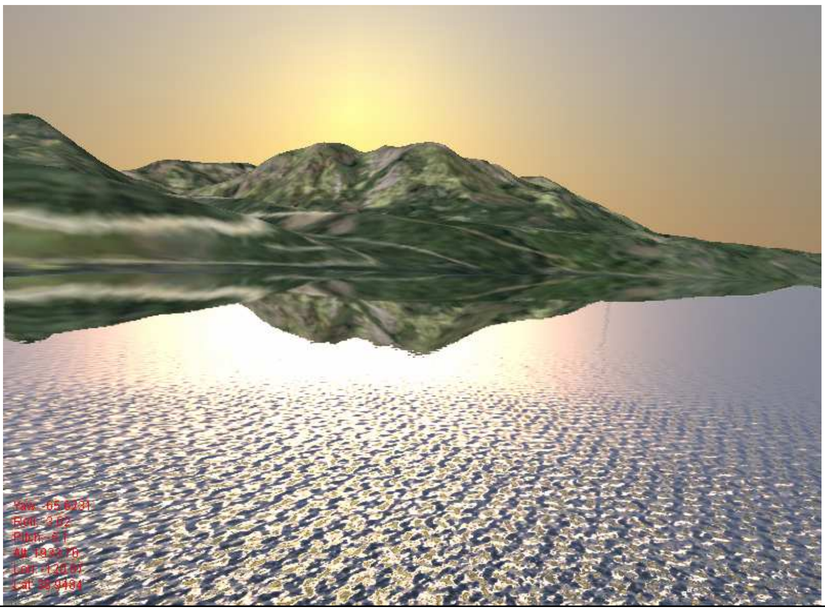
	Min	Max	Average
1. Run	15.8	42.5	25.5
2. Run	15.8	45.5	25.5
3. Run	15.8	43.4	26.0
Average	**15.8**	**43.8**	**25.7**

Still-photo frame-rates

	Location 1	Location 2	Location 3	Location 4	Location 5
1. Run	(17.7 - 32.8)	(24.7 - 29.8)	(17.7 - 19.2)	(19.8 - 21.0)	(25.0 - 30.7)
2. Run	(19.8 - 20.8)	(24.9 - 31.7)	(17.2 - 19.0)	(19.8 - 20.6)	(27.4 - 34.5)
3. Run	(19.8 - 20.6)	(24.6 - 33.3)	(17.1 - 19.0)	(19.4 - 20.8)	(25.0 - 30.3)
Average	**19.1 – 24.7**	**24.7 – 31.6**	**17.3 – 19.1**	**19.7 – 20.8**	**25.8 – 31.8**

Visual quality

- The local reflections look good with this texture resolution.
- The animation on the wave pattern is to repetitive.



Test 4 – Medium texture resolution – No Pbuffer

Specifications		
Global reflections:	*Cube map type:*	Dynamic
	Updated:	Each frame
	Texture resolution:	128x128
Local reflections:	*Texture resolution:*	512x512
	Terrain pixel scale:	1.0
	Rendering type:	glCopyTexImage2D()
Wave pattern:	*Type:*	3D animated
	Texture resolution:	256x256x64

Fly-through frame-rates

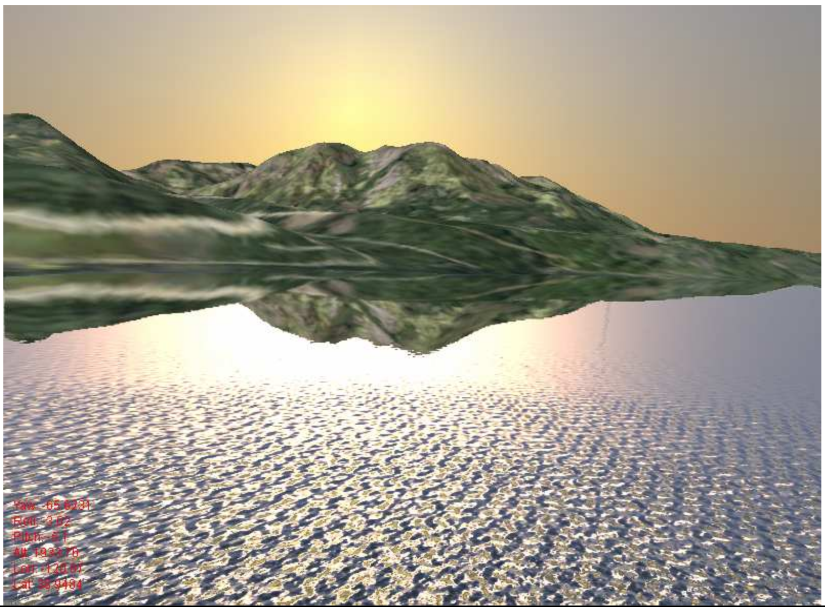
	Min	Max	Average
1. Run	15.4	32.3	22.6
2. Run	15.5	32.2	22.4
3. Run	15.4	34.4	22.8
Average	**15.4**	**33.0**	**22.6**

Still-photo frame-rates

	Location 1	Location 2	Location 3	Location 4	Location 5
1. Run	(15.6 - 32.7)	(20.0 - 20.8)	(15.7 - 16.2)	(19.4 - 20.8)	(25.0 - 32.7)
2. Run	(19.2 - 20.2)	(19.6 - 21.0)	(15.6 - 16.1)	(19.8 - 20.4)	(25.0 - 32.7)
3. Run	(19.4 - 20.2)	(19.8 - 20.6)	(15.6 - 16.2)	(16.2 - 20.4)	(25.0 - 32.2)
Average	**18.1 – 24.4**	**19.8 – 20.8**	**15.6 – 16.2**	**18.5 – 20.5**	**25.0 – 32.5**

Visual quality

- There are no visual differences between this test and test 3



Test 5 – Medium texture resolution – Static Cube Map

Specifications		
Global reflections:	*Cube map type:*	Static
	Updated:	Never
	Texture resolution:	512x512
Local reflections:	*Texture resolution:*	512x512
	Terrain pixel scale:	1.0
	Rendering type:	PBuffer
Wave pattern:	*Type:*	3D animated
	Texture resolution:	256x256x64

Fly-through frame-rates

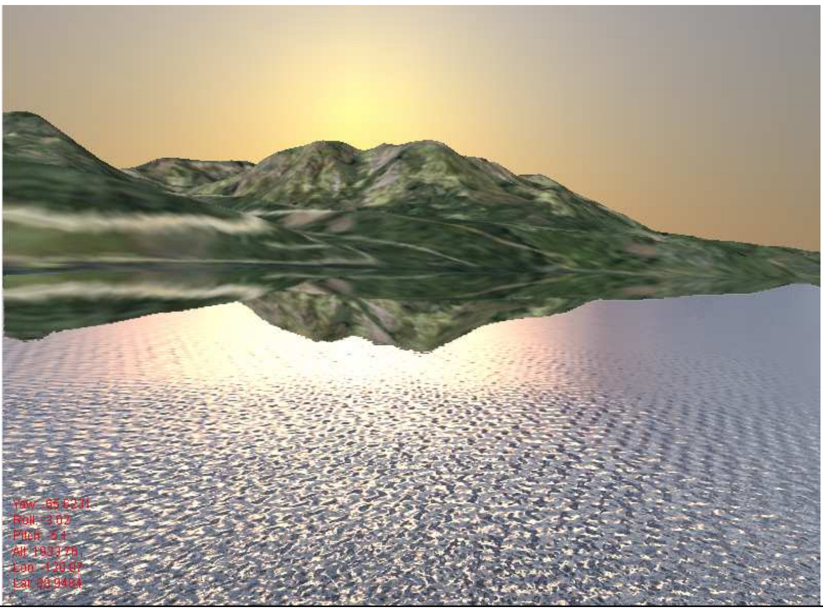
	Min	Max	Average
1. Run	19.6	44.4	32.6
2. Run	20.0	46.4	33.0
3. Run	20.0	47.6	33.2
Average	**19.9**	**46.1**	**33.0**

Still-photo frame-rates

	Location 1	Location 2	Location 3	Location 4	Location 5
1. Run	(19.8 - 44.3)	(24.9 - 33.2)	(19.8 - 20.4)	(24.9 - 25.6)	(41.6 - 44.3)
2. Run	(25.0 - 33.3)	(24.9 - 33.3)	(19.8 - 20.4)	(25.6 - 32.7)	(40.8 - 44.4)
3. Run	(24.9 - 27.4)	(25.0 - 32.7)	(20.0 - 20.4)	(25.0 - 32.8)	(40.7 - 46.4)
Average	**23.2 – 35.0**	**24.9 – 33.1**	**19.9 – 20.4**	**25.2 – 30.4**	**41.0 - 45.0**

Visual quality

- This approach looks pretty good as long as the static global reflection texture colors match the sky colors. The clouds encoded into the textures will not be consistent with the actual sky clouds.



Test 6 – Medium texture resolution – Dynamic Cube Map – less updates

Specifications		
Global reflections:	*Cube map type:*	Dynamic
	Updated:	Every 100th frame
	Texture resolution:	128x128
Local reflections:	*Texture resolution:*	512x512
	Terrain pixel scale:	1.0
	Rendering type:	PBuffer
Wave pattern:	*Type:*	3D animated
	Texture resolution:	256x256x64

Fly-through frame-rates

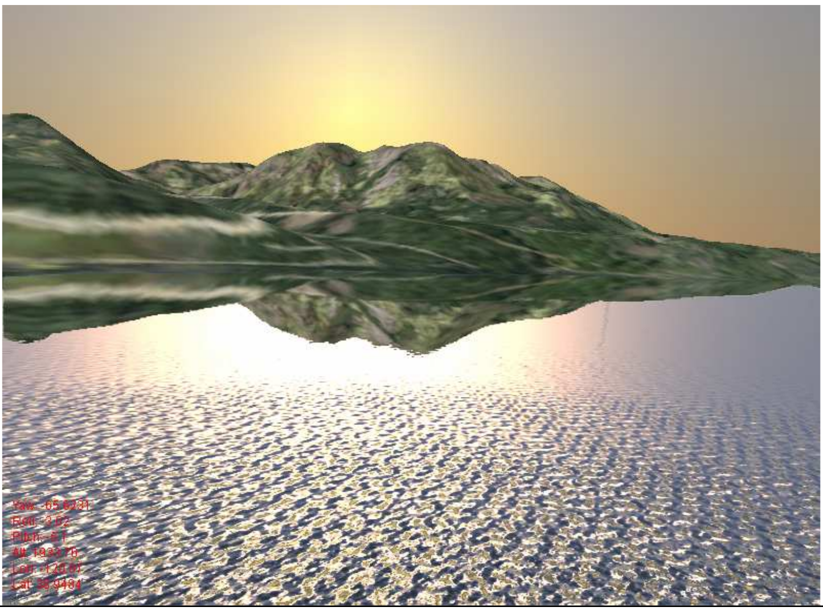
	Min	Max	Average
1. Run	18.3	45.4	30.3
2. Run	17.1	45.4	30.4
3. Run	18.1	44.3	30.4
Average	**17.8**	**45.0**	**30.4**

Still-photo frame-rates

	Location 1	Location 2	Location 3	Location 4	Location 5
1. Run	(18.5 - 43.5)	(24.9 - 33.3)	(19.2 - 20.8)	(25.0 - 33.2)	(39.9 - 45.4)
2. Run	(24.6 - 33.3)	(24.9 - 33.3)	(19.2 - 20.4)	(25.0 - 32.7)	(41.7 - 45.5)
3. Run	(24.6 - 30.3)	(25.0 - 32.8)	(19.2 - 20.4)	(24.7 - 32.8)	(40.7 - 47.6)
Average	**22.6 – 35.7**	**24.9 – 33.1**	**19.2 – 20.5**	**24.9 – 32.9**	**40.8 – 46.2**

Visual quality

- In our test data there are no changing global reflections so there are no visible differences between this test and test 3.



Test 7 – Medium texture resolution – Medium Terrain Pixel Scale

Specifications		
Global reflections:	*Cube map type:*	Dynamic
	Updated:	Each frame
	Texture resolution:	128x128
Local reflections:	*Texture resolution:*	512x512
	Terrain pixel scale:	4.0
	Rendering type:	PBuffer
Wave pattern:	*Type:*	3D animated
	Texture resolution:	256x256x64

Fly-through frame-rates

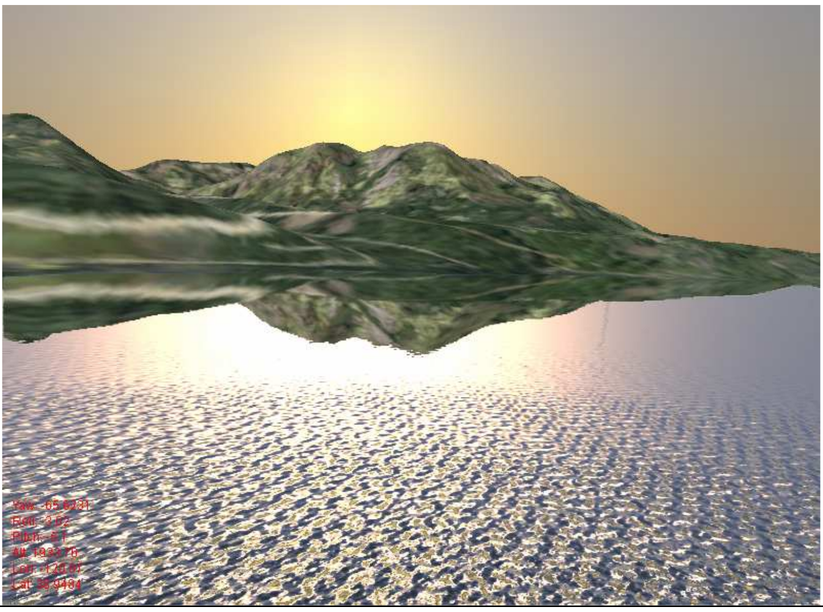
	Min	Max	Average
1. Run	15.5	44.3	26.7
2. Run	16.1	44.3	26.8
3. Run	16.0	47.6	26.5
Average	**15.9**	**45.4**	**26.7**

Still-photo frame-rates

	Location 1	Location 2	Location 3	Location 4	Location 5
1. Run	(19.0 - 44.3)	(24.7 - 32.7)	(19.4 - 20.6)	(24.1 - 29.8)	(39.1 - 44.3)
2. Run	(19.6 - 20.4)	(25.6 - 33.3)	(19.0 - 20.2)	(24.0 - 28.5)	(37.7 - 44.4)
3. Run	(20.0 - 20.6)	(24.9 - 33.3)	(19.2 - 20.4)	(24.7 - 31.2)	(37.7 - 43.4)
Average	**19.5 – 28.4**	**25.1 – 33.1**	**19.2 – 20.4**	**24.3 – 29.8**	**38.2 – 44.0**

Visual quality

- The higher pixel scale on the terrain local reflections leads to some popping on the local reflections, but the popping is minimal and does not have very much effect on the visual quality.



Test 8 – Medium texture resolution – High Terrain Pixel Scale

Specifications		
Global reflections:	*Cube map type:*	Dynamic
	Updated:	Each frame
	Texture resolution:	128x128
Local reflections:	*Texture resolution:*	512x512
	Terrain pixel scale:	8.0
	Rendering type:	PBuffer
Wave pattern:	*Type:*	3D animated
	Texture resolution:	256x256x64

Fly-through frame-rates

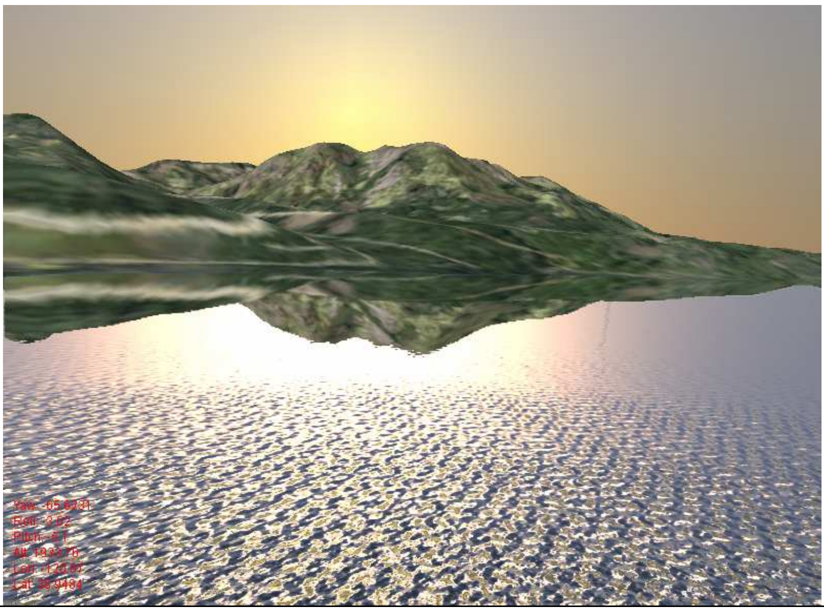
	Min	Max	Average
1. Run	15.6	45.4	27.0
2. Run	16.0	45.4	26.8
3. Run	16.1	44.3	27.1
Average	**15.9**	**45.1**	**27.0**

Still-photo frame-rates

	Location 1	Location 2	Location 3	Location 4	Location 5
1. Run	(16.5 - 45.5)	(24.9 - 32.7)	(19.0 - 20.2)	(24.7 - 31.7)	(39.2 - 44.4)
2. Run	(20.0 - 20.6)	(25.0 - 32.3)	(19.4 - 20.2)	(23.5 - 28.5)	(39.9 - 43.4)
3. Run	(19.4 - 20.6)	(25.3 - 32.2)	(19.2 - 20.2)	(24.4 - 31.7)	(39.1 - 44.4)
Average	**18.6 – 28.9**	**25.1 – 32.4**	**19.2 – 20.2**	**24.2 – 30.6**	**39.4 – 44.1**

Visual quality

- Even with the pixel scale set 8.0 the popping of the local reflections are not particularly distinctive



Test 9 – Medium texture resolution – Static Wave Pattern

Specifications		
Global reflections:	*Cube map type:*	Dynamic
	Updated:	Each frame
	Texture resolution:	128x128
Local reflections:	*Texture resolution:*	512x512
	Terrain pixel scale:	1.0
	Rendering type:	PBuffer
Wave pattern:	*Type:*	Static
	Texture resolution:	512x512

Fly-through frame-rates

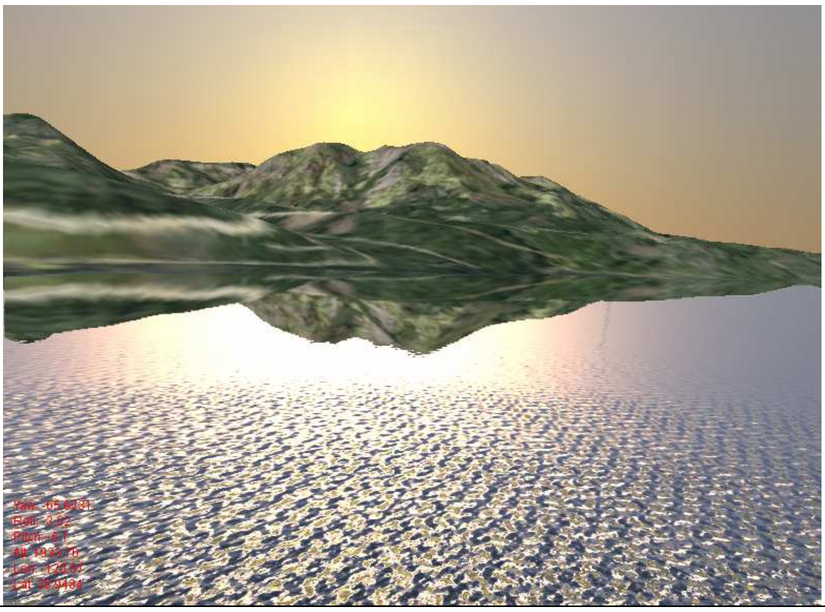
	Min	Max	Average
1. Run	15.0	44.4	25.3
2. Run	15.9	43.5	25.6
3. Run	15.6	43.4	25.6
Average	**15.5**	**43.8**	**25.5**

Still-photo frame-rates

	Location 1	Location 2	Location 3	Location 4	Location 5
1. Run	(18.0 - 30.8)	(24.7 - 33.2)	(18.2 - 19.6)	(19.8 - 20.6)	(26.3 - 33.9)
2. Run	(19.4 - 20.6)	(24.9 - 30.3)	(18.3 - 19.6)	(19.6 - 20.6)	(25.0 - 33.8)
3. Run	(19.8 - 20.4)	(25.3 - 33.3)	(17.4 - 18.8)	(19.6 - 21.0)	(25.0 - 32.7)
Average	**19.1 – 23.9**	**25.0 – 32.3**	**18.0 – 19.3**	**19.7 – 20.7**	**25.4 – 33.5**

Visual quality

- The fact that there is no animation on the wave pattern is very obvious when viewed up close. When viewed from a distance there are no visual differences between this test and test 3.



Test 10 – Medium texture resolution – No local reflections

Specifications		
Global reflections:	*Cube map type:*	Static
	Updated:	Never
	Texture resolution:	512x512
Local reflections:	*Texture resolution:*	None
	Terrain pixel scale:	None
	Rendering type:	None
Wave pattern:	*Type:*	3D animated
	Texture resolution:	256x256x64

Fly-through frame-rates

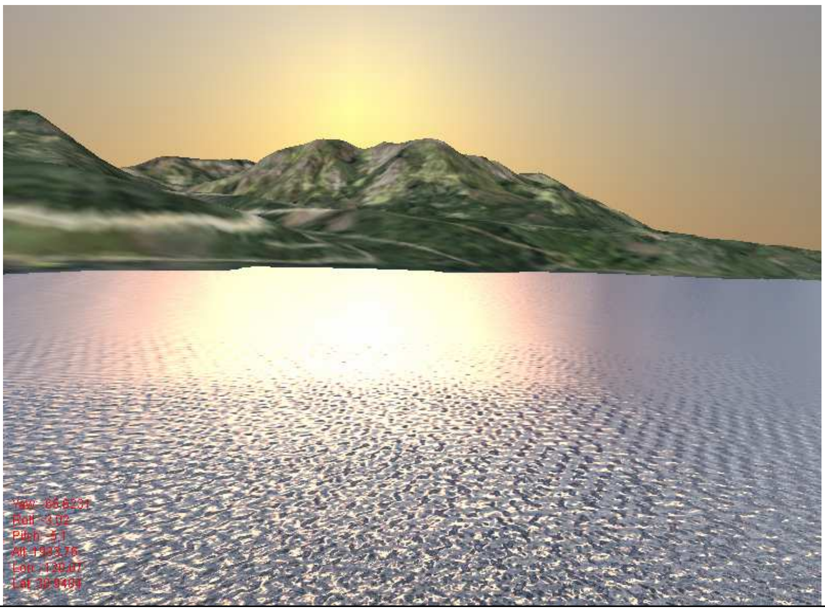
	Min	Max	Average
1. Run	22.4	166.7	61.2
2. Run	26.6	166.7	62.8
3. Run	27.7	181.8	61.7
Average	**25.6**	**171.8**	**61.9**

Still-photo frame-rates

	Location 1	Location 2	Location 3	Location 4	Location 5
1. Run	(29.4 - 133.3)	(49.9 - 100.0)	(29.8 - 31.7)	(37.7 - 40.8)	(95.2 - 125.0)
2. Run	(40.0 - 44.3)	(49.9 - 99.5)	(29.8 - 31.2)	(38.4 - 43.5)	(90.9 - 125.0)
3. Run	(40.8 - 43.5)	(49.9 - 73.8)	(29.4 - 31.3)	(38.4 - 42.6)	(95.2 - 142.9)
Average	**36.7 – 73.7**	**49.9 – 91.1**	**29.7 – 31.4**	**38.2 – 42.3**	**93.8 – 131.0**

Visual quality

- The lack of local reflections becomes more obvious with a calmer wave-pattern.



Summarize

Average fly-through frame-rates			
Test # - specifications	Min	Max	Average
Test 1 – high texture res.	17.1	43.4	25.5
Test 2 – low texture res.	16.4	44.4	26.8
Test 3 – medium texture res.	15.8	43.8	25.7
Test 4 – medium texture res. no Pbuffer	15.4	33.0	22.6
Test 5 – medium texture res. static cube map	19.9	46.1	33.0
Test 6 – medium texture res. less updates	17.8	45.0	30.4
Test 7 – medium texture res. medium pixel scale	15.9	45.4	26.7
Test 8 – medium texture res. high pixel scale	15.9	45.1	27.0
Test 9 – medium texture res. static wave pattern	15.5	43.8	25.5
Test 10 - medium texture res. no local reflections	25.6	171.8	61.9

Average fly-through frame-rates					
Test # - specifications	1.	2.	3.	4.	5.
Test 1 – high texture res.	19.5 – 24.7	24.8 – 32.5	18.5 – 20.1	19.1 – 20.6	25.2 – 31.7
Test 2 – low texture res.	18.2 – 28.2	25.1 – 32.4	19.2 – 20.5	23.6 – 28.0	39.5 – 43.9
Test 3 – medium texture res.	19.1 – 24.7	24.7 – 31.6	17.3 – 19.1	19.7 – 20.8	25.8 – 31.8
Test 4 – medium texture res. no Pbuffer	18.1 – 24.4	19.8 – 20.8	15.6 – 16.2	18.5 – 20.5	25.0 – 32.5
Test 5 – medium texture res. static cube map	23.2 – 35.0	24.9 – 33.1	19.9 – 20.4	25.2 – 30.4	41.0 - 45.0
Test 6 – medium texture res. less updates	22.6 – 35.7	24.9 – 33.1	19.2 – 20.5	24.9 – 32.9	40.8 – 46.2
Test 7 – medium texture res. medium pixel scale	19.5 – 28.4	25.1 – 33.1	19.2 – 20.4	24.3 – 29.8	38.2 – 44.0
Test 8 – medium texture res. high pixel scale	18.6 – 28.9	25.1 – 32.4	19.2 – 20.2	24.2 – 30.6	39.4 – 44.1
Test 9 – medium texture res. static wave pattern	19.1 – 23.9	25.0 – 32.3	18.0 – 19.3	19.7 – 20.7	25.4 – 33.5
Test 10 – medium texture res. no local reflections	36.7 – 73.7	49.9 – 91.1	29.7 – 31.4	38.2 – 42.3	93.8 – 131

Test 1, 2 and 3 shows that there is not much frame-rate gain by lowering the texture resolution. An average increase of 1.3 frames per second is gained in the fly-through sequence by setting the texture resolution to low in test 2, compared to using high resolution textures in test 1. The still-photo sequence only shows noticeable gain in location 1, 3 and 5. However, the low-resolution textures in test 2 significantly decrease the visual quality of the scene, subsequently reducing the water realism to an unacceptable level. Test 3 shows that there is practically no frame-rate difference between a medium texture resolution level and a high resolution level, and the visual quality is not significantly worse in test 3 even though the fewer layers in the Perlin Noise texture makes the wave-animation look more repetitive.

The small differences in frame-rate between test 1, 2 and 3 is probably due too the relative high amount of video memory on the video card. This makes it possible to store most of the textures in the video memory, which enables quick lookups without affecting the CPU. The noticeable frame-rate gain between test 2 and test 1 and 3 can most likely be linked back to a smaller viewport used to create the local and global reflections (*section 5.4*).

Test 4 shows that not using Pbuffers affects the frame-rate negatively. A relative high frame-rate difference of 3.1 frames in the average fly-through frame-rate compared to test 3 has been recorded. This verifies that Pbuffers are a faster method for capturing the local reflections to texture than the `glCopyTexImage2D()` method. Since using Pbuffers does not have an effect on the visual quality of the water simulation; the only reason not to use Pbuffers are the drawbacks described in *section 5.4*.

Test 5 has highest recorded average fly-through frame-rate and the highest frame-rate in most locations in the still photo-sequence, of the nine first tests which has all water features enabled. The average fly-through frame-rate in test 5 is 7.3 frames higher than test 3, which is a fairly significant difference. This confirms that creating dynamic cube maps are a computationally demanding task that steals a lot of frame-rate. The visual quality of using static cube map is the same as using dynamic cube maps as long as the texture colors are fitted to match the sky colors, except that the cloud pattern does not match the true cloud pattern generated by GeoGFX. This mismatching is most obvious when the water is calm. A rough wave-pattern obscures the global reflections and conceals the mismatching better. Test 6 is an alternative to using static cube maps. By only updating the dynamic cube map every 100th frame (approximately every 3 second) we gain 5.7 frames per second in the average fly-through sequence compared to test 3. We also notice a frame-rate gain in most of the locations in the still-photo sequence in this test compared to test 3. The visual quality of test 5 is approximately the same as test 3 as long as the global reflections do not change to rapidly.

Setting the terrain pixel scale to 4.0 in test 7 only buys us 1.0 frame per second in the average fly-through sequence compared to test 3. Location 4 and 5 in the still photo sequence reveals some frame-rate gain, but the visual quality is reduced since the pixel scale leads to “popping” in the local reflections. Test 8 shows that increasing the pixel scale gains a minimal amount of frames (only 0.3 compared to test 7), and reduces the visual quality on the scene by making local reflections “pop” more.

Test 9 reveals that there is no frame-rate gain with using a static wave-pattern instead of an animated wave pattern. This is probably due to the issue discussed earlier, with the video card used in the tests having a relative high amount of video memory. The visual quality of the scene is severely reduced when the viewer is close to the water surfaces because of the lack of wave-movement. When viewing the surfaces from a distance there are no differences in the visual quality because the eye does not percept the wave-movement.

Test 10 shows not surprisingly that by omitting the local reflections and by using a static cube map we get a drastic frame-rate improvement. The average frame-rate in the fly-through sequence is almost twice as high as in test 5 with a gain of 28.9 frames per second. This is of course due to the reason that omitting local reflections saves one triangulated terrain-rendering pass per frame, which is the most demanding task in GeoGFX. Visually we loose some realism in the scene, but if the wave-pattern is very rough local reflections are scarcely visible, and the lack of the reflections are not that evident. With a calm wave-pattern the missing reflections become very obvious and the water surfaces do not look very realistic.

Conclusion

Test 4 and *test 5* is the only two of the nine first tests that have an acceptable average frame-rate above 30 frames per second in the fly-through sequence. This indicates that in order to use the water visualization technique with local reflections presented in this thesis we must either use a static global reflection cube map, or use a dynamic cube map with less updates. *Test 5* updates the cube map approximately every 3 second, but we can probably have less frequent updates if the global reflections such as the cloud pattern do not change too rapidly. Another issue is that the tests performed does not include a rendering of cloud patterns in GeoGFX, introducing cloud pattern rendering to the global reflections might be such a heavy task that using static cube maps are the only realistic rendering alternative.

We notice that all nine first tests have frame-rate drops well below 30, which lead to discontinuity in the fly-through sequence. This indicates that the water visualization technique is not ideal for use with computers that does not have hardware better then the test computers. Even if we set all the options varied in the tests to the optimal level with regards to frame-rate we most likely will not maintain a stable frame-rate above 30.

If this is the case we might have to leave out local reflections as done in *test 10*. Even if this test also has frame-rate drops under 30, the average frame-rates are so high and the minimal frame-rates are above 24 that indicate that the discontinuities are barely visible. Leaving out the local reflections are not ideal, but a good alternative if we want to maintain a high frame-rate and still have visually good looking water surfaces on low-end computers.

7 Concluding remarks

- The water visualization technique presented in this thesis does not take into account the shorelines and reeves of the water areas. Future work should include wave-braking when water hits these regions.

- It is impossible to visualize waves that brake with foam because of the structure of normal maps. This is illustrated in *figure 7.1* where the red arrows are each point normal. Future work should include some sort of "fake" wave braking, for example adding foam to the waves when they are larger then a certain size.

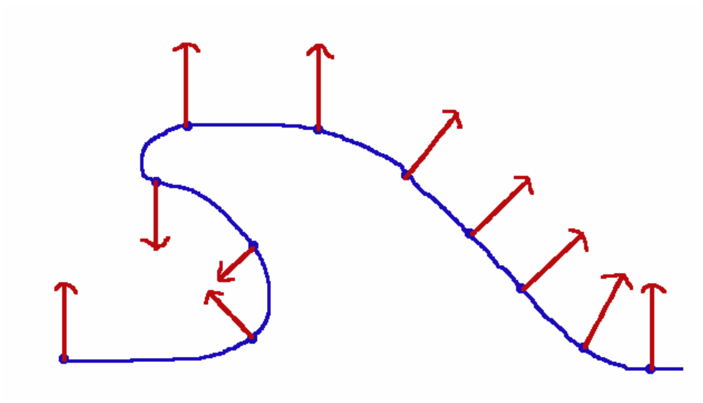


Figure 7.1 – Braking waves. Not possible with normal maps.

- Perlin Noise is an efficient, but not very realistic way of creating wave-patterns. Instead other wave algorithms like the Fourier transform should be used in order to create more realistic looking waves.

- The weather conditions such as wind and rain should have direct impact on wave pattern.

- The sun reflections should be computed relative to the amount of cloud and haze in the sky.

- Interaction between objects (such as boats) and the water should be implemented.

- The `closest()` function described in *code 6* is dependant on water areas composed of several vertices in order to locate the closest surface. This function should be modified so surfaces could be composed of only four vertices (each corner).

- Refractions of water floors should be implemented on shallow lakes.

- The mipmapping problem should be solved in order to use the water visualization technique on larger surfaces.

- Clip planes should be used to create local reflection if the seafloor or objects that are below the water surfaces are introduced.

- The problem with repetitive wave-patterns should be solved. Three probable solutions to this problem are:

- Use large, high resolution textures that cover the whole surface. However, this solution requires a lot of texture memory on the video card.
- Use a relatively small, seamless tile able texture, but apply some perturbation algorithm to the inner section of the texture. This produces tile able textures without looking repetitive. However, this solution also requires a lot of texture memory since each texture has to be treated independently.
- Use a large, coarse resolution texture as an underlying statistical model in addition to the seamless, tile able wave-pattern texture. Use a texture lookup to obtain values from the coarse texture and perturb the wave-pattern texture using these values.

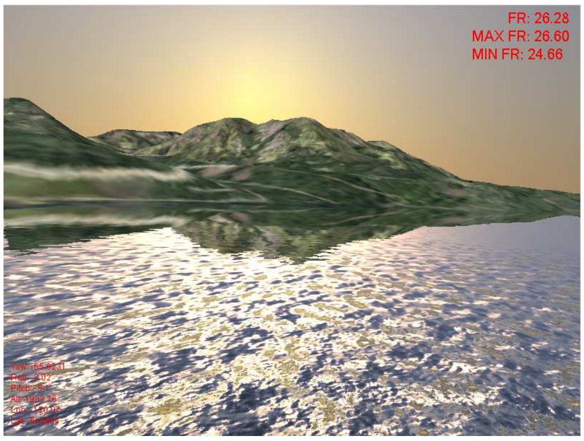
We where able to implement a low polygonal count water rendering technique in GeoGFX by using and modifying existing tools and classes, and introducing new tools like the OpenGL Shading Language binding **GngGLSL** and wave pattern generator **GgPerlinNoise**. The technique presented in this thesis is very well suited for simulating small to medium sized lakes. Because of the mipmapping problem described in *section 5.7* the technique is not very well suited for large surfaces. *Section 5.2* concludes that the generating wave pattern using "Per-Pixel Lighting" is most suited for use with flight-simulators since the "illusion" is revealed when the viewer is to close to the surface.

The *texture Level-Of-Detail with bump mapping* technique where implemented through the **GgWater** class which controls the rendering of the water surfaces. *Chapter 6* tested different options which can be adjusted in **GgWater**, and compared the frame-rate gain versus the visual quality. It was concluded that adjusting the texture detail level does not affect the frame-rate much, which enables us to keep a high level of detail on the local reflections and the wave-pattern. What mainly affects the frame-rate is the technique used for generating global reflections, and whether to display local reflections or not. These are the main two factors that **GgWater** have to adjust with respect to the graphic power on the utilized computer. More computer power makes it possible to produce even more realistic water simulations.

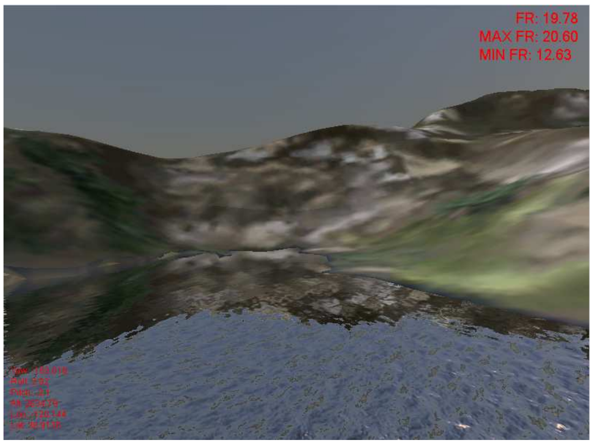
8 Appendix

Appendix A - Images

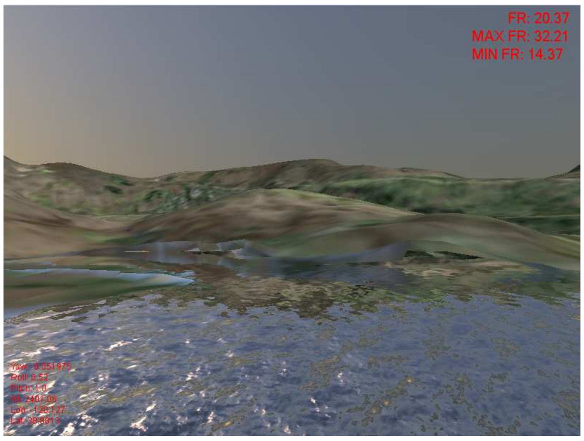
Still-photo test



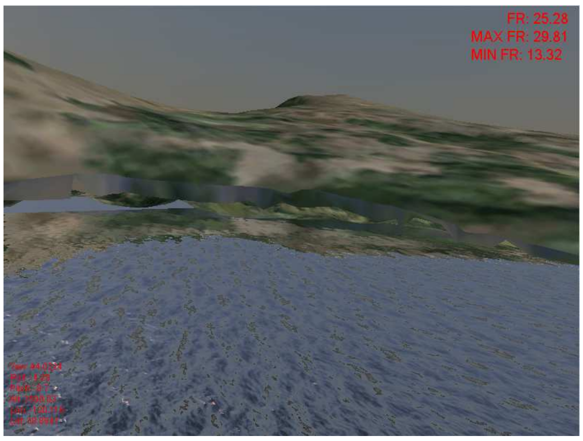
Location 1



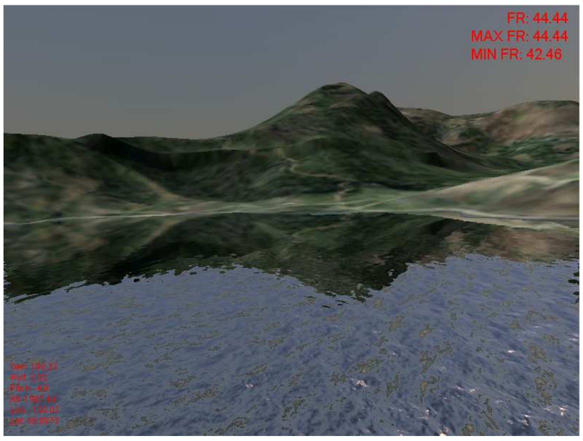
Location 2



Location 3

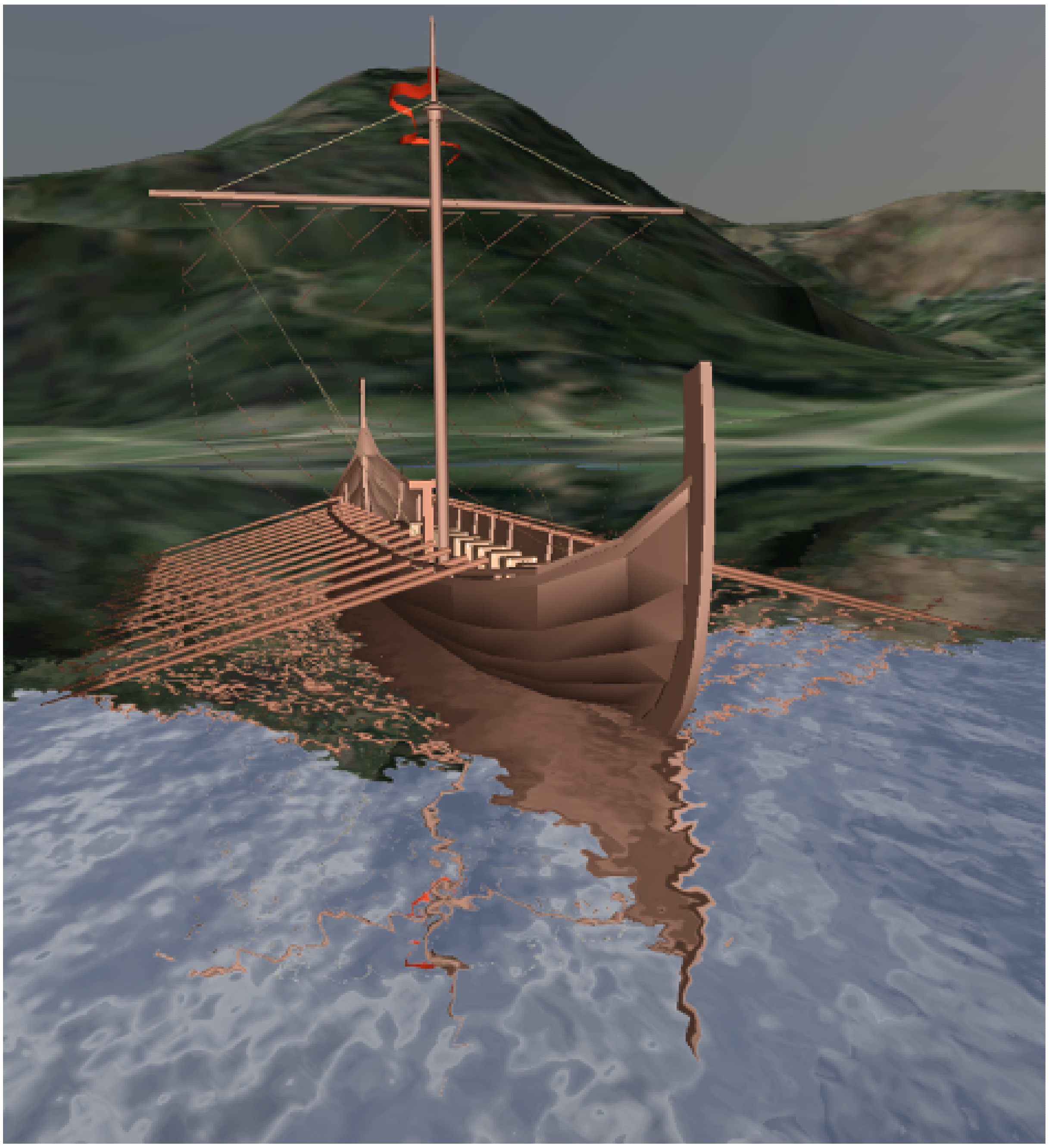


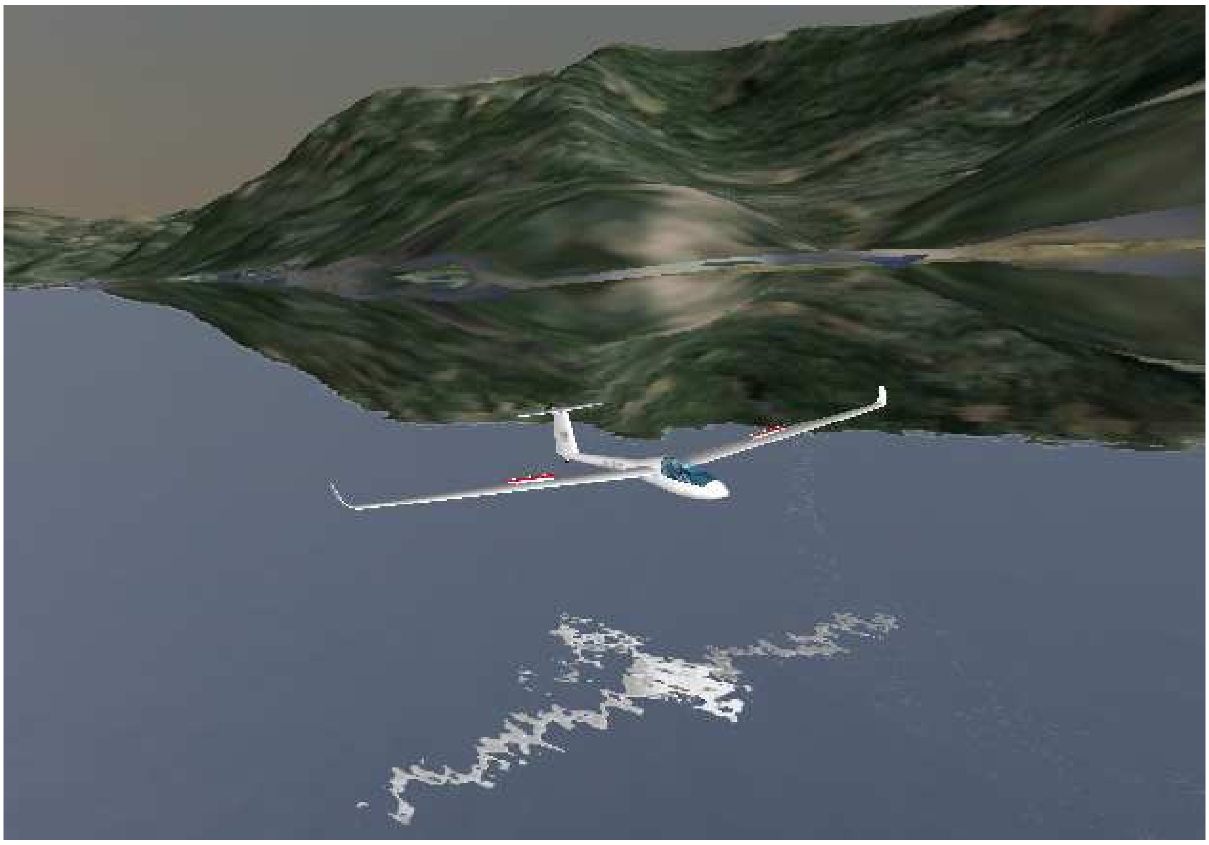
Location 4

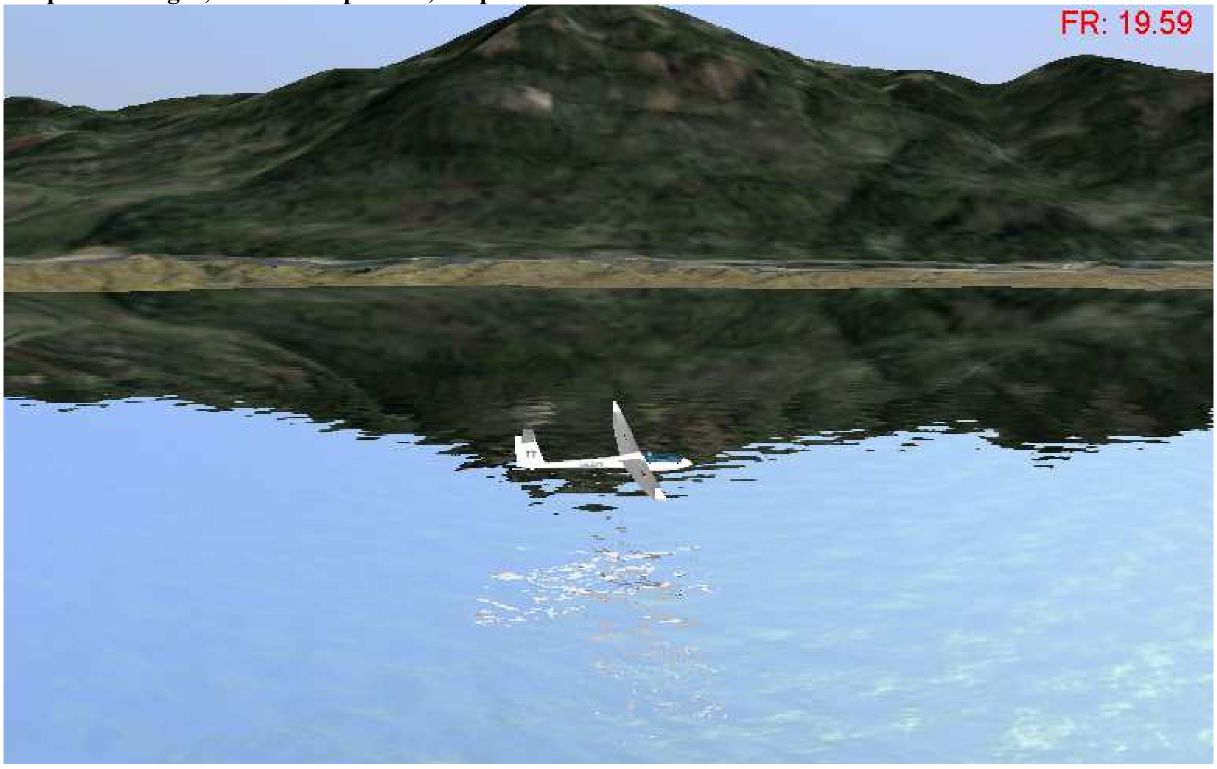


Location 5

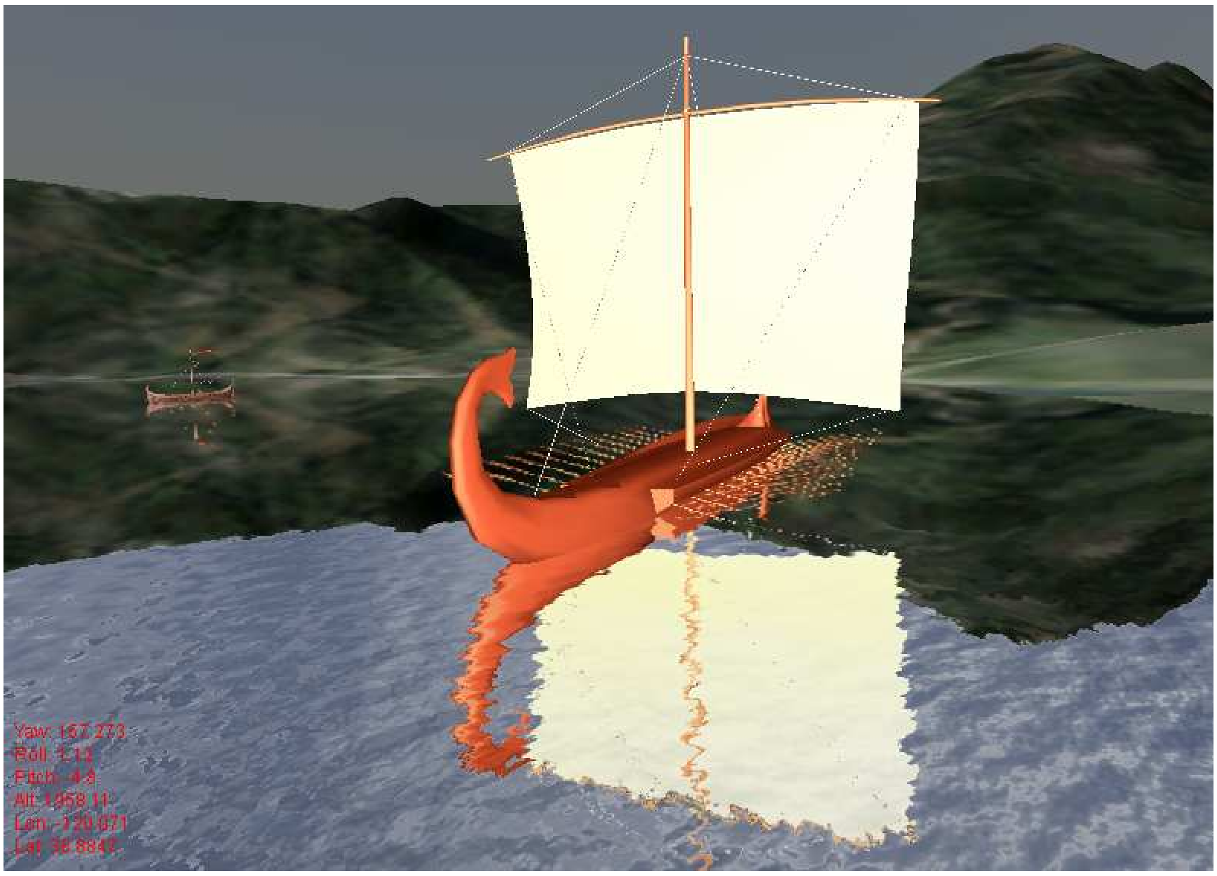
Other screenshots


Viking ship at night, medium rough wave-pattern, deep sea Fresnel color

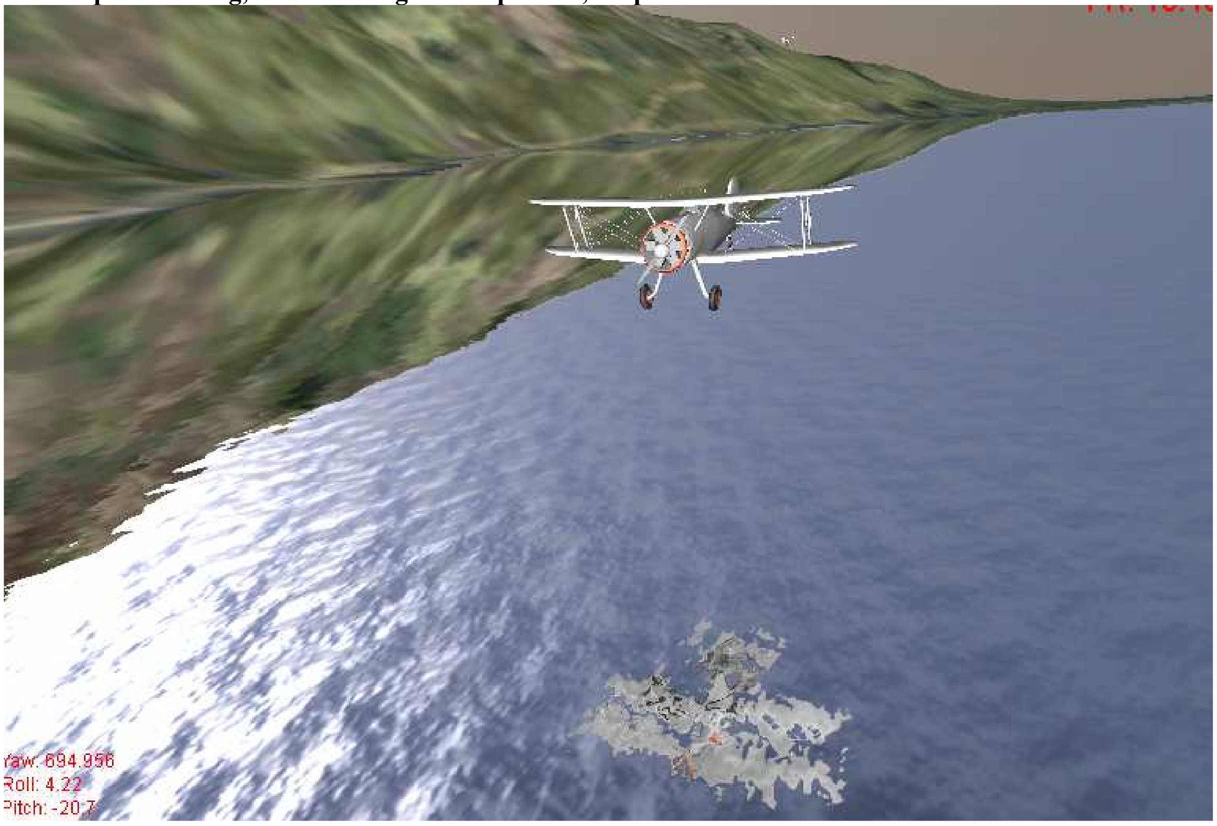

Sailplane at night, calm wave pattern, deep sea Fresnel color



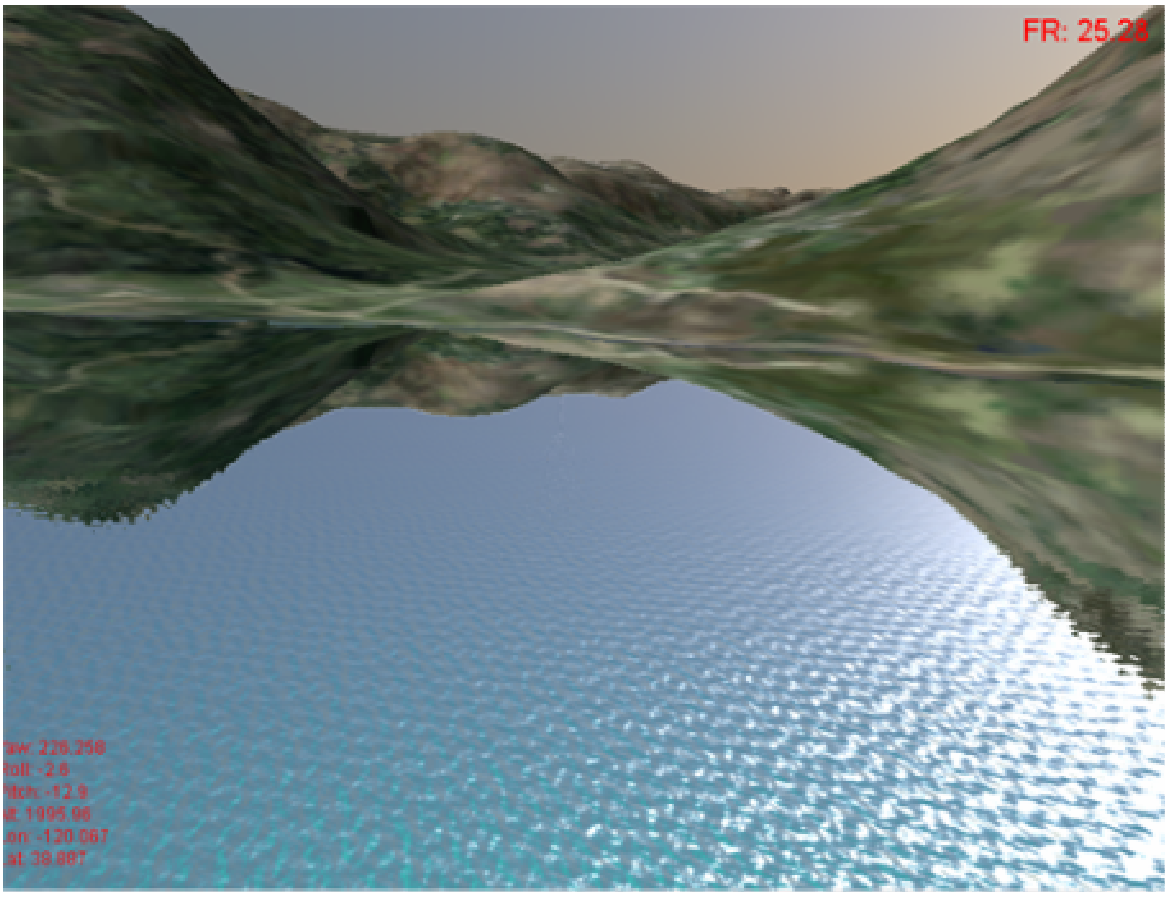
Sailplane at daytime, medium rough wave-pattern, deep sea Fresnel color



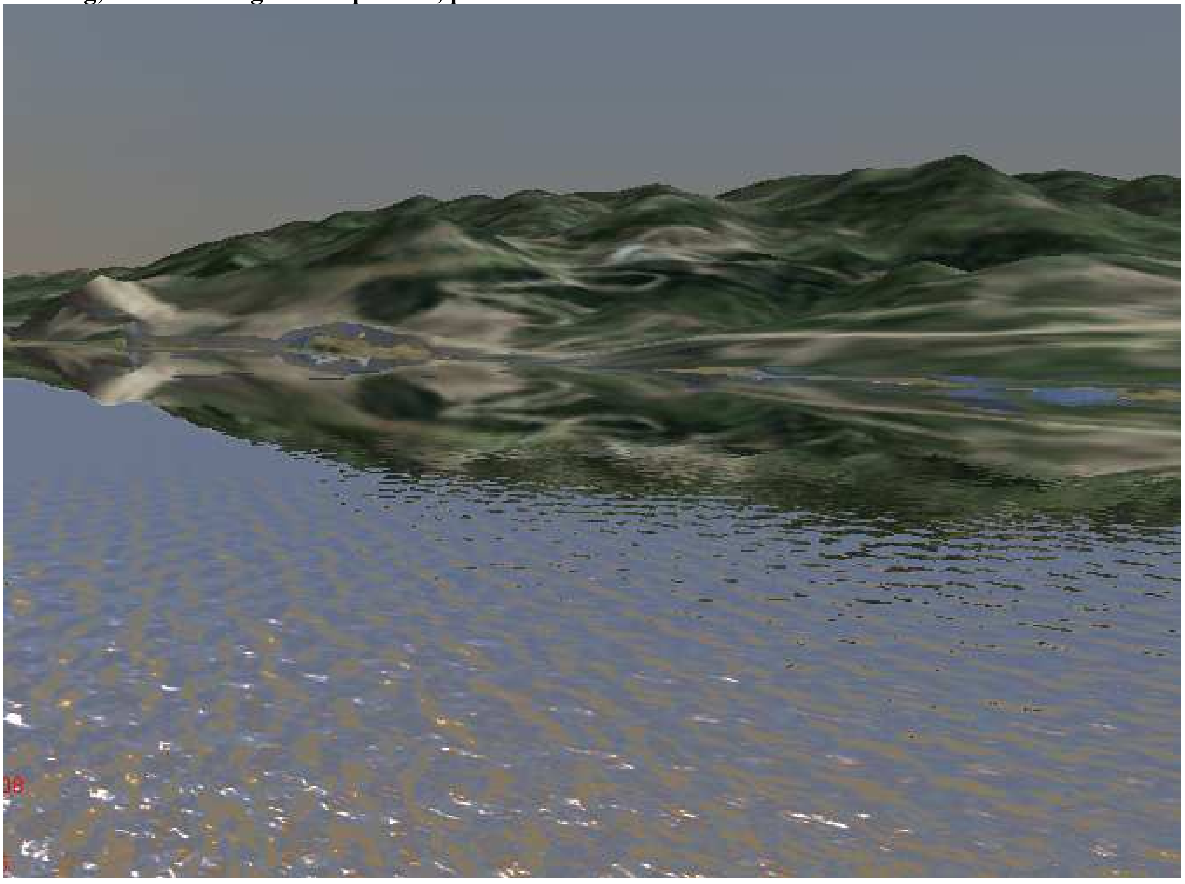
Two ships at evening, medium rough wave-pattern, deep sea Fresnel color



Propeller aircraft at evening, medium rough wave-pattern, deep sea Fresnel color



Evening, medium rough wave-pattern, pacific Fresnel color


Evening, medium rough wave-pattern, muddy Fresnel color

Appendix B – Implementation

The following code shows the most important parts of implementation of the object model described in *figure 4.4*. *Section 4.3* explains which classes are new and which classes that exists in GeoGFX and therefore only has been modified. Only implemented code is shown, unaltered code has been left out and marked with `/* ……………… Clipped code ………… */`.

GeoGFX code

```
//===========================================================================
// File: GgWater.cpp
//===========================================================================

#define PERLIN_WIDTH 256
#define PERLIN_HEIGHT 256
#define PERLIN_THICK 64

//===========================================================================
GgWater::GgWater(GngScene* scene) : GngRenderNode(scene)
{scene_ = scene;}
//===========================================================================

bool GgWater::initialize(const char* filename)
{

 FILE* fp = fopen(filename, "rb");
 if(!fp)
    return false;

//==== Pre-generated Perlin Noise Normal maps ============================
// Used instead of runtime generated normal maps
 /*GngImage img;
 GngImageIO*                              loader                          =
GngImageIO::create("c:/projects/dynamic_bumps/NoiseVolumeBump.tif");
 img.load(*loader);
 GngImage3D waves_3d_(img.getData(), 128,128,128, GngImage::RGBA_32);
 GngTexture3D::Ref waves = new GngTexture3D(&scene_->getTextureManager());
 waves->setImage(waves_3d_);
 waves->setMinFilter(GL_LINEAR);
 img.setDataPtr(NULL);*/
// ========================================================================

 // ============== Static wave pattern texture ============================
 // Used instead of 3D animated normal maps
 /*GngTexture2D::Ref waves_2d_ = (scene_->getTextureManager())
.getImageTexture("c:/projects/static_bumps/oceanBump.tif");
 waves_2d_->setMinFilter(GL_LINEAR);*/
// ========================================================================

// === Create Perlin Noise texture on runtime =============================
 perlin_noise    =    new    GgPerlinNoise(PERLIN_WIDTH,    PERLIN_HEIGHT,
PERLIN_THICK);
 noise3DTexPtr = perlin_noise->get3DPerlinNoise();
// ========================================================================

// === Create Normal map of Perlin Noise texture ==========================
 GngImage              perlin_wave(noise3DTexPtr,              PERLIN_WIDTH,
PERLIN_HEIGHT*PERLIN_THICK, GngImage::RGBA_32);

 // Get the four frequency channels
 GngImage red_bump, blue_bump, green_bump, alpha_bump;
 perlin_wave.decompose(red_bump, green_bump, blue_bump, alpha_bump);

 // Compose texture as sum of the four frequencies
 perlin_wave.composeSUM(red_bump, green_bump, blue_bump, alpha_bump);
 GngImage3D                    waves_3d_(perlin_wave.getData(),PERLIN_WIDTH,
PERLIN_HEIGHT,PERLIN_THICK, GngImage::MONO_8);

 // Filter texture through Normal map creation algorithm
 GngTexture3D::Ref waves = new GngTexture3D(&scene_->getTextureManager());
 waves->setMinFilter(GL_LINEAR);
 waves->initializeNormalMap(waves_3d_);

 perlin_wave.setDataPtr(NULL);
// ==========================================================================

 // ============= Blank texture with alpha = 0.0f =========================
 GngTexture2D::Ref              blank              =              (scene_-
>getTextureManager()).getImageTexture("c:/projects/blank.tga");
 // ========================================================================

// ============== Fresnel texture ==========================================
 GngTexture2D::Ref             fresnel              =              (scene_-
>getTextureManager()).getImageTexture("c:/projects/static_bumps/fresnel_wat
er_sRGB.bmp");
 fresnel->setMinFilter(GL_LINEAR);
// ==========================================================================

// =============== Static global reflections ==============================
 GngCubeMap*       enviroment_       =       new       GngCubeMap(&scene_-
>getTextureManager(),GngImage::RGBA_32);

 GngImageIO*                 front_loader                                 =
GngImageIO::create("c:/projects/cubemaps/clouds_front3.tif");
 GngImageIO*             back_loader                                      =
GngImageIO::create("c:/projects/cubemaps/clouds_back3.tif");
 GngImageIO*                 right_loader                                 =
GngImageIO::create("c:/projects/cubemaps/clouds_right3.tif");
 GngImageIO*             left_loader                                      =
GngImageIO::create("c:/projects/cubemaps/clouds_left3.tif");
 GngImageIO*           top_loader                                         =
GngImageIO::create("c:/projects/cubemaps/clouds_top3.tif");
 GngImageIO*                         bottom_loader                        =
GngImageIO::create("c:/projects/cubemaps/clouds_bottom3.tif");

 enviroment_->setLoader(left_loader, GngCubeMap::POSITIVE_X);
 enviroment_->setLoader(right_loader, GngCubeMap::NEGATIVE_X);

 enviroment_->setLoader(bottom_loader, GngCubeMap::POSITIVE_Y);
 enviroment_->setLoader(top_loader, GngCubeMap::NEGATIVE_Y);

 enviroment_->setLoader(front_loader, GngCubeMap::POSITIVE_Z);
 enviroment_->setLoader(back_loader, GngCubeMap::NEGATIVE_Z);


 enviroment_->setMinFilter(GL_LINEAR);
 enviroment_->setMagFilter(GL_LINEAR);
// ==========================================================================

// =================== Local reflection texture ===========================
 GngTexture2D::Ref    water_reflection_    =    new    GngTexture2D(&scene_-
>getTextureManager(), GngImage::RGBA_32);
```

```
 water_reflection_->setMinFilter(GL_LINEAR);
// ==========================================================================

// ==== Load all GgWaterBody surfaces ======================================
 int num_lakes = 0;
 fread(&num_lakes, sizeof(int), 1, fp);
 for(int i = 0; i < num_lakes; i++)
    {
    GgWaterBody::Ref lake = new GgWaterBody();
      lake->setSun(sun_);
      //Load lake
    if(!lake->load(fp,   waves,   enviroment_,   blank,   water_reflection_,
fresnel))
       {
       cerr << "failed to load lake " << i << endl;
       continue;
       }
    water_bodies_.push_back(lake);
      // Get geographical centers from lakes
      centers_.push_back(lake->centerGEO_);
    }
 fclose(fp);
// ==========================================================================

 return true;
}

// ==========================================================================
// Set GgWater's bsphere to infinitely large
void GgWater::recalcBSphere()
{
 bsphere_.setRadius(MAXDOUBLE);
 bsphere_dirty_ = false;
}

// ==========================================================================
// Calculate closest GgWaterBody object
int GgWater::closest()
{
      MtkPoint3f  cam_pos_  =  GgTools::geoToXYZ(lat_,  lon_,  alt_);  //Camera
position

      //First objects distance
      int wbClosest = 0;
      float dist0 = water_bodies_[0]->distance(cam_pos_);

      //Compare other surface distances
      for(int i = 1; i < (int)water_bodies_.size(); i++)
      {
                //Get objects distance from camera
                float dist = water_bodies_[i]->distance(cam_pos_);

                if (dist < dist0) // Closer than previously checked object
                  {
                        dist0 = dist;
                        wbClosest = i;
                  }

      }
      return wbClosest; //Closest object
}
```

```
// ==========================================================================
// Scene graph function that redraws object
void GgWater::redraw(const GngCamera& cam, const MtkTransform3f& m)
{
 // Render surfaces without writing to depth buffer
 glDepthMask(GL_FALSE);
 GngContext* ctx = GngContext::getCurrent();
 ctx->pushState();
 campos_ = GgTools::geoToXYZ(lat_, lon_, alt_); //Calculate camera position

 // Go through all GgWaterBody objects
 for(int i = 0; i < (int)water_bodies_.size(); i++)
 {
      // ------------ Compute GgWaterBody objects frustum sphere ----------
-
      GngBSphere b;
      MtkPoint3f t = GgTerrain::getCurrOrigin();
      MtkPoint3f ce = water_bodies_[i]->center_;
      MtkPoint3f  ce2  =  MtkPoint3f(ce.x()-t.x(),  ce.y()-t.y(),  ce.z()-
t.z());

    MtkPoint3f p = m.transform(ce2);
      b.setCenter(p);
      b.setRadius(water_bodies_[i]->radius_);
      //---------------------------------------------------------------------
-

      if (cam.frustumTest(b)) // If GgWaterBody inside frustum
      {
            water_bodies_[i]->setCampos(campos_); //Set camera position
            water_bodies_[i]->render(cam,m); //Render GgWaterBody object
      }

 }
 // Clean up states
 ctx->popState();

 ctx = GngContext::getCurrent();
 ctx->applyVertexArray(NULL);
 ctx->applyNormalArray(NULL);
 ctx->applyColorArray(NULL);
 ctx->applyTexCoordArray(NULL);

 //Turn depth buffer back on
 glDepthMask(GL_TRUE);
}
//==========================================================================

// Get local reflection texture from GgWaterBody object i
GngTexture2D::Ref GgWater::getRefTex(int i)
{
      //Show local reflections on selected GgWaterBody object i
      water_bodies_[i]->showRefl(true);

      return water_bodies_[i]->getRefTex();
}
//==========================================================================

// Get global reflection cube map from GgWaterBody object i
GngCubeMap* GgWater::getEnvCube(int i)
```

```
{
	return water_bodies_[i]->getEnvCube();
}

//===========================================================================
//============================================================================
// File: GgWaterBody.cpp
//============================================================================

#define MAX_CELL_SIZE 5000.0

//============================================================================

bool  GgWaterBody::load(FILE*  fp,  GngTexture3D::Ref  waves,  GngCubeMap*
enviroment2_,   GngTexture2D::Ref   blank,   GngTexture2D::Ref   reflect_,
GngTexture2D::Ref waves_2d_)
{
 //Read surface samples from file
 fread(&elev_,    sizeof(double), 1, fp);
 fread(&lat_min_, sizeof(double), 1, fp);
 fread(&lon_min_, sizeof(double), 1, fp);
 fread(&lat_max_, sizeof(double), 1, fp);
 fread(&lon_max_, sizeof(double), 1, fp);

 //Surface origin
 origin_ = GgTools::geoToXYZ(lat_min_, lon_min_, elev_);
 //Surface south east corner
 MtkPoint3d se = GgTools::geoToXYZ(lat_min_, lon_max_, elev_);
 //Surface north west corner
 MtkPoint3d nw = GgTools::geoToXYZ(lat_max_, lon_min_, elev_);

 double width  = (se - origin_).length(); //Surface width
 double height = (nw - origin_).length(); //Surface height
 rows = 1 + (int)(height/MAX_CELL_SIZE); //Number of rows
 cols = 1 + (int)(width/MAX_CELL_SIZE); //Number of columns

 double d_lat = (lat_max_ - lat_min_) / rows; //Latitude steps per row
 double d_lon = (lon_max_ - lon_min_) / cols; //Longitude steps per column

 //Surface radius, used with frustum sphere
 radius_ = sqrt(pow(width, 2) + pow(height,2));

 //Calculate XYZ center coordinates
 float x = ((se.x()+nw.x())/2);
 float y = ((se.y()+nw.y())/2);
 float z = nw.z();
 center_ = MtkPoint3f(x,y,z);

 //Calculate geographic center coordinates
 float x2 = ((lat_min_+lat_max_)/2);
 float y2 = ((lon_max_+lon_min_)/2);
 float z2 = elev_;
 centerGEO_ = MtkPoint3f(x2, y2, z2);

 //Calculate surface tangent vector
 tan_ = nw - origin_;
 MtkTransform3f earth_to_local2;
 GgTools::calcEarthToLocal(centerGEO_.x(),                   centerGEO_.y(),
earth_to_local2);
 tan_ = earth_to_local2.transform(tan_);

 //Triangulate surface, regular triangulation
 for(int i = 0; i <= rows; i++)
 {
   GngIndexArray16* index = new GngIndexArray16(GL_TRIANGLE_STRIP);
   double lat = lat_min_ + i*d_lat;
```

```
    for(int j = 0; j <= cols; j++)
    {
        //Calculate vertices
       double lon = lon_min_ + j*d_lon;
       MtkPoint3d  p = GgTools::geoToXYZ(lat, lon, elev_);
       MtkVector3d d = p - origin_;
       vertex_.append(d.x(), d.y(), d.z());

       //Calculate surface normal
       MtkVector3f n = (GgTools::geoToXYZ(lat, lon, elev_ + 100.0) - p);
       //Transform normal to local coordinates
       MtkTransform3f earth_to_local;
       GgTools::calcEarthToLocal(centerGEO_.x(),             centerGEO_.y(),
earth_to_local);
       MtkVector3f n_local = earth_to_local.transform(n);
       n_local.normalize();
       normal_.append(n_local.x(), n_local.y(), n_local.z());

       //Texture coordinates
       texture_.append(j,i);
       if(i < rows)
       {
          //Set index arrays
          index->append((i + 1)*(cols + 1) + j);
          index->append(i*(cols + 1) + j);
       }
    }

  indices_.push_back(index);
 }

 //Set local and global reflection textures
 enviroment_ = enviroment2_;
 water_reflection_ = reflect_;

 //Texture states
 bump_state_.enableTexture(GL_TEXTURE_2D, 0);
 bump_state_.enableTexture(GL_TEXTURE_3D, 1);
 bump_state_.enableTexture(GL_TEXTURE_CUBE_MAP, 2);
 bump_state_.enableTexture(GL_TEXTURE_2D, 3);

 bump_state_.setTexture(waves.getPtr(), 1);
 bump_state_.setTexture(enviroment_, 2);
 bump_state_.setTexture(waves_2d_.getPtr(), 3);

 //No local reflection state - load texture with alpha = 0.0
 refl_state_.setTexture(water_reflection_.getPtr(), 0);

 //Local reflection state
 no_refl_state_.setTexture(blank.getPtr(), 0);

 // Clean up states
 pass1_.disableTexture(GL_TEXTURE_CUBE_MAP, 2);
 pass1_.disableTexture(GL_TEXTURE_3D, 1);
 pass1_.disableTexture(GL_TEXTURE_2D, 0);
 pass1_.disableTexture(GL_TEXTURE_2D, 3);

 //Load shaders
 shader.setShaders("c:/projects/shaders/fs.frag",
"c:/projects/shaders/vs.vert");
```

```
 //Calculate sun
 calculateSunGlobalPosition();
 calculateSunGlobalColors(); //hack

 return true;
}

//==========================================================================
// Calculate distance from camera
float GgWaterBody::distance(MtkPoint3f camera)
{
  //Get first vertex distance
  MtkPoint3f   v0   =   MtkPoint3f(vertex_.getPoint(0).x()   +   origin_.x(),
vertex_.getPoint(0).y()    +    origin_.y(),    vertex_.getPoint(0).z()    +
origin_.z());
  float dist0 = MtkPoint3f::dist(camera, v0);

  for(int i = 1; i < vertex_.getSize(); i++)
  {
    MtkPoint3f   v   =   MtkPoint3f(vertex_.getPoint(i).x()   +   origin_.x(),
vertex_.getPoint(i).y()    +    origin_.y(),    vertex_.getPoint(i).z()    +
origin_.z());
    // Vertex distance from camera
    float dist = MtkPoint3f::dist(camera, v);
    //Test if last checked distance is shorter than previous checked
    if (dist < dist0)
    {
      dist0 = dist;
    }
  }
  return dist0; //Distance of closest vertex to camera
}
// =========================================================================

void GgWaterBody::render(const GngCamera& cam, const MtkTransform3f& m)
{
 //Draw vertices relative to terrain origin
 MtkVector3f t = origin_ - GgTerrain::getCurrOrigin();
 glMatrixMode(GL_MODELVIEW);
 glPushMatrix();
 glTranslatef(t.x(), t.y(), t.z());

 updateObjectSpaceVectors(); //Calculate eye vector

 //Update animation timer
 time += 0.009f;
 if (time > 120)
       time = 0.0f;

 //Enable and set shader variables
 shader.enable();
 setShaderValues();

 glDisableClientState(GL_NORMAL_ARRAY);
 glDisableClientState(GL_COLOR_ARRAY);
 GngContext* ctx2 = GngContext::getCurrent();
 ctx2->pushState();

 //Set water states
 ctx2->apply(bump_state_);
 //Set reflection state
 if (show_refl_)
      ctx2->apply(refl_state_);
 else
      ctx2->apply(no_refl_state_);

 drawWater();

 show_refl_ = false;

 //Release pbuffer textures
 water_reflection_->release();
 enviroment_->release();

 // Clean up states
 ctx2->apply(pass1_);
 ctx2->popState();

 glPopMatrix();
 shader.disable();
}
//============================================================================

void GgWaterBody::setShaderValues()
{
  //Get material and sun colors
  GngColor4f specular = global_sun_->getSpecular();
  GngColor4f refr_color = global_sun_refr->getDiffuse();

  //Uniform variables
  shader.setUniform3("light_vector",        global_sun_->getDirection().x(),
global_sun_->getDirection().y(), global_sun_->getDirection().z());
  shader.setUniform3("tangent_vector", tan_.x(), tan_.y(), tan_.z());
  shader.setUniform4("refr_color",      refr_color.r(),      refr_color.g(),
refr_color.b(), refr_color.a());
  shader.setUniform4("specular",  specular.r(), specular.g(), specular.b(),
specular.a());
  shader.setUniform1("specular_power", 32.0f);

  shader.setUniform1("time_0_X", time);
  shader.setUniform1("noiseSpeed", 0.34f);

  //Attribute arrays
  shader.setAttribArray("eye_vector", 3, GL_FLOAT, GL_FALSE, 0, e_.begin()-
>v());
  shader.enableAttribArray("eye_vector");

  //Textures
  shader.setTexture("reflection", 0); //Local reflections
  shader.setTexture("Noise", 1); //Waves
  shader.setTexture("enviroment", 2);     //Global reflections cube map
  shader.setTexture("fresnel", 3);  //Global reflections cube map
}

//============================================================================

void GgWaterBody::drawWater()
{
  glClientActiveTexture(GL_TEXTURE0);
  glEnableClientState(GL_VERTEX_ARRAY);
  glEnableClientState(GL_NORMAL_ARRAY);
  glEnableClientState(GL_TEXTURE_COORD_ARRAY);

  vertex_.apply();
  texture_.apply();
  normal_.apply();

  //Draw index arrays
  for(int i = 0; i < (int)indices_.size(); i++)
  {
      indices_[i]->drawElements();
  }

  glDisableClientState(GL_TEXTURE_COORD_ARRAY);
  glDisableClientState(GL_NORMAL_ARRAY);
  glDisableClientState(GL_VERTEX_ARRAY);

}

//========================================================================

void GgWaterBody::calculateSunGlobalColors()
{
  //Set refraction surface refraction color
  global_sun_refr = new GngLight();
  global_sun_refr->setDiffuse(GngColor4f(0.1078f,0.9762f,0.8218f, 1.0f));

  //calculate angle between waternormal and sunvector
  float angle = Mtk::rad_to_deg * MtkVector3f::angle(normal_.getVector(0),
global_sun_->getDirection());
  // Hack: setting specular color dependent on sunposition, colors are
precomputed
  if (angle < 20)
      global_sun_->setSpecular(GngColor4f(1.0f,1.0f,1.0f, 1.0f));
  if (angle <= 70 && angle >= 20)
      global_sun_->setSpecular(GngColor4f(1.1f,1.0f,0.9412f, 1.0f));
  if (angle > 70)
      global_sun_->setSpecular(GngColor4f(1.2f,0.4f,0.1f, 1.0f));
}
// ======================================================================

void GgWaterBody::calculateSunGlobalPosition()
{
  global_sun_ = new GngLight();
  // Find vector from vertex to lightsource
  // - Lightsource (Sun) is "infinitely" far away, so all vectors are
parallel on vertices
  //Sun direction is normalized and stored in local coordinates
  MtkVector3f light_pos_ = sun_->getDirection();
  global_sun_->setDirection(light_pos_);
}

//========================================================================

void GgWaterBody::updateObjectSpaceVectors()
{
  // Clear vector array
  e_.clear();

  // For all vertices...
  for(int i = 0; i < vertex_.getSize(); i++)
  {
      // ...find vector from vertex to eye position
      MtkVector3f vertex_position = vertex_.getPoint(i).createVector();
      MtkVector3f water_origin = origin_.createVector();
      MtkVector3f camera_position = campos_.createVector();

      // Vector from vertex to eye
      MtkVector3f eye = camera_position - (vertex_position+water_origin);

      //...Rotate eye vector into local coordinate system
      MtkTransform3f earth_to_local;
      GgTools::calcEarthToLocal(centerGEO_.x(),             centerGEO_.y(),
earth_to_local);
      e_.push_back(earth_to_local.transform(eye));

      }

}

//==========================================================================
//===========================================================================
// File: SceneWaterGlobalReflections.cpp
//===========================================================================

SceneWaterGlobalReflections::SceneWaterGlobalReflections(GgWater::Ref   ws_,
GngTransform::Ref geocentric_t_, GngScene* s_, MouseControl* cam_c)
{
  cam_control = cam_c;
  scene_ = s_;
  geocentric_trans_ = geocentric_t_;
  water_surfaces_ = ws_;
}

void SceneWaterGlobalReflections::redraw(GngCamera::Ref cam_, int w, int h,
int o_w, int o_h, double o_fovy)
{
  //Get viewpoint controller class
  const Observer* observer_ = cam_control->getObserver();
  //Save original camera position
  double viewpoint_longitude = observer_->getLongitude();
  double viewpoint_latitude = observer_->getLatitude();
  double viewpoint_altitude = observer_->getAltitude();
  //... and orientation
  double viewpoint_roll = observer_->getRoll();
  double viewpoint_pitch = observer_->getPitch();
  double viewpoint_yaw = observer_->getYaw();

  //Hand camera position to GgWater
  water_surfaces_->setCamPos(viewpoint_latitude,       viewpoint_longitude,
viewpoint_altitude);

  //Get  global  reflection  cube  map  texture  from  the  GgWaterBody  object
closest to the camera
  int closest = water_surfaces_->closest();
  GngCubeMap* env_ = water_surfaces_->getEnvCube(closest);

  //=========== Step 1 - set viewpoint position to water surface center ===
      cam_control->setPosition(water_surfaces_->centers_[closest].x(),
water_surfaces_->centers_[closest].y(),                       water_surfaces_-
>centers_[closest].z());
  //=========================================================================

  //===== Step 2 - set camera matrices ====================================
  cam_->setViewport(0, 0, w, h);
  cam_->setPerspective(90.0, (double)w/h);
  //=========================================================================

  //========== Step 3 - hide local objects ================================
  geocentric_trans_->setInvisible(true);
  //=========================================================================

  //=== Step 4, 5, 6, 7 - Render 6 cube map sides ========================
  pb_->enable();

  //Left
  env_->setPbufCube(pb_);
  env_->setOffscreenImageCube(1);
  cam_control->setOrientation(90, 0, 0);
  scene_->redraw(cam_());

  //Right
  env_->setOffscreenImageCube(2);
  cam_control->setOrientation(270, 0, 0);
  scene_->redraw(cam_());

  //Bottom
  env_->setOffscreenImageCube(3);
  cam_control->setOrientation(0, -90, 0);
  scene_->redraw(cam_());


  //Top
  env_->setOffscreenImageCube(4);
  cam_control->setOrientation(0,90, 0);
  scene_->redraw(cam_());

  //Front
  env_->setOffscreenImageCube(5);
  cam_control->setOrientation(0, 0, 0);
  scene_->redraw(cam_());

  //Back
  env_->setOffscreenImageCube(6);
  cam_control->setOrientation(180, 0, 0);
  scene_->redraw(cam_());

  pb_->disable();
  //========================================================================

  //===== Step 8 - Hand reflection cube map texture to GgWater ============
  /* env_ cube map is a pointer and do need to be handed back to GgWater */
  //========================================================================

  //Restore viewpoint to original position and orientation
  cam_control->setPosition(viewpoint_latitude,         viewpoint_longitude,
viewpoint_altitude);
  cam_control->setOrientation(viewpoint_yaw,                viewpoint_pitch,
viewpoint_roll);

  cam_->setViewport(0, 0, o_w, o_h);
  cam_->setPerspective(o_fovy, (double)o_w/o_h);

  //Unhide local objects
  geocentric_trans_->setInvisible(false);

}

//========================================================================
//============================================================================
// File: SceneWaterLocalReflections.cpp
//============================================================================

SceneWaterLocalReflections::SceneWaterLocalReflections(GgWater::Ref    ws_,
GgTerrain::Ref  t_,  GngTransform::Ref  sky_t_,  GngScene*  s_,  MouseControl*
cam_c)
{
  water_surfaces_ = ws_;
  terrain_ = t_;
  sky_trans_ = sky_t_;
  scene_ = s_;
  cam_control = cam_c;
  pix_scale = 2.5;           //Default pixel scale
  tex_scale = 1;        //Default texel scale
}

void SceneWaterLocalReflections::redraw(GngCamera::Ref cam_, int w, int h,
int o_w, int o_h)
{
  pb_->enable(); //Enable PBuffer

  //======Step 1 - Mirror scene ==========================================
  const Observer* observer_ = cam_control->getObserver();
  //Save original camera position
  double viewpoint_longitude = observer_->getLongitude();
  double viewpoint_latitude = observer_->getLatitude();
  double viewpoint_altitude = observer_->getAltitude();
  //... and orientation
  double viewpoint_roll = observer_->getRoll();
  double viewpoint_pitch = observer_->getPitch();

  //Hand camera position to GgWater
  water_surfaces_->setCamPos(viewpoint_latitude,       viewpoint_longitude,
viewpoint_altitude);

  //Get local reflection texture from the closest GgWaterBody object
  int closest = water_surfaces_->closest();
  GngTexture2D::Ref water_reflection = water_surfaces_->getRefTex(closest);

  //Get closest surface elevation
  double surface_elevation =  water_surfaces_->centers_[closest].z();

  //Mirror altitude
  double  mirrored_altitude  =  surface_elevation  -  (viewpoint_altitude  -
surface_elevation);

  //Mirror pitch
  double mirrored_pitch = -viewpoint_pitch;

  //Mirror roll
  double mirrored_roll = 0;
  if (viewpoint_roll == 0)
      mirrored_roll = 180;
  if (viewpoint_roll > 0)
      mirrored_roll = (180-viewpoint_roll);
  if (viewpoint_roll < 0)
      mirrored_roll = -180+abs(viewpoint_roll);

  //Set mirrored camera
  cam_control->setLocalReflPos(viewpoint_latitude,     viewpoint_longitude,
mirrored_altitude, mirrored_pitch, mirrored_roll, 0.1, 5000);
  //=========================================================================

  // == Step 2 Remove objects behind the reflective water surface =========
   /* Testing has shown that using clip planes to remove objects behind the
  reflective surfaces in GeoGFX are unnecessary as long as no objects exists
  underneath the water surfaces. GeoGFX has, as stated in section 5.4.4, no
  seafloor triangulation, and the parts of the triangulated terrain which
  have elevation lower than the mirrored surface inflicts no visible errors
  on the mirrored image. Subsequently we have decided to omit clip planes in
  this thesis.
   */
  // ========================================================================

  //====== Step 3 - Set camera viewport ===================================
  cam_->setViewport(0, 0, w, h);
  //=========================================================================

  //======== Step 4 and 5 - Hide water surfaces and sky cone ==============
  water_surfaces_->setInvisible(true);
  sky_trans_->setInvisible(true);
  //=========================================================================

  //Store original terrain pixel and texel scale
  double org_terrain_pixel_scale = terrain_->getPixelScale();
  double org_terrain_texel_scale = terrain_->getPixelScale();

  //Set new pixel and texel scale
  terrain_->setPixelScale(pix_scale);
  terrain_->setTexelScale(tex_scale);

  glClearColor(1.0f, 1.0f, 1.0f, 0.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  //========== Step 6 - Render mirrored scene =============================
  scene_->redraw(cam_());
  //=========================================================================

  pb_->disable(); // Disable Pbuffer

  //====== Step 7 - Capture mirrored scene to texture ====================
  //water_reflection->setOffscreenImage(0,0,w, h); //No Pbuffer solution
  //Pbuffer solution
  water_reflection->setOffscreenImageWGL(pb_, 0, 0, w, h);
  //=========================================================================

  //============ Step 8 - Hand reflection texture to GgWater ==============
  /* water_reflection texture is a pointer and do need to be handed back to
GgWater */
  //=========================================================================

  //Set viewpoint back to original position
  cam_control->setLocalReflPos(viewpoint_latitude,     viewpoint_longitude,
viewpoint_altitude, viewpoint_pitch, viewpoint_roll, 0.1, 30000);

  //Set water surfaces and sky cone visible
  water_surfaces_->setInvisible(false);
  sky_trans_->setInvisible(false);

  //Set camera viewport to original size
  cam_->setViewport(0, 0, o_w, o_h);

  //Restore terrain pixel and texel scale
  terrain_->setPixelScale(org_terrain_pixel_scale);
  terrain_->setTexelScale(org_terrain_texel_scale);

}
```

```
//=============================================================================
// File: SceneWindow.cpp
//=============================================================================

/* …………….. Clipped code ……….. */

void SceneWindow::buildScene(GgTerrain* terrain)
{
 /* …………….. Clipped code ……….. */

 // Set up transformation nodes
 water_trans_      = new GngTransform(scene_);

 water_ = new GgWater(scene_);

 // Initialize sky
 sky_.initialize(scene_, 64, 32);
 sky_.setLuminanceFactor(1.3);
 sky_.setTurbidity(4.0);
 sky_.setTimePos(39.0965, -119.996, 180, 17.50);

 // Light source
 sun_ = new GngLightSource(scene_);
 MtkVector3f vec_sun = sky_.getSunPosition();

 sun_->setPosition(MtkPoint3f(vec_sun.x(), vec_sun.y(), vec_sun.z()));

  water_->setSun(sun_());
 if(!water_->initialize("c:/projects/LakeTahoe/lakes.dat"))
 {
   cerr << "couldn't find lake file!\n";
 }

 /* …………….. Clipped code ……….. */

 surface_trans_    -> addChild(geocentric_trans_());
 geocentric_trans_      ->addChild(water_());

 //Create local and global reflections object
 scene_local_refl_      =       new       SceneWaterLocalReflections(water_,
terrain_,sky_trans_, scene_, curr_nav_ );
 scene_local_refl_->setPixelScale(1.0); //Local reflection pixel scale
 scene_global_refl_      =      new      SceneWaterGlobalReflections(water_,
geocentric_trans_, scene_, curr_nav_);


 //Send local and global object pointers to SceneWidget
 scene_w_->setSceneWLReflections(scene_local_refl_);
 scene_w_->setSceneWGReflections(scene_global_refl_);
 scene_w_->setWGRPbuffer(); //Initialize global reflection PBuffer
 scene_w_->setWLRPbuffer(); //Initialize local reflection PBuffer

}
```

```
//============================================================================
// File: SceneWidget.cpp
//============================================================================

/* ……………… Clipped code ………… */

void SceneWidget::paintGL()
{

/* ……………… Clipped code ………… */

  //Create dynamic global reflections
  scene_global_refl_->redraw(cam_,    GLOB_WIDTH,    GLOB_HEIGHT,    width(),
height(), 60.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  //Create local reflections
  scene_local_refl_->redraw(cam_,     LOC_WIDTH,     LOC_HEIGHT,     width(),
height());
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  //Draw whole scene
  scene_.redraw(cam_());

/* ……………… Clipped code ………… */

}

/* ……………… Clipped code ………… */
```

```
//===========================================================================
// File: GgPerlinNoise.cpp
//===========================================================================

/* Creates Perlin Noise rasters
Code copied from http://www.texturingandmodeling.com//CODE/PERLIN/PERLIN.C
and restructured by Fredrik Danielsen to fit in GeoGFX */

#define MAXB 0x100
#define N 0x1000
#define NP 12   // 2^N
#define NM 0xfff

#define s_curve(t) ( t * t * (3. - 2. * t) )
#define lerp(t, a, b) ( a + t * (b - a) )
#define setup(i, b0, b1, r0, r1)\
        t = vec[i] + N;\
        b0 = ((int)t) & BM;\
        b1 = (b0+1) & BM;\
        r0 = t - (int)t;\
        r1 = r0 - 1.;

#define at3(rx, ry, rz) ( rx * q[0] + ry * q[1] + rz * q[2] )

static int p[MAXB + MAXB + 2];
static double g3[MAXB + MAXB + 2][3];

int start;
int B;
int BM;

// ========================================================================
// Constructor
GgPerlinNoise::GgPerlinNoise(int w, int h, int t)
{
  make3DNoiseTexture(w, h, t);
}
// ========================================================================

// Makes noise textures tile able
void GgPerlinNoise::SetNoiseFrequency(int frequency)
{
  start = 1;
  B = frequency;
  BM = B-1;
}
//==========================================================================

//Normalize vectors used to create noise
void GgPerlinNoise::normalize3(double v[3])
{
  double s;

  s = sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
  v[0] = v[0] / s;
  v[1] = v[1] / s;
  v[2] = v[2] / s;
}
// =======================================================================

//Starts noise generation algorithm
```

```
void GgPerlinNoise::startNoise()
{
  int i, j, k;

  srand(30757);
  for (i = 0; i < B; i++)
  {
      p[i] = i;

      for (j = 0; j < 3; j++)
            g3[i][j] = (double)((rand() % (B + B)) - B) / B;
      normalize3(g3[i]);
  }

  while (--i)
  {
      k = p[i];
      p[i] = p[j = rand() % B];
      p[j] = k;
  }

  for (i = 0; i < B + 2; i++)
  {
      p[B + i] = p[i];
      for (j = 0; j < 3; j++)
            g3[B + i][j] = g3[i][j];
  }
}
//============================================================================

//3D noise algorithm
double GgPerlinNoise::noise3(double vec[3])
{
  int bx0, bx1, by0, by1, bz0, bz1, b00, b10, b01, b11;
  double rx0, rx1, ry0, ry1, rz0, rz1, *q, sy, sz, a, b, c, d, t, u, v;
  int i, j;

  if (start)
  {
      start = 0;
      startNoise();
  }

  setup(0, bx0, bx1, rx0, rx1);
  setup(1, by0, by1, ry0, ry1);
  setup(2, bz0, bz1, rz0, rz1);

  i = p[bx0];
  j = p[bx1];

  b00 = p[i + by0];
  b10 = p[j + by0];
  b01 = p[i + by1];
  b11 = p[j + by1];

  t  = s_curve(rx0);
  sy = s_curve(ry0);
  sz = s_curve(rz0);

  q = g3[b00 + bz0]; u = at3(rx0, ry0, rz0);
  q = g3[b10 + bz0]; v = at3(rx1, ry0, rz0);
  a = lerp(t, u, v);

  q = g3[b01 + bz0]; u = at3(rx0, ry1, rz0);
  q = g3[b11 + bz0]; v = at3(rx1, ry1, rz0);
  b = lerp(t, u, v);

  c = lerp(sy, a, b);

  q = g3[b00 + bz1]; u = at3(rx0, ry0, rz1);
  q = g3[b10 + bz1]; v = at3(rx1, ry0, rz1);
  a = lerp(t, u, v);

  q = g3[b01 + bz1]; u = at3(rx0, ry1, rz1);
  q = g3[b11 + bz1]; v = at3(rx1, ry1, rz1);
  b = lerp(t, u, v);

  d = lerp(sy, a, b);

  return lerp(sz, c, d);
}
//=========================================================================

//Make width*height*thick Perlin Noise texture
void GgPerlinNoise::make3DNoiseTexture(int width, int height, int thick)
{
  int f, i, j, k, inc;
  int startFrequency = 4;
  int numOctaves = 4;
  double ni[3];
  double inci, incj, inck;
  int frequency = startFrequency;
  GLubyte* ptr;
  double amp = 0.5;

  if  ((noise3DTexPtr  =  (GLubyte*)  malloc(width  *  height  *  thick  *  4))  ==
NULL)
  {
      cerr << "Couldn't not allocate 3d Perlin Noise texture\n";
  }
  for  (f  =  0,  inc  =  0;  f  <  numOctaves;  ++f,  frequency  *=  2,  ++inc,  amp  *=
0.5)
  {
      SetNoiseFrequency(frequency);
      ptr = noise3DTexPtr;
      ni[0] = ni[1] = ni[2] = 0;

      inci = 1.0 / (width / frequency);
      for (i = 0; i < width; ++i, ni[0] += inci)
      {
            incj = 1.0 / (height / frequency);
            for (j = 0; j < height; ++j, ni[1] += incj)
            {
                  inck = 1.0 / (thick / frequency);
                  for (k = 0; k < thick; ++k, ni[2] += inck, ptr += 4)
                        *(ptr  +  inc)  =  (GLubyte)  (((noise3(ni)  +  1.0)  *
amp) * 128.0);

            }
      }
  }
}
```

```
//===========================================================================
// File: GngGLSL.cpp
//===========================================================================


// ==========================================================================
//Enable shader
void GngGLSL::enable()
{

      if (prog != 3452816845)
            glUseProgramObjectARB(prog);
      else
            cout << "Shaders not enabled";

}
// ==========================================================================
//Disable shader
void GngGLSL::disable()
{
            glUseProgramObjectARB(0);
}
// ==========================================================================
// Get uniform location by string
GLint GngGLSL::getUniLoc(const GLcharARB *name)
{
      GLint loc;

      loc = glGetUniformLocationARB(prog, name);

      if (loc == -1)
            cout << "No such uniform named " << name;

      return loc;
}
// ==========================================================================
//Get Attribute location by string
GLint GngGLSL::getAttLoc(const GLcharARB *name)
{
      GLint loc;

      loc = glGetAttribLocationARB(prog, name);

      if (loc == -1)
            cout << "No such uniform named " << name;

      return loc;
}
// ==========================================================================
// Read shader code from file
GLcharARB* GngGLSL::readShader(const char * filename, int shaderType)
{
      char name[100];
    strcpy(name, filename);

      switch (shaderType)
    {
        case 1:
            strcat(name, ".vert");
            break;
        case 2:
            strcat(name, ".frag");
            break;
        default:
            printf("ERROR: unknown shader file type\n");
            exit(1);
            break;
    }

  FILE * pFile;
  long lSize;
  GLcharARB * buffer;

  pFile = fopen ( filename , "rb" );
  if (pFile==NULL) exit (1);

  // obtain file size.
  fseek (pFile , 0 , SEEK_END);
  lSize = ftell (pFile);
  rewind (pFile);

  // allocate memory to contain the whole file.
  buffer = (char*) malloc (lSize);
  if (buffer == NULL) exit (2);

  // copy the file into the buffer.
  fread (buffer,1,lSize,pFile);

  /*** the whole file is loaded in the buffer. ***/

  // terminate
  fclose (pFile);
  //free (buffer);
  return buffer;

}
// ===========================================================================
// Link shader program to OpenGL
bool GngGLSL::link()
{
      glGetObjectParameterivARB(prog, GL_OBJECT_LINK_STATUS_ARB, &linked);

      if (!linked)
            glLinkProgramARB(prog);

      glGetObjectParameterivARB(prog, GL_OBJECT_LINK_STATUS_ARB, &linked);

      if (!linked)
            return false;
}
// ===========================================================================
// Initialize shaders
bool GngGLSL::initShaders(const GLcharARB * fragShader, const GLcharARB *
vertShader)
{
if (initShader(vertShader, 1) && initShader(fragShader, 2))
{
      link();
      return true;
}
```

```
return false;

}
// ===========================================================================
// Read and compile shaders
bool GngGLSL::initShader(const GLcharARB * shader, int shaderType)
{
      GLint compiled;

      if (prog == 3452816845)
            prog = glCreateProgramObjectARB();

      switch (shaderType)
      {
      case 1:
            vs = glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
            glShaderSourceARB(vs, 1, &shader, NULL);
            glCompileShaderARB(vs);
            glGetObjectParameterivARB(vs,      GL_OBJECT_COMPILE_STATUS_ARB,
&compiled);
            if (!compiled)
            {
                  cout << "Vertex shader not compiled";
                  return false;
            }
            glAttachObjectARB(prog, vs);
            break;
      case 2:
            fs = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
            glShaderSourceARB(fs, 1, &shader, NULL);
            glCompileShaderARB(fs);
            glGetObjectParameterivARB(fs,      GL_OBJECT_COMPILE_STATUS_ARB,
&compiled);
            if (!compiled)
            {
                  cout << "Fragment shader not compiled";
                  return false;
            }
            glAttachObjectARB(prog, fs);
            break;
      }

      return true;
}
// ===========================================================================
//Initialize Fragment shader
void GngGLSL::setFragmentShader(const char *filename)
{
      initShader(readShader(filename, 2), 2);
      link();
}
// ===========================================================================
// Initialize vertex shaders
void GngGLSL::setVertexShader(const char *filename)
{
      initShader(readShader(filename, 1), 1);
      link();
}
// ===========================================================================
//Set shaders in GngGLSL object
```

```
void GngGLSL::setShaders(const char *fragmentfile, const char *vertexfile)
{
      initShaders(readShader(fragmentfile, 2), readShader(vertexfile, 1));
}

//Shader variables
//==========================================================================
// Float uniform variables
void GngGLSL::setUniform1(const GLcharARB *name, float x)
{
      glUniform1fARB(getUniLoc(name), x);
}
void GngGLSL::setUniform2(const GLcharARB *name, float x, float y)
{
      glUniform2fARB(getUniLoc(name), x, y);
}
/* ……………… Clipped code ………… */
//==========================================================================
// Integer uniform variables
void GngGLSL::setUniform1(const GLcharARB *name, int x)
{
      glUniform1iARB(getUniLoc(name), x);
}
void GngGLSL::setUniform2(const GLcharARB *name, int x, int y)
{
      glUniform2iARB(getUniLoc(name), x, y);
}
/* ……………… Clipped code ………… */

//4x4 Uniform matrix
void  GngGLSL::setUniformMatrix4(const  GLcharARB  *name,  GLuint  count,
GLboolean transpose, const GLfloat *v)
{
      glUniformMatrix4fvARB(getUniLoc(name), count, transpose, v);
}
//==========================================================================
// Uniform textures
void GngGLSL::setTexture(const GLcharARB *name, int number)
{
      glUniform1iARB(getUniLoc(name), number);
}
//==========================================================================
//Float atttributes
void GngGLSL::setAttrib1(const GLcharARB *name, float x)
{
      glVertexAttrib1fARB(getAttLoc(name), x);
}
void GngGLSL::setAttrib2(const GLcharARB *name, float x, float y)
{
      glVertexAttrib2fARB(getAttLoc(name), x, y);
}
/* ……………… Clipped code ………… */
//==========================================================================
// Integer attributes
void GngGLSL::setAttrib1(const GLcharARB *name, int x)
{
      glVertexAttrib1fARB(getAttLoc(name), x);
}
void GngGLSL::setAttrib2(const GLcharARB *name, int x, int y)
{
      glVertexAttrib2fARB(getAttLoc(name), x, y);
}
/* ……………… Clipped code ………… */
//============================================================================
//Short attributes
void GngGLSL::setAttrib1(const GLcharARB *name, short x)
{
      glVertexAttrib1fARB(getAttLoc(name), x);
}

void GngGLSL::setAttrib2(const GLcharARB *name, short x, short y)
{
      glVertexAttrib2fARB(getAttLoc(name), x, y);
}
/* ……………… Clipped code ………… */
//============================================================================
//Attribute arrays
void  GngGLSL::setAttribArray(const  GLcharARB  *name,  GLint  size,  GLenum
type, GLboolean normalized, GLsizei stride, const GLvoid *pointer)
{
      glVertexAttribPointerARB(getAttLoc(name),   size,   type,   normalized,
stride, pointer);
}

void GngGLSL::enableAttribArray(const GLcharARB *name)
{
      glEnableVertexAttribArrayARB(getAttLoc(name));
}
void GngGLSL::disableAttribArray(const GLcharARB *name)
{
      glDisableVertexAttribArrayARB(getAttLoc(name));
}
```

```
//==========================================================================
// File: GngTexture3D.cpp
//==========================================================================

/* ……………… Clipped code ………… */

//==========================================================================
/* 3D texture version of Normal map downsampling fucntion described in [3]
and used in GngTexture2D */
GngTexture3D::Normal*
GngTexture3D::downSampleNormalMap(GngTexture3D::Normal *old,
                                                        int  w2,  int  h2,  int  t2,
int w, int h, int t)
{
  /* ……………… Clipped code ………… */
}

// ========================================================================
/* 3D version of height field to normal map function described in [3] and
used in GngTexture2D */
GngTexture3D::Normal*   GngTexture3D::convertHeightFieldToNormalMap(GLubyte
*pixels, int w, int h, int t, int wr, int hr, float scale)
{
 /* ……………… Clipped code ………… */
}

// ========================================================================
/* 3D version on normal map initialization function decribed in [3] and
used in GngTexture2D */
bool GngTexture3D::initializeNormalMap(const GngImage3D& img, int unit)
{
  /* ……………… Clipped code ………… */
}

/* ……………… Clipped code ………… */
```

```
//===========================================================================
// File: GngTexture2D.cpp
//===========================================================================

/* ……………… Clipped code ………… */

void GngTexture2D::apply(GngTextureUnit& curr_tex_unit)
{
  /* ……………… Clipped code ………… */
  if (pbuf_rend)
  {
      //Bind PBuffer to texture
      if( !wglBindTexImageARB(pbuf_->pbuffer_, WGL_FRONT_LEFT_ARB ) )
      {
            MessageBox(NULL,"Could not bind p-buffer to render texture!",
                        "ERROR",MB_OK|MB_ICONEXCLAMATION);
            exit(-1);
      }
  }
  /* ……………… Clipped code ………… */
}

//===========================================================================
//PBuffer texture release function
void GngTexture2D::release()
{
  if (pbuf_rend)
  {
      if( !wglReleaseTexImageARB(pbuf_->pbuffer_, WGL_FRONT_LEFT_ARB ) )
      {
        MessageBox(NULL,"Could not release p-buffer from render texture!",
            "ERROR",MB_OK|MB_ICONEXCLAMATION);
        exit(-1);
      }
  }
}

//===========================================================================
//PBuffer version of GngTexture2D::setOffscreenImage()
void GngTexture2D::setOffscreenImageWGL(const GngPBuffer* pbuffer_, int x,
int y, int w, int h, int unit)
{
  /* ……………… Clipped code ………… */
  pbuf_rend = true;
  /* ……………… Clipped code ………… */
}
/* ……………… Clipped code ………… */
```

```
//=========================================================================
// File: GngCubeMap.cpp
//=========================================================================

/* ……………… Clipped code ………… */

// ========================================================================
//Initialize Pbuffer
void GngCubeMap::setPbufCube(const GngPBuffer* pbuffer_, int unit)
{
 setLock(false);
 unload();

      pbuf_  = pbuffer_;
pbuf_rend = true;

      GngContext::getCurrent()->setActiveTextureUnit(unit);
 glGenTextures(1, &handle_);
 glBindTexture(target_, handle_);
 applyTexParameters(target_);
 setMinFilter(GL_LINEAR);

 img_dirty_ = false;

 // Can't let texture manager handle this texture when
 // we don't have a dynamic image source.
 setLock(true);

}
// =========================================================================
//Cube map version of GngTexture2D::setOffscreenImage() function
void GngCubeMap::setOffscreenImageCube(int face)
{
 int              attr[]              =              {WGL_CUBE_MAP_FACE_ARB,
WGL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB+(face-1), 0};
 wglSetPbufferAttribARB(pbuf_->pbuffer_, attr);

}

/* ……………… Clipped code ………… */
```

```
//==========================================================================
// File: GngPBuffer.cpp
//==========================================================================

/* ……………… Clipped code ………… */

bool GngPBuffer::initialize(bool sharecontexts, bool sharelists)
{
      //   If   we   want   to   use   PBuffer   to   render   to   texture   set
WGL_BIN_TO_TEXTURE_RGBA_ARB = true
  int pf_attr[] =
  {
   WGL_SUPPORT_OPENGL_ARB, TRUE,       //P-buffer will be used with OpenGL
   WGL_DRAW_TO_PBUFFER_ARB, TRUE,      //Enable render to p-buffer
   WGL_BIND_TO_TEXTURE_RGBA_ARB, TRUE, //P-buffer will be used as a texture
   WGL_RED_BITS_ARB, 8,                // At least 8 bits for RED channel
   WGL_GREEN_BITS_ARB, 8,              // At least 8 bits for GREEN channel
   WGL_BLUE_BITS_ARB, 8,               // At least 8 bits for BLUE channel
   WGL_ALPHA_BITS_ARB, 8,              // At least 8 bits for ALPHA channel
   WGL_DEPTH_BITS_ARB, 16,             // At least 16 bits for depth buffer
   WGL_DOUBLE_BUFFER_ARB, FALSE,       // We don't require double buffering
   0                                   // Zero terminates the list
      };

// if PBuffer is set to render to a 2D texture use this properties array
  int properties[] =
  {
   // Our p-buffer will have a texture format of RGBA
   WGL_TEXTURE_FORMAT_ARB, WGL_TEXTURE_RGBA_ARB,
   // Our texture target will be GL_TEXTURE_2D
    WGL_TEXTURE_TARGET_ARB, WGL_TEXTURE_2D_ARB,
    0                                           // Zero terminates the list
  };
// else if Pbuffer set to render cubemap textures set texture target to
WGL_TEXTURE_CUBE_MAP_ARB
  if (cube_tex)
  {
   properties[3] = WGL_TEXTURE_CUBE_MAP_ARB;
  }
}

/* ……………… Clipped code ………… */
```

```
//============================================================================
// File: GngImage.cpp
//============================================================================

/* ……………… Clipped code ………… */

//============================================================================
//Add four image colors together
void GngImage::composeSUM(const GngImage& r, const GngImage& g,
                          const GngImage& b, const GngImage& a)
{
 copyHeader(r);
 setFormat(MONO_8);
 allocMem();

 unsigned char* r_ptr = r.getData();
 unsigned char* g_ptr = g.getData();
 unsigned char* b_ptr = b.getData();
 unsigned char* a_ptr = a.getData();

 int j = 0;
 for(int i = 0; i < npix_; i++)
 {
   data_[j++] = (r_ptr[i] + g_ptr[i] + b_ptr[i] + a_ptr[i] + 8) * 1.5;
 }
}
/* ……………… Clipped code ………… */
```

Vertex and pixel shader code

```
//==============================================================================
// File: Vertex shader
//==============================================================================

varying vec3 vEyeVector; //Object Space

attribute vec3 eye_vector; //Object Space
uniform vec3 tangent_vector; //Object Space

varying vec4 vTexCoordProj;
varying vec3 vTexCoord;
varying mat3 obj2tan;

void main(void)
{

  vEyeVector = eye_vector;

  // Tangent space conversion matrix ------------------------------------------------------------------
  obj2tan[0] = normalize(tangent_vector);
  obj2tan[1] = normalize(cross(normalize(tangent_vector), normalize(gl_Normal)));
  obj2tan[2] = normalize(gl_Normal);


 // Texture coordinates ---------------------------------------------------------------------------------
 vTexCoord = gl_Vertex*0.05;

  // Projective texture coordinates ----------------------------------------------------------------------
  mat4 Mr;
  Mr[0] = vec4(0.5, 0.0, 0.0, 0.0);
  Mr[1] = vec4(0.0, 0.5, 0.0, 0.0);
  Mr[2] = vec4(0.0, 0.0, 0.5, 0.0);
  Mr[3] = vec4(0.5, 0.5, 0.5, 1.0);


  mat4 Mprojtex = Mr * gl_ModelViewProjectionMatrix;

  vTexCoordProj = mul(Mprojtex, gl_Vertex);
  vTexCoordProj.s = -vTexCoordProj.s;

  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

```
//=============================================================================
// File: Fragment shader
//=============================================================================

uniform vec4 specular;
uniform vec4 refr_color;

uniform float specular_power;

uniform float noiseSpeed;
uniform float time_0_X;

uniform sampler3D Noise;
uniform sampler2D reflection;
uniform samplerCube enviroment;
uniform sampler2D fresnel;

varying vec3 vTexCoord;
uniform vec3 light_vector;
varying vec3 vEyeVector;

varying vec4 vTexCoordProj;

varying mat3 obj2tan;

void main(void)
{
   vEyeVector = normalize(vEyeVector);
  vec3 vLightVector = normalize(light_vector);

  //Tangent to object space matrix, inverse of object to tangent space
  mat3 tan2obj;
  tan2obj[0] = vec3(obj2tan[0].x, obj2tan[1].x, obj2tan[2].x);
  tan2obj[1] = vec3(obj2tan[0].y, obj2tan[1].y, obj2tan[2].y);
  tan2obj[2] = vec3(obj2tan[0].z, obj2tan[1].z, obj2tan[2].z);

  // Make 3d texture animate
  vTexCoord.z += 0.15 * time_0_X;
  vTexCoord.x += (noiseSpeed/8) * time_0_X;

  //Wave pattern
  vec3 noisy = texture3D( Noise, vTexCoord.xyz).xyz;
  vec3 smooth = vec3(0.5, 0.5, 1.0);
  vec3 bump = mix(smooth, noisy, 0.05);

  //Normalize normal map
  bump = normalize((bump * 2) - 1);
  //Rotate bump map normals to object space
  vec3 bump_obj = normalize(mul(bump, tan2obj));

  // Texture projection of local water reflections
  float offset = 7;
  vTexCoordProj.xy += offset * bump.xy;   //Disturb local reflection with wave pattern
  vec4 refl_ = texture2DProj(reflection, vTexCoordProj);

   /* Specular dotproduct with surface normal, used to avoid self-shadowing */
    float ndotl = dot( vLightVector, tan2obj[2]);

    float self_spec = 1;
    float c = 0.125;

   if (ndotl > c)
        self_spec = 1;
   if (ndotl > 0 && ndotl <= c)
    self_spec = (1/c)*ndotl;
   if (ndotl <= 0)
      self_spec = 0;

   //Reflection vector
   vec3 R = normalize(reflect(vEyeVector, bump_obj));

  //Half angle
  // vec3 H = normalize(vEyeVector + vLightVector);

    // Phong specular lighting
    vec3  sun  =   pow((vec3(specular)  *  2  *  self_spec  *  pow(max(  vec3(0.0),  dot(-
R,vLightVector)), vec3(specular_power) )), 1/2.2);

    //Blinn specular lighting
    //vec3 sun =  vec3(specular) * 2 * self_spec * pow(max( vec3(0.0), dot(H,bump_obj)),
vec3(specular_power) );

  // Fresnel lookup – setting value to max 0.99 since 1 will be mapped back to 0
  vec3 f = texture2D(fresnel, vec2(min(0.99,dot(-R,bump_obj)), 0.0));

   R.z = 0.5*R.z+0.5;
   vec3 env_ = textureCube(enviroment, -R.yzx);

   // Mixing local and global reflections using local reflection alpha value
   vec3 reflect_ = mix(env_+sun, refl_.rgb, refl_.a);

  // Mixing refraction and reflection using fresnel factor
  float fr = mix(0.0, f.r, dot(vEyeVector, tan2obj[2]));
  gl_FragColor = mix(vec4(reflect_, 1.0),refr_color, fr);

}
```

References

[1] Simon Premoze & Michael Ashikhmin (2000) - Rendering Natural Waters,
Eight Pacific Conference on Computer Graphics and Applications, October 2000,
http://www.cs.utah.edu/vissim/papers/water/waterColorPG.pdf

[2] Claes Johanson (2004) - Real-time water rendering introducing the projected grid concept,
Master of Science thesis, Lund University,
http://graphics.cs.lth.se/theses/projects/projgrid/projgrid-lq.pdf

[3] Mark J. Kilgard (2000) - A Practical and Robust Bump-mapping Technique for Today's GPUs,
NVIDIA technical document
http://developer.nvidia.com/object/Practical_Bumpmapping_Tech.html

[4] Matthias Wloka (2002) - Fresnel Reflection,
NVIDIA technical report,
http://developer.nvidia.com/object/fresnel_wp.html

[5] Yann Lombard - Realistic Natural Effect Rendering: Water I,
Gamedev article,
http://www.gamedev.net/reference/articles/article2138.asp

[6] Chris Wynn & Sim Dietrich (2001) - Cube Maps,
NVIDIA technical document,
http://developer.nvidia.com/object/cube_maps.html

[7] Cristopher Oat - Rendering to an off-screen buffer with WGL_ARB_pbuffer,
ATI research document,
http://www.ati.com/developer/ATIpbuffer.pdf

[8] Jens Schneider and Rüdiger Westermann (2001) - Towards Real-Time Visual Simulation of Water Surfaces,
In Proceedings from Conference on Vision, Modelling and Visualization (VM) 2001,
Stuttgart, Germany (p. 211 – 218, p. 525)
http://www.glhint.de/pub/data/VMV01Water.pdf

[9] Randi J. Rost, OpenGL Shading Language, Addison Wesley (2004)

[10] Woo, Neider, Davis, Shreiner - OpenGL Programming Guide, Addison Wesley (1999)

[11] Kristian M. Pettersen (2003) - Bumpmapping
http://www.ia.hiof.no/~borres/gb/exp-bump/p-bump.html

[12] M. Dæhlen, T. Sevaldrud (1999) - Hierarchical structures for real-time terrain rendering,
Preprint

[13] Silent Wings website - www.silentwings.no

[14] Nvidia Adobe Photoshop normal map plugin,
http://developer.nvidia.com/object/photoshop_dds_plugins.html