# Real-time water rendering

## Introducing the projected grid concept

Master of Science thesis

Claes Johanson <claes@vember.net>

March 2004

Lund University

Supervisor:
Calle Lejdfors
Department of Computer Science, Lund University

The thesis was done in cooperation with
Saab Bofors Dynamics
Supervisor at SBD: Magnus Anderholm, RTLI

# Abstract

This thesis will examine how a large water surface can be rendered in an efficient manner using modern graphics hardware. If a non-planar approximation of the water surface is required, a high-resolution polygonal representation must be created dynamically. This is usually done by treating the surface as a height field. To allow spatial scalability, different methods of "Level-Of-Detail" (LOD) are often used when rendering said height field. This thesis presents an alternative technique called "projected grid". The intent of the projected grid is to create a grid mesh whose vertices are even-spaced, not in world-space which is the traditional way but in post-perspective camera space. This will deliver a polygonal representation that provides spatial scalability along with high relative resolution without resorting to multiple levels of detail.

# Contents

# 1 Background

## 1.1 Introduction

The process of rendering a water surface in real-time computer graphics is highly dependent on the demands on realism. In the infancy of real-time graphics most computer games (which is the primary application of real-time computer graphics) treated water surfaces as strictly planar surfaces which had artist generated textures applied to them. The water in these games did not really look realistic, but neither did the rest of the graphics so it was not particularly distracting. Processing power were (by today's standard) quite limited back then so there were not really any alternatives. But as the appearance of the games increase with the available processing power, the look of the water are becoming increasingly important. It is one of the effects people tend to be most impressed with when done right.

Water rendering differ significantly from rendering of most other objects. Water itself has several properties that we have to consider when rendering it in a realistic manner:

- It is dynamic. Unless the water is supposed to be totally at rest it will have to be updated each frame. The wave interactions that give the surface its shape are immensely complex and has to be approximated.
- The surface can be huge. When rendering a large body of water such as an ocean it will often span all the way to the horizon.
- The look of water is to a large extent based on reflected and refracted light. The ratio between the reflected and refracted light vary depending on the angle the surface viewed at.

It is necessary to limit the realism of water rendering in order to make an implementation feasible. In most games it is common to limit the span of the surface to small lakes and ponds. The wave interactions are also immensely simplified. As mentioned in the introductory example, the reflections and refractions are usually ignored, the surface is just rendered with a texture that is supposed to look like water. Neither is it uncommon that water still is treated as a planar surface without any height differences whatsoever. But the state of computer generated water is rapidly improving and although most aspects of the rendering still are crude approximations they are better looking approximations than the ones they replace.

The remainder of this chapter will give a brief overview of how water rendering could be implemented.

## 1.2 The graphics pipeline

### 1.2.1 Overview

Real-time three-dimensional computer graphics is a field of technology that have matured substantially during the last decade. The mid 1990's saw the introduction of hardware accelerated polygonal scan-line rendering aimed at the consumer market. This allowed applications to offload the CPU by moving certain graphics-related operations to the graphics hardware. At first only the pixel processing were handled by the hardware but after a while the hardware started to support vertex processing as well. A problem with the graphics pipeline at this stage was that its functionality was fixed. It was possible to alter some states to control its behaviour but it could not do anything it was not built to do. The last two hardware generations, as of writing, (which support the features added by DirectX 8 and 9 [1]) have replaced this fixed function pipeline with programmable units at key locations, giving a somewhat CPU-like freedom when programming. The latest generation added floating-point capabilities and a significantly increased instruction count to the pixel-pipeline as well, making it even more generic. Although the programmability has increased, the basic data flow is still hardwired for use as a scan-line renderer.

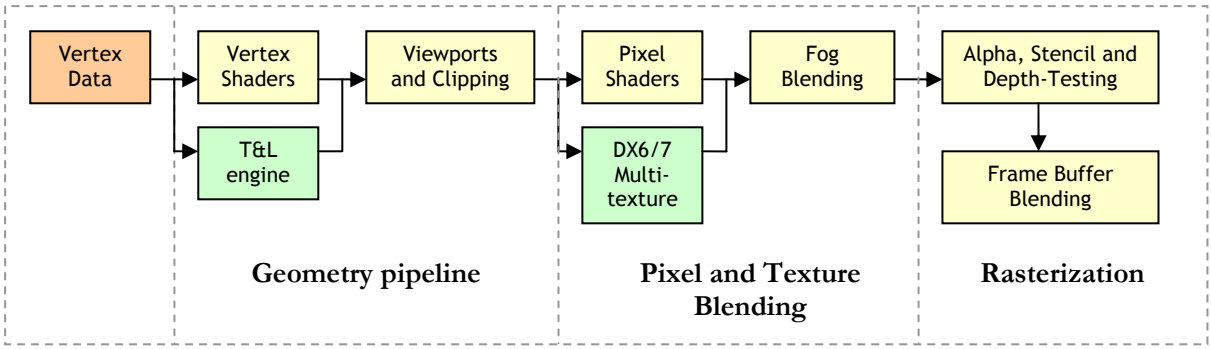



**Figure 1.1 Overview of the DirectX 8 pipeline**

### 1.2.2 Vertex transformations

The traditional vertex transformation used to transform the vertex data from its original state into coordinates on the screen will vary depending on implementation. The principle is that there are several spaces which geometry can be transformed between using 4x4 matrices.

In this thesis the matrix notations from DirectX 9 [1] will be used with the exception that 'projected space' will be referred to as 'post-perspective space' to avoid confusion. A left-handed coordinate system is assumed.

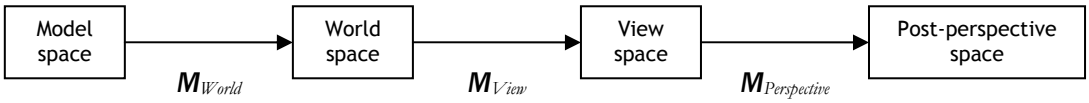



**Figure 1.2 Matrix notation used in the thesis.**

The camera frustum is defined as the volume in which geometry has the potential to be visible by the camera. It is often useful to see a geometric representation of the camera frustum. This can be accomplished by taking its corner points (±1, ±1, ±1) defined in post-perspective space and transform them to world-space.

## 1.3 Height fields

A water body can be represented as a volume. For every point within this volume we are able to tell whether it is occupied by water or not. But all this information is superfluous, as we really only need to know the shape of the surface (the boundary between the water body and the surrounding air) in order to render it. This surface can often be simplified further to a height field ($\boldsymbol{f}_{HF}$). A height field is a function of two variables that return the height for a given point in two-dimensional space. Equation **1.1** show how a height field can be used to displace a plane in three-dimensional space.

$$p_{heightfield}(s,t) = p_{plane}(s,t) + f_{HF}(s,t) \cdot N_{plane} \qquad \textbf{Equation 1.1}$$

Using a height field as a representation for a water surface does have its restrictions compared to a general surface. As the height field only represent a single height-value for each point it is impossible to have one part of the surface that overlap another. Figure 1.3 shows a two-dimensional equivalent of the limitation where the right image cannot represent the part of the curve that overlap.

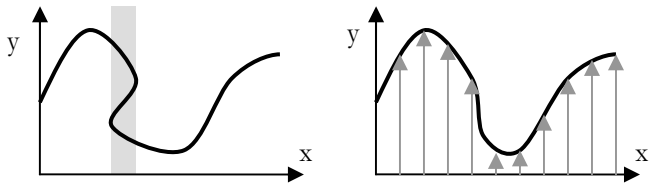



**Figure 1.3 Illustrations of the limitations that apply to height fields. (2D equivalent)**
**left: (x,y) = ($f_x$[s], $f_y$[s])**
**right: (x,y) = (x, f[x])**

The height field has many advantages as it is easy to use and it is easy to find a data structure that is appropriate for storing it. It is often stored as a texture, which is called height map. The process of rendering a surface with height map is usually called displacement-mapping as the original geometry ($\boldsymbol{P}_{plane}$ in Equation **1.1**) is displaced by the amount defined in the height map.

The rest of this thesis will only deal with rendering height fields and not discuss more general surfaces.

## 1.4 Task subdivision

As hardware is built with scan-line rendering in mind there is no way to get around that the surface must have a polygonal representation. Thus we need to approximate the surface with some kind of grid (not necessarily regular). As a height field representation will be used it is appropriate to generate a planar grid first, and then displace the points afterwards.

We can divide the water rendering task into these subtasks:

1. Generate the planar grid (tessellation schemes)
2. Generate the height field and sample the points in the grid
3. Render and shade the resulting geometry

Step 1 and 2 are not strictly separate as the generation of height-data may impose certain restrictions on how the grid must be generated. Neither do they have to be separate in an implementation. But it is good to treat them as separate tasks from an explanatory point of view.

## 1.5 Tessellation schemes

A surface of water is continuous in real life as far as we are concerned. But interactive computer graphics need a polygonal representation so the surface must be discretized. A tessellation scheme is the process of determining in which points the height field should be sampled.

The most straightforward way to represent a surface is to draw it as a rectangular regular grid in world-space.
The grid has a detail-level that is constant in world-space. But due to the perspective nature of the projective part of the camera-transform ($\boldsymbol{M}_{Perspective}$) through which the geometry is rendered, the detail-level in the resulting image will be dependant on the distance to the viewer. While this might work for a small surface like a lake or a pond, it will not be suitable in a scene where an ocean is supposed to cover the entire screen along with a horizon that is far away. The grid would have to be spread over such a large area that the resulting points in the grid would be too sparse to represent the level of detail we want.

The perspective effect have the implication that we can see both parts of the surface that are close, and parts of the surface that are distant simultaneously. In this thesis the distance between the nearest visible point on the surface to the most distant visible point will be referred to as distance dynamic range. A single rectangular grid is good when the dynamic range is small, but for large dynamic ranges, a more sophisticated solution is required.

A common solution is to use what is known as a LOD (level-of-detail) scheme. The surface is divided into a grid of patches. When rendering, the resolution of each patch will be determined by the distance to the viewer. The change in resolution is discrete and each level-of-detail usually has twice the resolution of the

previous one. There are many well developed implementations that are used for height map rendering that derives from the LOD concept which are used for landscape rendering. However, most of them rely on pre-processing to a certain amount. When using pre-processing, each patch does not even have to be a regular grid but can contain any arbitrary geometry as long as it matches the adjacent patches at the edges. However, as water is dynamically generated each frame that makes no sense to rely on any pre-processing all.

Figure 1.4 shows a suggestion how LOD could be used for water rendering. The suggested approach is similar to geomipmapping [6] with the exception that the level determination has been simplified. (Each LOD level has a constant parameter d whereas geomipmapping uses pre-processing to determine d. When the viewer is closer than d a LOD level switch occurs.)

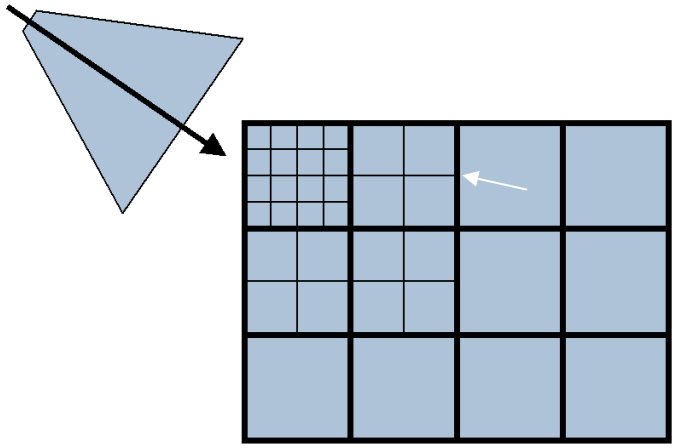


**Figure 1.4 Illustration of the LOD concept. The grid gets sparser further away from the camera.**

The implementation of this LOD-scheme is pretty straightforward. For all potentially visible patches, determine the level of detail depending on the distance between the patch and the camera, sample the height-data for the points in the patch and render the resulting patch.

The LOD-concept is not without difficulties though, as level-switches can cause a couple of artefacts. With a moving camera patches will frequently change level, causing a 'pop'. The pop occurs because the new detail level will look slightly different than the previous one (due to having more/less detail). As the switch is instantaneous it is often rather noticeable. There will also be discontinuities between patches of differing detail-levels (see the white arrow in Figure 1.4, the middle point to the left can have a different height than the interpolated value to the right.). This is usually solved by rendering a so-called "skirt" at the edges between detail-level boundaries and/or to interpolate between the detail-levels to make the switch continuous. The solutions to these issues are mentioned in far more detail in [6] but that is outside the scope of this thesis.

Chapter 2 will introduce a new tessellation-scheme called projected grid that is appropriate for water rendering.

## 1.6 Generating waves

There are a number of different approaches to choose from when generating the height-data for a displaced surface.

The Navier-Stokes Equations (NSE) is the cornerstone in the field of Fluid Mechanics and is defined by a set of differential equations that describe the motion of incompressible fluids. The NSEs are complex beasts, but they can be simplified and discretized to something we can use. In [7] it is suggested that a rectangular grid of columns are used to represent the volume of the water body. For every column, a set of virtual pipes are used to describe the flow of fluid between itself and the adjacent columns.

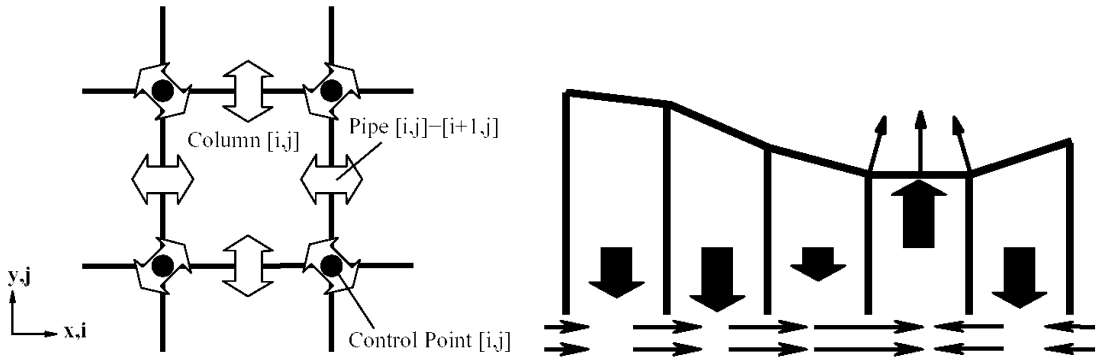



**Figure 1.5 Approximation of the NSE. The images are copied from [7].**

Although they are extremely realistic, the NSEs are somewhat impractical for our intended use. They require a world space grid to be calculated every frame and that grid cannot be big enough to cover an entire sea with enough detail. As the previous state of the surface has to be known, it is also hard to combine this with tessellation schemes that dynamically change their tessellation depending on the position of the viewer. Since the entire process of wind building up the momentum of waves has to be simulated it is unsuitable to open-water waves. A real-time implementation will have to be limited to smaller local surfaces. It is certainly useful in combination with another wave model. That way the NSE only have to take care of the water's response to objects intersecting it, and only at local areas.

With the purpose of rendering large water bodies, it is more practical to have 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 weather conditions. These spectrums can be used to filter a block of 2D-noise by using the Fourier transform. This method can be a computationally efficient way to generate a two-dimensional height map. This is explained in more detail in [4].

The method that was used for the demo implementations in this thesis is a technique called Perlin noise ([5] named after Ken Perlin) which gives a continuous noise. To generate Perlin noise it is required to have a reproducible white noise (a noise which can be accessed multiple times with the same value as the result). This can either be generated by storing white noise in memory or by using a reproducible random function. Perlin noise is obtained by interpolating the reproducible noise. Given the right interpolator, Perlin noise will be continuous. The basic Perlin noise does not look very interesting in itself but by layering multiple noise-functions at different frequencies and amplitudes a more interesting fractal noise can be created. The frequency of each layer is usually the double of the previous layer which is why the layers usually are referred to as octaves. By making the noise two-dimensional we can use it as a texture. By making it three-dimensional we can animate it as well.

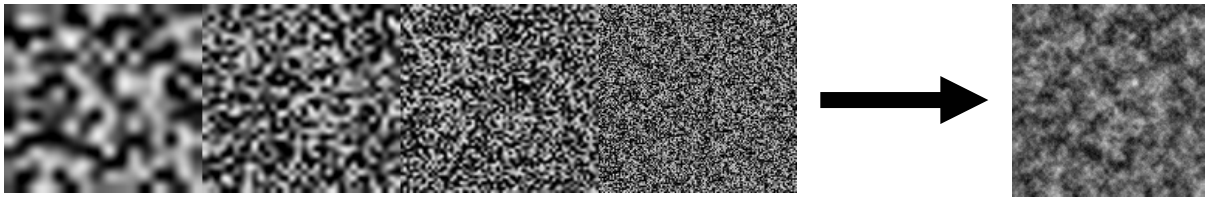


**Figure 1.6 Multiple octaves of Perlin noise sums up to a fractal noise. (α=0.5)**

In Figure 1.6 it can be seen how the sum of multiple octaves of Perlin noise result in a fractal noise. The character of the noise will depend on the amplitude relationship between successive octaves of the noise.

$$fnoise(x) = \sum_{i=0}^{octaves-1} \alpha^i \cdot noise(2^i \cdot x)$$ **Equation 1.2**

Equation 1.2 demonstrates how a fractal noise can be created from multiple octaves of Perlin noise and how all the amplitudes can be represented by a single parameter (α). Higher values for α will give a rougher looking noise whereas lower values will make the noise smoother.

## 1.7 The optics of water

To render a water surface in a convincing manner, it is important to understand how light behaves when we look at water in reality. Water itself is rather transparent, but it has a different index-of-refraction (IOR) than air which causes the photons that pass the boundaries between the mediums to get either reflected or transmitted (see Figure 1.7).

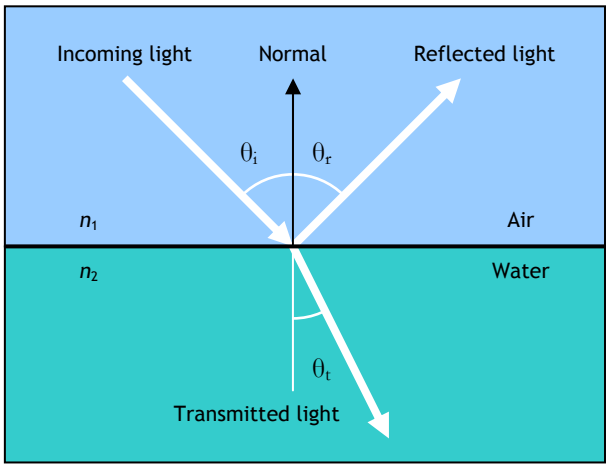



**Figure 1.7 Light transport through a air/water-boundary.**

The angles $\theta_i$ and $\theta_t$ can be obtained from Snell's law (assuming $\theta_i = \theta_1$ , $\theta_t = \theta_2$):

$$n_1 \sin(\theta_1) = n_2 \sin(\theta_2)$$ **Equation 1.3**

Snell's law is used to calculate how $\theta_i$ differs from $\theta_t$ depending on $n_1$ and $n_2$ which are the IORs of the two mediums.

The look of water depends greatly on the angle with which we view it. When looking straight down into the water we can easily see what's beneath the surface as long as the water itself is clear. But when looking towards the horizon it will almost act as a mirror. This phenomenon was studied by Augustin-Jean Fresnel which resulted in the Fresnel equations [2] which allows the reflection coefficient (the probability that a photon is reflected) to be calculated. As the light in most cases is non-polarized the reflection coefficient can be described as:

$$R = \left( \frac{\sin(\theta_i - \theta_t)}{\sin(\theta_i + \theta_t)} \right)^2 + \left( \frac{\tan(\theta_i - \theta_t)}{\tan(\theta_i + \theta_t)} \right)^2$$ **Equation 1.4**

The transmission coefficient (the probability that a photon is transmitted across the boundary into the new medium) is given by:

$$T = (1 - R)$$ **Equation 1.5**

In Figure 1.8 it can be seen graphically how this apply to the water ↔ air boundary.

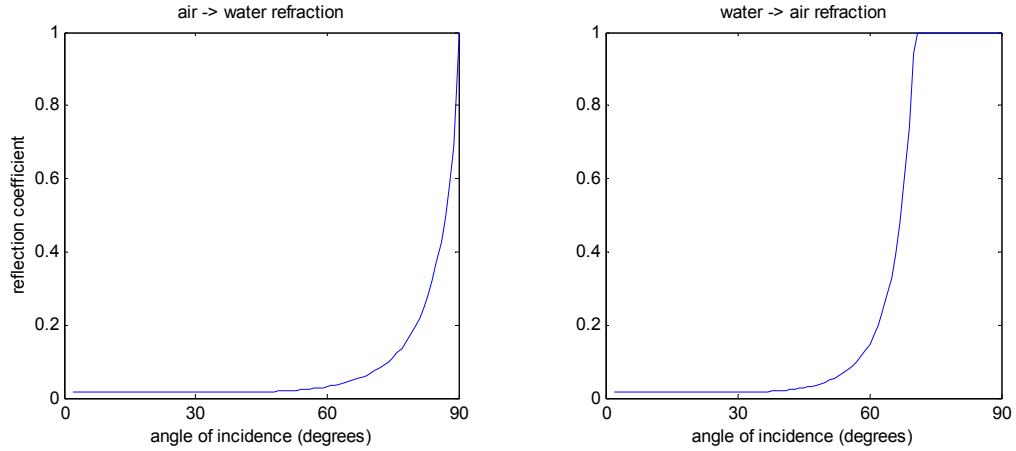



**Figure 1.8 Fresnel reflection coefficient for the air ↔ water boundary**

The IOR of water is dependent on several properties including the wavelength of the light, the temperature of the water and its mineral content. The documented IOR of sweet water at 20°C according to [4] is:

| wavelength | 700 nm (red) | 530 nm (green) | 460 nm (blue) |
|---|---|---|---|
| IOR at 20°C | 1.33109 | 1.33605 | 1.33957 |

As can be seen the differences are quite insignificant, and all three values could be approximated with a single value without any obvious degradation in realism. In fact, when quantitized to three 8-bit values (with the intention of using it as a texture lookup), the three values never differed from each other more than by a value of 1. This was true both in linear colour space and sRGB (the standard windows colour-space with a gamma of 2.2). It is safe to assume that all of these variations can be ignored for our purposes as it is unlikely they will ever cause any noticeable difference.

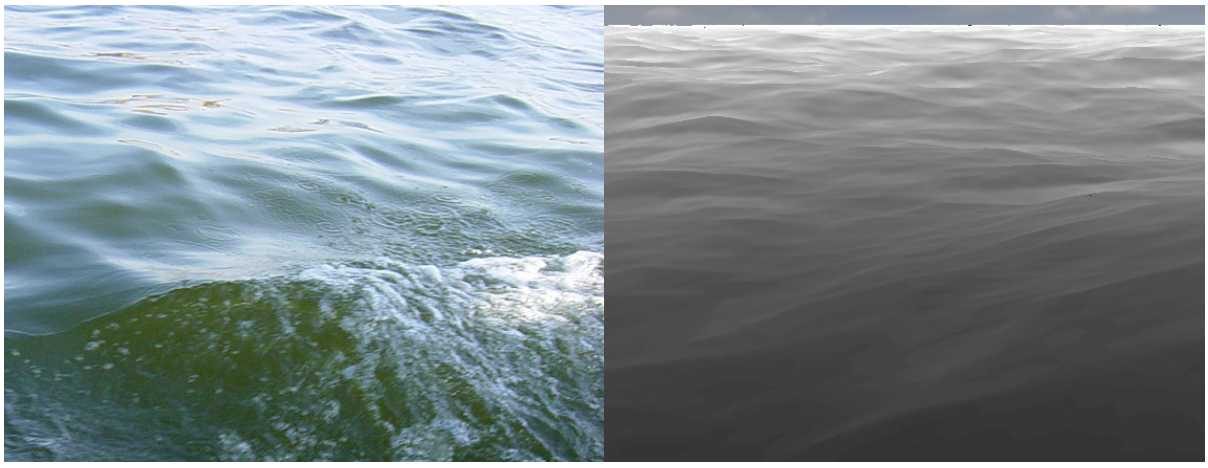


**Figure 1.9 Implication of the Fresnel reflectance term. The left image shows a photo of the Nile where you can see the reflectance vary depending on the normal of the surface. The right image shows a computer generated picture where the Fresnel reflectance for a single wavelength is output as a grey-scale (white = fully reflective, black = fully refractive).**

# 2 Projected Grid

## 2.1 Concept

The projected grid concept was developed with the desire of having a tessellation scheme that would generate a geometric representation of a displaced surface that as closely as possible matched a uniform grid in post-perspective space. What we're going to do is to take the desired post-perspective space grid and run it backwards through the camera's view and perspective transforms.

The "naïve" projected grid algorithm:

- Create a regular discretized plane in post-perspective space that is orthogonal towards the viewer.
- Project this plane onto a plane in world-space ($\boldsymbol{S}_{base}$).
- Displace the world-space vertices according to the height field ($\boldsymbol{f}_{HF}$).
- Render the projected plane

Without displacement the plane would, when rendered, result in the same plane that was created at the first step of the algorithm (as seen from the viewer). The depth coordinates will be different though. The ability to displace the coordinates in world-space is the whole purpose of the projected grid.

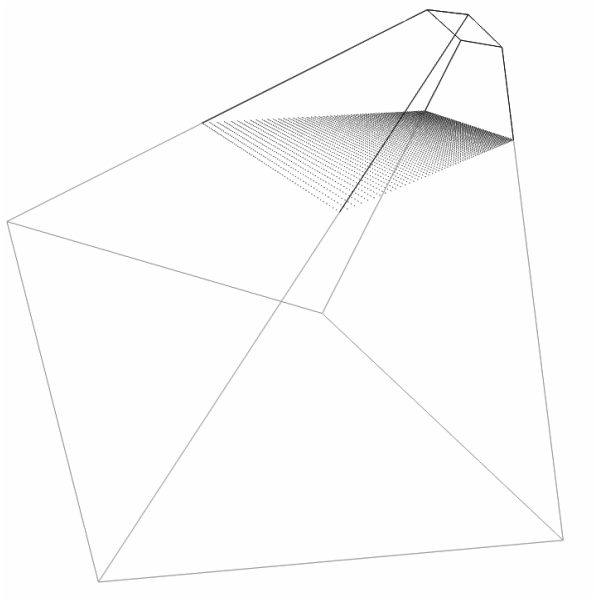


**Figure 2.1 Simple non-displaced projected grid**

The full projected grid algorithm is given in section 2.4. But to make it easier to understand (and motivate) the full algorithm we will showcase what makes the naïve algorithm unsuitable for general use in section 2.3.

## 2.2 Definitions

These are the definitions used in this chapter.

***Notation used:***

***p*** - point in homogenous three-dimensional space
***M*** - 4x4 matrix
***S*** - plane
***N*** - normal
***V*** - volume

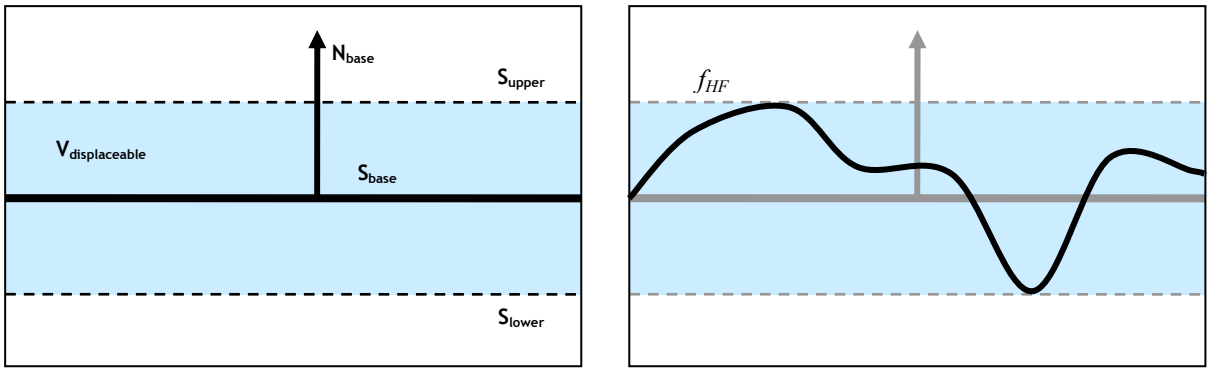



**Figure 2.2 2D illustration of the definitions**

***Base plane: $S_{base}$***
The plane that define the non-displaced placement of the grid in world-space. It is defined by its origin ($\boldsymbol{p}_{base}$) and its normal ($\boldsymbol{N}_{base}$). The points in the grid will be displaced along the normal of the plane, with the distance depending on the height field function ($\boldsymbol{f}_{HF}$).

***Height field function: $f_{HF}(u,v)$***
Given surface coordinates u and v, it returns the value of the height field at that position.

***Upper/Lower bound: $S_{upper}$, $S_{lower}$***
If the entire surface is displaced by the maximum amount upwards then the resulting plane defines the upper bound ($\boldsymbol{S}_{upper}$). The lower bound ($\boldsymbol{S}_{lower}$) is equivalent but in the downward direction.
Together the planes define the allowed range of displacement.

$$\boldsymbol{p}_{upper} = \boldsymbol{p}_{base} + \boldsymbol{N}_{base} \cdot \max(f_{HF}), \quad \boldsymbol{N}_{upper} = \boldsymbol{N}_{base}$$
$$\boldsymbol{p}_{lower} = \boldsymbol{p}_{base} + \boldsymbol{N}_{base} \cdot \min(f_{HF}), \quad \boldsymbol{N}_{lower} = \boldsymbol{N}_{base}$$

**Equation 2.1 Definition of the upper/lower bound planes. (Assuming that $\boldsymbol{N}_{base}$ is of unit length.)**

***Displaceable volume: $V_{displaceable}$***
The volume between the upper and lower bounds.

***Camera frustum: $V_{cam}$***
The frustum of the camera for which the resulting geometry is intended to be rendered by.

***Visible part of the displaceable volume: $V_{visible}$***
This is the intersection of $\boldsymbol{V}_{displaceable}$ and $\boldsymbol{V}_{cam}$.

# 2.3 Preparation

This section is intended to highlight some of the issues that plague the naïve algorithm. The proper projected grid algorithm (as described in section 2.4) will overcome these issues but it is crucial to have an understanding of them in order to understand the algorithm. As a step towards showcasing these issues, a more detailed explanation of the process that transform the grid and perform the projection onto the plane will be given.

## 2.3.1 A real-world analogy

We thought is could be useful to provide a real-world analogy. Imagine having a spotlight that illuminates a flat surface diagonally from above. If you were to put a transparent sheet of paper with a dotted grid on it in front of the spotlight, you would see the grid projected onto the surface.

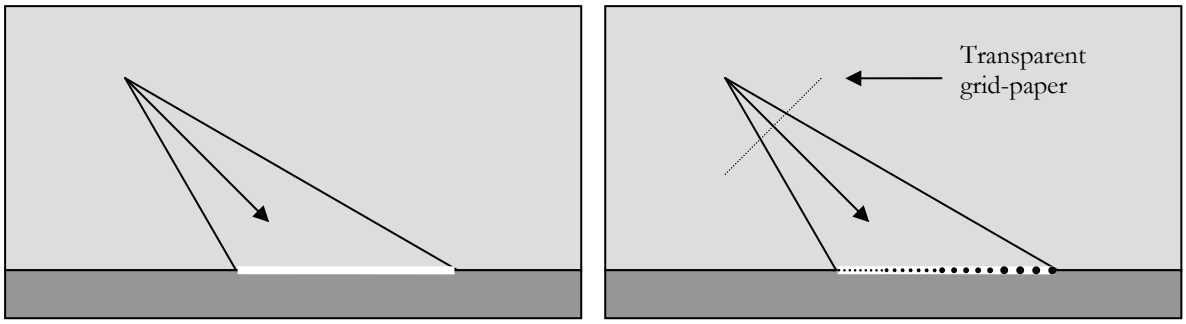



**Figure 2.3 Grid casting shadows on a surface.**

If you were to mark all the points of the grid with a pen, remove the spotlight and put your head in the same position as the light was located in, you would see a grid that would look as if it were right in front of you and not seen from an angle. This is exactly how the "naïve" projected grid algorithm works.

## 2.3.2 The projector matrix

But how does this real-world analogy transfer to math?

A point is transformed from world-space to post-perspective camera space by transforming it through the view and perspective matrices ($\boldsymbol{M}_{View} \cdot \boldsymbol{M}_{Perspective}$) of the camera. This is done for all geometry that is defined in world-space. If that transform is inverted, we obtain the following:

$$\boldsymbol{M}_{\mathrm{Pr}ojector} = \left[\boldsymbol{M}_{View} \cdot \boldsymbol{M}_{Perspective}\right]^{-1} \qquad \textbf{Equation 2.2}$$

$$\boldsymbol{p}_{world} = \boldsymbol{M}_{\mathrm{Pr}ojector} \cdot \boldsymbol{p}_{projector} \qquad \textbf{Equation 2.3}$$

Equations 2.2 and 2.3 show how a point is transformed from post-perspective space to world space. If we consider the z-coordinate (the depth) of each point in post-perspective space to be undefined, there will for each resulting point be a line in which the source point could be located (in world space). This is because the space of the source point has a higher rank. If the resulting line is intersected with the plane ($\boldsymbol{S}_{base}$) we will obtain the projected world-space position.

But as the resulting line is given in homogeneous coordinates (due to the perspective nature of the transform) a division by w is necessary to get the world space position. Appendix A shows how to do the intersection of a line in homogenous coordinates for planes parallel to the xz-plane.

While this approach works fine when the camera is aiming toward the plane, it will wreak havoc when aimed away from it. The resulting point will be where the line and the plane intersect. But because the camera is aimed away from the plane the intersection will occur behind the camera and not in front of it. The camera will 'backfire' so to speak (see Figure 2.4). If the points that are projected are a grid of vertices there will be a discontinuity at the plane's horizon. Discontinuity or not, these points are not supposed to be there in the first place.

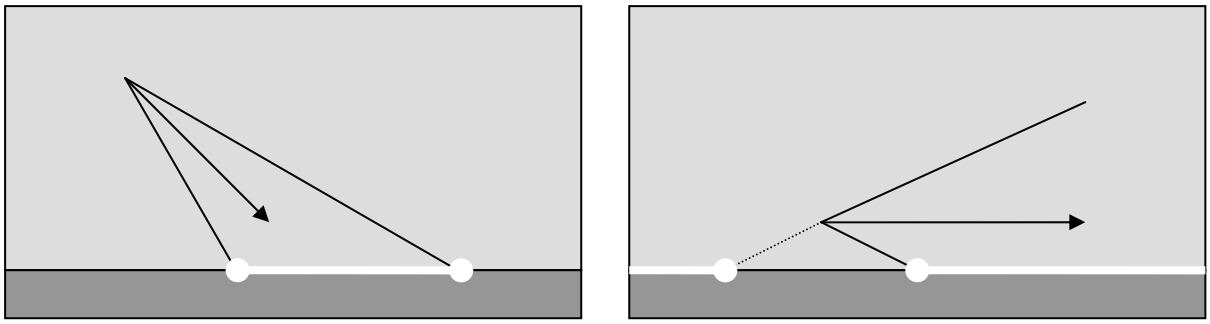


**Figure 2.4 Example of projector backfiring. In the left image everything is good but in the right image the projector has backfired. The points of the white line indicate the start/stop-span of the resulting projection. In the right image the projection will span into infinity on both sides.**

This issue could be fixed by treating the lines as rays instead, since we then would have a defined range where the projection is valid and could discard all polygons including invalid vertices. But it is desired to treat geometrical data on a macroscopical level thus any per-vertex operation is to be avoided. These points are not even doing anything useful so it is best to avoid them completely.

### 2.3.3 What can possibly be seen?

The "naïve" algorithm does not take into consideration whether the resulting geometry is actually supposed to be visible or not. An example where this fails is a scene where the camera is aimed towards the horizon (such as in Figure 2.4). It does not make any sense trying to create the projected grid in the areas of the screen (post-perspective space) where it cannot be visible. It will only cause back-firing as mentioned in section 2.3.2. To be more specific, the "naïve" algorithm will only work properly if the plane ($\mathbf{S}_{base}$) intersects the camera frustum and divides it into two parts without touching the near- or the far-plane (see the left image in Figure 2.5).

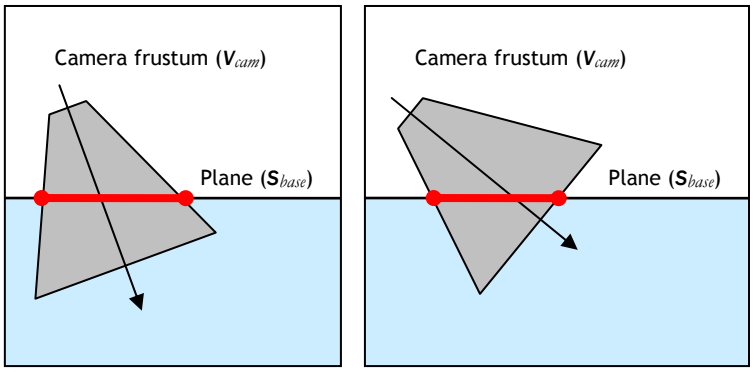



**Figure 2.5 The resulting grid's placement (in 2D) assuming no displacement took place.**

It becomes apparent in the right image in Figure 2.5 that the only part of the surface that has to be rendered is the part of $\mathbf{S}_{base}$ that lies within the camera frustum.

This is however only true if the surface is not displaced. When the surface is displaced and no longer limited to $\mathbf{S}_{base}$ the vertices can, as seen from the cameras viewpoint, be displaced inwards towards the centre of the screen. If this occurs to vertices at the edge of the grid, the edge will become visible and ruin the illusion that the entire surface is drawn. A safety margin must be added to avoid this. Instead of rendering the part of $\mathbf{S}_{base}$ that is within the frustum we must render the part of $\mathbf{S}_{base}$ that possibly could end up within the frustum after displacement has been applied.

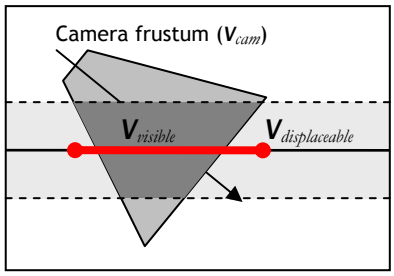



**Figure 2.6 The minimum required span of the projection**

As the distance between $S_{lower}$ and $S_{upper}$ from the cameras point of view increases, the span will become larger to ensure that the resulting geometry goes all way to the edges of the screen. Thus a lesser portion of the processed vertices will actually end up within the frustum. To avoid wasted geometry the span should not be larger than the requirements set by the height data.

It is very important that the resulting span does not intersect the near plane of the camera frustum as that would result in backfiring. This applies to the entire near plane and not just the part of it that is touching the frustum volume. This is dependent on both the position and direction of the camera but a good precaution is to keep the camera outside of $V_{displaceable}$ and always looking below the horizon. Although it will avoid backfiring, this is a serious limitation as the camera must be able to move freely.

## 2.4 The proposed algorithm

It was previously desired to use the inverse of the camera transform to do the projection, but that is not always suitable as the camera has to be able to look freely. The proposed solution is to have a separate projector. This projector will have the same position and direction as the camera, but will be adjusted if necessary to avoid backfiring. The post-perspective space of the projector will be known as projector-space. The placement of the generated grid in projector space will also be set to minimize the area drawn outside the camera-frustum while making sure the entire displaceable volume within the frustum is accounted for.

The proposed algorithm for generating the projected grid is as follows:

1. Get the position and direction of the camera and all other parameters required to recreate the camera matrices.

2. Determine if any part of the displaceable volume is within the camera frustum. Abort rendering of the surface otherwise. See Section 2.4.2 for implementation details.

3. Aim the projector as described in Section 2.4.1. This will provide a new position and direction for the projector (that could be slightly different from the camera). Use the standard "look at"-method to create a view-matrix ($M_{pview}$) for the projector based on the new position and forward vector. This will allow $M_{projector}$ to be calculated. $M_{Perspective}$ is inherited from the rendering camera.

   $$M_{\text{Projector}} = \left[M_{\text{pview}} \cdot M_{\text{Perspective}}\right]^{-1}$$

4. Consider the resulting volume ($V_{visible}$) of the intersection between $V_{cam}$ and $V_{displaceable}$. Calculate the *x* and *y*-span of $V_{visible}$ in projector space. Construct a range conversion matrix ($M_{range}$) that transforms the *[0, 1]* range to those spans. Update $M_{projector}$ with the range conversion matrix:

   $$M_{\text{Projector}} = M_{\text{range}} \cdot \left[M_{\text{pview}} \cdot M_{\text{Perspective}}\right]^{-1}$$

5. Create a grid with *x= [0..1], y = [0..1]*. A pair of texture coordinates *(u, v)* should be set to *u = x, v = y* for later use.

6. For each vertex in the grid, transform it two times with $\boldsymbol{M}_{projector}$, first with the z-coordinates set to -1 and then with the z-coordinate set to 1. The final vertex is the intersection of the line between these two vertices and $\boldsymbol{S}_{base}$. This intersection only has to be done for the corners of the grid, the rest of the vertices can be obtained by using linear interpolation. As long as homogenous coordinates are used, the interpolated version should be equivalent. (see Appendix B for the line-plane intersection of homogenous coordinates)

7. Displace the vertices along $\boldsymbol{N}_{base}$ by the amount defined by the height field ($\boldsymbol{f}_{HF}$) to get the final vertex in world space.

### 2.4.1 Aiming the projector to avoid backfiring

The considerations when moving and aiming the projector are the following.

- Never let the projector aim away from $\boldsymbol{S}_{base}$
- Keep the projector position outside of $\boldsymbol{V}_{visible}$
- Provide an as "pleasant" as possible projector transformation.

Since the $\boldsymbol{M}_{range}$ matrix will select the range within the projection plane where the grid eventually will be generated, it does not really matter whether the resulting grid actually match the projector's frustum.

A "pleasant" projector transformation is one that creates a projected grid that is pleasant to look at, usually attributable to high detail and lack of artefacts. It is desirable that the projection of $\boldsymbol{V}_{visible}$ is as close to an axis-aligned rectangle as possible because that will maximize the vertex efficiency of the solution. The vertex efficiency is defined as the percentage of the process vertices that actually end up within the camera frustum. Obviously, one could make $\boldsymbol{M}_{range}$ represent a non-rectangular transformation for better matching of the grid, but that has not been tested. The following solution has been developed mostly by trial and error. There are probably better ways to do this, but it has proven itself reliable.

First of all the position of the projector is restricted to stay above $\boldsymbol{S}_{upper}$. Due to the symmetry around $\boldsymbol{S}_{base}$ it does not matter if the projector is above or below $\boldsymbol{S}_{base}$ as long as the distance to $\boldsymbol{S}_{base}$ is the same. By keeping the projector above $\boldsymbol{S}_{base}$ at all times we do not have to worry about whether the indexing of the triangles in the grid should be clockwise or counter-clockwise depending on the projector position. As the projector can have a different position than the camera it is preferable to obtain a point (on the plane) for the projector to look at rather than a direction. The solution utilises two different methods for calculating this particular point.

Method 1 aims the projector at the point where the view-vector of the camera intersects the plane ($\boldsymbol{S}_{base}$). If the camera looks away from the plane it will mirror the view-vector against the plane before performing the intersection. Method 2 calculates a point at a fixed distance from the camera in its forward direction. The projector is aimed at this point projected onto $\boldsymbol{S}_{base}$.

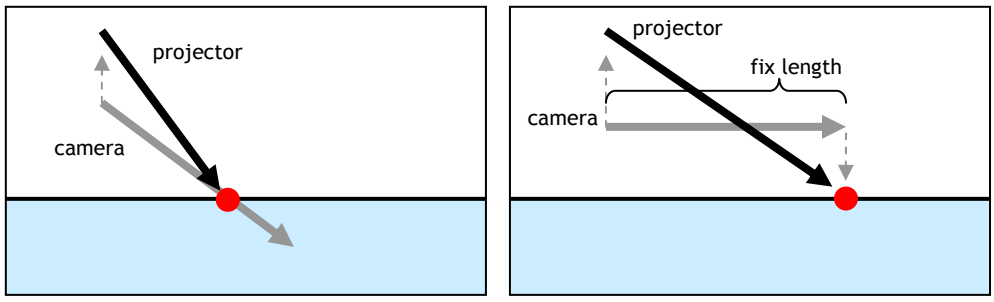



**Figure 2.7 Aiming of the projector using method 1 (left) and method 2 (right).**

Method 1 is better when looking down at the plane, but method 2 is more suitable when looking towards the horizon. The points are interpolated between depending on the direction of the camera.

## 2.4.2 Creating the range conversion matrix

In the algorithm, it is required to calculate the x and y span of $\boldsymbol{V}_{visible}$ in projector space. As mentioned earlier, $\boldsymbol{V}_{visible}$ is the intersection of the camera frustum ($\boldsymbol{V}_{cam}$) and $\boldsymbol{V}_{displaceable}$. The following method shows a suggestion of how this can be done:

- Transform the corner-points (±1, ±1, ±1) of the camera frustum into world-space by using the camera's inverted viewproj matrix.
- Check for intersections between the edges of the camera frustum and the bound planes ($\boldsymbol{S}_{upper}$ and $\boldsymbol{S}_{lower}$). Store the world-space positions of all intersections in a buffer.
- If any of the frustum corner-points lies between the bound-planes, add them to the buffer as well.
- If there are no points in the buffer, then $\boldsymbol{V}_{cam}$ and $\boldsymbol{V}_{displaceable}$ do not intersect and the surface does not have to be rendered.
- Project all the points in the buffer onto $\boldsymbol{S}_{base}$. (Figure 2.8, i-ii)
- Transform the points in the buffer to projector-space by using the inverse of the $\boldsymbol{M}_{projector}$ matrix. The x- and y-span of $\boldsymbol{V}_{visible}$ is now defined as the minimum/maximum x/y-values of the points in the buffer. (Figure 2.8, iii-iv)
- Build a matrix ($\boldsymbol{M}_{range}$) that transform the [0..1] range of the x and y-coordinates into the $[x_{min}..x_{max}]$ and $[y_{min}..y_{max}]$ ranges but leave the z and w values intact.

$$\boldsymbol{M}_{range} = \begin{bmatrix} x_{max} - x_{min} & 0 & 0 & x_{min} \\ 0 & y_{max} - y_{min} & 0 & y_{min} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

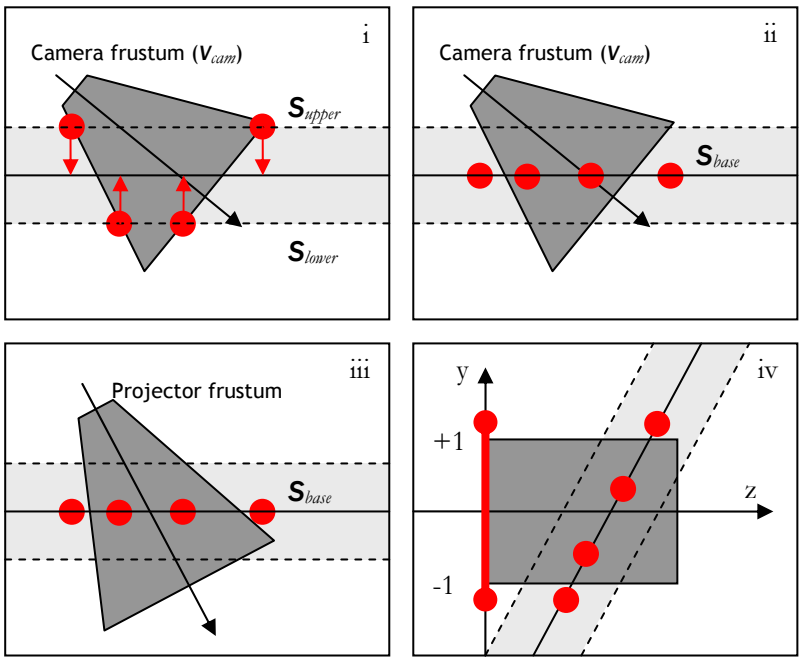



**Figure 2.8 Calculating the xy-span (2D simplification)**

## 2.5 Further restriction of the projector

The projected grid automatically makes the surface detail vary depending on the distance from the camera, which makes it somewhat comparable to a LOD based approach. Contrary to LOD based approaches, the projected grid has no maximum level of detail and will happily sample as detailed as it can because it does not care about detail in world-space, just in projector space. But if the height field (which the projected grid samples) has a limited detail, there is not really any point in creating geometry more detailed than the height field.

Furthermore, when the camera is very close to the surface and looking towards the horizon, the distance dynamic range (the difference between the maximum visible and the minimum visible distance on the surface) is really large. This will cause the projected grid to sample with less detail (in projector space) across the entire range. It is important to realise the difference between projector-space detail and world-space detail. A lower projector-space detail will have the effect that the detail in world-space is lower, but world-space detail is also relative to the distance from the viewer. The world-space detail can still be high compared to parts of the surface that are further away from the viewer. When the camera is close to the surface it will lower the overall projector-space detail in order to raise the world-space detail at close ranges. If that amount of detail is not present in the source data, the reduction of detail at the other distances are in vain. This might manifest it self as the surface having a blurred appearance.

It has already been stated that the projector should be kept outside of $\boldsymbol{V}_{displaceable}$ (Section 2.3.3). If we force it to keep an even larger distance away from the plane we will get a grid whose representation is somewhat in-between a projective and a regular grid (if the projector was forced to an infinite height, the grid would become rectangular). It will not be perfect, as some vertices (at close distances) will end up outside the screen (as many as 50% when really close), but it will avoid blurring across the entire surface. In the demo application there is a parameter labelled 'projector elevation' that lets the user control the minimum height distance from the plane.

There is an additional reason why the distance dynamic range becomes so great when the projector is near the plane. The projector has to project its grid outside the [0..1] range in order to make sure that any part of the displaced surface that reside in $\boldsymbol{V}_{visible}$ will be drawn. The geometry that is added (due to the safety margin) to the part of the grid that is closest to the camera will often increase the distance dynamic range tremendously. Elevating the projector takes care of this issue. This will have an increasing effect as the amplitude of the displacement increases.

It can be seen in Figure 2.9 and Figure 2.10 how the elevation parameter affect the resulting grid. The vertical resolution in the camera's view is increasing with distance when the projector is elevated, but it is apparent that some of the horizontal resolution at close ranges is lost in the process.

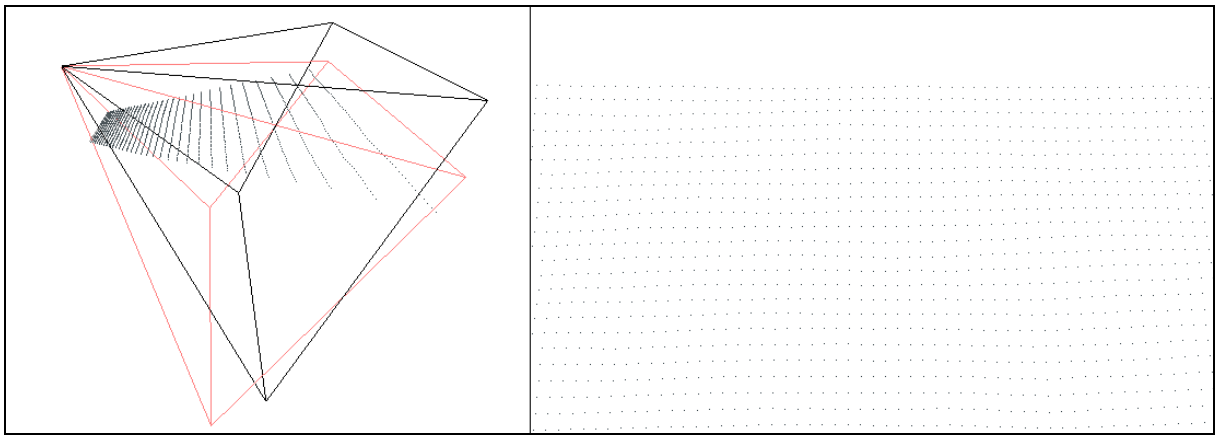


**Figure 2.9 (Left) Projected grid without elevation viewed by an observing camera. The camera's frustum is black and the projector's frustum is red. (Right) The grid as observed from the rendering camera.**

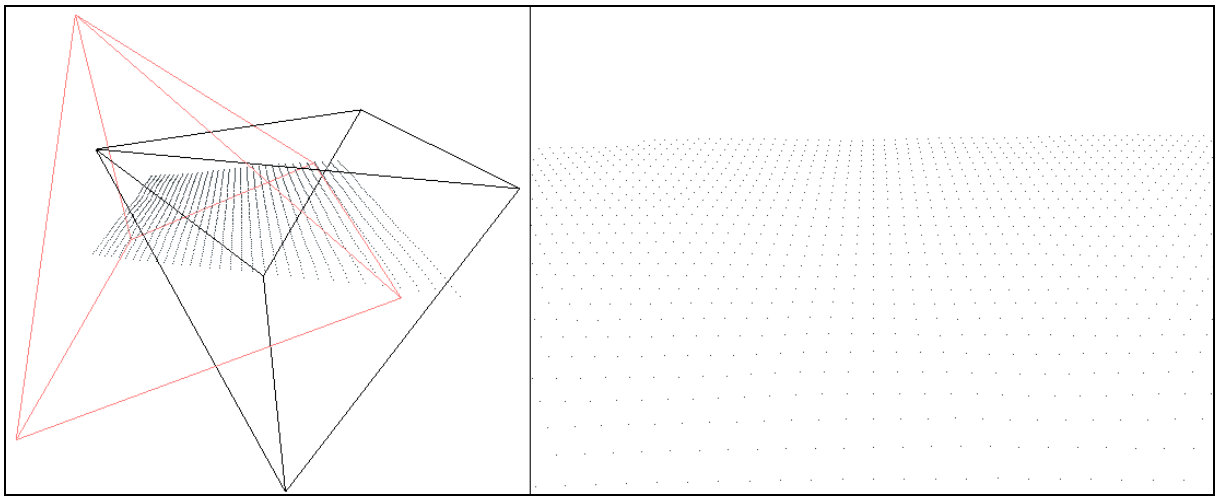


**Figure 2.10 (Left) Projected grid with elevation viewed by an observing camera. (Right) The grid as observed from the rendering camera.**

# 3 Implementation

Depending on the capabilities of the graphics hardware the projected grid algorithm can be implemented in different ways. Since it is designed for vertex processing it can be used on basically any type of hardware as long as the vertex processing is done on the CPU. Due to the rectangular nature of the representation, it can also be implemented using rasterization.

## 3.1 Implementation suggestions

This section provides three suggestions on how the projected grid could be implemented, depending on hardware capabilities. They do not have to be followed exactly. They're just a suggestion of how the work could be divided between the CPU and the graphics hardware.

### 3.1.1 CPU-based vertex processing

The first suggested implementation is as follows:

1. Create a uniform grid with a set of coordinates ($uv$) where $u$ and $v$ span across the $[0\ldots1]$ range. Store $uv$ as a texture coordinate.
2. Using the $uv$-coordinates, calculate the world-space position by interpolating the four corner-points given in step 6 of the algorithm in section 2.4.
3. Displace the world-position along $\mathbf{S}_{base}$'s normal according to $\mathbf{f}_{HF}$ (texture or procedural function).
4. Calculate normals for the surface using the cross product of two vectors that run along the surface. The first vector is between the world-space positions of the adjacent vertices in the $u$-direction of the grid. The second is likewise in the $v$-direction.
5. Render the grid.

This provide the graphics pipeline with positions, per-vertex normals and $uv$-coordinates to the rendering step so suitable shading can be performed. This is pretty much how the algorithm was described in Section 2.4. It requires no special hardware capabilities as all the vertex level processing is done by the CPU. The per-vertex transformations when rendering can obviously be done by a GPU, but the displacement step (and any prior step) requires CPU intervention as it is assumed that the GPU is not capable of reading $\mathbf{f}_{HF}$ by itself. Texture reads have to be implemented in software in order to read $\mathbf{f}_{HF}$.

### 3.1.2 GPU-based vertex processing using 'render to vertex buffers'

There are no consumer video cards at the time of writing that are capable of rendering this proposed implementation as intended, but there are plenty of indications that things will change during the coming months. This proposed implementation is how the projected grid was designed to be used and it is the one that will show its benefits to the fullest. It will give the same result as the CPU-based implementation but the bulk of the processing will be done on the GPU.

As the entire grid is basically a big square, we can process it through pixel-pipelines in a single batch just as we could if we had a single rectangular patch in world-space. The beauty of this method is that this is not just a rectangular patch, it is the entire part of the surface that is visible at the moment.

The suggested implementation is as follows:

1. Create a uniform grid with a set of two-dimensional texture coordinates ($uv$) where $u$ and $v$ are in the $[0\ldots1]$ range.
2. Using the $uv$-coordinates, calculate the world-space position by interpolating the four corner-points given in step 6 of the algorithm in section 2.4. This can be done by a vertex shader but then it will have to be done for each rendering pass that require the world-space position.
3. Render the displaced height map.
    a. Use the position within the grid ($uv$) as position for the rendered vertices.

b. Use the interpolated world-space position as a texture coordinate for the height-data lookup.

4. Render the normal map from the displaced height map.

   a. Use the position within the grid (*uv*) as position and as a texture coordinate.

   b. Offset the texture coordinate by ±1 texels in the u-direction. Obtain the corresponding displaced world-space coordinates and calculate the vector between them.

   c. Likewise in the v-direction.

   d. The normalized cross-product provides the normal.

5. Transfer the height map to a vertex buffer in some manner (or use a texture lookup on the vertex shader when rendering).

6. Render the grid.

This is identical to the method described for vertices except that we use the pixel-pipeline to do our calculations. This does put demands on the GPU, as it must allow high precision render-targets in order to avoid noticeable quantization artefacts. Textures with 8-bits per channel are not enough to get a smooth surface.

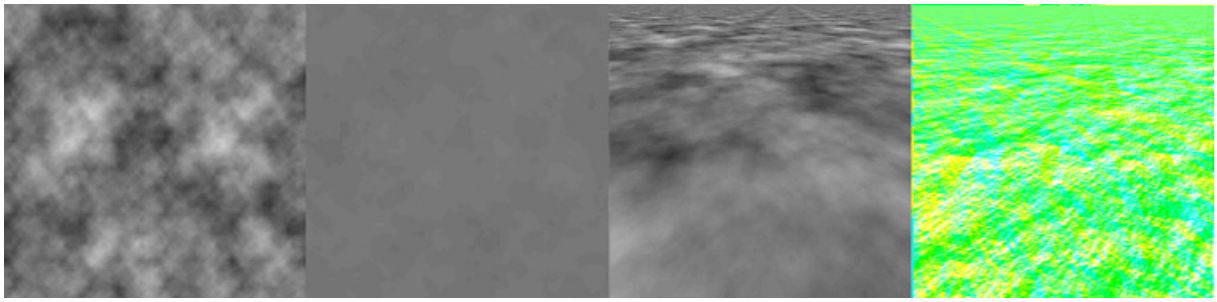


**Figure 3.1 Using the pixel-pipeline to generate a projector space representation.**
**i,ii) two textures containing four octaves of Perlin noise each**
**iii) generated height map**
**iv) generated normal map**
**Image contrast has been increased to improve visibility.**

As the GPU has to support render-targets with at least 16-bit precision per colour-channel this proposed implementation is exclusive to DX9-generation or newer GPUs. But the most severe issue with this proposed implementation is that it is not currently possible to use a vertex buffer as a render target for the pixel-pipeline. Even if the projected grid can be generated by the GPU, there is no way to get it into a vertex buffer without resorting to AGP reads and CPU involvement. This method has been tested successfully on a Radeon 9700 card but with disappointing performance due to pipeline stalls caused by the AGP read.

There are two different upcoming technologies that will remove this restriction. The first one is per-vertex displacement mapping. Matrox Parhelia is supposed to support displacement mapping, but the rest of the pipeline is not up to spec. Radeon 9500-9800 does support pre-sampled displacement mapping but that is not useful for our purposes, it is more suitable for vertex compression. The upcoming hardware generation (NV40 and R420) is assumed to allow texture reads on the vertex shader. However, with the upcoming OpenGL überBuffers extension, it should be possible to use a vertex buffer as a render target, which will allow this implementation to work properly even on the current generation (DX9) hardware. After all, this is more of a memory-management issue than a hardware limitation when it comes to the current generation hardware (Radeon 9500+ and GeForce FX).

The specifics of this suggested implementation is to be taken with a grain of salt as it is dependant on features that are not exposed by the hardware at the time of writing. The best way to do the implementation would differ depending on if the geometric displacement were done using texture lookups

on the vertex shader or by rendering directly to a vertex buffer. The important thing about this suggested implementation is the concept of using the pixel-pipeline to generate the height- and normal maps.

### 3.1.3 CPU-based vertex processing with GPU based normal map generation

The suggested implementation in 3.1.2 is unsuitable when the GPU's pixel-pipeline cannot write to a vertex buffer. But the normals for the surface do not have to be provided on a per-vertex basis, they can be provided by a texture instead. Using this approach, a normal map can be calculated on the GPU as in the previously suggested implementation (3.1.2). The GPU-generated height map is not used in the final rendering step, instead we will use CPU-calculated vertices as done in the first suggested implementation (3.1.1). In effect, this suggested implementation has to do all the work of the previous implementation (except step 5-6) and in addition to that calculate the vertices on the CPU. However the resolution of the CPU-generated grid can be lower than the resolution of the normals since it is not as important to the appearance to have high-resolution geometry as it is to have high-resolution normals.

This implementation is somewhat cumbersome as the height map has to be generated (and thus implemented) twice. However it delivers high-resolution normals at a fast rate and it works on today's hardware.

## 3.2 Height map generation

In all of the suggested implementations, data from a height map is required. The projected grid was designed to work with a method of height field generation that allows random access. If the height field generated require a particular kind of tessellation, the data has to be converted in some manner, most likely by storing it in an intermediary container (read: texture) and reading it by interpolation at the desired locations.

Perlin noise and a FFT-based method were mentioned in Section 1.6 as suitable methods for the creation of a height field that span over a large area. Both these methods are tileable and can be stored to a texture. This means that the lookup of the height field could be done on a pixel-pipeline as a texture read provided that the hardware is up to the task. Linear interpolation is extremely efficient on modern GPUs and mipmapping is provided for free. This is what makes the GPU-based implementations really shine. It is recommended that the resulting height-data is band-limited (not containing higher frequencies than the resulting grid can represent) to avoid aliasing. Using mipmapping is an efficient way of band-limiting the data.

## 3.3 Rendering additional effects into the height map

The height map can be generated by rasterization and at the same time represent the entire visible part of the surface. This gives the opportunity to render additional objects to the height map. Examples include boat wakes, rain-drops and big splashes caused by explosions. These effects could be pre-calculated (as an offset, possibly animated), stored in a signed texture and blended with the render-target using addition. The vertices used to draw these effects should be transformed from world-space to projector-space, but apart from that it's just regular rendering. This can be done at what in most cases would be considered an insignificant performance penalty. If a normal map is calculated from the resulting height map it will automatically be correct.

A software implementation of these effects can be a lot of work since it basically requires the implementation of a simple rasterizer. A GPU implementation on the other hand requires blending support when rendering to high-precision render-targets, which is not supported by any current consumer hardware(at the time of writing). There are workarounds for getting blending to work on current hardware but they're neither particularly efficient nor elegant. Due to these issues none of the mentioned effects were added to the demo implementations.

## 3.4 Shading

This subchapter will give a few brief suggestions on how shading of a water surface can be done using modern hardware equipped with vertex and pixel shaders. Simple HLSL code will demonstrate how the implementation could be done. HLSL is an acronym for High-Level Shader Language which is a c-like shader language that was introduced with DirectX 9.

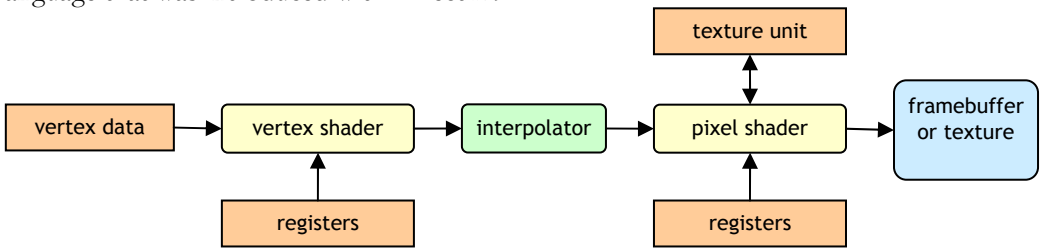



**Figure 3.2 Basic data-flow when using HLSL-shaders**

Shaders are small programs that are executed for every vertex/pixel. The data-flow of the shaders used in DirectX 9 is quite limited. For example, when processing a vertex/pixel the shader has no access to the data of the neighbouring vertices/pixels. Both vertex and pixel shaders do have access to an array of registers that are set by the CPU, but they remain constant during each rendering operation. The output values of the vertex shader are linearly interpolated for each pixel and are available to the pixel shader.

The suggested shaders are not specific for the projected grid algorithm. However, it will be assumed that the vertex stage of the pipeline delivers the following interpolated values to the pixel shader:

- Position
- Normals (unless normal maps are used, in either case they are referred to as N in this section)
- Position within grid (stored as texture coordinates u and v in the suggested algorithm)

Many of the values referenced to in the HLSL-code are constants and should be considered self-explanatory.

### 3.4.1 The view vector

The direction to every point of the surface are required, the general direction of the camera is not sufficient. This can be calculated per-vertex by using the position on the surface and camera position.

```
viewvec = normalize(surface_pos – camera_pos);
```

### 3.4.2 The reflection vector

The reflection vector is the view-vector mirrored against the plane defined by the surface normal. It will tell us from which direction the light we should see reflected in the surface originates from. If per-vertex normals are used then it could be done on the vertex-level but otherwise it should be a per-pixel operation.

```
R = reflect(viewvec,N);
```

### 3.4.3 The Fresnel reflectance term

As mentioned in Section 1.7 the Fresnel reflectance term is essential to achieve realistic shading of water. It is recommended to us the Fresnel term on a per-pixel basis as it can change quite rapidly. The most appropriate implementation is to use a one-dimensional texture as a lookup table.

It is advised to store the lookup texture so that instead of using the angle of incidence for lookup, the dot-product between the normal and reflection vector (alternatively the view vector) could be used. The lookup texture should be clamped to the [0..1] range to avoid wrapping.

```
fresnel = tex1D(fresnelmap, dot(N,R));
```

### 3.4.4 Global reflections

With global reflection we mean reflections from objects that could be considered to be infinitely far away. Because the reflected surface is infinitely far away, which part of that surface we will see is only dependent on the reflection vector of the surface we see the reflection through.
The position of the reflecting surface has no influence. The sky is a splendid example of an object where global reflections are appropriate. Although it is obviously not infinitely far away we can often consider it to be. Unless we travel great distances at great speed, we will not realise that we are moving looking at the sky alone. It is suitable to use a cube map lookup for these reflections.

```
refl_global = texCUBE(sky, R);
```

### 3.4.5 Sunlight

The direct reflection of sunlight is also considered a global reflection and could be put in the cube map used for the sky. However, it is usually better to use the classic per-pixel Phong lighting models specular highlight term instead. The sunlight, whose amplitude usually must be saturated to fit the dynamic range of the texture, will often appear bleak and washed out unless the dynamic range is preserved.

```
refl_sun = sun_strength * sun_colour * pow(saturate(dot(R,sunvec)), sun_shininess);
```

### 3.4.6 Local reflections

While global reflections are a good representation for objects that are part of the environment, they are not a good representation for object within the scene. Local reflections are defined as reflections from objects that cannot be considered to be infinitely far away. Put differently, reflections where the position on the reflecting surface in fact does matter, and the reflection are not only dependant on the angle from where it reflects. A cube map is capable of representing every possible angle of incoming light in a single point, but the reflection across an entire surface cannot be represented correctly using it.

There's no way around that local reflections and scan-line rendering is not a perfect match. Given the way scan-line rendering works, we have to store the possible reflections in a texture to use as a lookup when rendering. The feasibility of this manoeuvre depends on the complexity of the function that defines the reflections and more than anything else, its rank.

A local reflection is a function that depends on at least five variables. Two of them are occupied by the longitude and latitude of the reflected vector, and the other three are the world-space position of the point on the surface that is supposed to reflect its surroundings. Considering we only want to use the reflections on a water surface, we can probably drop one of the dimensions of the surface point's position. But there are still four variables left in the function which is way too many if we're supposed to be able to use it in real-time. As long as we're unable to use ray-tracing we will have to drop another two dimensions.

If we consider the surface to be a flat plane this can easily be accomplished. It will act just like a mirror. We mentioned that two dimensions had to be dropped. Since the mirror is flat, the angular variables do not contribute to the rank anymore since they can be derived from the surface position.
Implementing a mirror using graphics hardware is relatively easy. Just mirror the world-space along the desired plane and add a clip-plane that cuts away everything that is on the backside of the plane. Render the scene to a texture from the cameras point-of-view. When rendering the mirror, the texture can be used to provide the reflection by using a texture lookup with the actual screen coordinates as texture coordinates. It is a good idea to use the alpha channel of the texture to mark which parts of the image that has a local reflection. This way, it can be determined where global reflections should be used instead. This mirror effect has traditionally been done by rendering directly to the frame-buffer using stencilling. The texture approach is more flexible since the texture read coordinates can be altered. The ability to access the reflection on the pixel shader is another advantage compared with traditional blending.
Because it is unfeasible to get correct local reflections on a surface that is not a perfect mirror we will do the next best thing, approximating them. When we read the texture that contains the mirrored scene we can add an offset to the texture coordinates that is based on the xz-components of the surface normal (see the HLSL-snippet below). This will create a rippling distortion effect that is dependant on the normal, and

thus dependant on the waves. The offset is multiplied by a user-defined parameter to control the strength of the distortion. Since this is not a physically correct approximation, the parameter just have to be tweaked until it looks right.

```
offsetting the texture coordinate with the normal:
texcoord = screen_cord.xy + N.xz * rippling_strength;
```

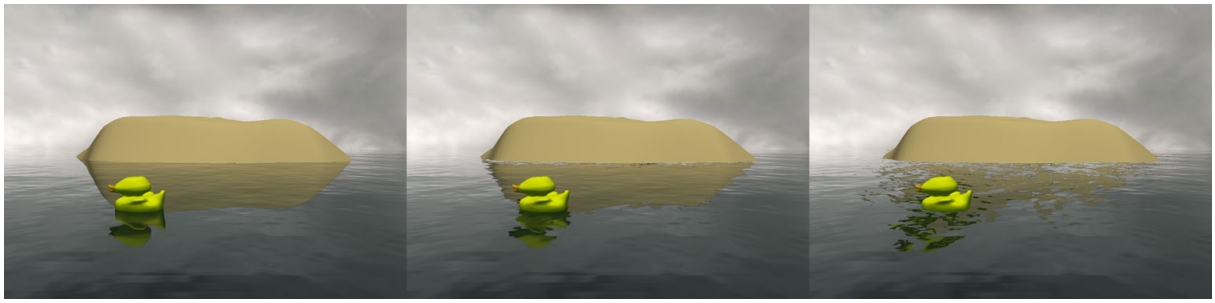


**Figure 3.3 Local reflections approximated as a mirror.**
**(Left) A perfect mirror.**
**(Middle) A sensible amount of distortion have been added.**
**(Right) Too much distortion ruins the illusion.**

It is advised to divide the distortion with the post-perspective z-coordinate, as this will make the distortion distance-dependant and make its post-perspective space magnitude lessen with the distance. We kept the normals in world-space even though they should have been transformed into view-space. Although the reflection was not distorted in the correct direction, it simply looked so chaotic that you would not really notice the difference.

When the distortions applied are too large it will cause artefacts in the image as can be seen in Figure 3.3. This is most noticeable on areas where the geometry intersects the surface as there will be seams on the edge in the reflection. It is recommended to keep the distortion at a sensible level to avoid this.

Since the water surface is actually a displaced surface and not a plane, there is another trick that can be used. Instead of using the actual screen coordinates for the texture lookup it is better to use the screen coordinates that would result if the surface were not displaced. This will give an approximation of the reflection offset caused by the lower frequencies of the height map. It is quite convincing and a good way to avoid that the water surface looks flat, especially when animated. This works well together with the texture coordinate offset-effect mentioned previously.

```
refl_local = tex2D(reflmap, undisplaced_screencoords.xy + refl_strength*N.xz/undisplaced_screencoords.z);
```

### 3.4.7 Refractions

Refractions can be approximated in a similar manner as local reflections. As with the reflections, we will assume that the surface is totally flat and add the ripples by distortion of the texture coordinates later. The process of rendering the refractions to a texture can be done by scaling the scene in the height-direction by 1/1.33, due to the differences of index-of-refraction across the boundary. This is why water always looks shallower when seen from above the surface. A clip-plane should be used to remove everything above the surface.

The distortion of the texture coordinates in the case of refraction can be implemented similarly to reflections. But the artefacts caused by geometry that intersect the surface will be even more noticeable when both reflections and refractions are used. The effect itself looks rather good, and it would work really nice in a scene that does not pay too much attention to intersections of the water. An alternative would be to cover up the edges with foam. If refractions are unwanted just set this variable to a constant colour.

```
refr = tex2D(refrmap, undisplaced_screencoords.xy + refr_strength*N.xz/undisplaced_screencoords.z);
```

### 3.4.8 Putting it all together

Using the Fresnel reflectance term to interpolate between the reflections and refractions and the local reflections alpha-value to interpolate between the global reflections + sunlight and the local reflections the following is obtained.

```
result = (1-fresnel)*refr + fresnel*(refl_local.rgb*refl_local.a + (1-refl_local.a)*(refl_global + refl_sun));
```

## 3.5 Proposed solutions for reducing overdraw

Because the projected grid was primarily developed to render a displaced surface defined as an infinite plane, it relies on the GPU's z-buffer to get sharp edges between itself and intersecting geometry. But although it is impractical to let the projected grid provide hard edges by itself, it is relatively simple to reduce overdraw in the areas where other geometry occlude the grid. It is also possible to limit the span of the surface to prevent it from appearing where it should not be. The volcano scene in Figure 3.4 demonstrates an issue that can appear when the surface is not limited in span. The water should only be placed within the volcano and not outside of it.

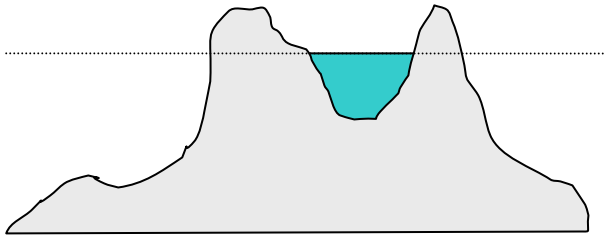


**Figure 3.4 Volcano scene**

### 3.5.1 Patch subdivision

By dividing the grid into a smaller number of patches, it is possible to prevent unnecessary processing to occur. Figure 3.5 looks very similar to approaches that could be used for LOD-based schemes. The main difference is that the division of the grid is done in projector-space, which among other things means that the rendered patches depend on the camera position.

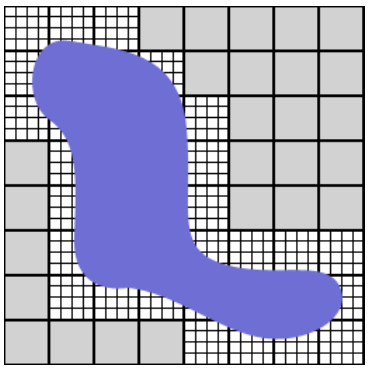


**Figure 3.5 Proposal for reduced overdraw by patch subdivision (seen in projector space). Blue is the visible part of the surface and light-grey are the patches that do not need to be rendered.**

The proposed solution is as follows:

- Subdivide the grid into patches
- For each patch, determine whether the projected grid surface could be visible.
- Render as usual the patches for which it is not certain that the projected grid will be occluded.

If the processing is done on the GPU, the process of rendering a selected set of patches could be achieved by creating an index-buffer that only include the visible patches. The decision whether a certain part of the surface should be visible or not can be made with several different approaches.

One could store low-poly outlines of lakes and other world-space limited water surfaces. The outlines should then be extruded to a volume whose height is depending on the amount of displacement. The resulting volume is transformed to projector space where it is determined which patches that are visible.

Another approach is to determine for each patch if it is occluded by other geometry. This will require a low-poly representation which fit entirely inside the occluding geometry. This might not be easy to obtain depending on the occluding geometry. The fact that the information is needed in projector space complicate matters further.

Patch subdivision will save both vertex and pixel bandwidth, and will give decent clipping which prevent the surface from expanding into unwanted areas. But the precision is limited depending on the number of patches the surface is divided into. As a result it might not be precise enough to make the volcano scene in Figure 3.4 feasible.

### 3.5.2 Using scissors

Scissors could also be used to limit the surface. The approach is simple. Transform the low-poly outlines mentioned in section 3.5.1 to post-perspective camera-space, and determine the span of the screen-coordinates. This will save fill-rate, but has no effect on the vertex-processing. Scissors will give better precision when limiting the span of the surface than the patch subdivision and they are not mutually exclusive so it's reasonable to use both.

### 3.5.3 Using the stencil buffer

The stencil buffer could be used if a more exact limit of the surface is desired. Rendering the outline of the lake inside the top of the volcano while setting the stencil buffer to a desired value will do the trick. If the surface is rendered with stencil testing enabled it will be strictly limited within the volcano. The volcano has to be rendered before the surface for this to work as the stencil pass require its z-buffer content.

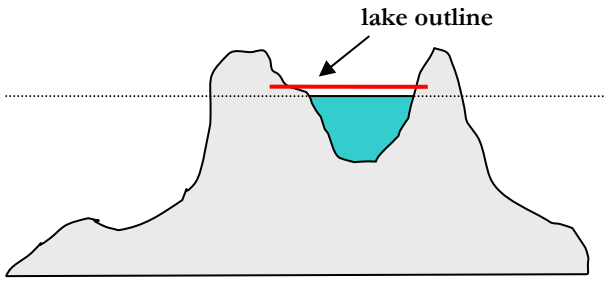



**Figure 3.6 Volcano scene with stenciling**

The outline should be located at the height of $\mathbf{S}_{upper}$ to make sure that the surface is not clipped improperly at the edges.

# 4 Evaluation

Although the title of the paper is "Real-time water rendering" its main focus is the description of the projected grid algorithm.

## 4.1 Implications of the projected grid

For its intended application the projected grid algorithm is efficient. The vertex efficiency (the amount of the processed vertices that end up within the camera frustum) is used as a measurement on how efficient the aiming of the projector is. It is also highly dependent on the height of the waves and position of the viewer. In most cases (and in the pictures that are included Appendix C) the vertex efficiency is usually in the range of 50 – 95 %. In extreme cases the efficiency can drop below 1 %. But these numbers are from the worst-case scenario, looking straight down on a height field with large waves at a low altitude. However, even when the efficiency is that low, it usually does not look bad. It will often give the appearance that the surface is less detailed because the viewer is too close, highlighting the lack of resolution in the source data. This is an extreme example, in most real-world cases the 50-95% figure is representative.

It is obviously not just important how many of the processed vertices that end up on the screen but also how much work that is required for each one. The only work that is required at the vertex-level is to do a linear interpolation of the intersected corner-points, do a texture/function-lookup and displace the vertex by the amount given by the lookup. This excludes the minor overhead of aiming the projector, but that should not have any relevance performance-wise as far as a real-world implementation is concerned. The texture units of modern graphics hardware are perfectly suited for such texture lookups. The demo application is generating a 512x1024 vertex height field complete with normals dynamically at roughly 100 fps on a Radeon 9500 Pro with 16-bit precision. This could certainly be improved further, but it is already a good indication of the performance obtainable. The fact that the amount of vertices is static makes the performance reliable.

However, due to the limitations of the graphics hardware it is not yet possible to generate the vertices entirely on the GPU. The required position and normal data is generated by the GPU in the demo application, but they cannot be used as vertices due to the limitations mentioned in section 3.1.2. This means that the vertex positions must be generated on the CPU which is not nearly as efficient. Consequently, the geometry must be rendered at a lower resolution than the normals to avoid drastically reduced performance.

We can compare the traditional LOD-approach mentioned in section 1.5, which is similar to geo-mipmapping, to the projected grid algorithm. To get rid of the discontinuities it is either required to add the so-called skirts to the detail-level boundaries and/or to interpolate between the detail-levels (in effect tri-linear reads) which counteract the popping artefact as well. A LOD-based approach will probably have some additional overhead due to its chunk-based nature. However, when future GPUs gets the capability of vertex-level texture reads many of the advantages a GPU-implementation of the projected grid enjoy will be given to LOD-techniques as well. The tri-linear filtering required to avoid popping is hardly expensive when performed by hardware and its usage is recommended together with the projected grid nonetheless. A similar bag of tricks are present for either tessellation scheme, the distinction between them is more a matter of how the geometry is distributed and the advantages that originate from the distribution rather than how they are implemented.

The main advantage of the projected grid is that it is continuous. It does not require any extra geometry to cover up changes in detail-levels and such. It does not need tri-linear filtering as badly as LOD-derivatives to avoid popping. It behaves more like traditional texturing in this aspect, where the mip-level switch is visible as a line. Depending on the relative magnitude of the displacement in the post-perspective space, the projected grid can be very efficient. In the extreme case, where the displacement approaches zero, the projected grid can claim 100% vertex efficiency in a good number of situations. This is obviously dependant on how the aiming of the projector is performed, but it is still a number a LOD-based

approach cannot reach unless you're cheating (by looking straight down, all rotations axis-aligned and making sure the camera frustum intersect the base plane only on chunk boundaries).

The LOD-based approach will render many chunks that are only partially within the camera frustum. Hence it is rather easy to grasp that the projected grid is more efficient than LOD-based algorithms when the displacement is small. But for larger displacements this is no longer true. The projected grid has to include an increasingly larger guard area to make sure that it renders everything within the frustum, and this will quickly lower the efficiency. A LOD-derived approach can have min/max-range of the displacement for every chunk and do not have to suffer due to a guard area as much as the projected grid do since the projected grid has to share the same min/max-range for the entire visible part of the surface.

Another disadvantage of the projected grid algorithm is that it suffers from the artefact known as swimming, which is somewhat similar (and related) to aliasing. When the camera is moving, the locations of the points of the grid (and consequently the sampled height-data) will change. Swimming has a look which reminds of that of sheet of fabric is lying on top of the real height field. When the fabric is moved, it will accentuate different parts of what lie beneath, and reveal that there is more information than what is visible. But this will not be that very noticeable if the detail of the grid is high enough. It is even less relevant if the surface is animated as that hides it pretty well. But this is still, along with the issue of not handling large displacements too gracefully, a reason why the projected grid algorithm is not the ideal choice for landscape rendering.

Yet another disadvantage is that the projected grid is slightly more complicated to restrict in world-space since it cannot exactly be limited to certain world-space span as its span is defined in projector-space. In a landscape rendering engine it can therefore be hard to use the projected grid for water rendering. However, if a scene was properly designed to include a lake and authored to avoid the pitfalls by using the techniques described in section 3.5 then it should not be an issue.

A definitive advantage of the projected grid is the simplicity that is obtained from it being a single rectangular grid. This gives the programmer a lot of freedom when implementing the projected grid algorithm and it also makes the algorithm hardware friendly. The projector space is a very useful representation which can be used for many other things than just vertex calculations. In the demo application it is used to calculate the height field and the normal map of the surface. But it is not limited to this. There are several situations where it is useful to be able to render things to a texture that represents the entire visible part of a plane. As mentioned in section 3.3 it could be of great use to render height-offset based effects into a height map. The ability to have a single texture that represent the entire surface could be used for other things as well. An example would be a texture map where objects that float on the water that are represented by a colour value. One could render spots of oil, foam or other entities that lies on the surface to this texture. This would then be accessed by pixel shaders (through a texture read) during the actual surface rendering pass. The alternative is to render another pass which might not be trivial as you might end up rendering the entire surface once again anyway. Besides, you might have z-biasing issues and will lose the capabilities given by the pixel shaders over frame-buffer blending.

## 4.2 Future work

The aiming of the projector could have been better. It tries to fit everything inside a rectangular span, and that is not always appropriate. A good example of this is when the camera is located close to the surface and is aimed towards the horizon. But it is crucial to remember that not only is it important to have a high vertex efficiency, it is also important to put those vertices where it counts. It's not trivial to develop an analytical model that says where the vertices should be located in order to look good.

There is also the question whether the algorithm should be generalised further. In the implementation it is assumed most of the time that the plane is parallel to the xz-plane. While there are no fundamental differences to a general plane, the simplification makes the implementation easier and more efficient. But does it have to be limited to planes? Projecting a grid onto a sphere could prove itself interesting. This would give a rendering a nice curvature towards the horizon. But it would also come with its own set of

issues. Intersecting a line with a sphere gives two points of intersection which should not that much of a problem, but what if the line misses the sphere entirely?

## 4.3 Conclusion

Water rendering is something that has become more interesting during the last years as newer generations of graphics hardware has made it possible to approximate the shading of water in an increasingly realistic manner. Water has often been troublesome as it did not directly apply to the standard lighting models that were traditionally used. Nowadays it is feasible to consider the Fresnel reflectance per-pixel when shading the surface and it is viable to provide reasonably realistic reflections and refractions. The processing power has increased to the degree that a surface no longer has to be approximated as a flat plane. It can even be rendered with very high detail.

The projected grid algorithm that is presented in this paper shows an alternative way to render displaced surfaces which can be very efficient. It is however, just another tool in the box for the programmer to consider as its efficiency depend on the application. But to summarize the usage of the projected grid for rendering a displaced surface will likely be beneficial if:

- The surface have small to medium displacements. They can be clearly visible but the surface should not be displaced by a significant distance compared to the size of the screen.
- The surface is animated, as that hides the swimming artefact.
- It is beneficial to alter the height map directly.
- Constant performance is desired.
- Graphics hardware can be used for the processing.
- The surface spans over a large area in world-space.

# Appendix A - Line - Plane intersection with homogenous coordinates

The line (x+Δx ·t, y+Δy ·t, z+Δz ·t, w+Δw ·t) is intersected with the plane y/w = h. (for simplicity's sake)

t is given by the following equality:

$$\frac{y+\Delta y \cdot t}{w+\Delta w \cdot t} = h \quad \Leftrightarrow \quad t = \frac{w \cdot h - y}{\Delta y - \Delta w \cdot h}$$

The point of intersection in homogenous coordinates is (x+Δx ·t, y+Δy ·t, z+Δz ·t, w+Δw ·t).

# Appendix B - The implementation of the demo application

The demo implementation that was developed during the work on this thesis was implemented using the implementation suggested in Section 3.1.3. It was implemented using DirectX 9 and the target platform is the ATi R300-class of GPUs (Radeon 9500 or higher).

It is available at the following URL:
http://graphics.cs.lth.se/theses/projects/projgrid

The animated Perlin noise is rendered on the CPU. The base resolution of each octave is 32x32 pixels and four octaves are stacked together in texture chunks with the resulting resolution of 256x256 pixels. Two of these chunks were used for a total of 8 octaves of noise.

The plane was forced to the xz-plane for simplicity's sake.

The vertices were (at default) calculated at a grid with the resolution of 65x129 vertices, resulting in 64x128x2 triangles. The line-plane intersections were only done at the corner vertices, the rest were interpolated using homogenous coordinates. The height-data was read by linear interpolation (but without mip-mapping) on the CPU before the vertex buffer were uploaded to the GPU as world-space coordinates.

The two chunks of Perlin noise was uploaded to the GPU and the same height map was generated on the GPU but with the resolution of 512x1024 pixels. Aliasing was avoided by using mip-mapping. The normal map was generated from and at the same resolution as the height map.

A simple demo scene consisting of an island and a moving rubber duck was mirrored against $\mathbf{S}_{base}$ and rendered to a texture for use as local reflections. The alpha channel was later used to determine which parts of the texture that was transparent.

A sky dome, the demo scene and the water surface was rendered to the back-buffer.

The water rendering was done by doing the following computations per-pixel:

- Using the normal map and the view-direction (at every point, not just the direction of the camera) the reflection vector was calculated.
- Using a cube map texture read the reflected part of the sky was sampled ($R_G$).
- The Fresnel reflectance (f) was obtained by doing a 1D texture read of the dot-product of the reflection vector and the normal.
- The sunlight contribution ($R_S$) was calculated with phong shading using the normal map and a light-direction vector.
- The local reflections ($R_L$) was approximated (cannot be done properly without ray tracing) by offsetting the texture coordinate with the xz-part of the normal.
- The refraction colour was set to a constant colour. ($C_W$)
- The output colour was: $C = (1-f) \cdot C_W + f \cdot (R_L.rgb \cdot R_L.a + (1- R_L.a) \cdot (R_G+R_S))$

A simpler implementation for the Nebula2 engine was also created. It's named nPGwater and should be available at the same location as the DX9-demo. It uses per-vertex normals and has no local reflections but is otherwise similar to the DirectX 9 demo implementation.

*HLSL-code for the water surface rendering:*

```
float4x4        mViewProj;
float4x4        mView;
float4          view_position;
float3          watercolour;
float           LODbias;
float           sun_alfa, sun_theta, sun_shininess, sun_strength;
float           reflrefr_offset;
bool            diffuseSkyRef;

texture EnvironmentMap, FresnelMap, Heightmap, Normalmap, Refractionmap, Reflectionmap;

struct VS_INPUT
{
   float3 Pos                   : POSITION;
        float3 Normal           : NORMAL;
        float2 tc               : TEXCOORD0;
};

struct VS_OUTPUT
{
   float4  Pos                  : POSITION;
        float2  tc              : TEXCOORD0;
        float3  normal          : TEXCOORD1;
        float3  viewvec         : TEXCOORD2;
        float3  screenPos       : TEXCOORD3;
        float3  sun             : TEXCOORD5;
        float3 worldPos         : TEXCOORD6;
};

samplerCUBE sky = sampler_state
{
   Texture = <EnvironmentMap>;
   MipFilter = NONE;    MinFilter = LINEAR;    MagFilter = LINEAR;
   AddressU  = WRAP;    AddressV  = WRAP;    AddressW  = WRAP;
};

sampler fresnel = sampler_state
{
   Texture = <FresnelMap>;
   MipFilter = NONE;    MinFilter = LINEAR;    MagFilter = LINEAR;
   AddressU  = CLAMP;    AddressV  = CLAMP;
};

sampler hmap = sampler_state
{
   Texture = <Heightmap>;
   MipFilter = LINEAR;    MinFilter = LINEAR;    MagFilter = LINEAR;
   AddressU  = CLAMP;   AddressV  = CLAMP;
};

sampler nmap = sampler_state
{
   Texture = <Normalmap>;
   MipFilter = LINEAR; MinFilter = LINEAR; MagFilter = LINEAR;
   AddressU  = CLAMP;  AddressV  = CLAMP;
};

sampler refrmap = sampler_state
{
   Texture = <Refractionmap>;
   MipFilter = LINEAR;    MinFilter = LINEAR;    MagFilter = LINEAR;
   AddressU  = CLAMP;              AddressV  = CLAMP;
};

sampler reflmap = sampler_state
{
```

```
    Texture = <Reflectionmap>;
    MipFilter = LINEAR;     MinFilter = LINEAR;     MagFilter = LINEAR;
    AddressU  = CLAMP;          AddressV  = CLAMP;
};

/* DX9 class shaders */

VS_OUTPUT VShaderR300(VS_INPUT i)
{
        VS_OUTPUT   o;
        o.worldPos = i.Pos.xyz/4;
        o.Pos = mul(float4(i.Pos.xyz,1), mViewProj);
        o.normal = normalize(i.Normal.xyz);
        o.viewvec = normalize(i.Pos.xyz - view_position.xyz/view_position.w);

        o.tc = i.tc;

        // alt screenpos
        // this is the screenposition of the undisplaced vertices (assuming the plane is y=0)
        // it is used for the reflection/refraction lookup
        float4 tpos = mul(float4(i.Pos.x,0,i.Pos.z,1), mViewProj);
        o.screenPos = tpos.xyz/tpos.w;
        o.screenPos.xy = 0.5 + 0.5*o.screenPos.xy*float2(1,-1);
        o.screenPos.z = reflrefr_offset/o.screenPos.z;                  // reflrefr_offset controls
                                                                        //the strength of the distortion

        // what am i doing here? (this should really be done on the CPU as it isn't a per-vertex operation)
        o.sun.x = cos(sun_theta)*sin(sun_alfa);
        o.sun.y = sin(sun_theta);
        o.sun.z = cos(sun_theta)*cos(sun_alfa);
        return o;
}


float4 PShaderR300(VS_OUTPUT i) : COLOR
{
    float4 ut;
        ut.a = 1;
        float3 v = i.viewvec;

        // depending on whether normals are provided per vertex or via a normal map, N is set differently
        //float3 N = i.normal;
        float3 N = 2*tex2D(nmap,i.tc)-1;

        float3 R = normalize(reflect(v,N));
        R.y = max(R.y,0);
        float4 f = tex1D(fresnel,dot(R,N));
        float3 sunlight = pow(sun_strength*pow(saturate(dot(R, i.sun)),sun_shininess)*float3(1.2, 0.4, 0.1), 1/2.2);
        float4 refl = tex2D(reflmap,i.screenPos.xy-i.screenPos.z*N.xz);
        float3 skyrefl;
        skyrefl = texCUBE(sky,R);
        float3 col = lerp(skyrefl+sunlight,refl.rgb,refl.a);
        float3 refr = watercolour;                                              // constant colour but
        //float3 refr = tex2D(refrmap,i.screenPos.xy-i.screenPos.z*N.xz);   // refraction could be used instead
        ut.rgb = lerp(refr, col, f.r);

    return ut;
}
```

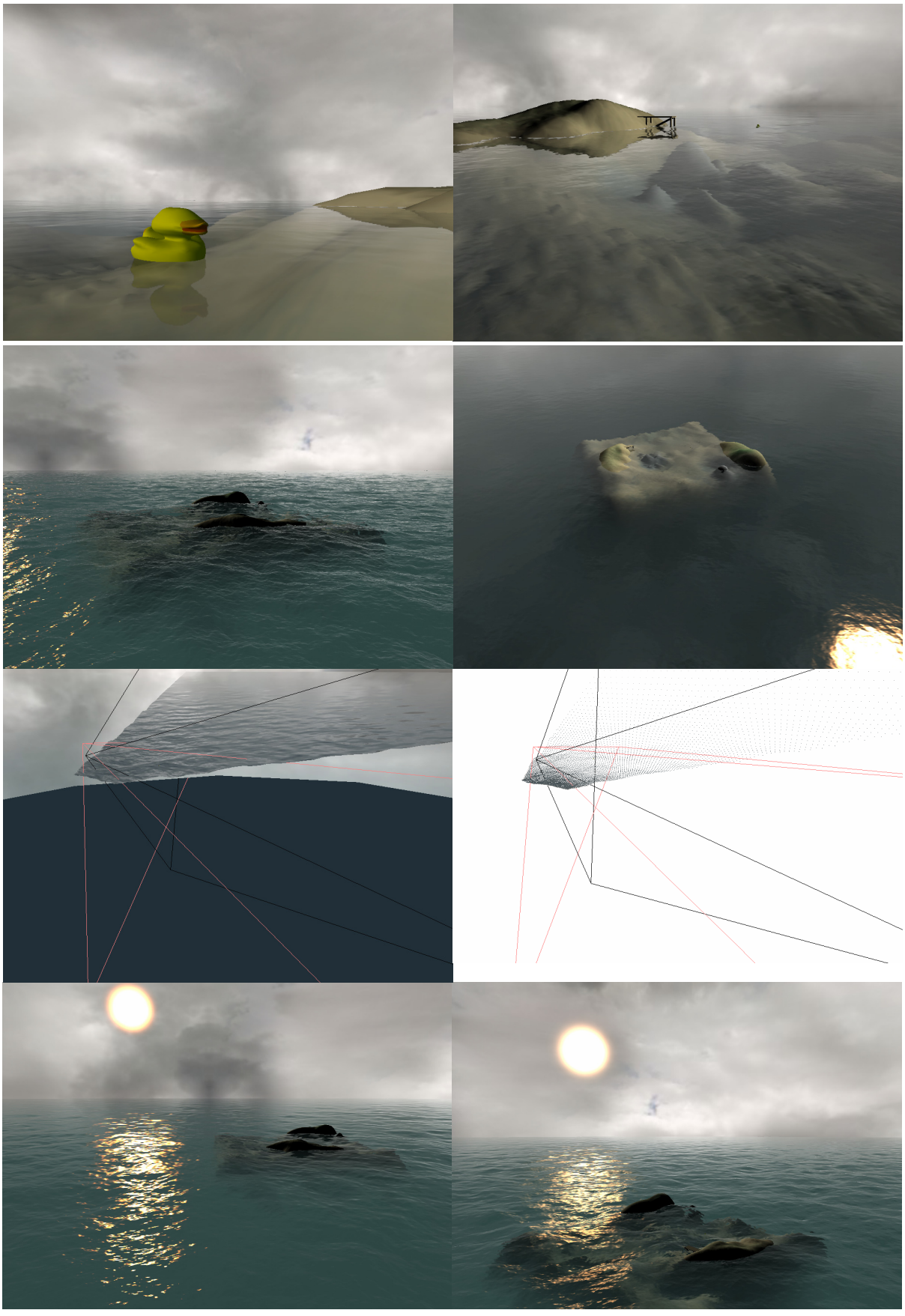
***HLSL-code for the heightmap generation (generates hmap):***

```
float           scale;  // the xz-scale of the noise

struct VS_INPUT
{
   float3  Pos    : POSITION;
   float2  tc     : TEXCOORD0;
};

struct VS_OUTPUT
{
        float4  Pos                 : POSITION;
        float2  tc0                 : TEXCOORD0;
        float2  tc1                 : TEXCOORD1;
};

texture noise0;
texture noise1;

// samplers
sampler N0 = sampler_state
{
        texture = <noise0>;
        AddressU  = WRAP;
        AddressV  = WRAP;
        MIPFILTER = LINEAR;
        MINFILTER = LINEAR;
        MAGFILTER = LINEAR;
        MipMapLodBias = -1;
};
sampler N1 = sampler_state
{
        texture = <noise1>;
        AddressU  = WRAP;
        AddressV  = WRAP;
        MIPFILTER = LINEAR;
        MINFILTER = LINEAR;
        MAGFILTER = LINEAR;
        MipMapLodBias = -1;
};

VS_OUTPUT VShader(VS_INPUT i)
{
        VS_OUTPUT o;
        o.Pos = float4( i.tc.x*2-1,1-i.tc.y*2, 0, 1 );
        o.tc0 = scale*i.Pos.xz*0.015625;
        o.tc1 = scale*i.Pos.xz*0.25;
        return o;
}


float4 PShader(VS_OUTPUT i) : COLOR
{
        return tex2D(N0, i.tc0) + tex2D(N1, i.tc1) - 0.5;
}
```

### *HLSL-code for the normalmap generation (generates nmap)*

```
float    inv_mapsize_x,inv_mapsize_y;
float4   corner00, corner01, corner10, corner11;
float    amplitude;      // the amplitude of the noise.. this determine the strength of the normals

struct VS_INPUT
{
   float3  Pos    : POSITION;
   float2  tc     : TEXCOORD0;
};

struct VS_OUTPUT
{
        float4  Pos             : POSITION;
        float2  tc              : TEXCOORD0;
        float3  tc_p_dx         : TEXCOORD1;
        float3  tc_p_dy         : TEXCOORD2;
        float3  tc_m_dx         : TEXCOORD3;
        float3  tc_m_dy         : TEXCOORD4;
};

texture hmap;

sampler hsampler = sampler_state
{
        texture = <hmap>;
        AddressU  = WRAP;
        AddressV  = WRAP;
        MIPFILTER = NONE;
        MINFILTER = LINEAR;
        MAGFILTER = LINEAR;
};

VS_OUTPUT VShader(VS_INPUT i)
{
        VS_OUTPUT o;
        o.Pos = float4( i.tc.x*2-1,1-i.tc.y*2, 0, 1 );
        float scale = 1;

        float2 tc = i.tc + float2(-inv_mapsize_x*scale,0);
        float4 meh = lerp(lerp(corner00,corner01,tc.x),lerp(corner10,corner11,tc.x),tc.y);
        o.tc_m_dx = meh.xyz/meh.w;
        tc = i.tc + float2(+inv_mapsize_x*scale,0);
        meh = lerp(lerp(corner00,corner01,tc.x),lerp(corner10,corner11,tc.x),tc.y);
        o.tc_p_dx = meh.xyz/meh.w;
        tc = i.tc + float2(0,-inv_mapsize_y*scale);
        meh = lerp(lerp(corner00,corner01,tc.x),lerp(corner10,corner11,tc.x),tc.y);
        o.tc_m_dy = meh.xyz/meh.w;
        tc = i.tc + float2(0,inv_mapsize_y*scale);
        meh = lerp(lerp(corner00,corner01,tc.x),lerp(corner10,corner11,tc.x),tc.y);
        o.tc_p_dy = meh.xyz/meh.w;

        o.tc = i.tc;
        return o;
}


float4 PShader(VS_OUTPUT i) : COLOR
{
        float2  dx = {inv_mapsize_x,0},
                dy = {0,inv_mapsize_y};
        i.tc_p_dx.y = amplitude*tex2D(hsampler, i.tc+dx);
        i.tc_m_dx.y = amplitude*tex2D(hsampler, i.tc-dx);
        i.tc_p_dy.y = amplitude*tex2D(hsampler, i.tc+dy);
        i.tc_m_dy.y = amplitude*tex2D(hsampler, i.tc-dy);
        float3 normal = normalize(-cross(i.tc_p_dx-i.tc_m_dx, i.tc_p_dy-i.tc_m_dy));
        return float4(0.5+0.5*normal,1);
}
```

# Appendix C – Images



# Terms used

Spaces:

World space

View space

Post-perspective space

Projector space – the space defined by world-space transformed by $\boldsymbol{M}_{projector}$

Matrices:

$\boldsymbol{M}_{View}$ view matrix of the camera (transforms from world to view space)

$\boldsymbol{M}_{Perspective}$ projection matrix of the camera (transforms from view to Post-perspective space)

$\boldsymbol{M}_{viewproj}$ $\boldsymbol{M}_{viewproj} = \boldsymbol{M}_{View} \cdot \boldsymbol{M}_{Perspective}$

$\boldsymbol{M}_{pview}$ view matrix of the projector (treating the projector as a camera object)

$\boldsymbol{M}_{range}$ range-scaling matrix of the projector (see section 2.4.2)

$\boldsymbol{M}_{projector}$ $\boldsymbol{M}_{\text{Projector}} = \boldsymbol{M}_{\text{range}} \cdot \left[\boldsymbol{M}_{\text{pview}} \cdot \boldsymbol{M}_{\text{Perspective}}\right]^{-1}$,

Abbreviations:

| | |
|---|---|
| FFT | Fast Fourier Transform |
| HLSL | High-Level Shader Language (Introduced with DirectX 9) |
| GPU | Graphics Processing Unit |
| IOR | Index Of Refraction |
| LOD | Level Of Detail |
| NSE | Navier-Stokes Equations |

# References

[1] Microsoft Corporation, (2003), DirectX 9.0 Programmer's Reference

[2] Wikipedia: Fresnel equations, http://en.wikipedia.org/wiki/Fresnel_equations

[3] Luxpop, For index of refraction values and other photonics calculations and analysis, http://www.luxpop.com/

[4] Lasse Staff Jensen & Robert Golias, (2001), Deep-Water Animation and Rendering, http://www.gamasutra.com/gdce/2001/jensen/jensen_01.htm

[5] Hugo Elias, Perlin Noise, http://freespace.virgin.net/hugo.elias/models/m_perlin.htm

[6] Willem H. de Boer, (2000), Fast Terrain Rendering Using Geometrical Mipmapping

[7] James F. O'Brien and Jessica K. Hodgins, (1995), Dynamic Simulation of Splashing Fluids