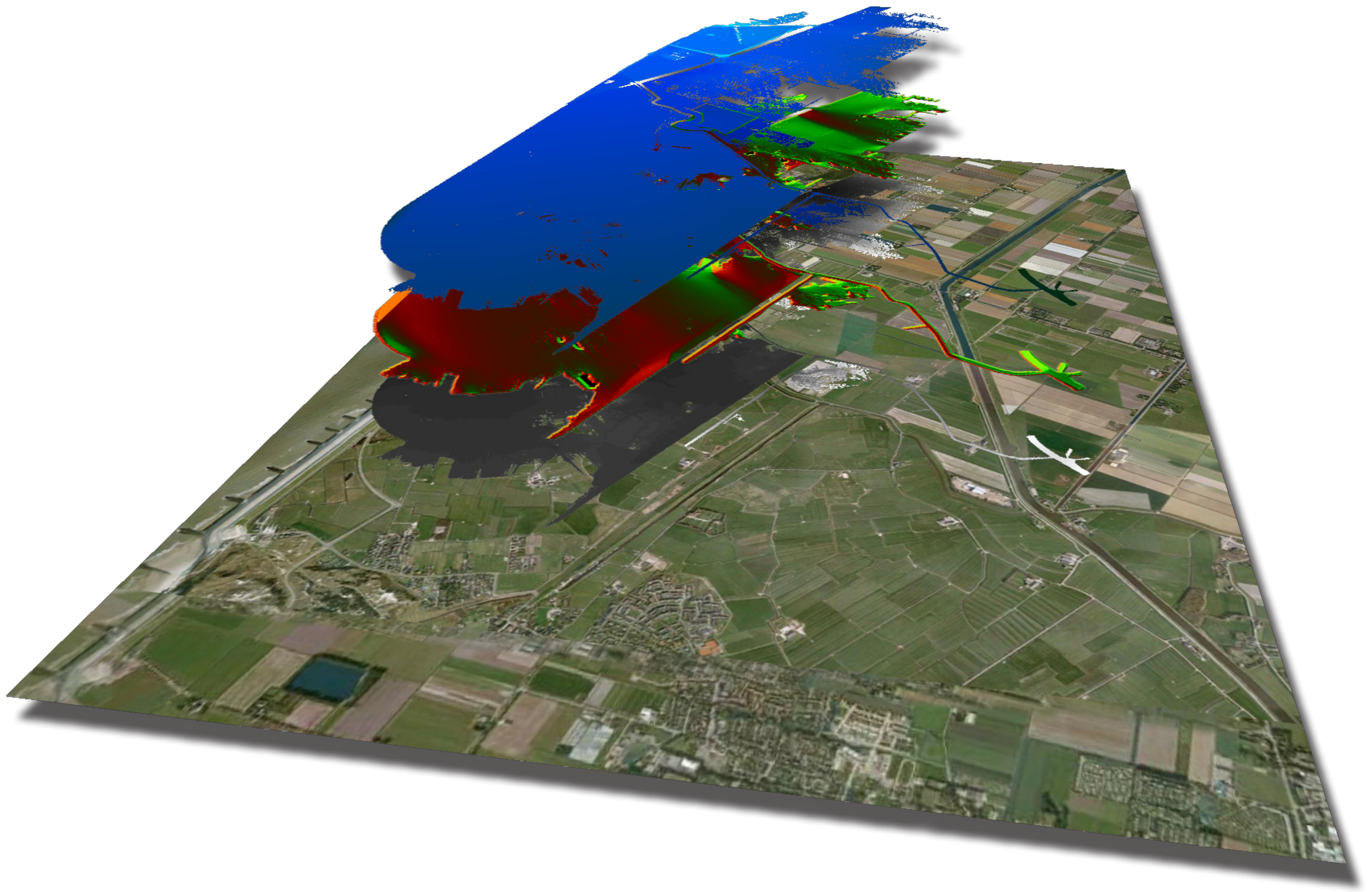
Experimentation Project:
Realtime visualization of water simulation output

By Almar Joling (3453243)



SUMMARY

In this document you will find the results of an experimentation project about water visualization. Using Unity3D and various textures which are used by pixel- and vertex shaders to render geometry, a good real-time approximation of flowing water based upon an externally calculated dataset is displayed.

INTRODUCTION

The aim of this experimentation project is to create an application that can load the output files from the water simulation software called Sobek, and turn this into a real-time 3D visualization. Currently the output consists of separate text files with fixed time intervals between them. Each text file contains the state of the water simulation for a given time for one type of variable. These variables are: water height, water level (terrain height + water level), flow direction u, flow direction v and velocity.

Sobek itself does provide the ability to export static images (image 2) but in the case of water, an animation can be a lot more convincing of the seriousness of an actual event or a precalculated “what if” scenario. Animations (as in videos) are normally made with Mathlab. This is in no way real-time and there is no interactive control whatsoever. The only features that are available in the resulting video is to start and pause the video on demand. There are no real differences in the data visualization, since the visualization can be changed on demand if needed. In this project the visualization will be focused on the “looks of the water” only, however. This project will try to bridge that gap by providing a visualization tool that allows to start and pause the time. But also to provide the abilities to zoom in or out, translate and rotate the camera, and make an video animation from any viewing angle. This makes it a good way to give flexible demonstrations during a presentation about future or past projects.
This project is part of a larger project which goes by the name “3DI”. The result of this project will eventually be merged together with another visualization project which uses 3D point cloud data, to provide a realistic interactive experience of moving water through Dutch landscapes or city areas.

GOAL OF THE PROJECT

In this project the aim is to create real-time realistic water based upon the output from the software package Sobek which simulates water flow.

TECHNOLOGY USED

In this project, the Unity3D engine (the free version) will be used as the visualization engine in combination with the development environment of Visual Studio 2010, which is available at Deltares. Unity3D is a flexible 3D engine that allows it’s developers to create their own dynamic meshes on the fly, yet it is more abstract than other 3D engines like Torque which rely more on custom programming. Next to this, Unity3D also supports compilation to different platforms like web browsers (using a plugin that needs to be downloaded once) and Apple’s OSX. At Deltares I have also used Unity3D before in other projects which reduces the learning curve and allows to focus more on the experimentation project itself. Unity3D uses Nvidia’s Cg shader programming language [7] for the pixel- and vertex shaders that will be used. Furthermore, all input data is provided by the 3DI partners (using Sobek) and no other additional software is required to generate input data. This project will not need Sobek itself to run.

3DI

This experimentation project is part of the 3DI project [1]. The 3DI project is about improving water management. Software and visualizations have a large role in water management. This project tries to enhance the software and visualization so that it is easier to perform faster and more accurate forecasts of what will occur during possible flooding situations. The 3DI project is spread over various subprojects. For example: Improving the calculation core of Sobek by utilizing the GPU (Graphics Processing Unit) of a video card. This could lead potentially to a performance increase of more than 30 times as seen in the GPU client of the popular distributed computing project Folding@Home. Another project is to use AHN2 data sets to provide more accurate calculations (predictions) of rainwater or flooding. This experimentation project is part of the interactive visualizations aspect of the project, together with another project which visualizes unfiltered LIDAR data in an open source 3D engine called “OpenSceneGraph”. This data is basically a very large set of 3D points. Buildings, trees and landscapes are clearly visible. More information can be found in the 3DI brochure [2]. For more background information like brochures (image 1) and downloads please go to the 3DI website [1].

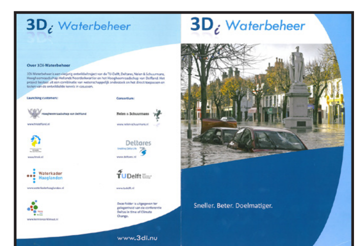


Image 1: 3DI brochure.

SOBEK

Sobek [3] is a software suite that was developed for the simulation of water flow in rivers and canals (image 2). It is very suitable for flood forecasting, optimizing water flow and river morphology. Besides for using it for calculations, Sobek is often used by the research institute Deltares to create visual impressions of flooding situations. For example, the effects of the dam break in Jakarta (March 2009), New Orleans flooding (August 2005) or the recent Japan tsunami (March 2011). Sobek is being developed by various Dutch public institutes and private consultants.

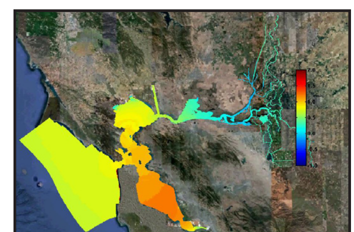


Image 2: Screenshot of Sobek output.

AHN DATA

AHN (“Actuele Hoogtekaart Nederland”) [5] or in English; “Actual Height map of The Netherlands” is a dataset which contains many points to accurately describe the terrain height of The Netherlands.

There are two versions of AHN: AHN version 1, which contains data points with a resolution of 5x5 square meters. The other version is AHN version 2, often shown as “AHN2”. These maps are still being created and have a resolution of 0,5x0,5 square meters. This means that for each 0,5x0,5 square meter a height value is given. These heights have a precision with a maximum error of approximate 5cm difference to their actual real sea-level height. The file format of AHN1 and AHN2 are 100% compatible with each other.

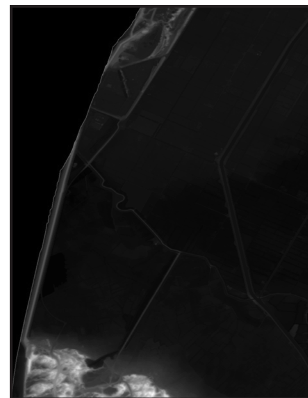


Image 3: AHN heightdata converted to a greyscale image.

The datasets are created by laser altimetry (LIDAR): A plane is systematically flying over The Netherlands. A laser emitting device is placed under the plane and emits pulses to the surface. By measuring the time it takes for the light to reflect back to the source it becomes possible to determine the height of the terrain, by using the already known plane orientation (rotation, altitude) and the known speed of the laser light. The AHN2 dataset of The Netherlands is not complete yet. This is also the case with the dataset provided in this project. The data provided is of a flooding situation at the area around the city of Petten in the province of Noord Holland, in AHN1 format. The AHN2 dataset is scheduled to be fully completed in 2012. Meanwhile there are various provinces already available. Unfortunately the data set does come with a price tag: About 40 Euro per 125 hectare. This price does get lower when ordering larger quantities. For non-commercial and educational institutes a reduced price is available.

WATER HEIGHT DATA

In this project the largest problem became the proper interpolation of the water values from the various output files. There are two types of data files: ASCII files (table 1), and incremental files (table 2). ASCII files contains all the water height values as double floating point values for a given time period separated by spaces. As shown below. Incremental files have the water heights categorized by a number of classes, and each line in the file contains just one change in the start of the dataset. The list of available classes start at “CLASSES OF INCREMENTAL FILE” until “END OF CLASSES”. Only one column is used, the other columns will have a value of -999. Following “END OF CLASSES” comes a list which specifies the timestamps on a line, and also the X, Y coordinates upon which the class follows. Basically it looks like:

Timestamp
X Y Class
X Y Class
Timestamp

The amount of [x,y] coordinates between each timestamp varies. Only changes per timestamp are in this file (hence the “incremental file” format).

```
ncols         886
nrows         1386
xllcorner     104339.2872918
yllcorner     525754.70683648
cellsize      5
NODATA_value  -9999
-9999 -9999 -9999 -9999 -9999 -9999 8.813987 8.846774 9.163333 9.06
-9999 -9999 -9999 -9999 -9999 -9999 9.292379 9.326129 9.634444 9.38
-9999 -9999 -9999 -9999 -9999 -9999 9.598536 9.664706 9.834445 10.06
-9999 -9999 -9999 -9999 -9999 9.453668 9.424715 9.398696 9.87 10.39
-9999 -9999 -9999 -9999 -9999 9.458859 9.389636 9.223077 8.73 10.92
-9999 -9999 -9999 -9999 8.786741 9.355749 9.325484 9.422222 9.17 11.26
-9999 -9999 -9999 -9999 8.92693 9.057838 9.481613 9.868889 9.7 11.7
-9999 -9999 -9999 8.40942 8.407746 8.991607 9.27389 10.00667 9.99 11.8
-9999 -9999 -9999 8.036201 7.992453 8.172927 8.33125 9.743333 10.58
-9999 -9999 -9999 7.586019 7.953158 7.795897 7.3 9.54 10.92 11.7 10.81
-9999 -9999 7.577547 7.584777 7.61353 7.687778 7.79 10.09 11.39 11.44
-9999 -9999 7.616163 7.597979 7.576087 7.86 8.11 10.41 11.69 11.25
-9999 7.489332 7.786044 7.731255 7.596667 7.21 8.73 10.9 11.85 10.9
-9999 7.486049 7.509097 7.77 7.786667 7.61 9.06 11.2 11.83 10.6 9.4
```

Table 1: ASCII file format.

```
INC1.0
MAIN DIMENSIONS                MMAX  NMAX
                                  886     1386
GRID                           DX      DY      X0      Y0
                                 5.00       5.00  104341.79  525757.21
domain                         END
START TIME T0: 2011.01.01 00:00:00
CLASSES OF INCREMENTAL FILE    Waterdepth(m) Velocity(m/s)  Waterlevel(m) U-velocity(m/s)  V-velocity(m/s)
                                   0.020     -999     -999     -999     -999
                                   0.100     -999     -999     -999     -999
                                   0.200     -999     -999     -999     -999
                                   0.300     -999     -999     -999     -999
                                   0.400     -999     -999     -999     -999
ENDCLASSES
.000000  0  1  1
1.000000  0  1  1
230 1033 1
230 1034 1
231 1033 1
231 1034 1
231 1035 3
231 1036 3
231 1037 5
```

Table 2:Incremental file which works with classes instead of real values. This file has "Water depth (m)" classes only. A class of 3 means a depth of 20cm.

For this project the incremental files were not used, since they lack precision because of the classes: These classes represent a range of values instead of an original value. In table 2 this can be seen, for "water depth (m)" the available classes are 0.020, 0.1, 0.2, 0.3 and 0.4. 0.2 could mean that the original floating point value could have been somewhere between >= 0.15 and <= 0.25 (assuming rounding). The ASCII format in table 1 provides the accurate values with floating point precision instead.

To convert the water height data to a usable format, a reader class was made which loads the entire file in a multi dimensioned array of the double data type.
After loading the data in memory, it is quite simple to write a bitmap file. The height values will need to be normalized to the byte data type, which has a maximum of 256 different values (from 0 to 255) per color channel. A bitmap supports three channels (red, green, blue) and therefore supports up to 256*256*256 colors. A simple grayscale bitmap (where all three channels contain the same color value) already gives a nice display of the data in each frame, as shown in image 3.

BITMAP CLASS

The program which creates all the textures for the visualization will have to create many textures with many pixels. To have a decent performance, the creation of the textures needs to be rather fast. The provided data in case "Petten" (image 3 shows the AHN map of this area) has a resolution of 886 x 1386 pixels (roughly 1.2 megapixel). C# provides a standard bitmap class which can read many common image formats. This class also provides two functions that provide read and write access to individual pixels. These SetPixel() and GetPixel() functions perform poorly when used in large quantities (image 4). Therefore I have created a new FastBitmap class which can lock the bitmap memory, and quickly change pixel values. When all the pixels have been set, an Unlock() function will be called which will copy the memory back in just one block and update the bitmap. This increases the performance significantly as shown in the graph (image 4).

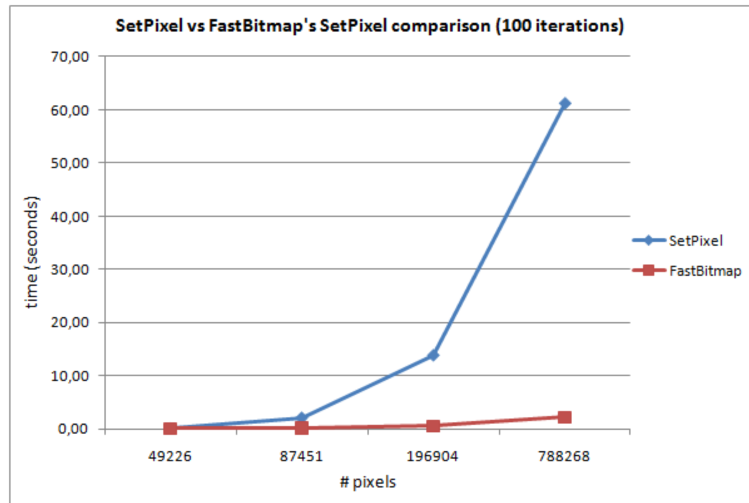


Image 4: SetPixel vs FastBitmap class performance.

8-BIT AND 16-BIT

The range of the data in this project is dynamic. For example, in case “Petten” the water level (which is AHN height + water height) has a range of [-3.00, 25.00]. In an 8-bit texture this value would be divided by 256 steps (8-bit = 1 byte = 256 values) giving a resolution of ~0.10 per value. Although this seems like a low value, it is enough to become visible. Image 7 in this document is an 8-bit image, and when enlarged (which is what happens with textures in a 3D world when zooming in) the so called ‘banding’ becomes relatively to see.

To extend the precision of the data in the water application, it is possible to use multiple color channels of an image to form a single value. In gaming applications 32-bit images (red, green, blue, alpha) are common and therefore allows us to use for example “32-bit floats” (combining all channels to create one value) or for example two 16-bit values, which already provide a (256*256) = 65536 values instead of the limited single byte 256 value range. By using such a technique, the precision of the terrain height can be increased and finer details in height become visible. In modern 3D engines it is also possible to use real floating point textures. Unfortunately Unity3D does not support these, nor can they easily be viewed in image editing software. Therefore the split channel method is used instead.

To convert a floating point value to 16-bit, the following bitwise actions are performed in C# on the variable “cb” which represents the value that needs to be stored over two bytes:

```
int b24 = ((cb >> 8) & 0xFF);
int a24 = (cb  & 0xFF);
```

In the shader the effect is reversed by doing the opposite operation when the textures are used in the pixelshader. If the shader would either use the blue or alpha channel the value would still be an 8-bit float with limited precision, while if the values are combined by bit shifting the two channels together, it would result in the original 16-bit floating point value as stored by the conversion program.

ANIMATING THE DATA

With all these “step files” as textures (image 5), Unity3D can import these and put them in the shader. The problem with these files is that they are not continuous. The files are created at fixed intervals. In case “Petten” this was 30 minutes. 30 minutes might not sound as a lot, but when a levee breaks there is a lot of water that wants to find it’s way from the sea to land. Eventually the water will slow down over time and almost stop entirely, when the water is at it’s lowest point.

The rushing water has become a real challenge, and needs to be interpolated in a way to look reasonably realistic. Finding the proper interpolation has posed an interesting problem since there are no algorithms that dealt with this kind of problem.

The first stepfile contains one single value which indicates the starting point of the flooding.

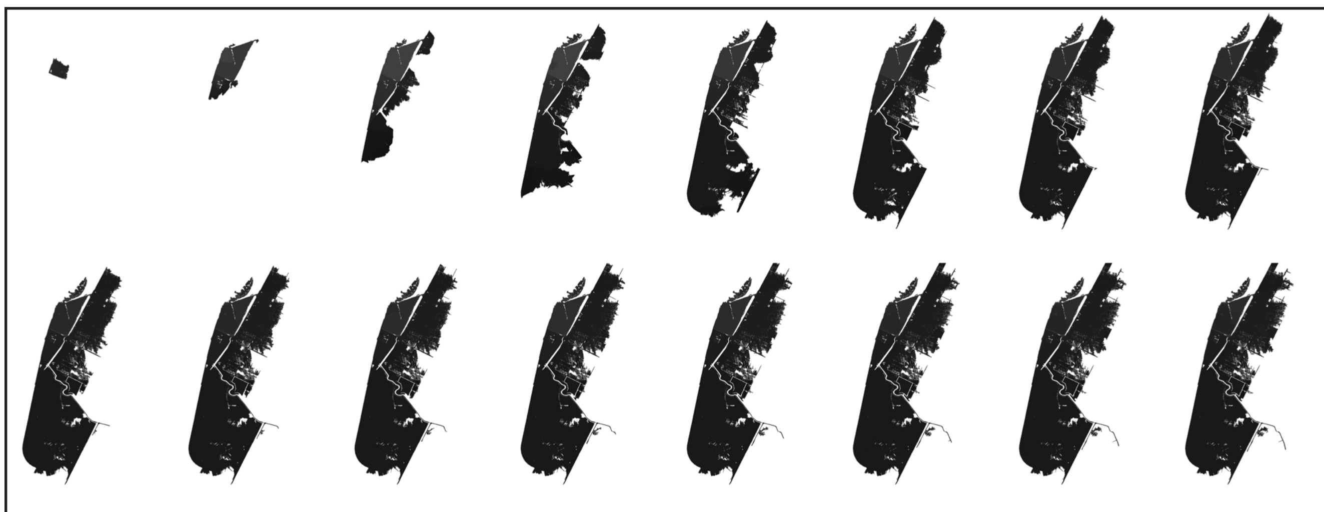


Image 5: Input ASCII stepfiles after conversion to bitmaps.

SEPARATE FRAMES

The first idea was to create many interpolated frames between every two step files. An interpolation of roughly 30 frames between two frames. Interpolation from frame 0 to frame 1, interpolated files from step 1 to step 2, and so on. This would mean that the animation would run at a maximum speed of 30 frames per second. Although the idea worked reasonably well (using linear interpolation), the files could not be used very well in the 3D engine: The interval between the step files was clearly visible and nothing like a smooth animation based upon the time between render frames. Next to this, textures had to be uploaded to the video card every frame which resulted in performance issues and timing issues (frame skipping). Therefore this technique was not an option at all.

LINEAR INTERPOLATION

At first, a linear interpolation was used. For each step from the origin, the distance of each cell (a cell represents a [x,y] location in the file). was increased, effectively creating a distance map. This method creates an interpolated grid, where the time is directly coupled to distance. But this is not how the water simulation results might actually be: the water might have slowed down over time, and in this case it had a constant speed.

Alternatively Euclidian distance was used between the start point, and each pixel's position. This created a very circular flow (image 6) of the water which also suffered from equal speed problems: The "velocity" of the water also had the same constant value, which reduces the illusion of watching water move over the land surface.

Image 6: Circular motion by using Euclidian distance.

EROSION

Erosion sounded like a good method: All step files would be stacked together to form one single array of values and then an erosion filter would 'bite away' around the edges of the water (so a reverse algorithm) creating new frames. The erosion could not be made directional. The erosion filter works with a NxN filter matrix to determine the smallest number in the matrix. The lowest value is set as the center value. By repeatedly iterating this way over the input data, the erosion filter does it's work. Although the NxN matrix does not have to be uniformly configured and for example use weights, it will not work reliably. Setting the weight to be important at the right side of the image could (and will) for example start to erode the empty values in the middle of the test case data (which can be seen in nearly all images in this document). Therefore the water is potentionally eroded on all sides, to a whole different start position than the real one. More like a reverse animation of a falling drop of water on a plane.

RENDER TEXTURE

Render textures are textures that are used as a render target. This means that everything that a 'camera object' renders will not be output to the screen, but rather to the texture. Render textures allow the creation of various real-time effects in popular games like: mirrors, water reflections, portals, etc.

Although Unity3D Free does not support render textures, the 30 day "Pro" trial license does. This trial was used to test the render texture experiment. In this case, the render texture could be used to interpolate between two step textures as mentioned in the previous experiment. This technique was inspired by the "Jump Flooding Algorithm" paper [9], although none of the described techniques in the paper were actually used. The interpolation is done using a linear interpolation in the pixel shader. The interpolation worked well, and was fast: the only problem is that shaders are stateless. It is not possible to return information from the pixel shader to the application (except for a new texture). Therefore it was impossible to tell to the application whether the interpolation was at 100% between two frames or not. Also, new textures need to be uploaded to the video card when advancing to a new frame, and the timing of this texture upload cannot be controlled. This could therefore lead to the skipping/chopping of frames because some frames might take a (noticeable) time longer to update than other frames. Also, Unity3D does not provide means to read back pixels from a RenderTexture. Therefore it is impossible to determine if the interpolated texture is pixel-wise equal to

the frame that was interpolated to.

THE SOLUTION

After many attempts, a suggestion came from a 3D artist at Deltares, proposing the idea to use some kind of gradient texture which would control the visibility of the water surface. An example was rather quickly made using the various step files created from the ASCII files and a “Gaussian blur” filter in Photoshop. This idea worked quite good, and this has become the visualization method for the water visualization.

It did mean that somehow a good “time control” texture would have to be generated from the input files. In this section the entire process will be explained and illustrated where possible.

MOTION THROUGH A TEXTURE GRADIENT

In this technique a gradient texture (image 7) is used to control the visibility of the water. The gradient has a height of just one single pixel since it is merely used as a lookup table. The gradient is basically linear with a large white “bump” (image 8). In Unity3D this texture is configured to clamp the texture coordinates [4]. Which means that it does not repeat itself when texture coordinates exceed the 0.0 to 1.0f boundary, in the so called texture space. Texture space coordinates start at [0,0] which is the top-left position of the texture. The bottom right pixels of the texture are located at [1,1]. With texture clamping the border pixels will repeat itself when texture coordinates are used outside the [0,0] to [1,1] range. If the last color at the right side of the image would be white, all pixels outside the range would also be white - at the right side.

Image 7: Gradient texture that controls the front wave intensity, and foam intensity.

The same gradient transformed to a two dimensional graph, results in a curve which looks like image 8:

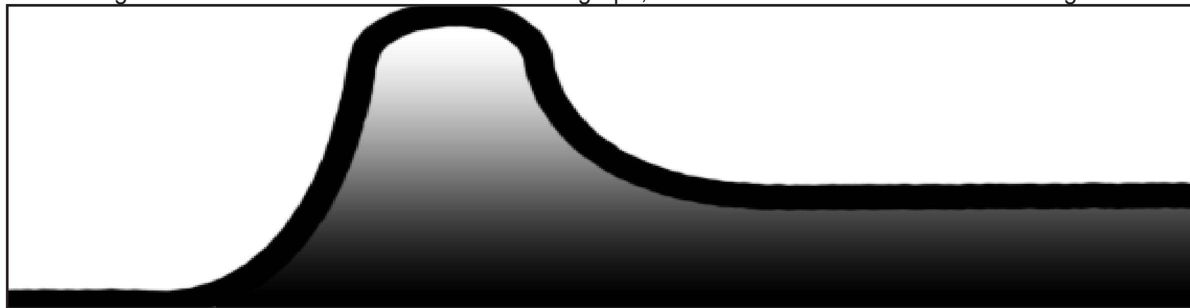


Image 8: 2D representation of the same gradient texture.

This gradient controls the visibility (black is invisible, and white is fully visible) and also the vector length of the normals used in the water to reflect the light. The normal values of the water are multiplied with this curve. At the peak of the curve (intensity = 1.0) the normal will be longer than afterwards (intensity = 0.5). In this case, it will give a smooth rolling look at the front side of the wave. Since the texture is clamped, after subtracting the time value from the texture (which is also ranged from 0.0 (start) to 1.0 (end)), this will automatically make the width of the foam smaller over time. Although this was actually a side effect of this technique, it does give an additional natural feel. The implementation of foam will be addressed later in this document.

At the tail (right side) of the gradient the color is grey (image 7 and 8). This value is necessary to give the water a constant visibility after the first frontal wave. This first frontal wave will be at the front of the moving water body, and besides having a curved shape when rapidly moving forward, the wave also brings a layer of foam at the front. This effect can typically be seen in large fronts of moving water which rushes downwards (which, in fact, is what happens during a levee break as in this project’s case) If it were black, like at the left side of the gradient, the water would disappear again after the front wave. Behind the frontal wave the water will look more restful which is also a commonly seen natural effect (image 9).

Image 9: Tsunami in Japan, the foam is actually dark by the dirty water. Image Courtesy of National Geographic “Witness of disaster” 2011.

INTERPOLATION OF FRAMES

Another texture is needed for the technique to work. The “time source” which controls the movement of the water using a texture that also looks like a gradient texture.

The first step is to crop the input frames to the **smallest** possible rectangle by removing all unnecessary empty cells (image 10 shows a representation of this). This means the NULL ‘values’ which do not have any data are removed, creating a tight bounding array. By creating a smaller rectangle the execution time of the algorithm will be reduced and also reduces the amount of triangles that are needed later in this application because the triangles and pixels are mapped with an 1:1 relation. After cropping, a simple text file is saved to disk which contains the crop rectangle values (left, top, right, bottom). These values are later used again to translate the water geometry to the new origin. The AHN height map is not cropped, and without translating the water geometry there would be a mismatch!

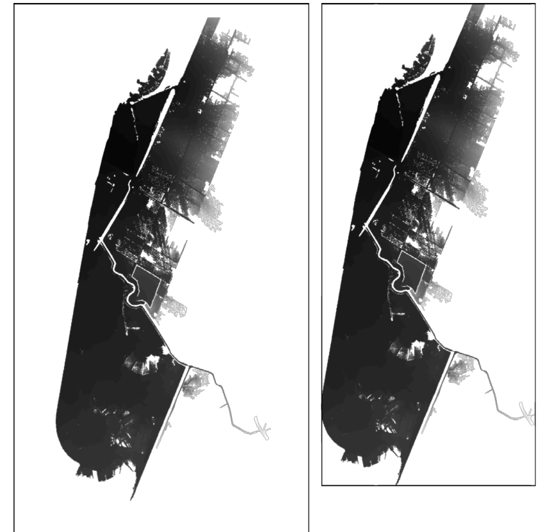


Image 10: Cropping the input data to a smaller rectangle

Since this rendering method uses a texture as ‘time source’ it will not be possible to use the separate frames combined in a single texture. After experimenting with various means of interpolation in one single texture, similar to the techniques used in the separate files (Euclidian distance, linear distance, erosion) the final result was to use a radial distance function (RDF) based upon the Pythagorean distance between the center point of a pixel, and an n x n block around it. The formula for the RBF is a Gaussian function (equation 1):

$$f(d) = ae^{-\frac{(d-b)^2}{2c^2}}$$

Equation 1: Gaussian function

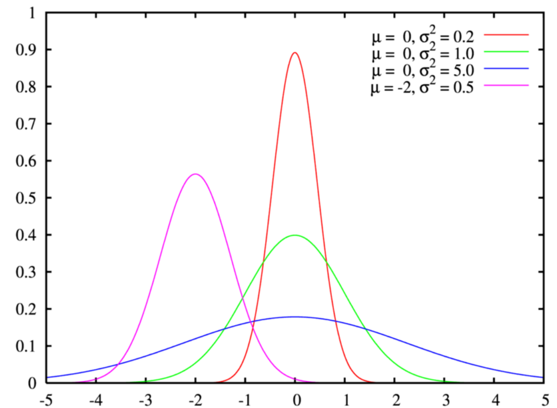


Image 11: Gaussian function used in the interpolation algorithm. b = μ (center point, always 0 for this application, c = σ (steepness of bell curve. user configured)

Whereby the following variables are used:

- *a* as a constant factor to scale the results
- *d* is the Euclidian distance between the cell coordinates [x,y] and the offset coordinates [x,y]
- *b* defines the center of the Gaussian bell curve, which is 0 in this case (the position is always the cell [x,y] which is being processed at the moment.
- *c* is a value which controls the slope of the function. This value is based on the maximum [x,y] offset from the center. This value is user configurable.

With this function the values will look similar to a bell shape (or normal distribution) as seen in image 11, which will reduce the straight lines (image 12) of the final interpolated control texture. The steepness of the curve will determine how smooth (or ‘rounded’) the flowing water will be.

Image 12: Water moving in straight lines when not using the normal distribution but a more standard flood fill.

INTERPOLATION ALGORITHM

To create the time (interpolated) texture the following algorithm is used:

- First determine all cells in all input step files that have a value (not a NULL value). This is simply done by looping through all Sobek output files of the type that contain the water height. For each of these files there will be a loop through all cells (each [x,y] coordinate).
- After this, all the values of the cells are accumulated together for each [x,y] position on which they belong. So, around the starting position it is very likely that there will be as much values added as the number of input step files. Cells which are only added at the end of the stepfile sequence will therefore not have any cumulative values, but just the values of the last step file. Of course this could be different if the water were standing still for a number of frames on exactly the same place.
- Then the starting cell is searched (the first stepfile has just one value indicating the starting [x,y] coordinates) and add it to

a queue of cells.

» Loop through the queue (at start this is just the first starting cell) and increase a “height” property each step using the RDF. For each step it is also determined if the water is already “flooding” over the next stepfile in the sequence. If it does, then those cells are added to the same queue. The height is increased using the RDF also on those cells based on the distance to the input (center) cell of this function.

» If all NxN (n is an input value) cells around one other cell have become active, the center cell is marked as finished and register on which loop iteration this happened (this is an optimization to the loop).

» After all cells have been processed, loop once more through all cells to add the difference between the last iteration and the iteration on which the cell was considered finished. This value is multiplied with the RDF to get the exact same value as if it were never marked completed.

Image 13 shows a process diagram of this algorithm:

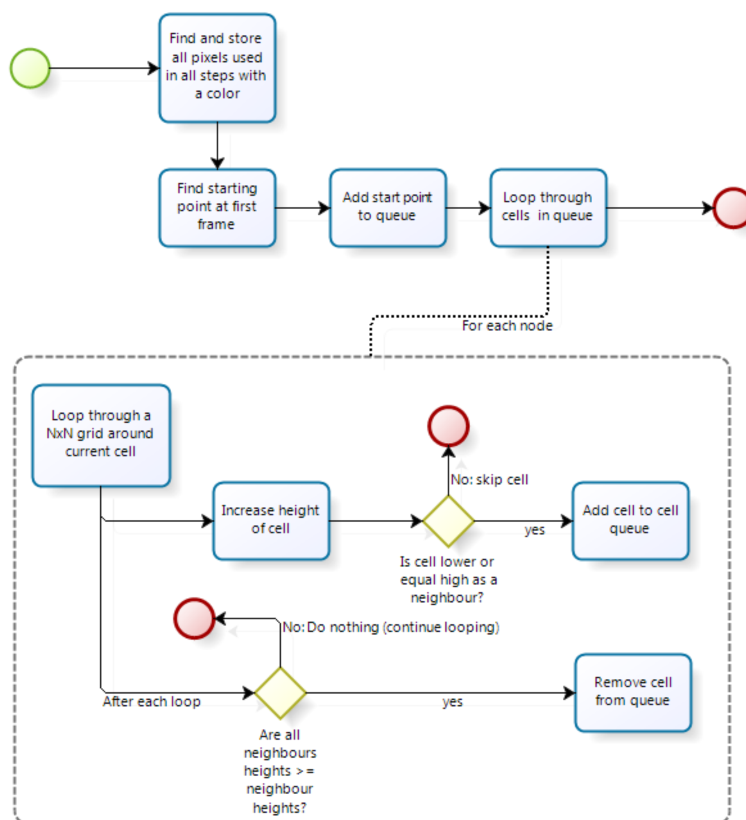


Image 13: Process diagram of the algorithm.

Image 14 shows what happens over time for a small number of output step files: The circular shape represents the time which increases each iteration of the loop in a bell shaped curve.

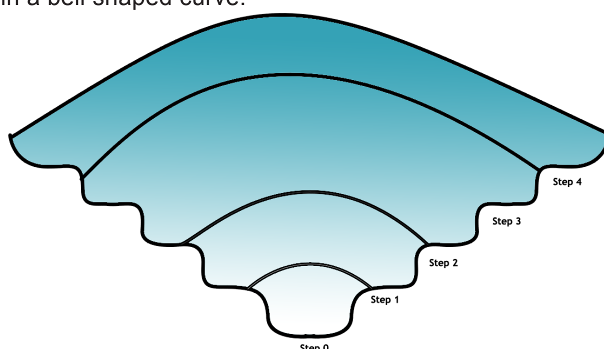


Image 14: Side view of the water filling algorithm. The algorithm continues to pour water until the last step has “flooded”.

The final output of the algorithm can be seen next (image 15). The left texture is the final control texture used to interpolate over time. The second image is the final image in 16-bit. The 16-bit image’s “fingerprint” effect does give an idea of the water flowing over the terrain.

Image 15: Time texture in greyscale and in 16-bit spread over the blue channel and the alpha channel.

CREATION OF THE WATER SHADER

INTRODUCTION TO WATER

The representation of water has always been a subject that received a lot of research in game development. The graphics have improved a lot since the first real 3D games and the approximation of water has become very realistic. The screenshots below give an indication of the recent improvements in graphics.

Image 16: Unreal Tournament (1998, Unreal Engine 1, Epic Megagames) at the left. Crysis 2 (2011, using the CryEngine2 by Crytek) at the right.

In this experimentation project, the goal is to create water which has about the graphical quality of games from the past three years (image 16). To do this the water shader is based upon papers which are written in the last couple of years. In this case the “Using Vertex Texture Displacement for Realistic Water Rendering” [8] and the “Water Flow in Portal 2” [6] papers provide a good start, and the shader will be based upon the papers. Image 17 shows how the water Left4Dead2 (shares nearly the same engine as Portal 2) look like.

To create realistic water in this experimentation project, the following elements will be used in the water shader:

» Reflection of the cube map (a sky dome)
» Refraction of the terrain underneath
» Creation of waves (using normal maps)
» Directional waves (flow maps)
» Water becomes more transparent at the edges (depth buffer)
» Foam at the edges (using the gradient texture created in this project)
» Lighting, like the sun (using normal maps)

In this report these items will not be handled in detail, since most of these elements are quite common in the rendering of (real-time) water.

Image 17: Water of Left4Dead2. Screenshot courtesy of Valve.

REFLECTION

This water demo uses a cube map to give the environment a "sky". The cube map acts as a so called "sky dome" to enhance the concept of viewing a virtual world. For reflection and refractions to work, a view vector is needed. Unity3D comes with a function which does just this for us. Now that the "eye" vector is known and a (per pixel normal) for the water surface itself (either calculated in a shader or from a normal map), the Fresnel term can be calculated.

Using a Fresnel term the amount of reflected light by the viewing angle can be changed. The lower the viewing angle, the more visually reflective the surface gets. In shader programming commonly an approximation is used, since a real Fresnel integration would be too much of a computational strain to calculate per pixel. The Fresnel term is calculated in the shader using:

```
float3 calcfresnel(float3 viewdir, float3 normal, float R0)
{
	float fresnelBias = 0.2f;
	float fresnelScale = 0.1f;
	return fresnelBias + fresnelScale * pow(1 + dot(viewdir, normal), R0);
}
```

Unfortunately "RenderTextures" cannot be used in Unity3D Free, which means real-time reflection is impossible to perform at the moment. But in this project there are no other dynamic elements in the environment, so this is not much of a problem.

REFRACTION

Refraction is the change of direction of a wave due to its speed or due to crossing a different medium. Since light also travels in (very small) waves, refraction also applies. Which means that the land underneath the water should be somewhat refracted. Water has a common refraction index of 1.33. The Cg shader language has a standard function to perform this calculation:

```
	// Refract the reflection vector, to make the reflection less perfect:
	float3 Refract = normalize(refract(normalize(i.eyeDir), finalnormal, 1.333f));
```

Eventually the refraction was removed from the shader: Refraction works well with transparent water (where you can see through it), and not for the dark water in this project (Dutch water typically is not really transparent).

WAVES

The waves of the water in this project come from two sources. First, there is the large "tidal" wave which is the water spreading over the land. The second type of waves are the smaller ones which are at the top of the water surface.
The large moving water body is created using a sliding gradient texture. Smaller waves are created using flow maps. Flow maps are a technique introduced by Alex Vlachos [6] in Valve's recent games like Left 4 Dead 2 (image 17) and Portal 2.

The method behind these flow maps works as follows:
The opacity of two normal maps are oscillating on a half-time interval (image 18). At T=0, normal map 1 will be completely visible, and normal map 2 will be completely invisible. At T=0.5 this will be exactly the opposite. These normal maps represent directional vectors which are distorted by using the flow map. This distortion can only be done for a small amount, otherwise the distortion will be too much, and become noticeable. This is the reason why there are two normal maps interpolating over time: The animation

needs to be reset every half of the animation phase, and this way it becomes (nearly) invisible.

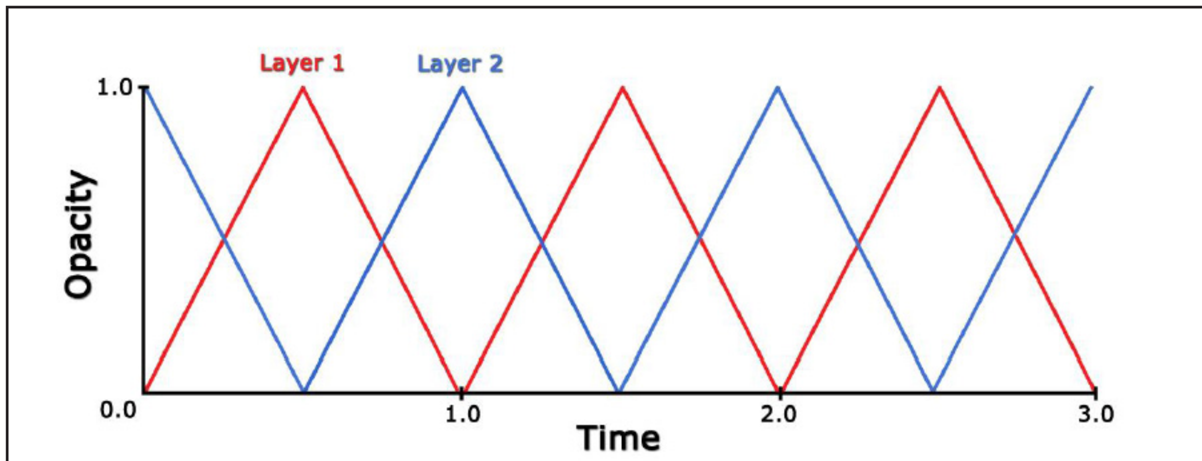


Image 18: Two textures interpolate with a half phase difference. Image courtesy of Valve.

WAVE SCALE

As distance increases, objects tend to get smaller. The same applies for the water. Nearby small detailed waves can be seen. When looking from the sky in a helicopter, the waves would not be recognizable. This poses a problem for the water visualization: The water should look like water from a large distance. Dynamic scaling results in strange artifacts while moving, so the water wave size is set at a fixed size. This size is larger than what would be realistic: The waves in the water have more the size of ocean water, but this does not look that odd to the viewer. The water normal map texture (as seen below) is tiled a fixed number of times in the shader. The more tiles, the smaller the scale of the water. The 256x256 pixels water normal map is tiled 50 times on the horizontal and vertical axis. This value is purely based on a ”what looks best” case.

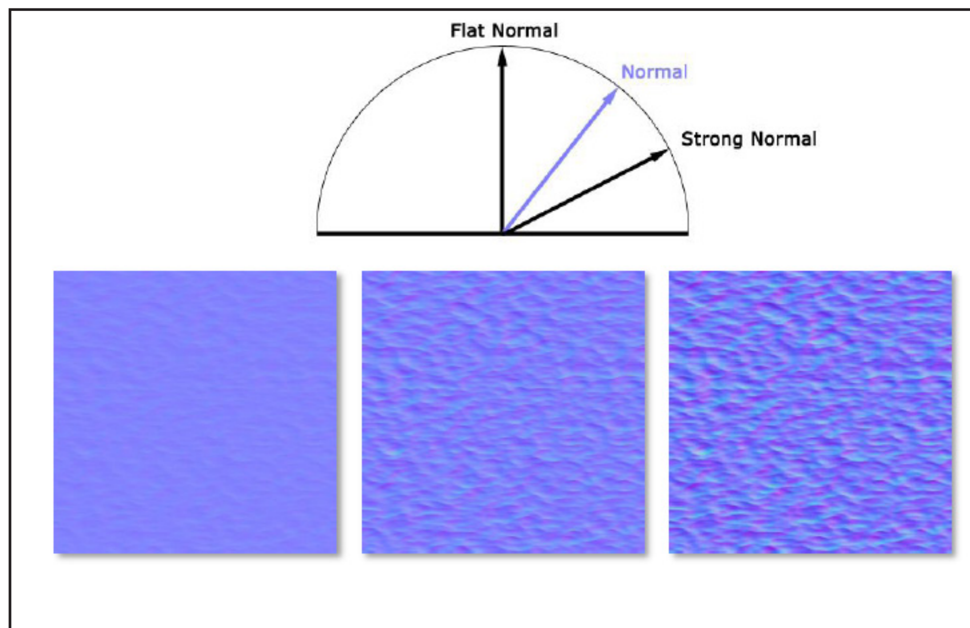
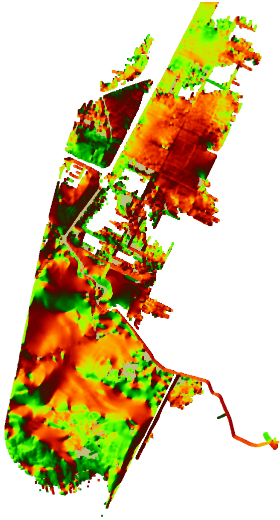


Image 19: Two textures interpolate with a half phase difference. Image courtesy of Valve.

To hide the remaining “pulsing effect” which occurs, an extra noise texture value is added to the normal. This effectively reduces most of the repeated pulsing. The normal map of the water is used in all the light calculations and the controlled distortion will therefore create the flowing effect of the water. The strength of the normal is controlled by the Euclidian length of the normal vector, as shown in image 19.

A flow map is created using the gradient map (image 15). By simply adding all direct neighbour values (3x3 excluding the center) and using the difference between the center and neighbour value as the length of the vector, effectively creating a vector which points to the (average) direction which the water flows. This method works in the case of this visualization better than the already existing “ u and v” direction ASCII files. These files are (just like the water height) in steps of a certain time. Also, after converting the final frame to a normal map, it was clear that using a custom flow map gives better accuracy and smoother “flow” directions than the original frame. One of the issues was that the directions seem to rather change over time, based upon the time step. Something that the setup of this shader could not cope with. In theory it could be possible to integrate multiple flow textures in the shader except that it would result in similar issues as with the step file interpolation method: Textures would have to be uploaded to the video card very often and again would have to be interpolated over a certain amount of time (probably 30 fps similar to the steps). Therefore it is not feasible to interpolate such an amount of detail. A combined version of all time frames stacked together resulted in an image that contained many sudden (unrealistic) direction changes as seen on image 20.



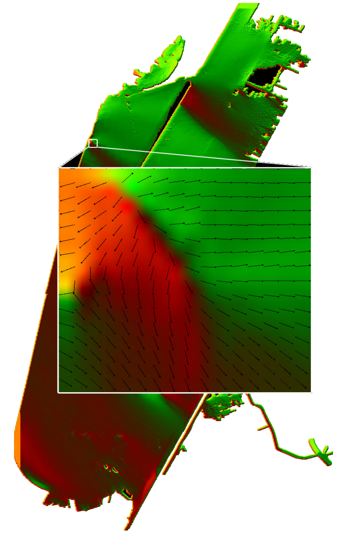


Image 20: Flow map based on ASC file at the left. At the right the custom generated flow map, with indication of the directions using arrows.

LIGHTING

Lighting is calculated based upon the water normal map (image 21), the flow normals, and a specified light color and direction. The actual lighting calculation is based upon Lambertian lighting ("N dot L") which means that the dot product of the normal vector and the lighting direction vector are used to determine whether a pixel should be lighted or not. The intensity of light reflected is calculated by the angle of incidence the light has on the surface. This type of lighting has a rather uniform intensity and the distance does not matter (an approximation of directional lighting like a sun).

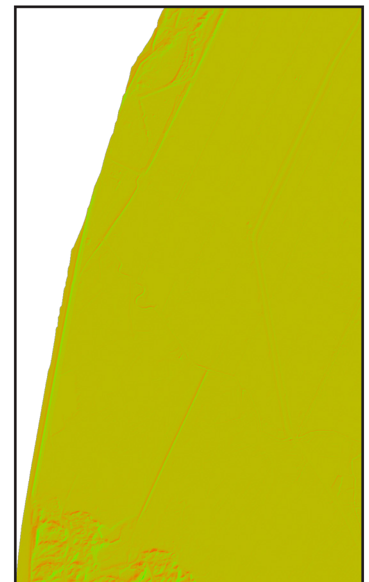


Image 21: Normal map of the water.

WATER TRANSPARENCY

Unfortunately Unity3D Free does not have the ability to use render textures, which means that a depth texture cannot be used. With a depth texture the water transparency could have been altered dynamically based on the distance to the camera. When the camera is closer to (not so deep water) the transparency should normally increase. In this case the water transparency could be rendered using the water depth texture itself created by the interpolation tool. Although this does not give as smooth effects as desired, the results are about the same.

FOAM

Foam (image 22, 23) is visible at the front of the large water waves that spread over the land (like the sea), but also on the water surface itself. Foam occurs normally when water mixes with air. This happens when the water reaches the shoreline at a coast, but also because of water flowing across uneven surfaces, various directions of water merging each other (like at sea) and so on.

In the shader, foam is also created using the gradient texture (image 7). This is somewhat logical: The 'front' of the water is located where the gradient is at its maximum. This front side is exactly the place where the most foam should be. To give an extra form of foam illusion, the foam is moved using the similar flow map method as mentioned before but without any extra tiling. This makes the foam flow together with the water without any seams.

Image 22: At the left water with foam. At the right without foam.

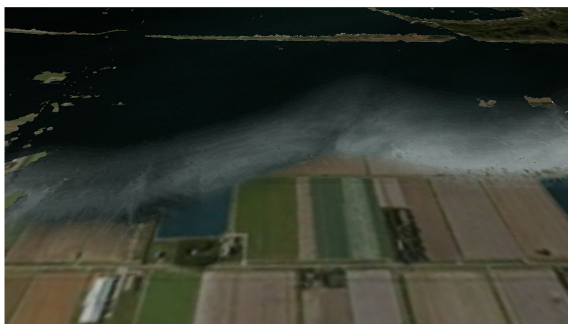


Image 23: Front view. At the left water with foam. At the right without foam.

TERRAIN RENDERING

The terrain consists of a plane made of triangles which are displaced by a greyscale height texture. This texture is created using the AHN data and is static. Unity3D has a limited triangle count of 65k triangles per mesh and therefore the terrain resolution is limited to a maximum of 256x256 triangles. In this experimentation project the terrain will not be optimized using LOD (Level of Detail) functions.
The terrain diffuse textures are based upon screenshots from Google Earth. These are not dynamically acquired (since this is against the policy of Google Earth) and therefore were manually captured and stitched together.
To light the terrain, a normal map is created for the terrain when converting the AHN height file to a bitmap. Creating a normal map from a height map can be done in various ways. One method is to sample four points, subtract these, and create a cross product. Another method is to use a Sobel filter, which is exactly the method used in this project. Basically, a Sobel filter finds the direction of the largest possible increase of the values and how much, for each point in the AHN height file.

Then the values are multiplied by two, and subtracting one. Now the normal map channels are in 0.0f to 1.0f range in the shader (which means actually 0 and 255 in normal image programs), but actually represents -1.0f to 1.0f. In the shader process is reversed to get the actual normal value back. This allows us to use per pixel lighting in the vertex shader. In this project a standard lighting model is used: “N dot L” lighting (Lambertian reflectance). This application does not require an highly realisttic lighting model.

CONCLUSION

The goal of this project was to create a water visualization based upon precalculated output from Sobek. After experimenting with various attempts to solve the interpolation between various frames, one technique worked out well. By creating a custom flood fill based upon layers created by the stacking of the output files, a proper time based interpolation was possible. After turning this ‘flood fill’ to a texture, and by using a gradient texture to control the shape of the wave the vertices where displaced probably on the video card. By implementing a pixel shader based upon Alex Vlachos’s [6] paper the water’s surface became animated and started to look like a proper visualization of a levee break with the water flowing over the terrain.

FUTURE WORK

This project’s purpose was to see if it was possible to create reasonable realistic water based upon the output from the Sobek calculated step files. There are various improvements to be done.

DYNAMIC TERRAIN TEXTURES

Right now the Google Maps terrain image is the only file that is not automatically created using the processing tool. This texture was made from many large screenshots taken by hand in Google Earth which where composited together. It is also against Google’s policy to automatically download images from their services. Fortunately, the end goal of the project is to be integrated in a point cloud system which already uses terrain images from Dutch air camera photographers. When used outside The Netherlands, this could be a problem again. An additional fallback on purchased satellite images could work in these situations.

INCREASE TERRAIN RESOLUTION

The maximum resolution of the terrain is now limited by the resolution which Unity3D can handle (4096x4096 pixels) since the program uses one single texture for the entire landscape. It could be investigated how this can be improved, possibly by dividing the terrain in chunks which use different terrain textures of the maximum resolution.

ADDING PARTICLE EFFECTS

To increase the graphics quality, particle effects could be used to add an extra level of detail. For example, to create water splashes at the base of the dam break, or when the water travels around corners or obstacles.

LEVEL OF DETAIL (LOD)

To optimize the rendering of the terrain, screen space texturing could be used [5], together with a mesh which has an angle of few degrees more than the camera’s field of view (FOV):

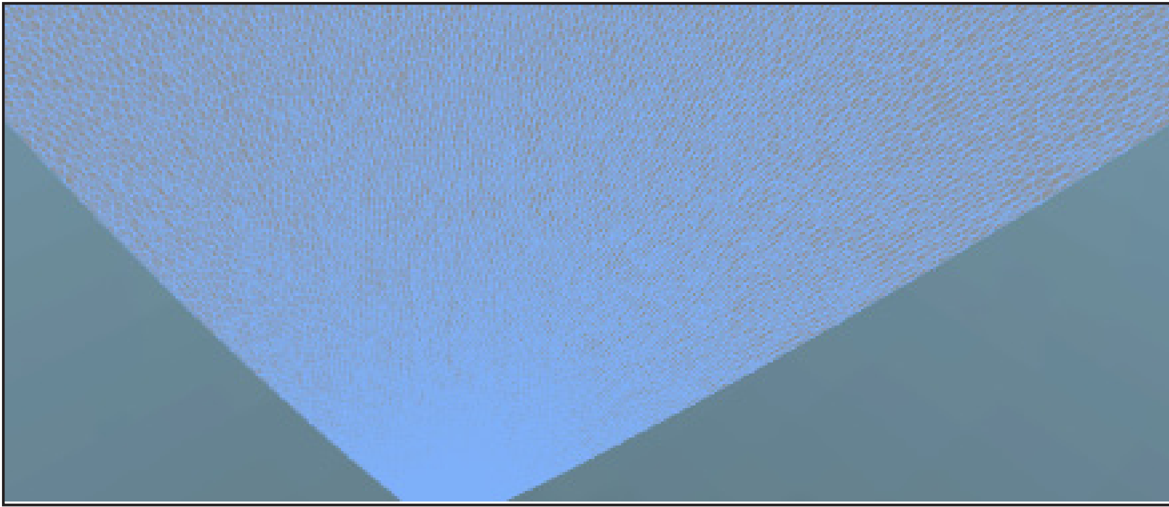


Image 24: Level of Detail mesh. More detail (more triangles) at the front of the camera.

This mesh (image 24) allows us to concentrate the detail near the player camera, while the detail gets less over distance. A relative simple and efficient way to create a Level of Detail (LOD) system for the terrain. This technique does potentially have some issues when viewed from a high altitude view.

SCREENSHOTS

INTERPOLATION COMMAND LINE TOOL

```
file:///D:/unity/Interpolation/InterpolationCMD/3DInterpolate/bin/Release/3DInterpolate.EXE
3DInterpolate Instructions
Created by Almar Joling (2011)

This program requires SOBEK output files in either INC or ASC format
and a matching AHN ASC file to succesfully convert data

/h                  These instructions
/ahn:<ahn file>             Input AHN ASC file
/steps:<directory>              Input directory where ASC or INC files are locat
ed
/output:<directory>             Output directory where to write files to
/flow               Create flow maps
/ahn                Create AHN maps
/all                Create all maps (Default)
/radius:<number>                Radius function, increases time dramatically if
set too high! (Default: 14)

Error:
AHN input file: None specified
Steps input directory: None specified
Output directory: None specified
```

```
file:///D:/unity/Interpolation/InterpolationCMD/3DInterpolate/bin/Release/3DInterpolate.EXE
3DInterpolate Instructions
Created by Almar Joling (2011)

This program requires SOBEK output files in either INC or ASC format
and a matching AHN ASC file to succesfully convert data

Creating AHN height texture...done
Creating AHN normal texture...done
Creating stack texture...done
Creating fill texture (this will take a while):
Exploring world...done
Filling world...35%
```

Image 25: Command line interface of interpolation tool.

/h	These instructions
/ahn:<ahn file>	Input AHN ASC file
/steps:<directory>	Input directory where ASC or INC files are located
/output:<directory>	Output directory where to write files to
/flow	Create flow maps
/ahn	Create AHN maps
/all	Create all maps (Default)
/radius:<number>	Radius function, increases time dramatically if set too high! (Default: 14)

Table 3: Command line arguments

WATER RENDERING

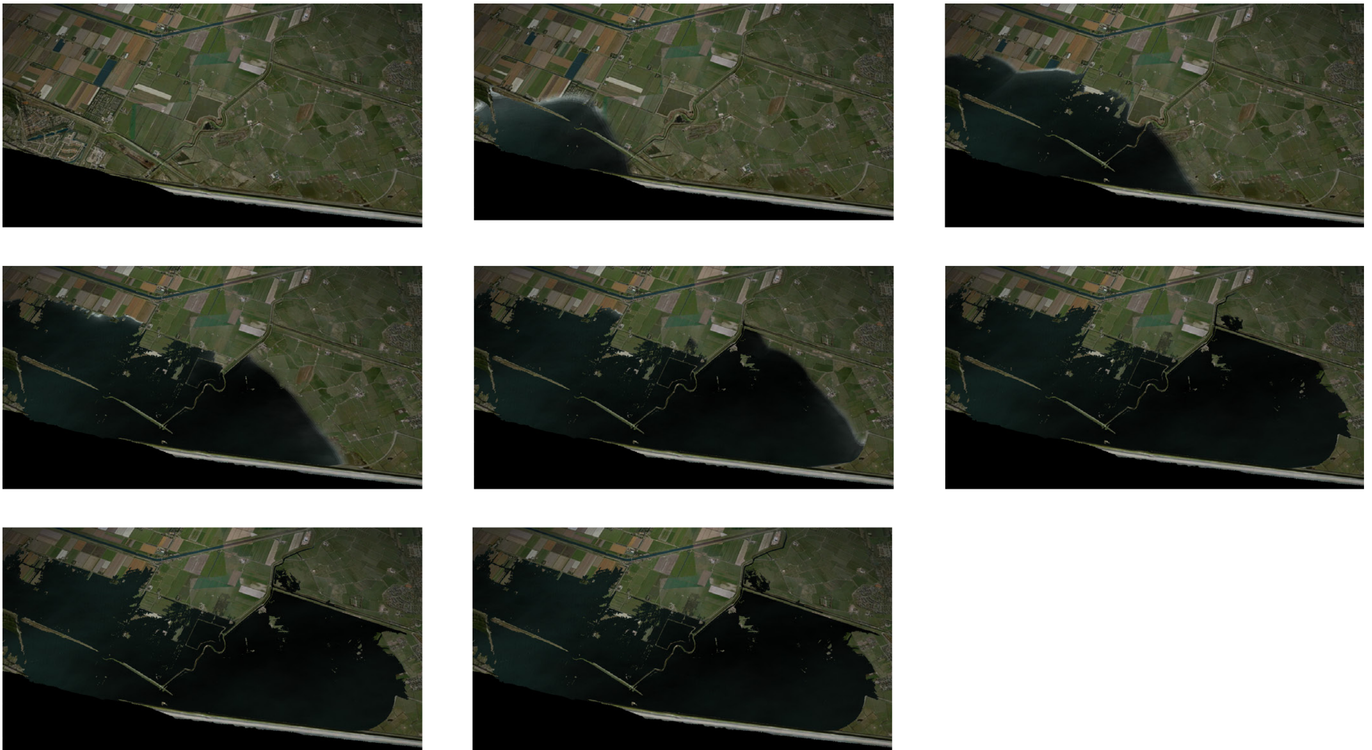


Image 26: The animation seen in a number of frames. The amount of foam can easily be changed inside the shader, and in these screenshots the amount was set a bit lower. Still the foam is quite large compared to the terrain underneath!

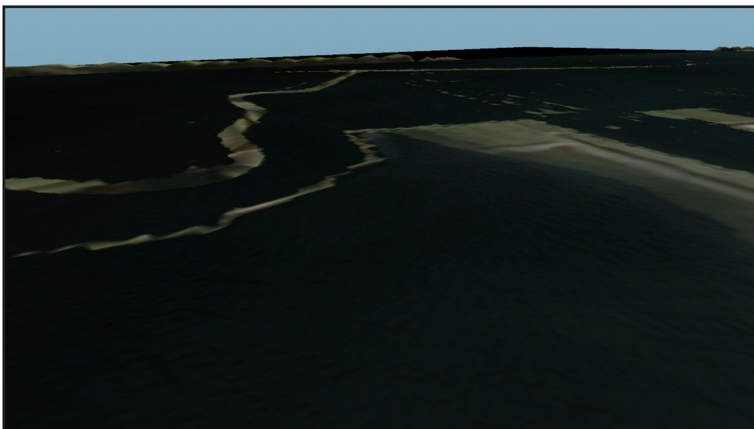


Image 27: Water spreading over the land area. The waves on top also follow the direction.

REFERENCES

[1] Website: http://3di.nu/. Visited at 13-05-2011
[2] Website: http://www.deltares.nl/nl/software/108282/sobek-suite Visited at 09-07-2011
[3] Brochure of 3DI: http://3di.nu/media/cms_page_media/5/Folder_tweede%20druk_Engels_webversie.pdf (English version) Visited at 09-07-2011
[4] Website: Microsoft MSDN. http://msdn.microsoft.com/en-us/library/aa911203.aspx. Visited at 10-07-2011.
[5] Website: http://www.ahn.nl/. Visited at 13-05-2011
[6] Water Flow in Portal 2. Alex Vlachos, Valve. July 2010, Presented at SIGGRAPH 2010. Website: http://www.valvesoftware.com/publications/2010/siggraph2010_vlachos_waterflow.pdf. (Visited at 28-06-2011)
[7] Nvidia Shader CG reference website: http://http.developer.nvidia.com/CgTutorial/cg_tutorial_appendix_e.html (Visited at 28-06-2011)
[8] Using Vertex Texture Displacement for Realistic Water Rendering, Yuri Kryachko (Maddox Games) in GPU Gems 2. (online: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter18.html Visited at 13-05-2011)
[9] Jump Flooding Algorithm on Graphics Hardware and it’s applications by Rong Guodong , 2007. Thesis submitted to national University of Singapore.