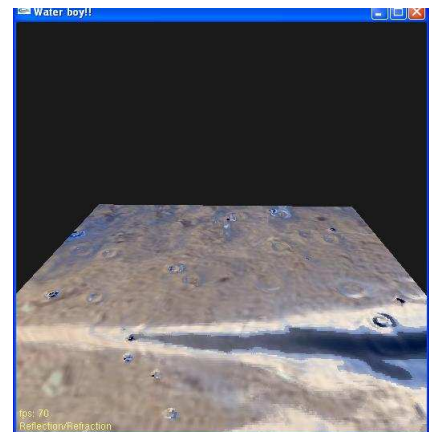
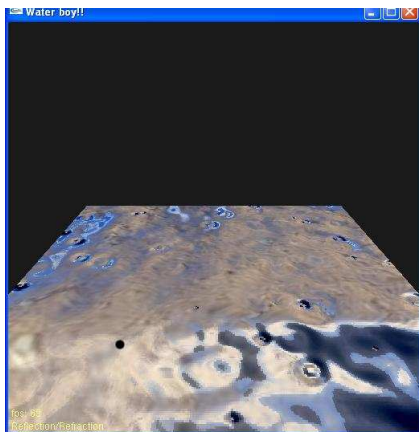
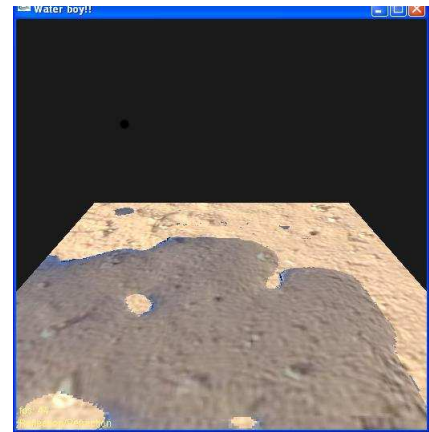
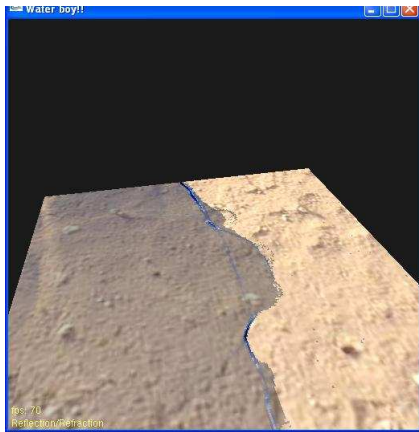


# Implementing Rapid, Stable Fluid Dynamics on the GPU

Karsten Ø. Noe, 20001995

Implementation of project done in cooperation with Peter Trier

Fall 2004



# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Previous work</b>	<b>2</b>
<b>3</b>	<b>Rapid, Stable Fluid Dynamics for Computer Graphics</b>	<b>3</b>
3.1	An explicit expression . . . . .	6
<b>4</b>	<b>GPU implementation</b>	<b>6</b>
4.1	Simulation . . . . .	6
4.1.1	Jacobi iteration . . . . .	7
4.1.2	Jacobi iteration on the GPU . . . . .	8
4.1.3	An overview of a simulation time step . . . . .	9
4.2	Visualization . . . . .	10
4.2.1	Refraction/reflection . . . . .	10
4.2.2	Moving a grid of vertices . . . . .	11
<b>5</b>	<b>Results</b>	<b>12</b>
<b>6</b>	<b>Discussion</b>	<b>13</b>
6.1	Conclusion . . . . .	13
6.2	Future Work . . . . .	14
<b>A</b>	<b>Screen shots</b>	<b>14</b>
<b>B</b>	<b>Running the program</b>	<b>15</b>
<b>C</b>	<b>Fragment and vertex programs</b>	<b>16</b>
C.1	Calculation of depth . . . . .	16
C.2	Calculation of coefficients (vertical) . . . . .	17
C.3	Jacobi iteration (vertical) . . . . .	17
C.4	Validating depths . . . . .	17
C.5	Reduction in volume-calculation . . . . .	18
C.6	Distribution of volume . . . . .	18
C.7	Visualization with reflection / refraction . . . . .	19
C.8	Moving Vertices . . . . .	21
<b>D</b>	<b>Literature</b>	<b>21</b>

# 1 Introduction

For many applications in virtual reality and computer animation, a realistic looking simulation of water is useful. In this project we investigate a model for water simulation proposed by Michael Kass and Gavin Miller in which a water surface is represented as a height-field (a 2D grid of heights). This makes the computational complexity much smaller than when using full three dimensional model while still retaining some realism for large water surfaces like oceans. The bottom of lake or ocean is also represented as a height map, which means that bottom conditions can influence the simulation.

Our focus is the implementation of this model of water simulation on a Graphics Processing Unit and the subsequent visualization of the result. In the simulation we use Jacobi iteration for solving linear systems. For the visualization we use a method based on the refraction and reflection of light on the water surface.

# 2 Previous work

As mentioned in the introduction, the work done in this project is mostly based on the paper *Rapid Stable Fluid Dynamics for Computer Graphics* [KASS] by Michael Kass and Gavin Miller. The water model in this paper will be described in detail in section 3. Here follows a description of some related projects.

Many others have used the Kass and Miller model in their work on water simulation. Rory Middleton and Sabrina Nielsen use the model in their masters thesis on real-time simulation and visualization of water. Here they even experiment with using the GPU for an explicit version of the simulation. Failing at this, they develop another explicit model that is more stable.

In 'Linear Algebra Operators for GPU Implementation of Numerical Algorithm' [LINALG], Jens Krüger & Rüdiger Westermann describe their work on developing a framework for performing arithmetic operations on matrices and vectors using the GPU. Here they represent matrices as sets of diagonal vectors, which are stored as textures. As an example they use the 2D wave equation, which when discretized and approximated by finite differences will lead to a linear system, in which the coefficient matrix is banded with 6 bands. These banded matrices are especially efficiently stored using their model. They use a GPU version of the conjugate gradient method for solving the systems. In our work we solve a similar, but slightly more complex problem. We solve the problem first in one direction then in the other, which reduces it to one-dimensional problems. This leads to tridiagonal matrices, and these can be stored even more efficiently using the four colour components of an RGBA-texture. Krüger and Westermann also discuss a reduction operation that we use in this work. This reduction technique is also treated in Ian Buck and Tim Purcell's "A Toolkit for Computation on GPU's" [GEMS] exemplified by a max reduction.

Emil Persson also known as 'humus' has made a water simulation that runs

entirely on the GPU (<http://www.humus.ca/index.php?page=3D&&start=8>). The wave movement is based on a spring-field. Although this is an entirely different approach from ours, his visualization has been an inspiration. For our visualization, we have also been inspired by the refraction demo in “Cg Toolkit: Users Manual” [CG].

### 3 Rapid, Stable Fluid Dynamics for Computer Graphics

In this project we have decided to focus on a model proposed by Michael Kass and Gavin Miller in [KASS]. In this model a water surface (of e.g. a lake or an ocean) is represented as a discrete two-dimensional grid of the water-heights. At the same grid points, the heights of the bottom of the lake (or the sea floor) is stored. Using a very simplified version of the Navier-Stokes equations the simulation of the water-surface can be described either as an explicit expression or as a number of linear systems - one for each row and one for each column in the grid - depending on the integration method used.

We now start out by looking at a single one-dimensional strip of water heights. To derive the equations used for the simulation in the Kass model, one starts with making some assumptions that will be used to simplify the Navier-Stokes equations:

- The water surface can be described as a height field
- The vertical component of the velocity of water molecules can be ignored
- The horizontal component of the velocity of the water in a vertical column is the same all over the column

The first two assumptions are especially erroneous when there is rough water (i.e. when the waves are very steep or even turn over). The third assumption fails when there is turbulent water or for example when there is much friction at the bottom. These assumptions lead to the *shallow water equations*:

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + g \frac{\partial h}{\partial x} = 0, \tag{1}$$

$$\frac{\partial d}{\partial t} + \frac{\partial}{\partial x}(ud) = 0 \tag{2}$$

Here  $u$  is the vertical velocity,  $h$  is the water surface height, and  $d$  is the water depth given by  $d(x) = h(x) - b(x)$ .  $b$  is a height map of the sea floor.

Equation 1 is an expression of Newton’s law  $F = ma$  while equation 2 expresses conservation of volume. Equations 1 and 2 can be further simplified if the second term of eq. 1 is ignored and a linearization around a constant value of  $h$  is made. The results are:

$$\frac{\partial u}{\partial t} + g \frac{\partial h}{\partial x} = 0, \quad (3)$$

$$\frac{\partial h}{\partial t} + d \frac{\partial u}{\partial x} = 0 \quad (4)$$

The above simplifications are justified if the bottom is slowly varying and the velocities are small. By differentiating eq. 3 by  $x$  and differentiating eq. 4 by  $t$  and combining the resulting expressions, the following equation can be derived:

$$\frac{\partial^2 h}{\partial t^2} = g d \frac{\partial^2 h}{\partial x^2} \quad (5)$$

To be able to solve the above numerically, a discretization is needed. When working with the discrete one-dimensional arrays of heights, Kass and Miller report that the following discretizations of eq. 3 and 4 are the most stable they have found:

$$\frac{\partial u_i}{\partial t} = -g \frac{(h_{i+1} - h_i)}{\Delta x} \quad (6)$$

$$\frac{\partial h_i}{\partial t} = \frac{(d_{i-1} + d_i)}{2\Delta x} u_{i-1} - \frac{(d_i + d_{i+1})}{2\Delta x} u_i \quad (7)$$

Using these expressions in eq 5 one arrives at the following:

$$\frac{\partial^2 h}{\partial t^2} = -g \frac{(d_{i-1} + d_i)}{2(\Delta x)^2} (h_i - h_{i-1}) + g \frac{(d_i + d_{i+1})}{2(\Delta x)^2} (h_{i+1} - h_i) \quad (8)$$

Now we need a discretization of the left hand side of the equation into time steps. This is found by twice applying a first order derivative discretization:

$$\begin{aligned} \ddot{h}_i(n) &= \frac{\dot{h}_i(n) - \dot{h}_i(n-1)}{\Delta t} \\ &= \frac{1}{\Delta t} \left( \frac{h_i(n) - h_i(n-1)}{\Delta t} - \frac{h_i(n-1) - h_i(n-2)}{\Delta t} \right) = \\ &= \frac{h_i(n) - 2h_i(n-1) + h_i(n-2)}{\Delta t^2} \end{aligned} \quad (9)$$

Where the dots mean differentiation with time and the index in the parenthesis denotes which time step is used.

Using this we have a fully discretized expression, we can use for the simulation:



for each row in the other direction (the y-dimension). In the following we call these directions the horizontal and vertical directions. This kind of integration method is called alternating dimension integration.

Because the water surface during the simulation can come into contact with the bottom, it is necessary to take active measures to conserve the water volume. To do this Kass and Miller suggest scanning the surface to find connected pieces of water, and then finding the volume of these. Comparing this volume with the initial volume (found before the simulation started), one can then distribute the difference over the heights in a connected area.

To prevent the simulation from exploding it is also necessary to prevent the water surface from moving below the bottom. When having water heights below the bottom Kass and Miller suggest placing the height at bottom-height minus  $\epsilon$  (a very small number).

### 3.1 An explicit expression

According to Kass, the shallow water equations are very badly conditioned for explicit integration. To examine the stability of the simulation when integrating explicitly, we will try using the following reformulation of 10:

$$\begin{aligned}
 h_i(n) = & 2h_i(n-1) - h_i(n-2) - \\
 & g(\Delta t)^2 \frac{(d_{i-1}(n-1) + d_i(n-1))}{2(\Delta x)^2} (h_i(n-1) - h_{i-1}(n-1)) + \\
 & g(\Delta t)^2 \frac{(d_i(n-1) + d_{i+1}(n-1))}{2(\Delta x)^2} (h_{i+1}(n-1) - h_i(n-1)) \quad (16)
 \end{aligned}$$

Here all heights and depths on the right hand side are all evaluated from the previous expression. This expression was proposed by Rory Middleton and Sabrina Nielsen in [RORYSAB]. Here we will again try to use an alternating dimension scheme.

## 4 GPU implementation

This section contains a description of our GPU-implementation of the water simulation model described in section 3.

### 4.1 Simulation

Programming on a GPU is different from programming on a CPU. Instead of chunks of random access memory one uses textures, and instead of looping over this memory to perform a task, a lot of small fragment programs are run in parallel by rendering them, and each find a little part of the solution.

In our implementation we use Cg (C for graphics) and RenderTextures by Mark Harris (<http://www.markmark.net/misc/rendertexture.html>). These make it possible to render the result of a fragment program into a pbuffer that can later be used as an input-texture in another fragment (or vertex) program. Where in a texture to read input is controlled by texture coordinates that can be interpolated by the graphics hardware. In this way one can render a piece of geometry and give each fragment of this different input by setting the appropriate texture coordinates in the vertices. The output of a fragment program is the color of the fragment.

To perform a task in a GPU the problem must be formulated in such a way that it is divided in many small task, that can each find their own little part of the result *without knowing the results of the other programs* and *without changing the result of the other programs*. In Gaussian elimination, when treating a row to find the echelon form it is necessary to change the values of all matrix-coefficients (and the right sides) in the rows below. This is not easily transformed to fragment programs without needing to render the same geometry very many times. As we shall see, Jacobi iteration is very suited for a GPU implementation.

#### 4.1.1 Jacobi iteration

To solve the linear system from eq. 11 we use the numerical method of Jacobi iteration. The method is derived by looking at each of the equations in a linear system  $Ax = b$  one at the time, and isolate  $x_i$  while holding all the other  $x$ 's constant. This means the equation

$$\sum_{j=1}^n a_{ij} = b_i \quad (17)$$

is used to find an iterative formula where, provided some conditions are met, all  $x_i^{(k)}$  are closer (or at least as close) to the correct solution as  $x_i^{(k-1)}$ :

$$x_i^{(k)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(k-1)}}{a_{ii}} \quad (18)$$

For the Jacobi iteration method to be guaranteed to converge towards the correct solution for any starting vector, the coefficient matrix  $A$  must be diagonally dominant.

As mentioned before this is the case for the matrix used in the water simulation. In the following we will show this for the center rows of the matrix. There are special cases in the first and last row of the matrix - here similar arguments apply. We formulate the condition of diagonal dominance in the following way:

$$|a_{ii}| > \sum_{j=0, i \neq j}^{n-1} |a_{ij}| \Rightarrow |e_j| > |f_{j-1}| + |f_{j+1}| \quad (19)$$

Here we have used the tridiagonal structure of the matrix. To simplify the following discussion we introduce the quantity  $\rho$ :

$$\rho = g(\Delta t)^2 \frac{1}{2(\Delta x)^2} \quad (20)$$

In the following we assume that we have nonnegative depths ( $d_i \geq 0$ ). The assumption of nonnegative depth is justified because even though the water height can be set below the bottom, we make sure not to get negative numbers when calculating the depths. Inserting the expressions for  $e_i$  and  $f_i$  in eq. 19 we get:

$$\begin{aligned} |e_i| > |f_{i-1}| + |f_{i+1}| \Rightarrow \\ |1 + (d_{i-1} + 2d_i + d_{i+1})\rho| > |(d_i + d_{i+1})\rho| + |(d_{i-1} + d_i)\rho| \end{aligned}$$

Using the above assumption of nonnegative depths we can remove the numerical signs:

$$\begin{aligned} 1 + \rho d_{i-1} + 2\rho d_i + \rho d_{i+1} > \rho d_i + \rho d_{i+1} + \rho d_{i-1} + \rho d_i \Rightarrow \\ 1 > 0 \end{aligned}$$

And we conclude that the matrix is diagonally dominant.

#### 4.1.2 Jacobi iteration on the GPU

The Jacobi iteration method is very interesting in the context of general purpose GPU-computation, because it is very parallelizable. For each step, the calculation of the new value  $x_i^{(k)}$  is not dependent on the other  $x$ 's in step  $k$ . Therefore each  $x_i^{(k)}$  can be calculated in a fragment shader that uses a texture containing the  $x_i^{(k-1)}$ 's as input. The result will be stored at the place in the texture, where the fragment is rendered. Now this texture can be used as input in a new iteration.

Notice that in the Jacobi iteration method, no matrix coefficients are changed. This means that once the coefficients have been calculated and stored in a texture, this texture can be used as input to all the following iteration steps.

In our case, because the matrix is tridiagonal, we know that the maximum number of nonzero entries in each row of  $A$  is 3 and how these are positioned relative to the diagonal. We have decided to store the parts of eq. 11 in a texture in a very compact way.

For each position in this texture, corresponding to the position where  $h_i$  is stored in the height texture, a RGBA color is placed containing all the information needed to evaluate eq. 18.

- In the red component  $a_{i,j-1} = f_{i-1}$  is stored (when  $j = 0$ , zero is stored)

- In the green component  $a_{ij} = e_i$  is stored
- In the blue component  $a_{i,j+1} = f_{i+1}$  is stored (when  $j = n - 1$ , zero is stored)
- In the alpha component the right hand side  $b_i = 2h_i(n - 1) - h_i(n - 2)$  is stored

Now to calculate  $x_i^{(k)}$  in a Jacobi iteration step, one only needs to do the following:

- lookup  $x_{i-1}^{(k-1)}$  and  $x_{i+1}^{(k-1)}$  in the texture containing the previous heights
- make a lookup in the texture containing matrix coefficients to get  $a_{i,j-1}$ ,  $a_{ij}$ ,  $a_{i,j+1}$ , and  $b_i$
- evaluate eq. 18 and store the result.

#### 4.1.3 An overview of a simulation time step

In our simulation we at time step  $n$  maintain RenderTextures with only red color components containing the heights at time steps  $n - 1$  and  $n - 2$ . The vertical part of a simulation step proceeds as follows:

- From the heights  $h(n - 1)$  and a texture containing the bottom heights we calculate the depths. For this we use a fragment program that simply reads the surface height and the bottom height from the textures and returns the difference. If the surface height is less than the bottom height zero is returned. See appendix C.1 for the fragment program.
- Now a RenderTexture containing the coefficient matrix is constructed as described above. This is done using the fragment programs in appendix C.2. Special programs is used in the boundaries where one of the coefficients are set to zero. In the 6 coefficient programs (three in each direction) a factor containing the term  $\frac{g(\Delta t)^2}{2(\Delta x)^2}$  is sent as an argument.  $\Delta t$  is calculated from the frame rate and  $\Delta x$  is found by dividing the real physical size of the simulated water by the number of grid points in each direction.
- After this a number of steps of the Jacobi iteration are performed. See appendix C.3 for the fragment program.
- Now using another fragment program (see appendix C.4) all heights that are below the bottom are set to bottom-height minus  $\epsilon$ .

The horizontal part of the simulation step proceeds identically except that the fragment programs used here use offsets in the x-direction instead of the y-direction on the texture coordinates when looking up neighbours.

When using this scheme it is necessary to maintain an extra RenderTexture for temporary heights. During each iteration the heights at  $n - 1$  remain unchanged while the RenderTextures containing the temporary heights and the heights at  $n - 2$  are used for rendering the Jacobi program back and forth. At the end of the iteration the old heights at  $n - 1$  become the heights at  $n - 2$  and the result of the Jacobi iteration becomes the new heights at  $n - 1$ .

For the mechanism maintaining a constant water volume we need to calculate the current volume. This is done by first adding all heights and then multiplying by the area covered by each grid rectangle.

The addition of heights is a reduction operation as described in [LINALG]. To calculate the sum we repeatedly render quads using a program that sums up four neighbouring values and returns the result. In this way at each reduction iteration we produce a quad of half the size in both directions. Although this is a little costly, it is necessary for a correct simulation. For the fragment program see appendix C.5.

The volume calculation is the only thing in our implementation that makes assumptions about the dimensions of our grid - because of the reduction, the grid must be the same power of two in both directions. The reduction ends in a texture of size 1x1 that is read back from the graphics card and used to determine how much water (possibly a negative amount) to distribute. Half of this water<sup>1</sup> is distributed evenly over the entire quad (even at places that are dry) using the fragment program in appendix C.6.

## 4.2 Visualization

No simulation of physical phenomena is any good if the result is not in some way shown to the user. In our program the simulation can be visualized in multiple ways. The heights, depths, and matrix coordinates can be directly projected onto the screen. This is done by using a very simple fragment program that just looks up a value in the relevant texture and returns it.

The most sophisticated visualization we have implemented is based on reflection and refraction of light. More on this in section 4.2.1. Finally we have been experimenting with displaying the heights by displacing vertices. Unfortunately we had performance problems with this. See section 4.2.2

### 4.2.1 Refraction/reflection

We have implemented a visualization that is based on the physical laws of reflection and refraction. To find the color in a fragment, we start by finding a normal to the water surface at that point. This done by looking up the

---

<sup>1</sup>compensation for all the water difference in one step did sometimes lead to instability

horizontal and vertical neighbours in a height-map-texture and constructing a tangent and bi-normal from these. By using a cross product and normalizing we get a normal  $\vec{n}$  to the surface. As an input to the fragment program we send a view vector  $\vec{v}$  indicating from which direction, the fragment is seen. This view vector is reflected<sup>2</sup> on the surface using the calculated normal, and the result is used to make a lookup into a cube map containing the surroundings. From this we get the color of the reflected light.

The refraction vector  $\vec{r}$  is found using the Cg-function `refract` using  $\vec{n}$ ,  $\vec{v}$ , and a relative index of refraction. This is used to make a lookup in a bottom texture. To find the correct texture coordinates, the following parallax like method is used:

- A three dimensional point  $\vec{p}$  that describes the position of the surface element seen on the fragment is found as  $\vec{p} = (tx, height(tx, ty), ty)$ , where (tx,ty) are texture coordinates. Notice that we use texture rectangle coordinates here - using ordinary texture coordinates between 0 and 1 would change the effect. Most ideally we should send the real x and y coordinates as separate texture coordinates, use these for the first and third components, and then compensate in the lookup.
- A scaling factor  $t$  is calculated as the ratio between the depth  $d$  of the water (at the point of the fragment) and the y-component of the refraction vector  $t = d/(\vec{r}.y)$ .
- Now we have found a primitive measure of the point  $\vec{h}$  where the refracted light hits the bottom:  $\vec{h} = \vec{p} + t\vec{r}$ . Using the first and third component of  $\vec{h}$  as texture coordinates we look up the color of the refracted light in a texture.

To scale between the two colors, we use the following approximation to the Fresnel equation:  $fast\_Fresnel(\vec{v}, \vec{n}, \alpha, \beta, \gamma) = \alpha + \beta(1 - \vec{v} \cdot \vec{n})^\gamma$ , where  $\alpha$ ,  $\beta$ , and  $\gamma$  are constants. Linear interpolation is now made between the two colors using the Fresnel term as ratio.

When implementing the above, care must be taken not to mix up world and object space coordinates.

Because we want to be able to clearly see when there is no water above the bottom at a particular position, we simply use only the color from the bottom texture when the water depth is nearly zero.

#### 4.2.2 Moving a grid of vertices

Using the vertex shader profile `CG_PROFILE_VP40` it is possible to look up information from a texture in a vertex program. This we have used in the vertex

---

<sup>2</sup>Using the Cg-function `reflect`

program shown in appendix C.8 to lookup a height from a texture and displacing the vertex accordingly.

Unfortunately this resulted in a strange performance penalty, dropping the frame rate to below 1 frame per second when moving only 4 vertices. Because of this we decided not to spend time implementing a grid with one vertex per height-value in the simulation.

## 5 Results

The result of the implementation work described in section 4 is a program that simulates a water surface area of 100x100 meters. How to obtain and operate this program is described in section B.

We have measured the frame rates (and thereby the simulation rates) of the program using a machine with an Intel P4 3.0GHz and a NVIDIA geforce 6800 with 128 MB ram. We have experimented with different grid sizes and number of Jacobi iterations in each direction (for a discussion on the number of Jacobi iteration steps needed see section 6). We have used the reflection/refraction visualization. The results are summarised in the following table:

Frames per second:

grid size \ iteration steps	26	11	6
64x64	78	136	178
128x128	83	135	175
256x256	47	93	132
512x512	13	28	49 (*)
1024x1024	3	7	12 (*)

The grid sizes here are chosen because these are the ones compatible with our volume calculation algorithm. In the runs marked with (\*) there were graphical artifact probably indicating that a larger number of iteration steps were needed.

It is seen that it has practically no (and in one case a negative) effect on the frame rate to go from a grid size of 64x64 to one of 128x128. Our hypothesis is that because of these rather small texture sizes, the performance is dominated by the many context switches between different RenderTextures and not the actual rendering.

We have also tried using only 2 iteration steps for the 256x256 grid size. Here the simulation rate was 202 frames per second. As can be seen, the simulation runs at interactive rates.

## 6 Discussion

The simulation combined with the visualization based on reflection and refraction gave a visually very pleasing result. Phenomena like lens magnification effect are captured by the simple model for visualizing the water bottom. Experimenting with different initial height maps for the bottom and water surface revealed a very different animation of the surface when the water was rather deep compared to when it was quite shallow.

As mentioned earlier we have also been experimenting with varying the number of Jacobi iteration steps. It was revealed that even very few steps (5 steps or less) resulted in a stable simulation, although the result became more realistic with a larger number of steps (until the number of steps was large enough for it to converge to the right solution). This means one can find a tradeoff between performance and reliability.

We have been experimenting with using the explicit expression of eq. 16 for the simulation. We again first used the equation in one direction and then in the other. Although we expected we needed to use smaller time-steps when using this expression, we were surprised how much less stable than the implicit solution it was. Using time steps even remotely comparable to the ones in the implicit simulation, it would explode when making small perturbations to the water surface.

The Kass and Miller model offers a realistic animation of water in situations that are often used in virtual reality applications (oceans for example). This it does at a computational complexity that makes it suitable for interactive applications. In situations where interaction with the water is not needed one might be better off with a model, where the wave movement can be procedurally calculated e.g. by sine functions. One situation, where one might find an interactive simulation of water interesting is in computer games, but here using the GPU for the simulation is maybe not a good idea, because the graphics processor is often well occupied.

### 6.1 Conclusion

As described, we have successfully transferred the water simulation model proposed in [KASS] into an implementation that runs on a GPU with little computational load on the CPU. Screen shots of the running program can be seen in appendix A. For information about frame rates see section 5. An introduction to the program and a link to the running code can be found in section B.

In the simulation process the numerical method of Jacobi iteration was used because it is very parallelizable and therefore very suited for use on a GPU. The iterative process involved in the Jacobi iteration is very useful in this context because the tradeoff between realism and simulation speed can be made to fit a particular use. The simulation is very stable - especially compared to an explicit expression covering the same model.

We visualize the water in a way that reflected light displays the surrounding sky and refracted light displays the sea floor. These are combined based on the viewing angle. The visual appearance of the water is very convincing.

## 6.2 Future Work

The following are suggestions to what could be done to continue the work described in this report

- The visualization using vertex displacement would provide a much more accurate view of the water surface height - especially compared to the bottom. Therefore it would be useful to find out the source of the performance penalty we got and make the visualization work.
- When calculating the water volume we use a simple midpoint rule for the integration. Using a more elaborate method would give a more precise estimate of the volume and thereby a more smooth volume conservation.
- In the current implementation the volume distribution is done evenly all over the grid. It would be better to first investigate which places were not dry and distribute over these. This investigation could be done as another reduction operation. Even better would be to identify connected regions of the surface and then maintaining the volume of these individually (as suggested in [KASS])
- For compatibility with ATI graphics adaptors and to investigate speed differences, it would be interesting to make a version of the simulation using ordinary texture coordinates (relative) instead of texture rectangle coordinates (absolute).
- We have been using 32 bit precision in our calculations. It could be interesting to try using 16 bit precision and make a comparison on frame rate and stability
- There are other methods for solving or finding an approximate solution to a linear system. It might be worth the experiment to try e.g. the conjugate gradient method.
- Comparing the GPU implementation with a CPU implementation might also give valuable information.

## A Screen shots

Screen shots demonstrating the implementation can be seen in figure 1. In the top left picture a 'beach' has been created by initializing the bottom height

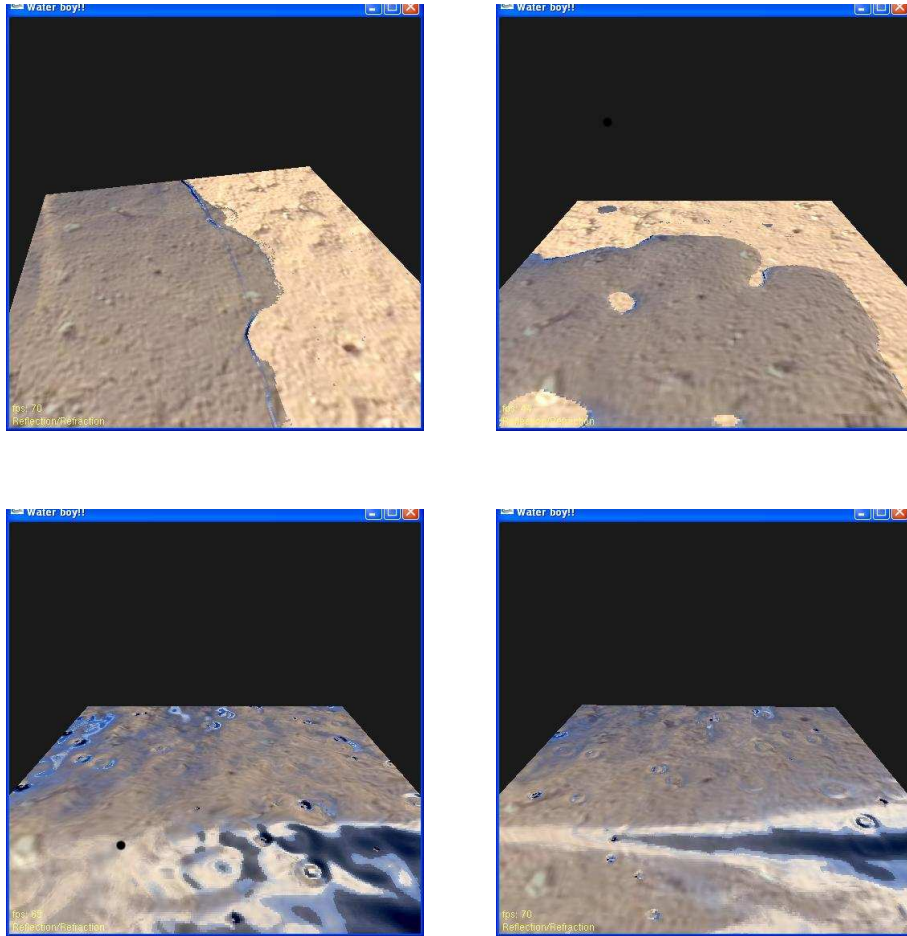


Figure 1: Screen shots

map with values increasing by  $y$ -value and then lowering the amount of water. In the picture on the top right the bottom has been initialized from a bitmap, which results in a little puddle being formed. On the lower left turbulent water is shown and on the lower right a rather large wave can be seen. For the screen shots we have chosen 26 Jacobi iteration steps in each direction.

## B Running the program

A running version of our implementation running on a geforce 6800 can be found on the URL [www.daimi.au.dk/~kn/gpu-water/gpu-water.zip](http://www.daimi.au.dk/~kn/gpu-water/gpu-water.zip). It requires Windows XP / 2000, OpenGL drivers, Cg, devIL, glew, and glut. A Visual

Studio .NET solution and source code is included.

The executable is called 'Kasswater.exe'. When the program is running the user has some options. These can be accessed using the mouse (press the right mouse button for a menu) or using the keyboard:

- 't' changes the visualization mode
- 'r' toggles the 'rain' that drips into the water to make movement
- 'escape' or 'q' closes down the program
- '+' and '-' changes the amount of water in the simulation
- 'e' switches over to simulation using the explicit expression. The simulation explodes immediately (the same time step size is used as in the implicit simulation).
- With the mouse menu three different water levels can be selected, and the simulation can be reset
- After clicking the left mouse button the quad containing the refraction / reflection visualization can be rotated.

When selecting a new water height a wave will emerge. This is to make an immediate animation of the water. It is highly recommended to move the quad around and to try toggling off the rain and see the resulting animation of the water.

## C Fragment and vertex programs

### C.1 Calculation of depth

```
fragment_out calculate_depths( vertex_fragment IN,
                              uniform samplerRECT h_tex,
                              uniform samplerRECT b_tex)
{
    fragment_out OUT;
    float h = texRECT(h_tex, IN.Text_0).r;
    float b = texRECT(b_tex, IN.Text_0).r;
    if(h<b)
        OUT.Color.r = 0;
    else
        OUT.Color.r = h - b;

    return OUT;
}
```

## C.2 Calculation of coefficients (vertical)

```
fragment_out calculate_coeffs_vert_middle(vertex_fragment IN,
                                         uniform samplerRECT h_1_tex, // h-1
                                         uniform samplerRECT h_2_tex, // h-2
                                         uniform samplerRECT d_tex,
                                         uniform float factor)
{
    fragment_out OUT;
    float d_here = texRECT(d_tex, IN.Tex_0).r;
    float d_plus_1 = texRECT(d_tex, float2(IN.Tex_0.x,IN.Tex_0.y+1)).r;
    float d_minus_1 = texRECT(d_tex, float2(IN.Tex_0.x,IN.Tex_0.y-1)).r;
    float right = 2.0 * texRECT(h_1_tex, IN.Tex_0).r - texRECT(h_2_tex, IN.Tex_0).r;
    float f_here = -factor*(d_here+d_plus_1);
    float f_minus_1 = -factor*(d_minus_1+d_here);
    float e = 1.0 + factor * (d_minus_1 + 2*d_here + d_plus_1);
    OUT.Color = float4(f_minus_1, e, f_here, right);
    return OUT;
}
```

## C.3 Jacobi iteration (vertical)

```
fragment_out jacobi_iteration_vert(vertex_fragment IN,
                                   uniform samplerRECT coeff_tex,
                                   uniform samplerRECT h_tex)
{
    fragment_out OUT;
    float h_plus_1 = texRECT(h_tex, float2(IN.Tex_0.x,IN.Tex_0.y+1)).r;
    float h_minus_1 = texRECT(h_tex, float2(IN.Tex_0.x,IN.Tex_0.y-1)).r;

    float4 coeffs = f4texRECT(coeff_tex, IN.Tex_0);
    float sum = h_minus_1*coeffs.r + h_plus_1*coeffs.b;
    OUT.Color.r = (coeffs.a - sum) / coeffs.g;

    return OUT;
}
```

## C.4 Validating depths

```

fragment_out check_depths( vertex_fragment IN,
                           uniform samplerRECT h_tex,
                           uniform samplerRECT b_tex)
{
    fragment_out OUT;
    float h = texRECT(h_tex, IN.Tex_0).r;
    float b = texRECT(b_tex, IN.Tex_0).r;

    if (h<b)
        OUT.Color.r = b-EPSILON;
    else
        OUT.Color.r = h;

    return OUT;
}

```

## C.5 Reduction in volume-calculation

```

fragment_out reduce(vertex_fragment IN,
                    uniform samplerRECT texture)
{
    fragment_out OUT;

    float2 tex2 = float2(2*IN.Tex_0.x + 0 , 2*IN.Tex_0.y - 1);
    float2 tex3 = float2(2*IN.Tex_0.x - 1 , 2*IN.Tex_0.y + 0);
    float2 tex4 = float2(2*IN.Tex_0.x - 1 , 2*IN.Tex_0.y - 1);
    float a = texRECT(texture,2*IN.Tex_0).r;
    float b = texRECT(texture,tex2).r;
    float c = texRECT(texture,tex3).r;
    float d = texRECT(texture,tex4).r;
    OUT.Color.r = a+b+c+d;
    return OUT;
}

```

## C.6 Distribution of volume

```

fragment_out distribute(vertex_fragment IN,
                       uniform samplerRECT texture,
                       uniform samplerRECT bottom,
                       uniform float offset)
{
    fragment_out OUT;

```

```

float a = texRECT(texture,IN.Tex_0).r;
float bot = texRECT(bottom,IN.Tex_0).r;
if(a >bot)
    OUT.Color.r = a + offset;
else
    OUT.Color.r = a;
return OUT;
}

```

## C.7 Visualization with reflection / refraction

```

fixed fast_fresnel(float3 I, float3 N,
                  float3 fresnel_value)
{
    fixed power = fresnel_value.x;
    fixed scale = fresnel_value.y;
    fixed bias = fresnel_value.z;

    return bias + pow(1.0 - dot(I,N), power) * scale;
}

vertex_fragment vertex_main( app_vertex IN ,
                            uniform float4 cam_pos)
{
    vertex_fragment OUT;
    // Get OpenGL state matrices
    float4x4 ModelView = glstate.matrix.modelview[0];
    float4x4 ModelViewProj = glstate.matrix.mvp;

    // Transform vertex
    OUT.Position = mul( ModelViewProj, IN.Position );
    // Pass through texture coordinate
    OUT.Tex_0 = IN.Tex0;
    OUT.view_vec = normalize(mul(ModelView, IN.Position ));
    // OUT.view_vec = (cam_pos -IN.Position ).xyz;
    return OUT;
}

fragment_out fragment_main(vertex_fragment IN,
                            uniform samplerRECT Texture2D : TEXUNIT0,
                            uniform samplerCUBE cube_map :TEXUNIT1,
                            uniform samplerRECT floor_tex : TEXUNIT2,
                            uniform samplerRECT depths : TEXUNIT3)

```

```

{
    fragment_out OUT;
    float4 col1 = texRECT(Texture2D, IN.Tex_0).xyzw;
    float t1 = texRECT(Texture2D,float2(IN.Tex_0.x + 1,IN.Tex_0.y)).x;
    float t2 = texRECT(Texture2D,float2(IN.Tex_0.x - 1,IN.Tex_0.y)).x;
    float t3 = texRECT(Texture2D,float2(IN.Tex_0.x ,IN.Tex_0.y + 1)).x;
    float t4 = texRECT(Texture2D,float2(IN.Tex_0.x ,IN.Tex_0.y - 1)).x;
    float4x4 mit = glstate.matrix.invtrans.modelview[0];
    float4x4 inverse = glstate.matrix.inverse.modelview[0];

    // we transform to world space
    float3 tangent = mul(mit,normalize(float4(0,t1 - t2,1,0))).xyz;
    float3 binormal = mul(mit,normalize(float4(1,t3 - t4,0,0))).xyz;

    float3 normal = normalize(cross(tangent,binormal));
    float3 fresnel_term = fast_fresnel(-IN.view_vec,normal,float3(5.0,1.0,0.15));

    float4 tempVec;
    tempVec.xyz = reflect(IN.view_vec, normal);
    tempVec.w = 0;
    float3 reflVec = tempVec.xyz;

    float4 refl = texCUBE(cube_map, reflVec);
    float3 refrVec = normalize(refract(IN.view_vec, normal, 0.9));
    tempVec = float4(refrVec.x,refrVec.y,refrVec.z,0);
    refrVec = mul(inverse,tempVec).xyz;

    float3 point = float3(IN.Tex_0.x ,texRECT(Texture2D,IN.Tex_0).x, IN.Tex_0.y);

    float depth = texRECT(depths, IN.Tex_0);

    float t = depth / refrVec.y ;

    float2 texcoord = (point + t*refrVec).xz;
    float4 refr = texRECT(floor_tex,texcoord);

    if(depth > 0.009)
        OUT.Color.rgb = lerp(refr, refl, fresnel_term);
    else
        OUT.Color.rgb = refr*1.1;
    return OUT;
}

```

## C.8 Moving Vertices

```
vertex_fragment vertex_mover( app_vertex IN,
                             uniform float4 cam_pos,
                             uniform samplerRECT h_tex : TEXUNIT6 )
{
    vertex_fragment OUT;

    // Get OpenGL state matrices
    float4 position = IN.Position;
    float4x4 ModelViewProj = glstate.matrix.mvp;

    position.y += texRECT(h_tex, IN.Tex0).r;

    // Transform vertex
    OUT.Position = mul( ModelViewProj, position );
    // Pass through texture coordinate
    OUT.Tex_0 = IN.Tex0;
    return OUT;
}
```

## D Literature

[GEMS] Ian Buck & Tim Purcell, A Toolkit for Computation on GPU's, 2004, chapter 37 from the book 'GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics'

[CG] NVIDIA Corporation, 2004, Cg Toolkit: Users Manual Release 1.2, NVIDIA Corporation.

[HARRIS] Mark Harris, Fast Fluid Simulation on the GPU, 2004, from the book 'GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics'

[KASS] Michael Kass & Gavin Miller, 1990, Rapid, Stable Fluid Dynamics for Computer Graphics, SIGGRAPH 1990.

[NA] David Kincaid & Ward Cheney, 2002, Numerical Analysis: Mathematics of Scientific Computing, Brooks/Cole.

[LINALG] Jens Krüger & Rüdiger Westermann, 2004, Linear Algebra Operators for GPU Implementation of Numerical Algorithms, Technical University of Munich.

[RORYSAB] Rory Middleton & Sabrina Nielsen, 2004, Real-time Simulation and Visualization of Water, DAIMI.

[WEISSTEIN] Eric W. Weisstein et al. "Jacobi Method." From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/JacobiMethod.html>  
<http://mathworld.wolfram.com>