Linköpings Universitet

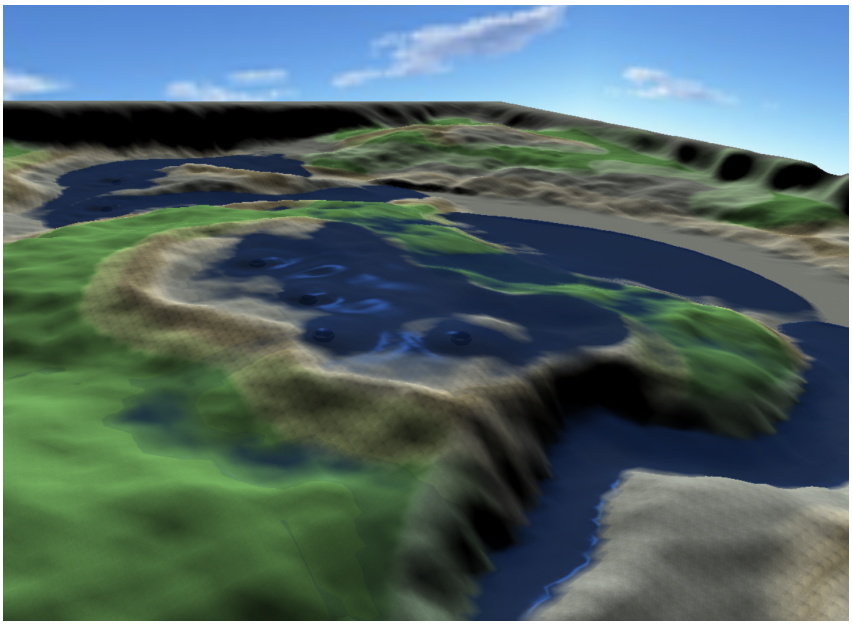
TSBK03

Height Field Water Simulation

Author:
Marcus Wallin

LiU-id:
marwa079

December 2019



Contents

1 Introduction

This report is made for the course *Advanced Game Programming—TSBK03* at Linköping University, as a description of the final project for the course. The project that has been done is a water simulation program, which is to a large extent based on M. Müller and N. Chentanez work in *Real-time Simulation of Large Bodies of Water with Small Scale Details* [1].

The result of the project is a water simulation that is using a grid-based approach, which makes it able to handle water moving through arbitrary terrain. In Fig. 1 an example of the result can be seen, where the water has been moving down a slope and is creating waves and ripples on a small lake.

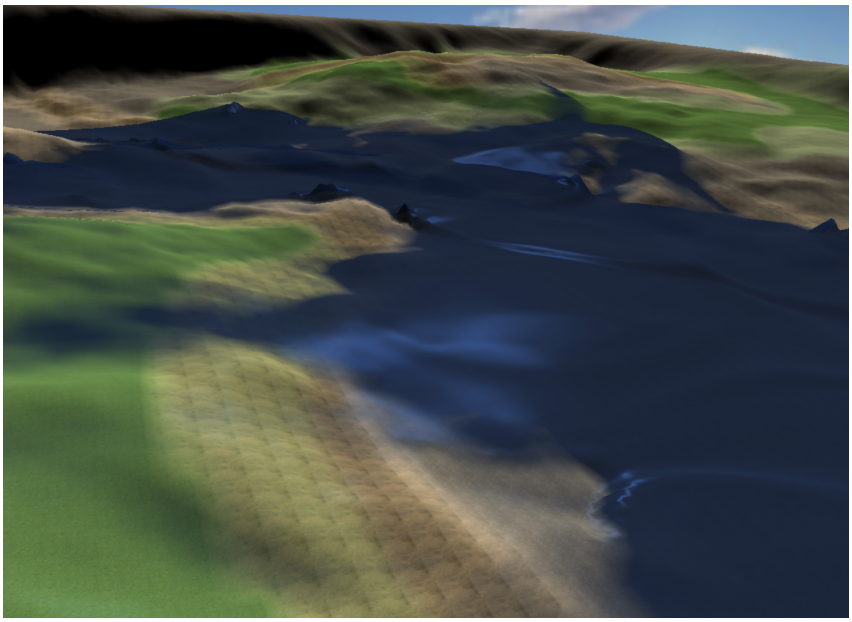


Figure 1: Final result of the water simulations.

In Table 1 the initial requirements can be seen, where it can be seen that the mandatory requirements has been fulfilled, and some of the non-obligatory as well. The status of partially on requirement 7 comes from the implementation of "wells" that can be generated on the terrain to act as water sources and drainages.

2 Background Information

This chapter will present previous work in the field of water simulation. First the Navier-Stokes equation will be presented, then this will be re-written and simplified for the use in staggered grids (a.k.a. MAC-grid).

Table 1: Requirement specification of the project.

Req. No.	Description	Importance	Status
Req. 1	Visualise water represented as a 2D-grid.	**Mandatory**	**OK**
Req. 2	Calculate the water movement with Shallow Water Equation.	**Mandatory**	**OK**
Req. 3	The water shall be in a nature environment.	**Mandatory**	**OK**
Req. 4	The water shall have a bottom with height differences.	**Mandatory**	**OK**
Req. 5	The water shall be able to move through height differences.	**Preferred**	**OK**
Req. 6	At certain conditions water particles shall be created from the water mass.	**Preferred**	**-**
Req. 7	Object shall be able to interact with the water.	**Addition**	**partially**

2.1 The Navier-Stokes Equation

The movement of water is described by the Navier-Stokes equations [2],

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} + \nu \nabla^2 \vec{u} \tag{1}$$

$$\nabla \cdot \vec{u} = 0 \tag{2}$$

where $\vec{u}$ in the equations is the velocity of the fluid, $\vec{g}$ is external forces such as gravity, p is the pressure, ν is the kinematic viscosity, and ρ is the density of the fluid. Eq. (1) is called the momentum equation and describes the movement of the fluid, and Eq. (2) is called the incompressibility condition.

The term $\vec{u} \cdot \nabla \vec{u}$ is called the *advection term* and explains that the divergence of the velocity will affect itself. This will for example mean that the fluid will move faster in narrow sections [3]. The term $\frac{1}{\rho} \nabla p$ is called the *pressure term*, and describes that areas with high pressure will flow towards places with low pressure. The $\nu \nabla^2 \vec{u}$ term is called the *viscosity term*, and describes the thickness of the fluid. Fluids with high viscosity ν will affect surrounding regions more than those with a low value.

2.2 Staggered Grid

The Navier-Stokes equations can describe any fluid, but solving the equation analytically and receiving the general solution for any system is impossible. Therefore, simplifications need to be done, and the first of such is to discretize the water domain into a fixed number of points. This is done by Müller and Chentanez in a staggered grid, a.k.a. a MAC grid [1] in two dimensions, but it can also be done in three dimensions if wanted. For real time applications such as games, the results of the 2D simulations can be very realistic, and since the computational cost is much lower than for 3D, this is what was chosen here.

The staggered grid implementation used for this project can be seen in Fig. 2. At each cell center (i,j) two heights are stored, h is the water depth and H is the height of the terrain, as can be seen in Fig. 3. On the border to every cell the perpendicular velocity is stored,

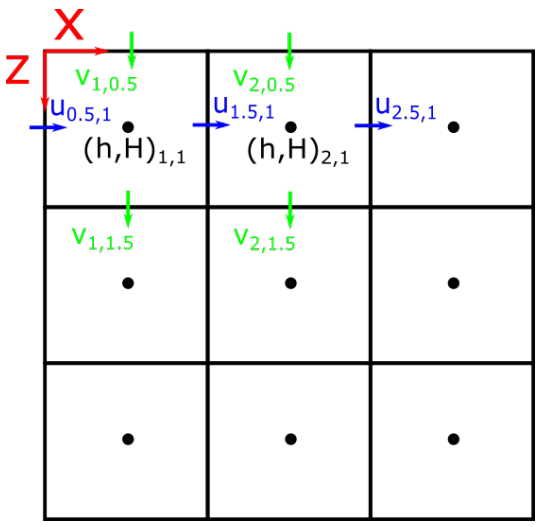


Figure 2: Staggered grid in 2D. h is the height of the water and H is the height of the underlying terrain. The velocities are stored in

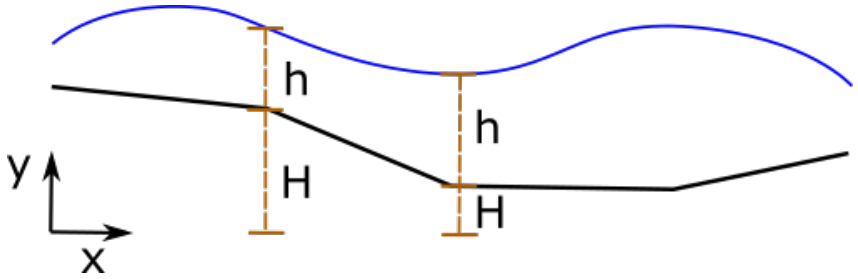


Figure 3: Sideways view of the terrain and water heights.

where u is the horizontal velocity in x-direction and v is the same in z-direction. Note here that the notation of (i+0.5, j) and (i, j+0.5) is used to denote the velocity index.

2.3 Simplification of Navier-Stokes Equation

Using the staggered grid representation presented in the previous chapter, the Navier-Stokes equation can be simplified to the following expressions [4]:

$$\frac{\partial h}{\partial t} = -\vec{u} \cdot \nabla h - h \nabla \vec{u} \tag{3}$$

$$\frac{\partial u}{\partial t} = -\vec{u} \cdot \nabla u - g \frac{\partial h}{\partial x} \tag{4}$$

$$\frac{\partial v}{\partial t} = -\vec{u} \cdot \nabla v - g \frac{\partial h}{\partial y} \tag{5}$$

As previously, $\vec{u} = (u, v)^\intercal$ is the horizontal velocity in x and z directions (the notation is taken from [4]). The first term on the right side in eqs. (3) to (5) are the advection

terms, and the second is the pressure, which can be compared to the original Navier-Stokes equation in Section 2.1.

3 Calculations

Calculation of eqs. (3) to (5) in a staggered grid can be done in many ways, but the problem is to make the system relatively stable. This can be achieved with implicit calculations [5], meaning that the equations are calculated several steps into the future—by for example using Runge-Kutta integration. For the sake of simplicity and speed, Müller and Chentanez are using simple Euler-integration for the pressure terms [1], and a more advanced semi-Lagrangian method for the advection.

3.1 The Height Pressure Term

For each grid-point (i,j), the pressure part of Eq. (3) can be calculated as [1]:

$$\frac{\partial h_{i,j}}{\partial t} = -(\frac{(\bar{h}u)_{i+\frac{1}{2},j} - (\bar{h}u)_{i-\frac{1}{2},j}}{\Delta x} + \frac{(\bar{h}v)_{i,j+\frac{1}{2}} - (\bar{h}v)_{i,j-\frac{1}{2}}}{\Delta x}) \tag{6}$$

Here Δx is the grid size and $\bar{h}$ is an approximation of the height between two nodes, which is calculated using Eq. (7) for the x-direction, and similarly for the z-direction [1].

$$\bar{h}_{i+\frac{1}{2},j} = \begin{cases} h_{i+1,j} & \text{if } u_{i+\frac{1}{2},j} \leq 0 \\ h_{i,j} & \text{if } u_{i+\frac{1}{2},j} > 0 \end{cases} \tag{7}$$

After calculating this, the height $h_{i,j}$ is integrated with Euler-integration:

$$h_{i,j} \mathrel{+}= \frac{\partial h_{i,j}}{\partial t}\Delta t \tag{8}$$

3.2 The Velocity Pressure Term

The velocity term in Eq. (4) in the x-direction is integrated using the following equation, also with Euler-integration [1].

$$u_{i+\frac{1}{2},j} \mathrel{+}= (\frac{-g}{\Delta x}(\eta_{i+1,j} - \eta_{i,j}) + a_x^{ext})\Delta t \tag{9}$$

Here $\eta = h + H$ is the total height of the water and the terrain. This calculation is done similarly in the z-direction.

3.3 The Advection Terms

Due to the short time-span of this project, there was no time investigate a stable method to calculate the advection terms in eqs. (3) to (5), and therefore the implemented formula was done in the most straight forward fashion.

The advection term from Eq. (3) in the height direction was implemented in the following way, and was integrated with Euler integration.

$$\frac{\partial h_{i,j}}{\partial t} = -\frac{(h_{i+1,j} - h_{i-1,j})}{2\Delta x}\frac{(u_{i+\frac{1}{2},j} - u_{i-\frac{1}{2},j})}{2} - \frac{(h_{i,j+1} - h_{i,j-1})}{2\Delta x}\frac{(u_{i,j+\frac{1}{2}} - u_{i,j-\frac{1}{2}})}{2} \tag{10}$$

Attempts at calculating the advection terms in the velocity equations (eqs. (4) & (5)) was made, but the result was too unstable to be used. Müller and Chentanez are writing in [1] that they are for the advection: *"using an unconditionally stable modified MacCormack method ... and fall back to the semi-Lagrangian method if the resulting value is not within the bounds of the velocity values used for bilinear interpolation of the first semi-Lagrangian sub-step"*.

The MacCormack and semi-Lagrangian methods are both ways of performing stable calculations of the advection, and are both performing error compensation and correction by a combination of back and forward tracing the advection in time [6].

If there would have been more time available for the project, the semi-Lagrangian advection would be the next step in implementation, but at the time being it is not included in the final result.

3.4 Boundary Conditions

Just like Müller and Chentanez in [1], the boundaries of the water simulation are treated as reflective, which is done by setting the velocities to 0 at the end of every time step. Additionally: a face in the x-direction (and similarly for the z-direction) is considered reflective if either of the following cases are true,

- $h_{i,j} \leq \epsilon$ and $H_{i,j} > \eta_{i+1,j}$
- $h_{i+1,j} \leq \epsilon$ and $H_{i+1,j} > \eta_{i,j}$

where $\epsilon = 10^{-4}$ in the simulations. What this condition means is that the water must be higher than the neighbouring dry terrain if the water should be able to flow there.

4 Implementation

The implementation of the program was done on Windows in Visual Studio using C/C++. Most of the water calculations are done on the CPU, but could quite easily be extended to be done on the GPU instead.

4.1 Objects & Data Storage

The largest part of the project was created in an object oriented way, objects are responsible for keeping track of the models, textures, calculations etc. In Fig. 4, an overview of which objects contains which can be seen. The world object is the main object and is holding the terrain and water object, which are both inheriting from the abstract class *Grid* that essentially works as a two dimensional array. The terrain object is initiated along with a black and white texture that functions as a bump map that will create the terrain according to the values of the image, which will be stored in the height_array. When the terrain has been generated, this will be input to the water object that will copy a part of the terrain to it's own height_array, and if the resolution is higher than the grid size of the terrain, will use bilinear interpolation to create grid points in between the nodes of the larger terrain resolution. The water object also contains another Grid object that stores the velocities. Since these are stored on the boundary of the nodes (see Fig. 2), these are stored in a special way in the 2D array, which is explained in Fig. 5. If the grid size of the height nodes

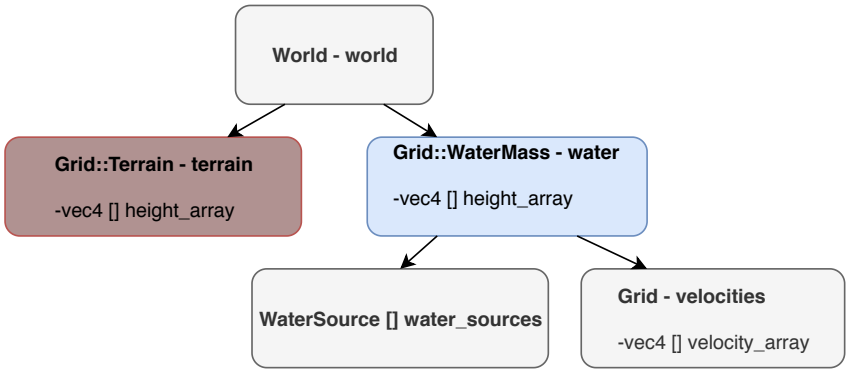


Figure 4: Object overview of the main part of the simulation. An arrow from an object means that it contains the object it is pointing to.

are (s_x, s_z), then the size of the velocity array will be $(s_x + 1, 2s_z + 1)$. The water object also holds an array of WaterSources, which can be placed by the user and will essentially override the water calculations and set the water height to a fixed value at the position where it is placed, making it into some sort of pump or drain.

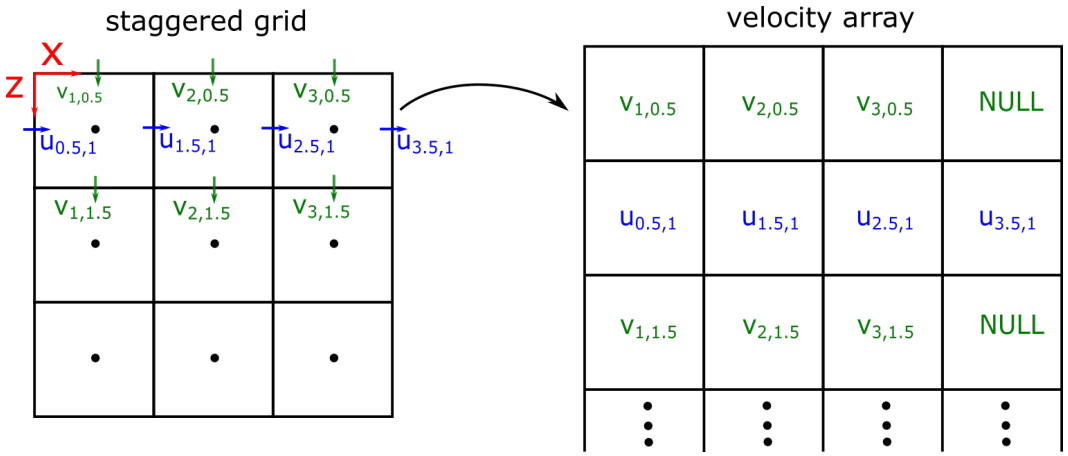


Figure 5: Explanation of how the velocities are stored in the grid, where the left image is the staggered grid and the right represents how the data is stored in the 2D grid. A special function for returning the correct velocity was implemented.

4.2 Calculations & Visualisation

The movement calculations of the water are handled by the WaterMass object, and are performed on the CPU. Attempts were made at doing the calculations on the GPU instead, but the time was not sufficient to finish this implementation. How this would be done will be explained in Section 5.1.

The first thing that happens during each time step is that the velocity_array is integrated using the current heights of the water. After this, the water height_array is calculated using

the equations in Section 3 by looping over every node in the array. The new values are then uploaded as a texture to the GPU. The water shaders therefore has access to both the water height and ground height, which are used to calculate the transparency of the water (explained more in Section 5.2). The vertex shader is also calculating the normal of each vertex at every time step using the height data.

5 Interesting Problems

There has appeared many interesting problems during this project, and here I will list a few of them that I find worth mentioning.

5.1 Calculations on the GPU

Simulating the movement of water using the technique specified in this report is very well suited for being performed on the GPU with OpenGL. Why? Because the calculations at each node (i,j) are:

- easy to represent as texture values.
- only dependent on the previous state of the height and velocity.
- only needing memory access from nearby elements in the texture memory (efficient).

Due to time restrictions I had no time to implement this, but for the sake of redeeming myself I will here present a methodology of doing this with the current calculations (without velocity advection).

Four FBO objects will be needed:

- heights1—texture unit 1
- heights2—texture unit 2
- velocities1—texture unit 3
- velocities2—texture unit 4

At each time step one of the two height and velocity FBO's are the current active FBO, and the other one is containing the old values of the simulation. On the CPU there will be two integers stored—height_unit and vel_unit—that is either 1 or 2 depending on which FBO is currently active. The following sequence will explain what needs to be done during each time step:

Step 1—integrate velocities: Velocity integration is done by rendering on a square with a resolution equal to the size of the velocity_array (see Section 4.1). This rendering is done with velocities1 as input to the FBO velocities2, where the result will be stored. The calculations will be done in the fragment shader, and by using the texture coordinates of the current fragment we can know which data to access to perform the integration. It is important to be careful here because the velocities are stored alternately in x and z direction.

Step 2—swap active velocity FBO: When the integration has been performed, the new values should be stored in FBO velocities2. By changing the vel_unit to 2 and uploading this to the GPU we can inform the height integration shader, as well as the drawing shader

that texture unit nr. 4 should be used for velocities. Also swap the pointer in the CPU of the two FBO objects to make the next time step use the same FBO calls.

Step 3—integrate heights: Just like in step 1, we now draw a quad on the sreen but with the resolution of the water grid and into the FBO heights2, with heights1 as input data. Now vel_unit is 2 and a function that retrieves the surrounding velocities from texture unit 4 can be created.

Step 4—swap active height FBO: Now the current height texture is texture number 2, so we change height_unit to 2 and upload it to the shaders, and swap the pointers of the height FBOs on the CPU just like in step 2.

Step 5—draw: Now draw the scene in the normal way, and make sure the water vertex shader has the correct height texture.

repeat: At the next time step this process can be repeated, but we change the height_unit and vel_unit to 1 instead.

5.2 Transparency & Water Realism

When the project had progressed until the point where the simulations started to behave like some sort of liquid, the problem of making it look like water became apparent—at this point the water looked more like some blue blob. After thinking about this problem I came up with four main characteristics that would improve the esthetics:

1. The transparency of the water should depend on its depth and the angle you are looking at it.
2. Water is reflective, especially if still.
3. When looking through water, Snell's law will cause an angle change of the light, making objects appear at places where they not really are due to refraction.
4. The color of the water should depend on the depth and velocity of the fluid, an example is that fast shallow water should be more brown and muddy than still water.

5.2.1 Parallax Mapping

Improvement Nr. 1 was solved by using *Parallax mapping* [3]. It is a technique for approximating the water depth, which is taking into account the angle at which you are looking through the liquid. This is normally used for bump maps, but is suitable here as well.

In Fig. 6 an explanation of parallax mapping can be seen, where L' would be the ideal distance (if we do not take Snell's law into account) to the bottom, and L is the approximated distance by parallax mapping. As can be seen, Parallax mapping estimates the length L as a right-angled triangle, and if $\vec{y}$ is the y-axis and $\vec{v}$ is the direction to the view position (both should be normalized), then the formula for L can be written as Eq. (11). The transparency of the water A should therefore be a function of L, and in the project the function used is Eq. 12.

$$L = \frac{h}{\vec{v} \cdot \vec{y}} \tag{11}$$

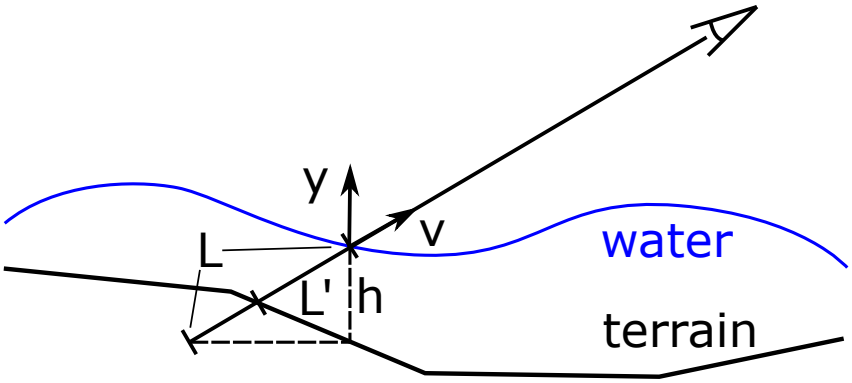
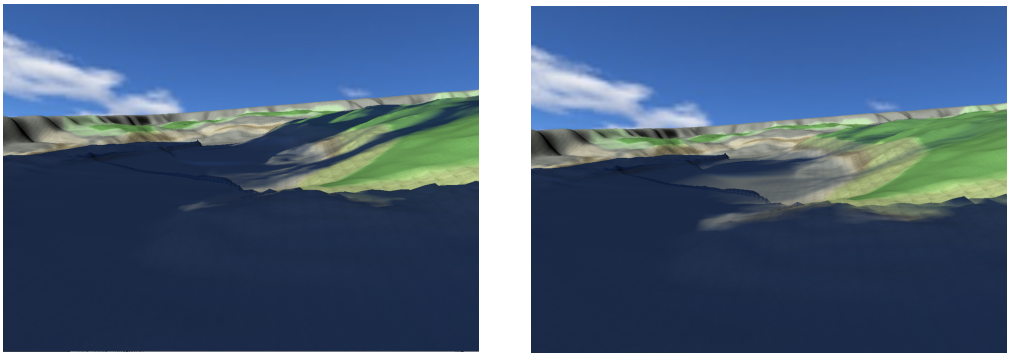


Figure 6: An explanation of parallax mapping, where L' is the ideal distance to the bottom and L is the approximated distance by Parallax mapping.

$$A(L) = clamp(L, 0.1, 0.99) \tag{12}$$

The result of using parallax mapping instead of using the height h straight down is good. A comparison between the two can be seen in Fig. 7, where the exact same scene is rendered with parallax mapping and with $L = h$ in Eq. 12. As can be seen, the transparency of the water that is straight below the viewer is exactly the same, but to the far right where the angle to the water is large, the water is nearly invisible where $L = h$, giving parallax mapping a much more realistic result.



(a) Parallax mapping transparency, $L = \frac{h}{\vec{v} \cdot \vec{y}}$

(b) Height transparency, $L = h$

Figure 7: A comparison of the same scene with Parallax mapping and the vertical height as transparency function.

5.2.2 Reflections

Improvement Nr. 2 was to an extent implemented with simple phong shading in the project, making it so that the sunlight is reflecting on the water surface. A further improvement would be to use environment mapping to also reflect the color of the skybox in the water. This is relatively simple to implement, we already have the view vector $\vec{v}$, and it is simply to

reflect this vector on the surface and look where this would hit on the cube-mapped skybox texture [7].

5.2.3 Refraction

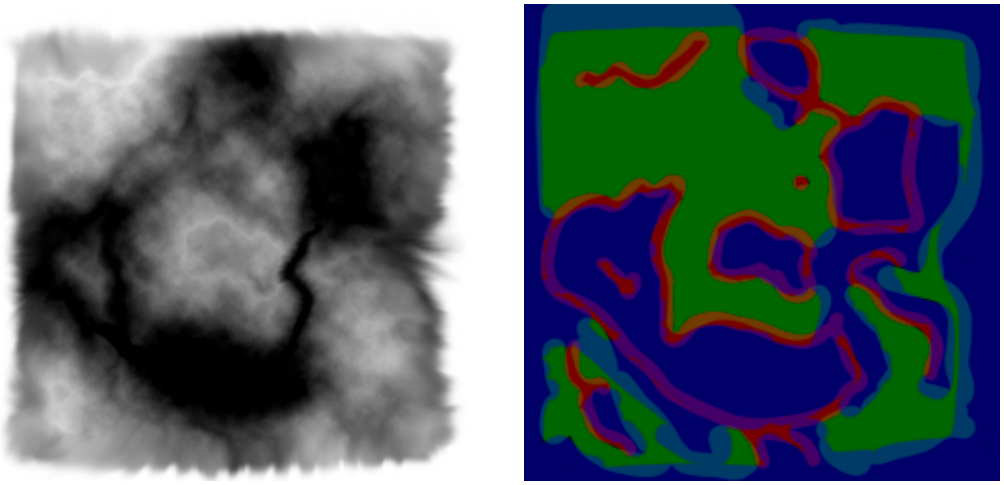
Improvement Nr. 3 is requiring refraction of the light in some way. I personally do not know of any other way than performing ray-tracing of some sort to achieve this effect, which is quite tricky to implement in a real-time scenario. A possible trick that i can think of that could be used to achieve this look without the costly ray-tracing, would be to distort the terrain under the water in some smart way—depending on the viewing position and depth of the water—which would be relatively simple and possibly yield good results.

5.2.4 Water Color

Improvement Nr. 4 would be relatively simple to implement. Simply upload the velocities to the water shader and let the output color be dependent on height and velocity in some way. This could be implemented in a few hours in the current stage of the project.

5.3 Terrain & Ground Texture

To generate the terrain, a bump map was used where the color indicates the offset of the ground. For the sake of trying it out, I asked my girlfriend Julia Jobson (who is good at Adobe Photoshop) to design a map with a river and a lake with mountains around. The result can be seen in Fig. 8a.



(a) The bump map used for terrain generation.

(b) Ground texture selection.

Figure 8: Displaying the self designed

To make the scene a bit more realistic, I wanted to have different types of ground textures as well, and I chose three: sand, grass and gravel. To select which texture should be used we designed a texture with RGB-colors indicating how much of the three textures should be used that can be seen in Fig. 8b, where R=sand, G=grass and B=gravel. The result can

be seen from above in Fig. 9, where it can be noted that the image is mirrored compared to Fig. 8 due to a swap in coordinate systems.

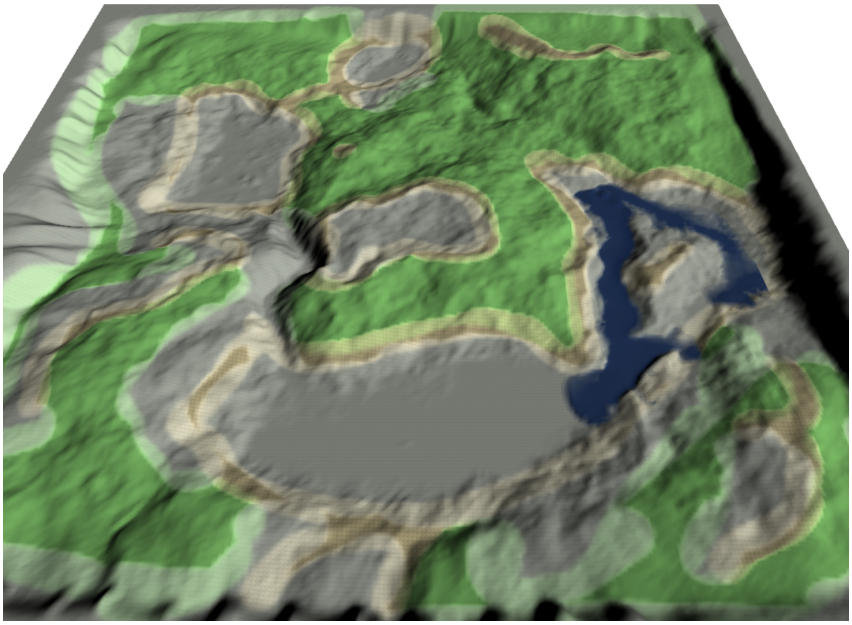


Figure 9: Top view of the terrain, with a bit of water pouring on the right side.

Right now the edges of the colors in Fig. 8b are a bit too rough for making the result realistic, but a bit more work would most likely have yielded excellent results.

6 Conclusions

The project was fun and has an essentially endless amount of improvements that can be made relatively easily.

6.1 Results & Discussion

The results of the simulation are relatively life like, and I am generally happy with the results. All the obligatory requirements from Table 1 was fulfilled, and I did not expect to get any further with the project due to the short time frame of only $\sim$7 weeks.

During this project I have most of all improved my knowledge in the Navier-Stokes equation, and a lot about solving this kind of differential equations. Topics such as stable integration has never before been a problem during any of my projects, but after this one i realized that this is not always a trivial thing, such as in the case for advection which I never really had the time to investigate deep enough to properly understand it.

6.2 Possible Developments

I have already mentioned some future developments in Section 5.1 and 5.2, where I would move the calculations to the GPU and add more realism to the water with environment mapping and velocity dependent color. Some other things that i have not mentioned but that is feasible to do with the model is the following, all of which is done by Müller and Chentanez [1]:

- Interactions with objects (such as boats and logs).
- Velocity advection.
- Splashing particles, generated when waves are breaking or at waterfalls.
- Moving bump maps on the surface of the water to imitate wind.

6.3 Usage in Games

The height field technique has from my perspective a lot of potential for the use in games, and there appears to be a few games that actually do use it (I am only guessing from the looks however):

- Cities: Skylines 2015 [8]
- Hydrophobia 2011 [9]
- Arcane Worlds 2014 [10]

Of course the biggest game here is Cities: Skylines, where rivers can be created that looks very nice. Other large game studios find other ways of making good looking water, but the trend in game programming has always been to make the graphics more and more based on real life physics—so who knows, maybe in a few years we will see game studios like Rockstar Games or DICE including this kind of water in their games.

References

[1] Nuttapong Chentanez and Matthias Müller. "Real-time Simulation of Large Bodies of Water with Small Scale Details". In: *ACM SIGGRAPH Symposium on Computer Animation*. 2010.

[2] Robert Bridson and Matthias Müller-Fischer. *FLUID SIMULATION—SIGGRAPH 2007 Course Notes*. Tech. rep. 2007. URL: http://www.cs.ubc.ca/.

[3] Ingemar Ragnemalm. *So How Can We Make Them Scream*. 2017.

[4] Nils Thürey et al. "Real-time breaking waves for shallow water simulations". In: *Proceedings - Pacific Conference on Computer Graphics and Applications*. 2007, pp. 39–46. ISBN: 0769530095. DOI: 10.1109/PG.2007.54.

[5] Matthias Müller et al. *Real Time Physics Class Notes*. Tech. rep. 2008.

[6] Andrew Selle et al. *An Unconditionally Stable MacCormack Method*. Tech. rep. Stanford University, 2007.

[7] Ingemar Ragnemalm. *Polygons Feel No Pain*. 2017.

[8] *How To Make Realistic Rivers - Cities Skylines Tutorial - YouTube*. URL: https://www.youtube.com/watch?v=D3O7gPOfazA.

[9] *Hydrophobia's Water Physics - YouTube*. URL: https://www.youtube.com/watch?v=c6AsDUXbqZ0.

[10] *Arcane Worlds — Steam*. URL: https://store.steampowered.com/app/269610/Arcane_Worlds/.