Stegen Sander

Simulating real-time interactions between water and dynamic objects

Graduation work 2021-2022

Digital Arts and Entertainment

Howest.be

ABSTRACT

Throwing objects in water is something everyone did as a kid. In this paper we will simulate this interaction between an object and a body of water to be used in games. Games require a real-time solution to be able to run smoothly, thus the focus will be a developing a lightweight solution. The simulation will show how the waves get generated on top of a water surface together with how realistic buoyancy forces get applied to simulate sinking and floating objects. In computer sciences multiple approaches exist to simulate water, with each having their specific use-case. For our interactive water a height-map approach will be combined with a three-dimensional grid for detecting the collision on the displaced surface. Archimedes' Principle will be followed to apply the buoyancy forces back onto the dynamic object when colliding with water. This results in a water simulation where details can be easily controlled by parameters with performance as trade-off. The biggest performance bottlenecks aren't the calculation themselves, but rather the data transfers necessary to perform the calculation on the graphics card.

The final project and source files can be viewed on https://github.com/StegenSander/Water-Interactions

CONTENTS

TABLE OF FIGURES

INTRODUCTION

Water is a common aspect of games; for some games water is a core aspect of the gameplay and for other games its sole purpose is to make the environment look pretty. Getting realistic water in games is not the easiest task because high quality water is expensive to calculate real time. Games and simulations often use low accuracy approximations of reality to strike a good balance between credibility and performance. There are a lot of different methods whit each method having its specific use case. The focus in this project lies with getting visual and physical feedback from objects that collides with water. The visual feedback is common in most water simulation, this visual feedback being waves or ripples on the water whenever an object collides. A proper implementation of the physics part of the simulation is less common and those that exist often only support cubic shapes for collision reasons. This will be the focus of the project. How can interactive water be simulated in real-time together with realistic physics responses on the shape interacting with it?

To achieve this goal multiple wave propagation methods will be investigated and the most suitable one will be chosen and implemented. Wave propagation on water with high details sounds like an expensive task. Thus, the project will be developed with performance in mind. In the end the trade-off between realism and performance will be investigated. Which factors in the water simulation influence the performance the most and how do these compare to the visual quality increase they provide?

A last point of investigation will be the shape of the object and waves. The shape of the waves generated on the water surface should depend on the shape of the object. A cube falling in the water should generate a differently shaped wave than a sphere.

RESEARCH

1. TYPES OF WATER SIMULATION IN COMPUTER SCIENCES

Before going further into depth on waves or specific implementation it is important to define different types of water and implementation. During a presentation by Nvidia about real-time water simulation [1] in 2008. They talked about three different approaches in water simulation. The first one is procedural water; procedural water is the water generated by mathematical functions like trochoid and sine waves. This approach is mainly used for unbound surfaces and oceans. The second approach is particle systems. Particle systems do not work very well on a big scale, so they are mainly used for smaller scale simulation, like splashes or puddles of water. The last approach is using height field. Height fields work best for ponds and lakes. A follow up presentation was given by Nvidia in 2011 [2]. This presentation goes more in depth about the pros and cons of each approach and going into detail about possibly combining multiple approaches based of the required level of detail. Combining approaches is outside the scope of this project.

1.1. PROCEDURAL WATER

Procedural water, like oceans, rely on mathematical functions to simulate waves. The two most used functions for this are the sine and trochoid waves. Trochoid and sine waves are very similar, however there is a subtle difference. A sine wave represents the line created while rotating a circle, while a trochoid wave represents the line created by rolling a circle. Both functions generate a repeating wave-like pattern, but the trochoid wave generates a wave pattern that resembles water waves best [3]. Comparing the two, trochoid waves will always have the upper hand in simulating water waves.

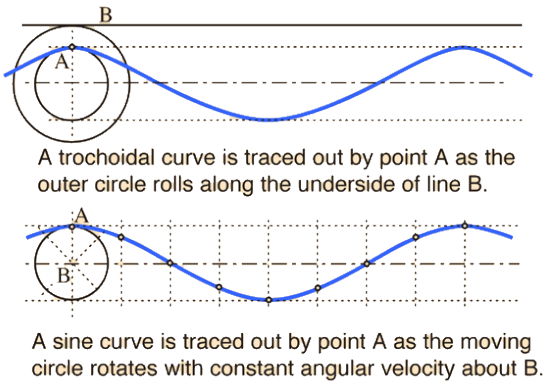

Figure 1: Difference between trochoid wave and sine wave

Trochoid and sine waves can achieve realistic ocean waves with ease. Especially if they get combined and stacked on top of each other. Yet, they will not be used in this project. Using a procedural approach like trochoid and sine waves generate a consistent and repeating wave pattern. Which is very useful for ocean waves but not as useful for interactive water. With interactive water the simulation will be permanently changing by falling or moving objects. To apply all these changes and approximate the water simulation with trochoid and sine waves would, although not impossible, be a very difficult task.

1.1. PARTICLE BASED WATER

In particle based the water gets represented by a collection of individual particles. Each of these particles behave based on a few simple rules and then get rendered together as water. Particle simulation is fast and quite simple, but there is a limited number of particles that can be used without crashing the performance of the program. The particle-based water simulation is mainly used for small scale simulation, like puddles and splashed and is not suitable for the bigger scale simulations that are wanted in this project.

1.2. HEIGHT FIELDS

In the case of height maps the surface of a body of is represented by a two-dimensional (2D) image. Each pixel on this image represent the height of a part of the water. It reduces the complexity from three-dimensional (3D) to 2D. One of the biggest limitation is that breaking waves are not possible because there is no 3D information available. The positive aspect of using height maps is that full control of the simulation is possible. Every pixel of the height maps represent an height in the simulation. Thus these values can be modified however wanted based on set behaviors.

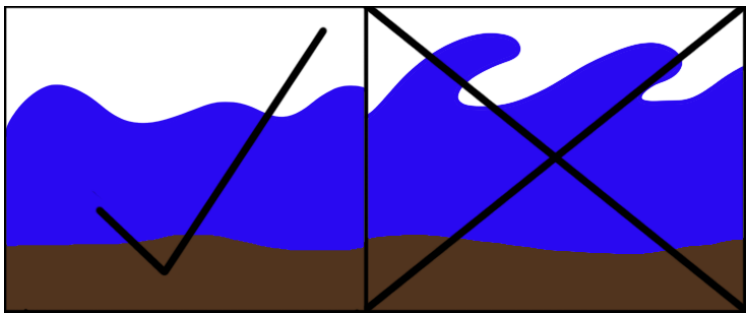

Figure 2: Water represented by height above an underlying terrain

2. WAVES

Now that it is decided to use height map to represent the surface of the water, the question that remains is how this height map need to behave and be updated to represent realistic waves on a water surface.

2.1. WHAT IS A WAVE?

Wave is a broad term and can be interpretated in a lot of different ways depending on the field of interest. For the sake of this paper, the interest lies with water waves which are a form of physical waves. “A physical wave is a dynamic disturbance that propagates over time” [4] . This is the type of behavior that will be desired as result in the wave propagation model.

2.2. BOW AND STERN WAVES

Bow and Stern waves are two types of waves that get generated whenever an object moves through water [5]. Bow and stern waves are not a replacement for physical (water) waves, but they should be seen as a type of physical wave. An object moving forward in water pushes the water aside generating bow waves. At the location where the object moved from is now an open space and water starts flooding in generating stern waves.

Figure 3: Bow waves get generated in front of the boat (1). Stern waves get generated behind the boat (2)

2.3. PHYSICAL WAVES

“A physical wave is a dynamic disturbance that propagates over time” [4]. Thus, this needs to be modelled inside the wave propagation function. Looking at the core definition of a physical wave, the physical waves can be modelled as a circle that grows overtime. To represent this circle in data there are two concrete options. The first option is to represent the waves as a collection of pure circles which can be defined with a center and a growing radius. The second option would be to represent circles and how they grow inside a grid.

When storing waves as a circle with a center and a growing radius, all the most up to date data will always be available and the waves can be drawn onto a height map with high precision. The downside of this approach is that every wave needs to be stored separately in memory. Meaning that during complex simulation with a lot of waves, performance will crumble. A second problem with storing waves as a circle is that it is very hard for waves to interact with static object or other waves. Whenever two waves pass each other, they should collide on only the overlapping parts of the circle. But the circle consist only out of a single part, meaning that to be able to make collision work the circle needs to be split up in smaller segment.

Storing waves directly into a grid already fixes a lot of problem that the previous method would generate. The grid has a consistent size so the simulation can never blow up. And the collision between two waves couldn’t be easier. Two waves will automatically collide whenever a part of their waves tries to move to a grid cell where another wave is already present. The remaining problem here is that the wave needs to properly propagate over the grid cells and interact with the other objects.

2.4. NAVIER-STOKES EQUATION

The Navier-Stokes equation is a precise mathematical model for most flows occurring in Nature. It can be used to model ocean currents, the weather and air flow around the wing of airplane. This equation sounds like a great candidate for interactive water simulation, it can simulate fluids based on 2- or 3-Dimensional flow fields. Which means that multiple velocities and velocity sources can be active at the same time and still get to a precise result.

Navier Stokes equation are perfect for this project, too perfect. Solving Navier Stokes equations by hand was nearly impossible, until in 1950 the help of computers was introduced and to this day these equations still require too much computational power to be handled in real-time. Solving the complex Navier Stokes equations are not suitable for real-time simulations.

2.5. REAL TIME NAVIER-STOKES EQUATION

Navier Stokes equations can come to near perfect results but are too expensive. However, Games don’t require perfection, they require good enough. Instead of solving the equation, try approximating it. The paper “Real-Time Fluid Dynamics for Games” by Stam J. [6], talks about how to simplify the Navier Stokes equation so they can be used in real time. It can not only be used to solve the water in our water simulation, but it can expand to 3D smoke or other simulations.

Instead of trying to solve perfectly, use a density and a flow field (Velocity field) as 2D grids (Or 3D depending on the use case). The flow field represent the flow of the current frame and the Density field represent how much of the particle is present in a specific grid cell. Density in this project will represent the height of the water in a specific grid cell. Every frame the density and the flow field get update in 3 steps. Addition of new sources, things that cause disturbance in the current simulation. Diffuse, every cell spreads a bit of its value to its neighboring cells. Move, move the densities or velocities along the flow. Moving the flow field will move the

flow field along itself, this may seem weird at first, but it works great. Using Stam's Navier-Stokes solver is a suitable and realistic way to make the waves propagate and behave in a 2D grid.

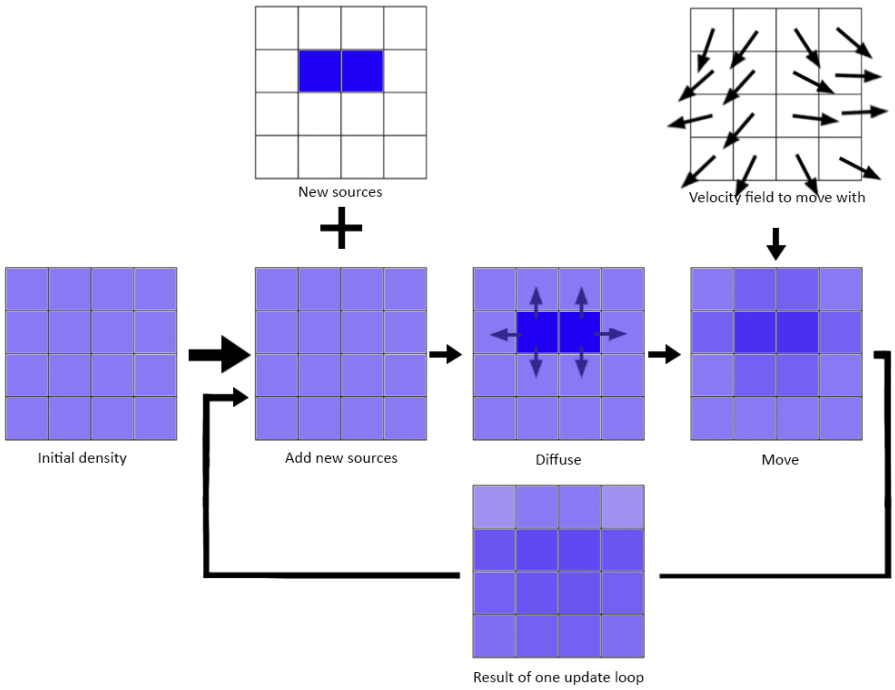

Figure 4: The process of updating a density texture

3. HOW TO VISUALIZE THE HEIGHT OF THE WATER?

There are two ways to visualize the height of the water. The first one is moving the vertices of the mesh itself. Every frame loop over the points of the mesh surface and update their height. The second method is not moving the vertices in the mesh but using a shader which acts like it moved the vertices. The mesh would never actually change but it would still look at if it did. Although the shader is way faster, another crucial factor to take into consideration is collision. By moving the vertices of the mesh, the mesh data can be sent to the physics API for accurate physics detection with the water. However, in the Physics API that will be used in this project, PhysX, it is not possible to make a trigger out of a mesh collider. Meaning that colliding with the water would be possible but going inside of the water would not be possible. Obviously the wanted behavior is to be able to go inside of the water making this method obsolete. When using a shader to represent the height, the mesh and the collider will always remain a cube. Thus, some more advanced collision detection will be necessary.

4. COLLISION DETECTION

Using a shader to visualize the height of the water, comes at a cost. The physics engine doesn't know the shape of the water because this information is only stored inside the heightmap of the mesh and not in the mesh itself. This means that to be able to generate waves from object colliding with the water surface, the physics engine cannot be relied upon, a custom collision detection system is mandatory.

4.1. CAMERA METHOD

The paper "Real-Time Open Water Environments with Interacting Objects" [7] goes into detail on how to detect and handle collision between a body of water and dynamic objects. The most important piece of information to take out of this paper is the use of collision maps. A collision map is a map which contains the position of all the objects that are currently active in the world and that can interact with the water. One collision map on its own does not provide much usable information but compare a collision map with the map from the previous frame and the move information of all objects are available. On the places where previous frame no object was present and now and object is present, water should get pushed away and bow waves are generated. On places where an object is no longer present, water can flood in, and stern waves get generated. To generate these collision maps a camera can be used.

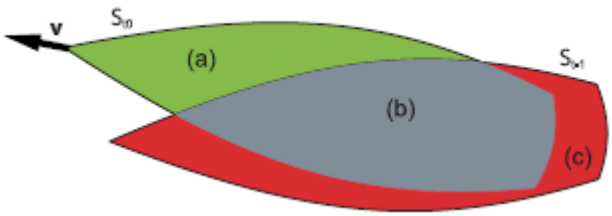

Figure 5: Waves generated by intersection of the current and previous time step collision data. Bow waves are generated in (a) and stern waves in (c) [7]

To use a camera to render collision some requirements need to be met. The camera needs to be orthographic, facing downwards in the water and be perfectly aligned with the water surface. If the camera meets these requirements, a render command can be executed to render only the interactive objects onto this texture. The object that will be rendered onto this texture are all the objects that currently are inside of the water. This resulting texture can be used as a collision map.

The camera method is limited. The camera would render all object that are inside the water. Meaning that if an object is completely underwater, it would still render as an object that is colliding with the water surface. Another limitation is that for the camera to work it needs to be perfectly aligned with the water surface. If a part of the water surface gets displaced by a wave, the collision data is no longer accurate. To fix these issues the depth buffer of the camera can be used to compare it with the height map of the water. But, yet again it causes issues because only the nearest faces of the object will be visible on the camera.

The camera approach is only suitable for certain setups. For the camera to generate accurate collision maps the water surface height needs to be close to constant, if there would be a lot of height variation on the water the camera can give a lot of fake results. Object can be rendered without intersecting with the water, Objects could be blocking the camera's view of intersecting objects and it is hard to tell whether an object is completely submerged or not. The best use-case of this collision camera would be boats on a lake. For our use-case more depth data is required.

4.2. GRID METHOD

Using a camera to render a collision map is not suitable for this project because it lacks depth information. So instead of using a 2D collision map, a 3D collision grid can be used. A 3D grid can store the information of whether there is an object at a certain position or not. This information can be overlayed with the height map to check on which points there is an actual collision with the water surface. This data can be baked into a collision map which can then be used for spawning waves. An important note when comparing this frame's collision data with last frame's collision data. No longer will the 2D collision map be compared, but rather the actual grid data will be compared. To know where new objects appeared in the simulation, it is necessary to know how the objects moved in 3D space, rather than how they moved compared to the water.

5. WATER PHYSICS

The physics of water is a complicated concept in its entirety, but the only interesting part of the water physics is how forces get applied to object, which is a much simpler concept. The physical behavior of the water itself will be handled by the Navier-Stokes equations. Applying forces to objects involves Newton's second law. Newton's second law state that the acceleration of an object, is the force being applied on the object divided by its mass [1].

$$a = \frac{F}{m}$$

The mass of the object is known, but the force being applied on the object is still missing. The force on the object originates from the water. Following Archimedes' principle which states the force of the water is equal to the density of the fluid multiplied by gravity and the submerged volume of the object.

$$F = \rho * g * V$$

5.1. CALCULATING VOLUMES OF TRIANGLE MESHES

To use Archimedes' principle the volume of the objects is required. Calculating the volume of cubic or spherical shapes is easy, but once triangle meshes get involved the calculation are less obvious. The paper "A Symbolic Method for Calculating the Integral Properties of Arbitrary Nonconvex Polyhedra" [8] solved this issue in 1984. Meaning there is a simple and quick solution available to calculate the volume of meshes. It works by calculating the signed volume of every triangle in a mesh. The sum of all the signed volumes results in the precise volume of the mesh. For more details on this topic refer to the mentioned paper.

5.2. FACTORS THAT INFLUENCE THE GENERATED WAVES

To know which factors influence the waves generated by an object intersecting with water, it is important to observe the water in real life. That is exactly what the "Experimental Hydrodynamics of Spherical Projectiles Impacting On a Free Surface Using High Speed Imaging Techniques" [9] paper did. Using a high-speed camera, photos were taken of how the water behaves when an object hit the surface. From this experiment two interesting conclusions were made which are relevant to this paper. The velocity with which an object strikes the water surface influence the height of the water, while the size of the object influences how wide the wave is. These are two factors that will be kept in mind while creating the water simulation.

CASE STUDY

Research pointed out that the best approach for our use-case is by using a 2D height map that represent the surface of the water. On this height map waves will be represented by a density and a velocity according to Stam's implementation of real-time Navier-Stokes equations. To add waves to the simulation a collision grid will be used to check whether objects are colliding with the displaced water surface or not. If an object is colliding or inside of the water, waves get generated accordingly and forces from the water get applied back onto the object based on Archimedes' Principles.

1. WORKFLOW

The water simulation process consists out of three big steps

The first step is the physics step. The calculation of the physics step happens mostly on the CPU and can be split up into two parts. The first part consists of checking all the collision and spawning waves wherever necessary. The second part is responsible for applying the water forces back to the bodies colliding with the water

The second step is the wave propagation step. This step is responsible for updating the state of the water and generating a new height map for the water. Because this step updates and processes a lot of data, this step happens mostly on the GPU in the form of compute shaders.

The third and final step is render step. The render step is a shader which uses the latest height map to visualize and render the water.

2. COMPUTE SHADERS

To be able to process a lot of data at high speeds, compute shaders will be used. A compute shader excels in multithreading repetitive tasks, such as processing textures. Which will be necessary to be able to handle the water simulation in real-time.

While working in compute shaders everything is multithreaded. Every thread that executes a function gets a thread ID assigned, this ID can be used to access the current pixel of the texture it should work on. Because a compute shader is multithreaded it is important that each thread only makes changes to the pixel that matches its ID. Its neighbor pixels are easily accessed by offsetting this ID, but the neighbor pixels should only be read from to guarantee thread safety.

3. DOUBLE BUFFERING

Another problem that can occur while looping over textures is that some data in the texture is already update, while other data is still being processed. This means that if calculation need to look at neighboring cell, they can sample a combination of old and new data, creating fake results. To prevent this the double buffering pattern will be used. Every calculation involving in and output, will read from the input buffer, process the data, and then write to the output buffer. After this the buffers gets swapped, so the output from the previous calculation will always be the input for the next one. Working with two buffer guarantees that old and new data will never get mixed.

4. STANDARDS

To properly understand the upcoming pages, it is important to set some standards and rules the project follows to make everything clear.

- The project is made in Unity 2020.3.13f1, using C#, HLSL and the Unity shader graph
- Unity uses a y-up system

- Textures are assumed to have an aspect ratio of 1by1, other aspect ratios are not supported and result in unexpected behavior
- Texture sizes are easy changeable during compile/build time, but must stay consistent during runtime
- Density means the height of the water at a certain cell
- The density map contains values from [0,1] where 0.5 is the default water height.
- The density map has a default value of 0.5 and the total density needs to be constant across the entire simulation. No water will be added or removed from the simulation
- The velocity map contains values from [-1,1] remapped to [0,1] for visualization.
- The resulting height map contains a float3 for every cell, in the red and green channel the x and y velocities are stored. In the blue channel the density is stored.

5. PHYSICS STEP

5.1. COLLISION DETECTION

To check whether an object is colliding with the water a grid is placed at the top of the water. The grid is not just randomly placed at an area around the top of the water. The lowest and highest point of the grid are aligned with the height that the water will have when its heightmap contains a value of 0 and 1. This makes sure the grid only checks for the necessarily collision and it can ignore any other object.

Every frame all the bodies that are inside the bounding box of the grid are gathered and processed to check which grid cells contains an object and which don't. Whenever a grid cell is inside an object it gets assigned a value between 0 and 1. The value is based on the velocity the object has when colliding with the water. This is because the height of the waves should be dependent on the velocity the object collided with the water as stated in the 'Factors that influence the generated waves' paragraph. The result is a good approximation of where objects are around the water surface. The accuracy of the collision detection can easily be increased or decreased by changing the size/cell count of the grid. In the example below a grid of 32x10x32 cells is used for 10x3x10m bounding box. Increasing grid size will lead to more expensive but more accurate collision checks.

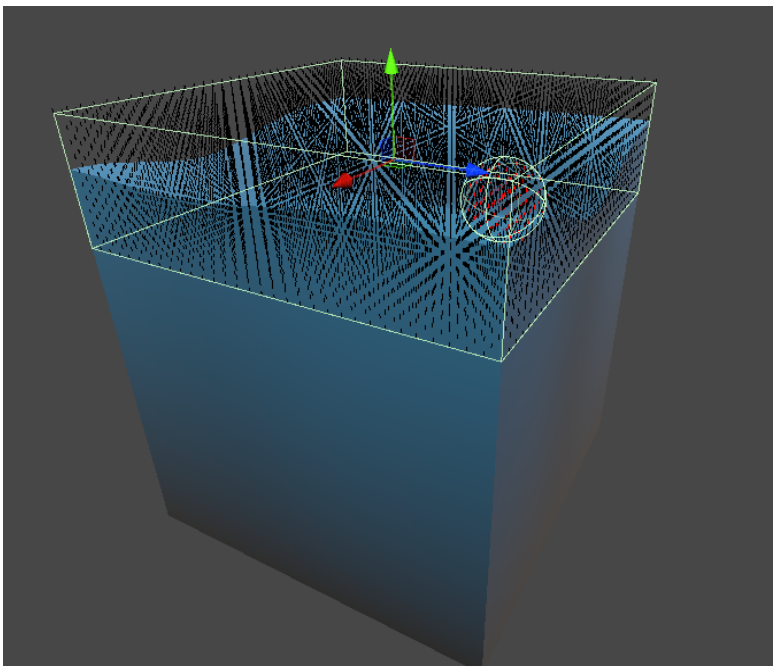

Figure 6: Water with collision grid visualized

To speed up the collision check, not every grid cell gets checked for collision. Only the grid cells inside of the overlapping bounding boxes between the water and the object gets checked. This leads to a major performance increase and no accuracy loss.

Just having a 3D grid and knowing where objects are inside the grid is not enough information to generate waves from. To generate waves a collision map is required. The collision map shows exactly where objects are intersecting with the water. To calculate this collision the height map gets overlayed in the grid and because the height of the water is known, a 3D grid point can be calculate based on UV coordinate and the height. However, this point will rarely be perfectly aligned with a single grid cell, so weighted samples are taken based on all the nearby cells.

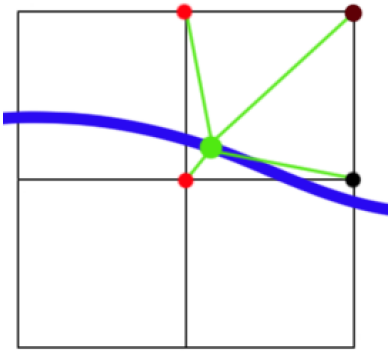

Figure 8: 2D example of a weighted sample

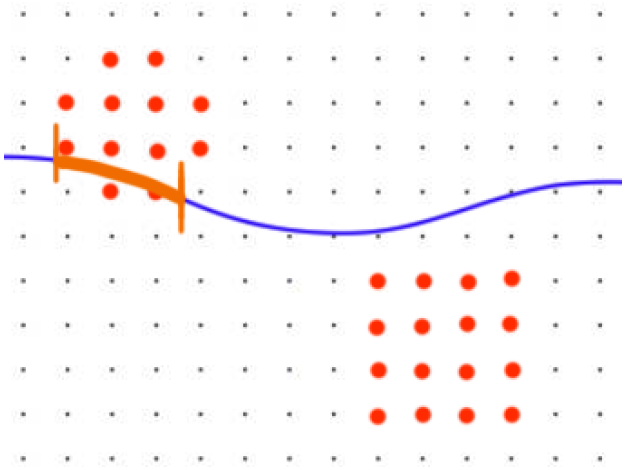

Figure 7: 2D example of water getting overlayed on a grid

An important detail that needs to be mentioned is that the above method works for calculating a general collision map. But for generating waves the desired information is not where the collisions are, it is where the new collision are. So before actually sampling the grid values, subtract the current grid with the grid from previous frame. This makes sure that the collision maps that gets baked is a map that represents all new collision instead of all collisions.

5.2. WATER PHYSICS

The force the water applies onto the object follows Archimedes' principle. Archimedes' principle can be led back to one simple function.

$$F = \rho * g * V$$

The resulting force (F) is dependent on the density of the liquid (ρ), the gravity (g) and the submerged volume (V). The density and the gravity of a liquid body will rarely change in this project. The used liquid in this project is water which has a density of $0.997\ g/cm^3$ and gravity will be a constant of $9.81 m/s^2$. So, the only factor that can influence the force coming from the water is the submerged volume.

To calculate the submerged volume of an object, it is first necessary to calculate the volume of each of the object that can get thrown into the water. Calculating the volume of triangle mesh is a solved issue. It can be done by using the quick and precise implementation discussed in "A Symbolic Method for Calculating the Integral Properties of Arbitrary Nonconvex Polyhedra" [8]. Sample code of this can be found in the appendices [Figure 22]. To further optimize the volume calculation, the volume gets cached for each object because the volume would only change if the object got scaled. Explaining how exactly this volume calculation works is beyond the scope of the project. To go into depth on this topic refer to the original paper on the topic.

Calculating exactly how much of the total volume is submerged is an expensive task. It would require calculating the volume of the overlapping part of two collider. For some object this may seem easy, but once triangle meshes are involved the complexion grows. Rather than calculating the exact overlapping volume, collision particles gets used. These collision particles can be placed all over the mesh and each of these particles represents a part of the total volume. The total submerged volumes can now be defined based on how many particles are submerged. This is only an approximation, but the level of detail can be controlled by adding more collision particles.

To calculate if a particle is submerged a comparison with the height map needs to be made. To be able to do this an expensive function call needs to happen. Currently the height map and all the calculations are located on the GPU. But now the height map is required on the CPU. A GPU readback needs to happen so the data can be stored on the CPU. This needs to happen only once per frame because every object can sample the same height map. But GPU readback are not cheap and performance analysis later on, pointed out that this is one the most expensive functions called during the update loop, but more about that during the analysis.

The collision particles have another great use case. These particles can also receive the forces of the water. By applying the water force at the location of the particles, objects can now tip over if there is more water on one side compared to the other.

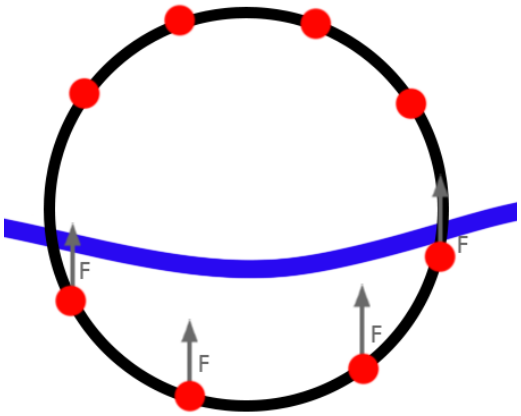

Figure 9: Forces being applied on collision particles of a sphere

The particles can be placed by hand for each mesh for maximum details and efficiency. But for this project they get generated for each mesh. A defined amount of sample points gets taken on the faces of the mesh and then the samples get filtered based on distance to the other samples. Only the samples which are far enough away from the other samples remain. This leads to a surprisingly good results and details can easily be increased by decreasing the minimum distance between points and by increasing the amount of sample points.

6. WAVE PROPAGATION

The wave propagation is based on the paper “Real-Time Fluid Dynamics for Games” [6] by Stam J. It showed how to simplify the Navier Stokes equation to turn it into a real-time solution for computer graphics. A lot of the things mentioned in the paper were directly applicable to the simulation. However, some implementations caused minor problems or minimal changes could be made to increase performance. Overall, all the steps achieve the same goals, the code is just modernized and moved to compute shaders.

Like discussed before the wave propagation exists out of three steps. First all the new data from the physics steps gets added to the simulation. Second the texture gets diffused and for the final step everything gets updated with the velocity field. To make sure all these steps can properly take place. Five buffers or textures in this case gets used. Two density or height textures, two velocity textures and one texture for the final heightmap. The reason two density and velocity texture get used, is to properly implement the double buffering concept mentioned earlier. An overview of the wave propagation process during one frame:

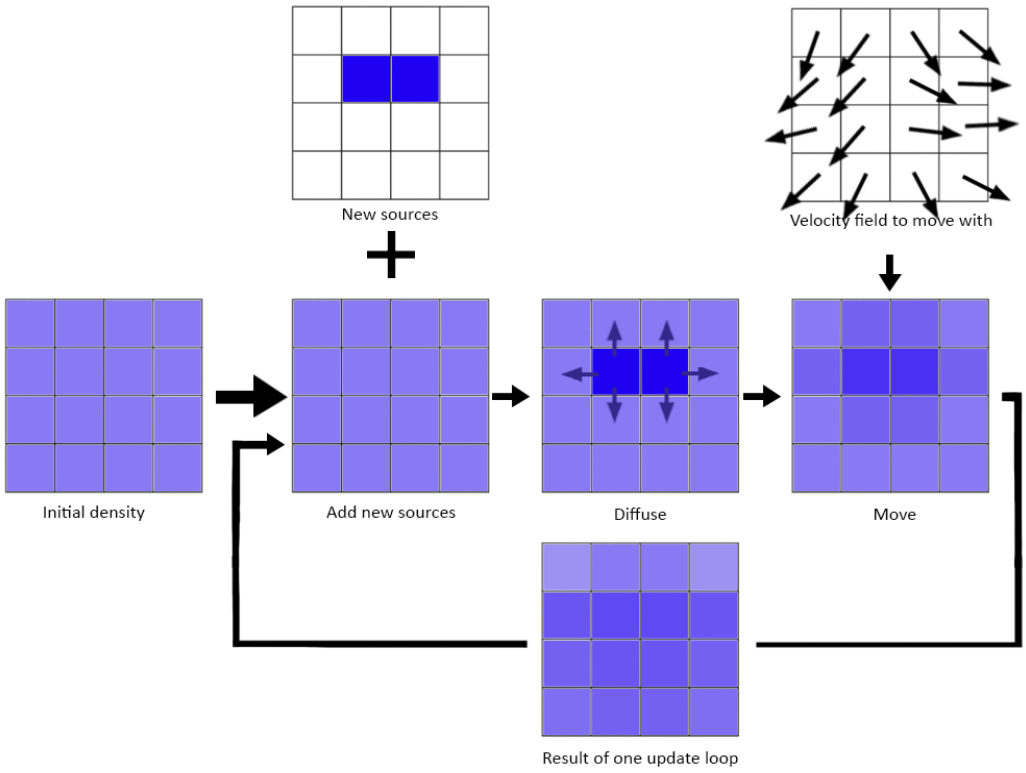

Figure 10: The process of updating a density texture

6.1. ADDING NEW WAVES

To start the simulation, waves need to be added, this gets done by using the output of the physics step. The output of the physics step rendered a collision map of all the new collision data that happened during the past frame. This collision maps gets first transformed into a velocity map containing the resulting velocity caused by the collision and then applied to the current velocity map.

Converting the collision map into a velocity map is a simple step, for every grid cell all the neighbors get checked. If a neighbor is a cell where collision happened, the current cell will apply a velocity to itself pointing away from that neighbor. Doing this for all neighbor and taking the sum of the velocities for each cell will result in a velocity map containing how new waves should be added. This velocity map can then be added to the active velocity map and the simulation can start.

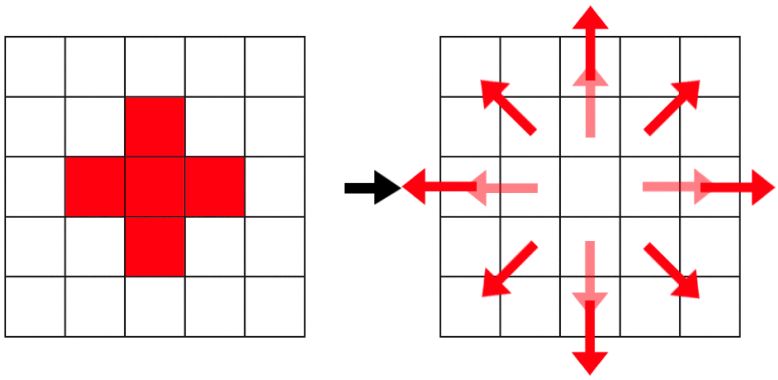

Figure 11: A collision map turned into a velocity map

6.2. DIFFUSING TEXTURES

Diffusing textures is used to make sure the water spreads out overtime and gets back to its normal state of a flat surface. Similar to the converting the collision map to a velocity map, the neighbor cells will get used. The concept is that every cell will give a part of its density to all the nearby cells and will receive a part its neighbors density.

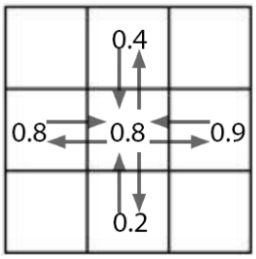

Figure 12: Diffusing a texture

The most obvious way to calculate this is the with the equation below, in which *s* is the diffuse scalar and *d* is the density map

$$float\ s = diffuseRate * deltaTime\ *\ textureSize\ *\ textureSize;$$
$$dOut[x][y] = d[x][y] - 4 * s * d[x][y] + s * (d[x-1][y] +\ d[x+1][y]\ +\ d[x][y+1] +\ d[x][y-1]);$$

However, this implementation doesn't work as well as you might expect. The problem with this equation is the value of the scalar. The scalar is meant to help control the speed of the diffuse. But for big scalar values things start going wrong. Values can become negative and eventually diverge to infinity. Thus, for this reason another method of diffusion is required. This problem and its solution are explained in more detail in the "Real-Time Fluid Dynamics for Games" [6] paper. The solution is to use a more stable method.

$$float\ s = diffuseRate * deltaTime\ *\ textureSize\ *\ textureSize;$$

$$dOut[x][y]\ = \frac{d[x][y]\ + s * (d[x-1][y] +\ d[x+1][y]\ +\ d[x][y+1] +\ d[x][y-1])}{1\ + 4 *\ s}$$

Using this diffuse solver gives a way more stable result and the simulation will no longer blow up for bigger diffuse scalar values. The simulation will no longer blow up, but it does converge to a maximum diffuse speed.

The best way to make the simulation go faster than this maximum diffuse speed, is to do multiple iteration of the diffuse solver each frame.

This diffuse solver works for diffusing the density as well as the velocity. The only difference is that the density map contains one float, the height. And the velocity map contains two floats, the x and y velocity. The above-mentioned diffuse solver is one directly from the “Real-Time Fluid Dynamics for Games” [6] paper. In the actual project very small changes are made, to get more control over certain parameters, but the core of the solver remains the same.

6.3. UPDATING ALONG VELOCITY FIELD

The density and velocity now get diffused, but the water density doesn’t move yet. To move the water, or the density representing the water, the velocity map gets used. Velocities were initially set in the adding waves step now it time to start using them.

The most obvious way to update the density map along the velocity field would be to loop over all the cells and trace the velocity of that cell and check in which cell it ends up. However, to make sure everything is thread safe this is not a valid solution. There would be a write command to a pixel which does not match the current thread ID. So, rather than check where the current cell velocity ends up in, check which velocities of the nearby cells end up in the current cell.

To check if a velocity ends up in the current cell and how much of that velocity end up in that cell, let’s go back to the weighted sample. For each of the surrounding grid cells trace the velocity (scaled with delta time). This traced point will end up somewhere in between four grid cells. If the current cell is one of these four points, a part of the velocity ends up in the cells. Then using the weighted sample, calculate which percentage of the velocity end up in the cell and thus which percentage of the density will be transferred.

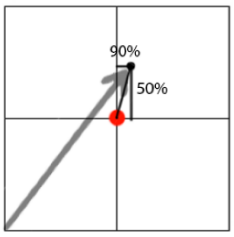

Figure 13: Checking how much a velocity influences a specific cell

Doing this for all surrounding cells will lead to a properly updated density field. There is a catch however, if the velocity would become extremely big, more cells will have to be checked because the velocity can come from cells much further away.

Not only the density map will get updated along the velocity map, also the velocity map will get updated along itself. This may seem weird at first, but the velocity map describes how everything moves so it is the correct way of making sure the velocities propagate together with the water height.

6.4. VOLUME LOSS

None of the wave propagation steps are not a 100% perfect when it comes to maintaining the total volume constant. Water often disappears at the borders of the simulation. However, there is an easy a bullet proof fix [1]. The rule is simple the volume needs to be constant. Compare the current volume of the simulation with the initial volume of the simulation. If there is water missing or water needs to be added. Add or remove water from the simulation spread evenly across all cells. Doing this every frame will result in an invisible, quick and solid solution to keep the water volume constant.

7. THE WATER SHADER

The water shader is made with the Unity shader graph. To properly understand what exactly is going on, it is important to understand the mesh [Figure 24] and its UV space [Figure 23] first. The mesh itself is a regular cube with a heavy subdivided top plane. The more subdivisions this top plane has, the more height details can be shown on the water surface.

The top plane covers the entire UV space. With the side plane directly attached to it. This gives us less control over the texture on the side planes. But this makes sure that the highest points of the side plane are directly overlapped with the edges of the top plane making sure the points sample the same value from the height map. Thus, becoming the exact same height after running the shader.

A last detail that is added to the mesh for optimal shader results, is that the red channel of the vertex color of each points represent the amount of influence the height map has on the point. For the points at the bottom of the mesh this will be 0 the points on the top plane this value will be 1. This makes sure that only the surface points get displaced.

The shader itself exist out of two important steps. The first one is using the height value from the height map to displace the vertices based on the red value of the vertex color, this happens in the vertex shader [Figure 25]. The second step is recalculating the normals for these points. The vertices get displaced so the normals are no longer correct. The Unity shader graph has a very helpful node for this. The node “Normal from height” gives the exact wanted behavior and make sure the shader properly bends the light. Besides moving the points and bending the normal, a slight color change happens based on the height value. Bending the normal and applying the color change happens in the pixel shader [Figure 26].

The shader used for the water is nothing special and is big area of improvement but the scope for this project is generating a good height map. The shader in this project is used to proof that the height map is of good quality and does the job it is supposed to do. Which is mimicking realistic water.

EXPERIMENT & DISCUSSION

1. PERFORMANCE TEST EXPERIMENT

In real-time simulations performance is a very important aspect. An analysis has been made comparing the performance of all the most crucial functions in the project and how they scale when the core parameters change. These tests are run on a Lenovo Legion 5 15IMH05H laptop with following specifications: Intel core i5-10300H (2.50GHz), 16GB DDR4 ram, Nvidia RTX 2060 6GB GDDR6 ram.

The most important functions in the simulation are the functions related to the physics and to the wave propagation. The following functions will be measured:

- Bake collision function (GPU) -> The conversion from 3D collision grid to 2D collision map.
- Adding collider to grid (CPU)-> Mapping the collider position to the grid
- Water forces (CPU) -> Applying forces back to the object based on the heightmap
- Adding new velocities (GPU) -> Converting the collision map to velocity map and applying it in the wave propagation
- Diffusing the textures (GPU) -> Spreading a small part of every cell to its neighbors
- Update along the velocity field (GPU) -> Update the density of the water along the velocity field
- Convert Render Texture to Texture 2D (CPU/GPU)-> Convert the GPU render texture to a CPU Texture 2D for CPU height map sampling

The parameters that will be investigated are the following:

- Texture size (Default: 128 by 128)
- Collision grid size (Default: 32 x 10 x32)
- The number of objects (Default: 1)
- Amount of collision particles (Default: 16/Object -> 16 total)

The performance test itself will goes as follows. Each parameter will be measured separately, meaning that all parameters will always have their default value except the one that is being measured. The chosen parameter will be measured four to five times with different values that make sense in the given context. For each of these different values the simulation gets ran and ten time-measurements of every function are made. From these ten measurements of every function the smallest and biggest value are removed to avoid outliers. The remaining values with be averaged and the estimated time will be presented in milliseconds (ms). The numbers in red indicate interesting changes in the data.

Texture size:	32x32	128x128	512x512	1024x1024	4096x4096
Bake collision	0,0365	0,0368	0,0353	0,0347	0,0400
Adding collider to grid	0,0944	0,0908	0,0903	0,0945	0,0727
Water forces	0,0328	0,0357	0,0339	0,0299	0,0245
Adding new Velocities	0,0139	0,0114	0,0107	0,0120	0,0080
Diffusing Textures	0,0090	0,0071	0,0066	0,0103	0,0082
Update along velocity field	0,0060	0,0057	0,0055	0,0074	0,0069
Convert Render Texture to Texture2D	1,3353	1,5497	4,1053	6,6052	41,5531

Figure 14: Analysis of the influence of the texture size on the performance, measured in milliseconds

Collision grid size:	16x5x16	32x10x32	64x20x64	128x20x128
Bake collision	0,0282	0,0368	0,1245	0,3710
Adding collider to grid	0,0154	0,0908	0,5844	2,7129
Water forces	0,0234	0,0357	0,0347	0,0427
Adding new Velocities	0,0133	0,0114	0,0150	0,0253
Diffusing Textures	0,0086	0,0071	0,0092	0,0125
Update along velocity field	0,0065	0,0057	0,0061	0,0078
Convert Render Texture to Texture2D	1,7651	1,5497	1,8692	2,0243

Figure 15: Analysis of the influence of the collision grid size on the performance, measured in milliseconds

Number of collision particles:	6	16	29	61
Bake collision	0,0391	0,0368	0,0433	0,0436
Adding collider to grid	0,0972	0,0908	0,1064	0,1112
Water forces	0,0240	0,0357	0,0639	0,1090
Adding new Velocities	0,0133	0,0114	0,0176	0,0169
Diffusing Textures	0,0098	0,0071	0,0108	0,0114
Update along velocity field	0,0060	0,0057	0,0070	0,0078
Convert Render Texture to Texture2D	1,8091	1,5497	1,8023	2,0694

Figure 16: Analysis of the influence of the collision particles count on the performance, measured in milliseconds

Number of objects:	0	1	5	10
Bake collision	0,0433	0,0368	0,0473	0,0396
Adding collider to grid (per object)	0,0000	0,0908	0,1271	0,0993
Water forces (per object)	0,0000	0,0357	0,0368	0,0248
Adding new Velocities	0,0149	0,0114	0,0223	0,0169
Diffusing Textures	0,0108	0,0071	0,0140	0,0080
Update along velocity field	0,0076	0,0057	0,0096	0,0068
Convert Render Texture to Texture2D	1,5139	1,5497	2,7936	1,6214

Figure 17: Analysis of the influence of the object count on the performance, measured in milliseconds

2. PERFORMANCE TEST DISCUSSION

In the first measurement [Figure 14] the influence of texture size gets measured. The function that gets influenced the biggest is the conversion function. From around 1.5ms for a 128 by 128 to 6ms for a 1024 by 1024. It is already clear that this function is a major bottle neck for the simulation. Besides the conversion function no noticeable changes are present. This makes sense for the physics related functions, but this is especially surprising for the wave propagation functions (Adding new velocities, diffusing textures, and updating along velocity field). The wave propagation functions perform calculations on each cell and by increasing to total cell count by more than 64times, still no changes are noticeable. By looking at the time of these functions, it is also noticeable that the three wave propagation functions combined take no longer than 1/20 of a millisecond. Meaning they're extremely fast. This leaves room for more detailed calculations. More steps and checks could be added during this wave propagation loop to improve the realism and overall quality of the heightmap.

The second measurement [Figure 15] is the influence of the grid size. The 2 functions that get influenced are the collision bake and adding colliders to the grid. The influence on adding the collider makes sense, because there are more grid points to loop over. The reason duration of the collision bake function gets increased is not because it must do more calculations. The texture size defines the number of calculations the collision bake must do, because for each height map value a value from the grid gets sampled. The increase in time can be explained because more grid cells need to get send from the CPU to the compute shader.

The third [Figure 16] and fourth [Figure 17] measurement show the expected results. More objects influence the total time of adding colliders to the grid, and the total time it takes to apply the force to the objects. However, the time per object stays constant. By changing the amount of collision particles, it takes longer to apply the force per object, which makes sense because more particles, means more points to check.

Only looking at performance is useful but what do these parameters influence exactly and when is it useful to change them. The number of objects is not really a parameter that will be discussed here, because it will always be defined on the user's end. It still is an interesting factor in measuring the performance. Increasing the particles density is useful for getting more detailed force feedback on triangle meshes. For spheres and cubes, only a few particles are more than enough. For precise force feedback this value can be increased on the necessary meshes. Changing the grid size will influence the quality of the initial waves on the water. The grid is a pixelated representation of the collision world. With increased pixel density, round or complex shapes are better visible in the waves generated by the object. Increasing the texture size is a tricky one. There are two ways texture size can be used, the first way is to increase the general quality. The other way is to allow for bigger objects. In this project increasing the texture size will result in being able to cover a bigger object with the same texture. This is because currently the speed is defined in pixel per second. To have texture size generate more details the speed needs to be handled in world space.

All these parameters can influence the detail quality of the simulation, but only if the other parameters support it. There is no use in setting the grid size bigger than the texture size. Increasing the texture size also need to be done with the right reason. If the water looks too blocky it is not the texture size of the height map that needs to be improved, but rather the vertex density of the mesh.

3. WAVE VISUALS EXPIREMENT

After implementing Stam's three step simplified Navier-Stokes implementation [6] the following result are achieved. The results would be best visible in video format because it is a simulation. This is not possible in a paper so after recording the footage, six screenshots were taken out of it, to showcase the visual results of the project.

The following values represent the settings with which this simulation took place.

- The water object itself had a size of 20x10x20 meters
- The texture size is 128x128 pixels
- The grid size is 64x10x64 grid cells
- The collision particles were generated with a minimum distance of 0.8m between 2 particles

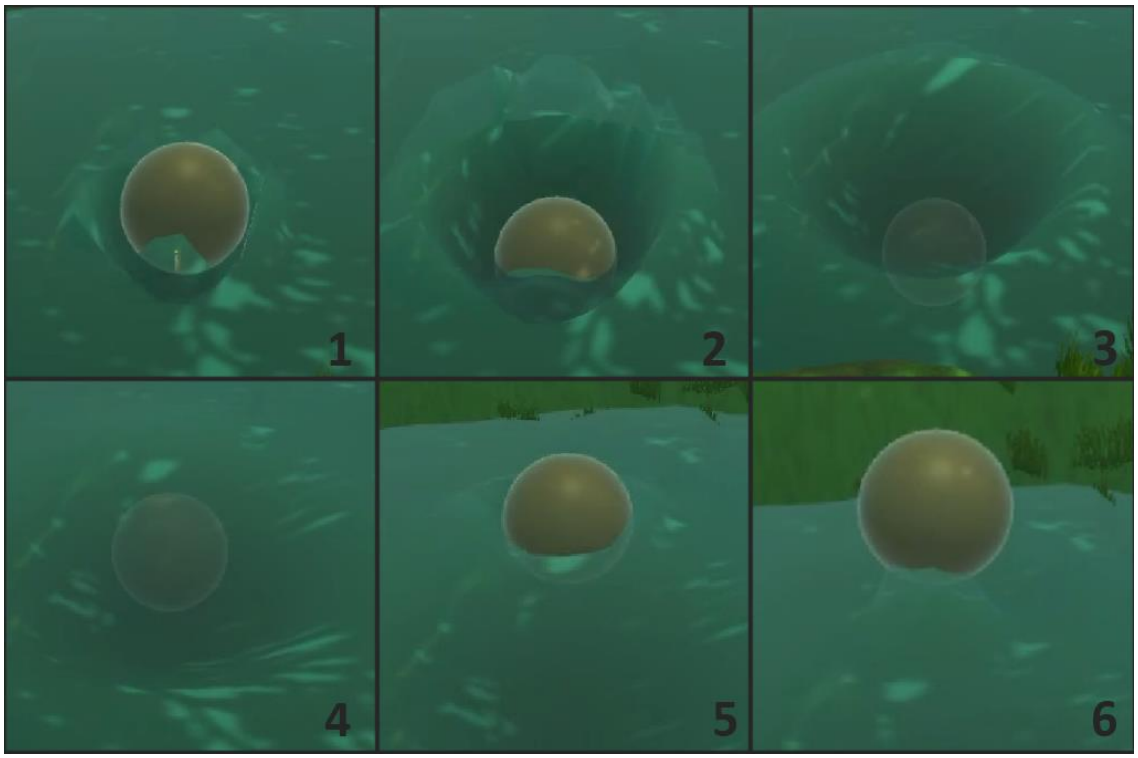

Figure 18: The waves generated by a floating sphere on the water surface

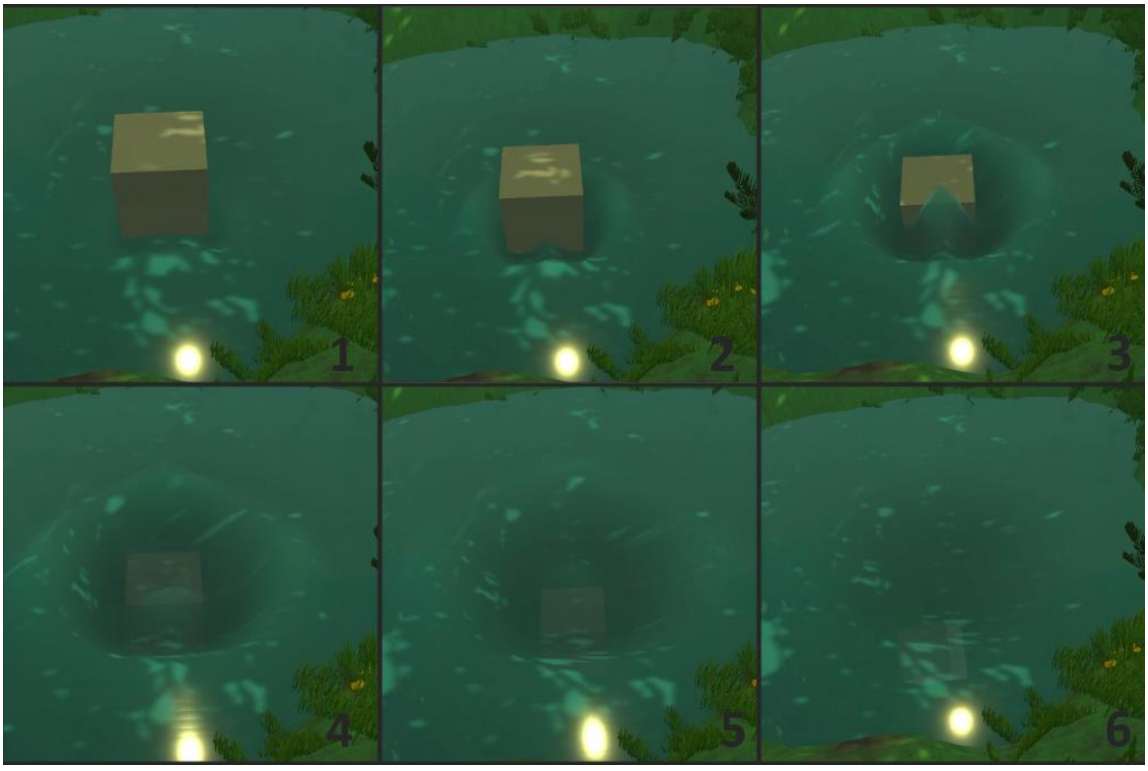

Figure 19: The waves generated by a sinking cube on the water surface

Figure 21: The waves generated by a torus shaped triangle mesh on the water surface

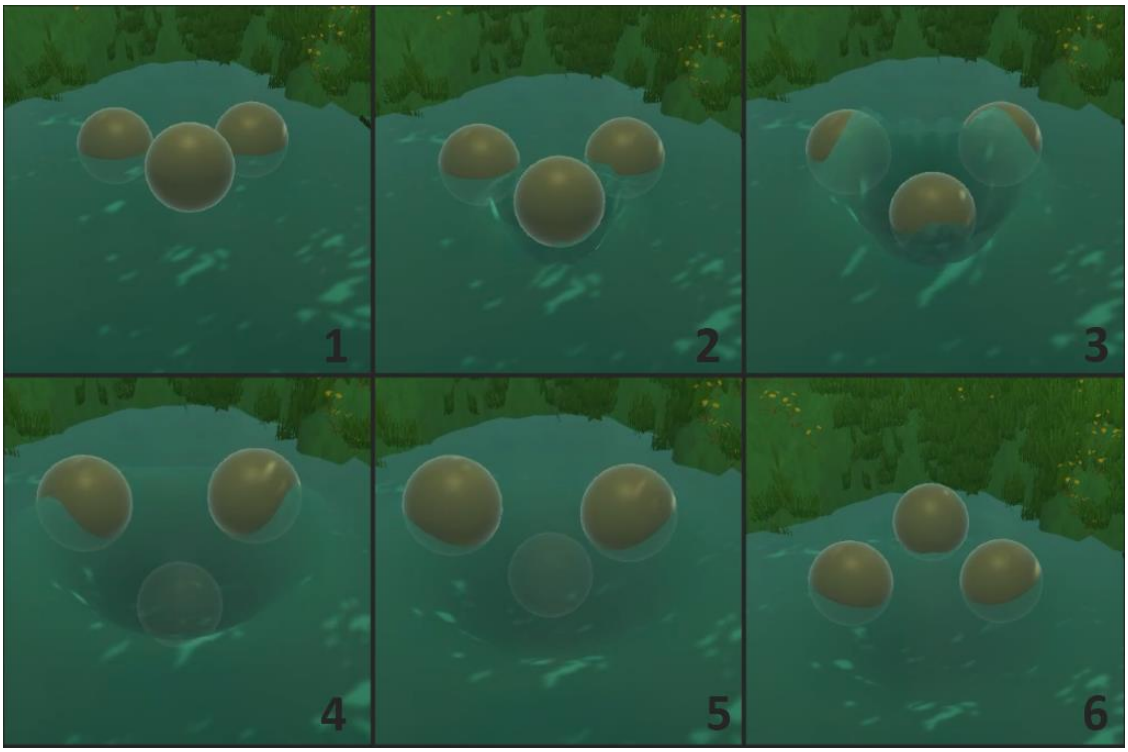

Figure 20: Objects get lifted upwards based on the wave height

4. WAVE VISUALS DISCUSSION

Looking at the visuals of the project showcased in the experiment the wave propagation does what is expected. Waves gets generated based on the shape of the object and the object react on forces applied by the water. At first sight everything looks good, however the simulation however has some flaws.

[Figure 18] The first image showcases a floating sphere being dropped in water. On the moment the sphere touches the water, bow waves appear all around the object. This is visible in the first three steps of this figure. But it is also noticeable that the waves generated by the object look very spiky and triangle shaped in the first two steps. After the simulation has had some time to run, the waves gets smoothened and it looks like the expected result. The last step of this figure shows that whenever an object comes out of the water, a water trail follows the object this is due to the stern waves filling up the gap behind the object and overflowing.

The reason behind the spiky waves on the initial contact with water is due to the velocity map that gets added to the simulation on contact. The velocity map is based on the collision map generated by the collision grid. When generating this velocity map on all the cells that neighbor to a collision cell a velocity gets applied away from this cell. This means that the initial velocities of an object are only one pixel wide. This is a very abrupt change in a very small number of cells. Resulting in a disturbance in the smoothness of the simulation until the diffuse function spreads out this velocity further.

[Figure 19] The next image showcases the same as the previous one except instead of a floating sphere, a sinking cube is thrown. As expected, the waves generated by the cube have a more cubic shape than the one generated by the sphere. Another phenomenon appears here that is best visible in the third step in this image. A stern wave of water is generated in the middle of the initially generated wave. This stern wave although looking weird is correct by the provided rules. The cube is colliding with the water, but still has a velocity on the ZX-plane which means that behind the object water starts flooding in again. The biggest problem why this looks weird is because the collision grid does not store move direction. The simulation sees this wave as an object that fell straight down in the water and left behind a stern wave. The expected result would still be a stern wave but a stern wave that is influenced by the direction the object is moving in.

[Figure 21] Cubes and sphere generate accurate waves based on their shape. The same is true for more complex meshes like a torus. The torus does not only generate waves on the outside of its shape, but also in the hole in the middle. In this figure it also visible how the physics get applied on more complex shapes. As result of the physics being applied on the collision particles, the object gets a rotational force applied inside the water. Coming out of the water in a vertical way causing rectangular waves.

[Figure 20] In the last figure it is visible that object properly gets forces applied based on the height of the water. An object gets thrown next to two other objects generating a wave that passes through the other objects lifting them up in the process.

Although the results are very interesting, the waves aren't perfect. The waves don't propagate very far across the water surface and only one big visible wave gets generated from each object. The simulation leans more towards volume simulation rather than wave propagation. The main reason for this is the method used in the implementation. The Navier-Stokes equation is a method to predict the flow of water and other volumes like smoke. This does not mean the water in this simulation is bad or not useable, but there can be improved upon. The biggest room of improvement is the wave propagation step. In the performance analysis became clear that this step is only a very small fraction of the entire simulation, meaning that a lot more calculation can be added without hurting the performance.

The volume simulation does not give all the wanted or expected visual results, but it does contain all the physics information necessary to make sure that objects can interact with it properly. This water simulation shows roughly which shape the water would have while interacting. Thus, it can still be used as a base layer for water

visuals where more water data can be rendered on top of the water like foam and water ripples. While this base layer can act like the physics backend of the water.

LIMITATIONS

The water simulation works and the wanted results are visible, but it still has some shortcomings. Most of these limitations are fixable but are just not implemented yet in the current simulation.

Lack of interaction with static objects. Static object such as land masses or islands, don't influence the simulation at all. The water can just pass through the static objects which result in weird visual results. The fix for this one would be to implement a mask of where all the static objects are. This mask contains the necessary information to make sure that the wave propagation function can properly interact with static objects. This mask can also be used to give the water different shapes than square or rectangular.

Velocity in pixel space. Like mentioned earlier in the performance analysis, the velocity of the water and diffusing the texture is very pixel dependent. Scaling the texture size will not automatically result in higher simulation quality. Increasing texture size will make the texture fit to be used on bigger surfaces rather than increase the quality. Although this is not completely unwanted behavior. It should be possible to increase the quality of the simulation without messing up the speed of the waves in the simulation.

Waves don't travel far. The water waves generated by object don't travel as far as one would expect. They quickly lose all their height and disappear from the water surface. This is caused by one of the more complex issues. When an object touches the water, velocity gets added to the simulation. This velocity gets diffused and updated along itself. The problem is that only a part of the velocity gets transferred every time, meaning that after some time has passed there still is an active velocity force pushing the water away from the original point of impact. If the velocities remain water cannot flow back into those cells, this will result in dents being present in the water. Dents in water are not expected, the water is supposed to showcase the wave and then reset to its original height. To counter these dents the diffuse rate of the velocity was increased drastically removing the dents but also causing the waves to travel less far.

Reading back the heightmap from the GPU. In the performance analysis it is clear how inefficient reading back the heightmap is from the GPU. With the current simulation setting it doesn't take that long. But to use this simulation on a bigger scale with more or bigger texture, the readback needs to be avoided. The readback is necessary to be able to apply the forces on the objects in the world. If this readback can be avoided, the simulation would speed up tremendously. This could be solved by either calculating everything multithreaded on the CPU or calculation everything on the GPU by implementing it as an extension of the PhysX physics library.

CONCLUSION

Stam's Navier-Stokes implementation provides a solid base for physics-based water simulation. Visually the waves generated don't look as realistic as expected, this is due the fact that the implementation focusses on simulating the behavior of a volume rather than water specifically. To improve the behavior quality of water more calculation steps and rules need to be added to the wave propagation step. This should result in no performance problems because the performance tests pointed out that the wave propagation step is in fact the best performing and the best scaling step out of the entire simulation, remaining almost constant no matter how big the texture got. The biggest performance loss comes from data transition from CPU to GPU and back.

Using a grid-based approach for collision detection produces very accurate results, especially when turning up the grid cell count. Turning up this cell count does affect performance. It will take longer for colliders to register their location inside this grid, and it will take longer to send this grid data to the GPU to calculate the collision map. Changing this grid cell count is very easy, thus performance and accuracy can easily be adjusted based on device and use-case of the water.

The waves might not look like what is expected from water waves, the waves miss detail. It is a major area of improvement for this implementation. However, everything that was meant to be researched is visible in the simulation. Bow and stern waves get generated based on the shape of the object whenever it passes through the water. Even holes in object get proper waves generated from them. The quality of the waves coming from triangle meshes can be controlled by changing the grid cell count. Thanks to Archimedes' principle and volume calculation, accurate physics responses happen on objects. Making objects jump up out of the water or sink based on their density. The implementation of collision particles makes sure that objects get a rotational force applied to them when only partially inside the water.

FUTURE WORK

1. 3D NAVIER-STOKES EQUATION

The core of the project is the Navier-Stokes equation but simplified and put in a 2D shape. With the current implementation it would be possible to change this 2D Navier-Stokes equation to 3D. Some more rules need to be applied like gravity rules. But going 3D could result in much more interesting physics interactions. Objects moving underwater can result in small waves on the surface. Objects could move along underwater velocity. Using 3D Navier-Stokes would overall result in a more detailed and realistic simulation. The performance of the current wave propagation also shows that expanding further on this is possible.

3D Navier-Stokes would also expand the use case. Not only can the 3D implementation work on water, but it can also go back to the original use case. Representing volumes like smoke and cloud together with airflow.

2. IMPLEMENTATION IN THE PHYSICS ENGINE

The project turned out more physics than water heavy than originally anticipated. The performance analysis is a proof of this. The wave propagation takes up only a very small performance of the total performance. The most time-consuming steps are calculating where the objects are located compared to the water and making the heightmap readable on the CPU for applying the forces. Optimizing these two big steps would result in major performance improvement. A lot of the calculations feel dull and unnecessary. The current flow goes like this.

Physics API (PhysX) -> Collider to Grid (CPU) -> Grid to collision map (GPU) -> Texture wave propagation (GPU) -> Readback texture (CPU) -> Apply forces (CPU) -> Physics API (PhysX).

Considering this flow and considering the fact the PhysX does most of its calculations on the GPU. Means that there is a GPU -> CPU -> GPU -> CPU -> GPU flow. The initial data is stored on the GPU get read back to the CPU to then be send to the GPU again, to read back to the CPU again to be send to the GPU. This is a very obnoxious flow. Together with the fact that the most expensive functions are the functions that require data to be send to and readback from the GPU, like the collision bake and converting the texture. Means that there is room for improvement.

A better and more efficient approach would be to implement the wave propagation directly into the physics engine. This would mean that everything happens all in one place and would result in a major performance increase. The only challenge would be to get the heightmap from the physics engine into the water shader. Both happen on the GPU so this shouldn't be impossible, but it is something that needs to be investigated further.

BIBLIOGRAPHY

[1] M. Müller-Fischer, “Fast Water Simulation for Games Using Height Fields Post doc MIT: Simulation in CG 2002 Co-founder NovodeX (physics middleware) 2004-2008 Head of research Ageia (SDK features) 2008 Research lead PhysX SDK at Nvidia Zürich office: R&D PhysX SDK currently Height Field Fluids,” 1999.

[2] A. R. Brodtkorb, M. L. Sætra, and M. Altinakar, “Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation,” *Computers and Fluids*, vol. 55, pp. 1–12, Feb. 2012, doi: 10.1016/j.compfluid.2011.10.012.

[3] R Nave, “Ocean Waves.” http://hyperphysics.phy-astr.gsu.edu/hbase/Waves/watwav2.html (accessed Dec. 30, 2021).

[4] Wikipedia, “Wave.” https://en.wikipedia.org/wiki/Wave (accessed Jan. 24, 2022).

[5] Wikipedia, “Bow wave.” https://en.wikipedia.org/wiki/Bow_wave (accessed Jan. 25, 2022).

[6] J. Stam, “Real-Time Fluid Dynamics for Games.”

[7] O. Staadt, H. Cords, and O. Staadt, “Real-Time Open Water Environments with Interacting Objects. Real-Time Open Water Environments with Interacting Objects,” 2009. [Online]. Available: https://www.researchgate.net/publication/221314832

[8] S.-L. Lien and J. T. Kajiya, “A Symbolic Method for Calculating the Integral Properties of Arbitrary Nonconvex Polyhedra,” 1984.

[9] S. M. Laverty and S. M. Laverty, “Experimental Hydrodynamics of Spherical Projectiles Impacting On a Free Surface Using High Speed Imaging Techniques,” 2004.

[10] N. Chentanez and M. Müller, “Real-Time Eulerian Water Simulation Using a Restricted Tall Cell Grid.”

[11] “Realtime Water Simulation on GPU Nuttapong Chentanez NVIDIA Research 2.”

[12] Wikipedia, “Dispersion (water waves)”, Accessed: Jan. 24, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Dispersion_(water_waves)

[13] Wikipedia, “Wind wave.” https://en.wikipedia.org/wiki/Wind_wave (accessed Jan. 24, 2022).

[14] K. Crane, I. Llamas, and S. Tariq, “Real-Time Simulation and Rendering of 3D Fluids.” https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-30-real-time-simulation-and-rendering-3d-fluids (accessed Jan. 24, 2022).

[15] “Dynamic Water.” https://john-wigg.dev/DynamicWaterDemo/ (accessed Dec. 30, 2021).

[16] C. Ming, “FAST, INTERACTIVE WATER WAVE SIMULATION IN GPU FOR GAMES.”

[17] M. Zamith *et al.*, “Sound Wave Propagation Applied in Games.”

[18] J. Tessendorf, “Simulating Ocean Water.” [Online]. Available: http://www.finelightvisualtechnology.com

[19] L. Sun, G. Ma, C. Nie, and Z. Wang, “The simulation of dropped objects on the offshore structure,” in *Advanced Materials Research*, 2011, vol. 339, no. 1, pp. 553–556. doi: 10.4028/www.scientific.net/AMR.339.553.

[20] B. Nikunj Raghuvanshi, C. Lauterbach, A. Chandak, D. Manocha, and M. C. Lin, “Crowd simulation in an urban scene. (Geometric Algorithms for Modeling, Motion, and Animation Research Group,” 2007. [Online]. Available: www.havok.com/

[21] “Design of Boat Racing Game using Buoyant Force,” *Journal of The Korea Society of Computer and Information*, vol. 23, no. 9, pp. 21–26, 2018, doi: 10.9708/jksci.2018.23.09.021.

[22] A. D. D. Craik, “The origins of water wave theory,” *Annual Review of Fluid Mechanics*, vol. 36, pp. 1–28, 2004, doi: 10.1146/annurev.fluid.36.050802.122118.

[23] B. Ottosson, “Real-time Interactive Water Waves Water Wave Simulation for Computer Games.”

APPENDICES

```
static public float CalculateVolumeOfMesh(Mesh mesh)
{
    float volume = 0f;

    int[] triangles = mesh.triangles;
    for (int i = 0; i < triangles.Length; i += 3)
    {
        float tempVolume = CalculateVolumeOfTriangle(mesh.vertices[triangles[i]]
            , mesh.vertices[triangles[i + 1]]
            , mesh.vertices[triangles[i + 2]]);

        volume += Mathf.Abs(tempVolume);
    }

    return volume;
}
2 references
static private float CalculateVolumeOfTriangle(Vector3 vertex1, Vector3 vertex2, Vector3 vertex3)
{
    var v321 = vertex3.x * vertex2.y * vertex1.z;
    var v231 = vertex2.x * vertex3.y * vertex1.z;
    var v312 = vertex3.x * vertex1.y * vertex2.z;
    var v132 = vertex1.x * vertex3.y * vertex2.z;
    var v213 = vertex2.x * vertex1.y * vertex3.z;
    var v123 = vertex1.x * vertex2.y * vertex3.z;
    return (1.0f / 6.0f) * (-v321 + v231 + v312 - v132 - v213 + v123);
}
```

Figure 22: Implementation of the used volume calculation algorithm

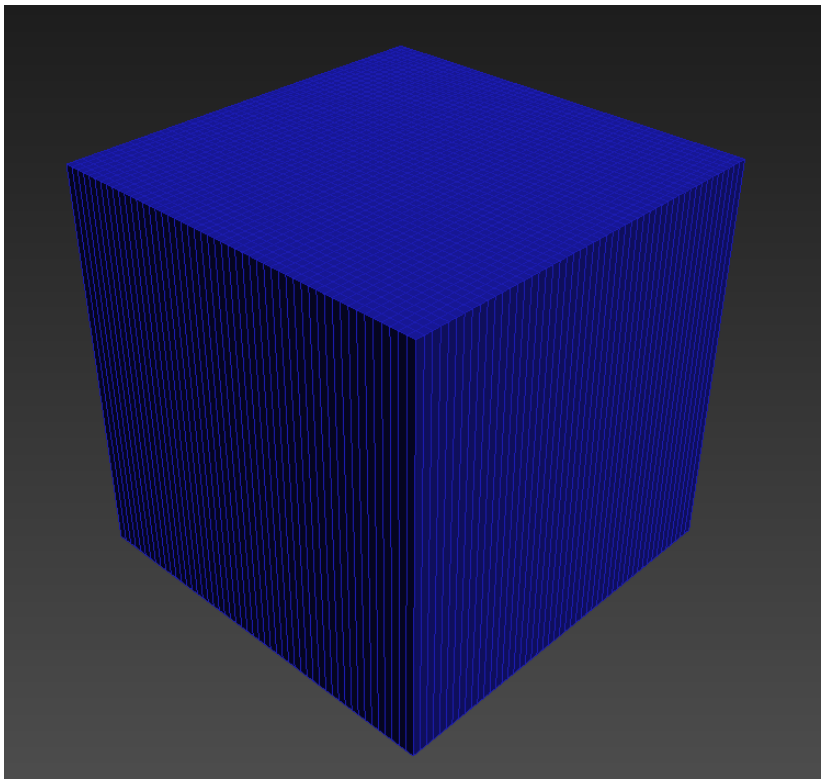

Figure 24: The mesh used to represent the water in this project

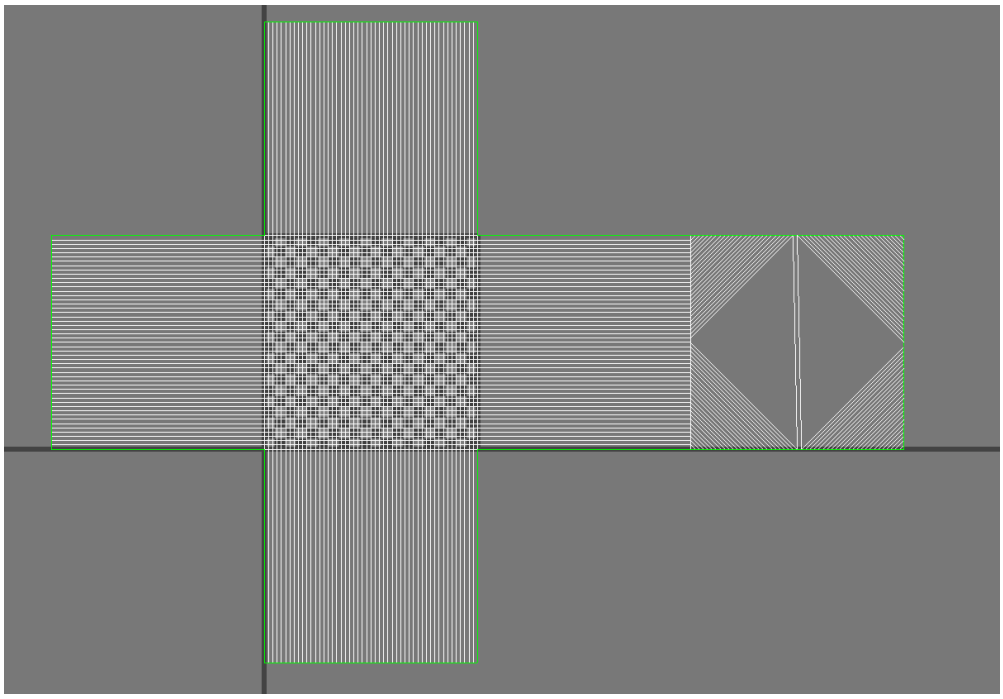

Figure 23: The UV space of the used mesh, with in the center the top plane which acts like the water surface

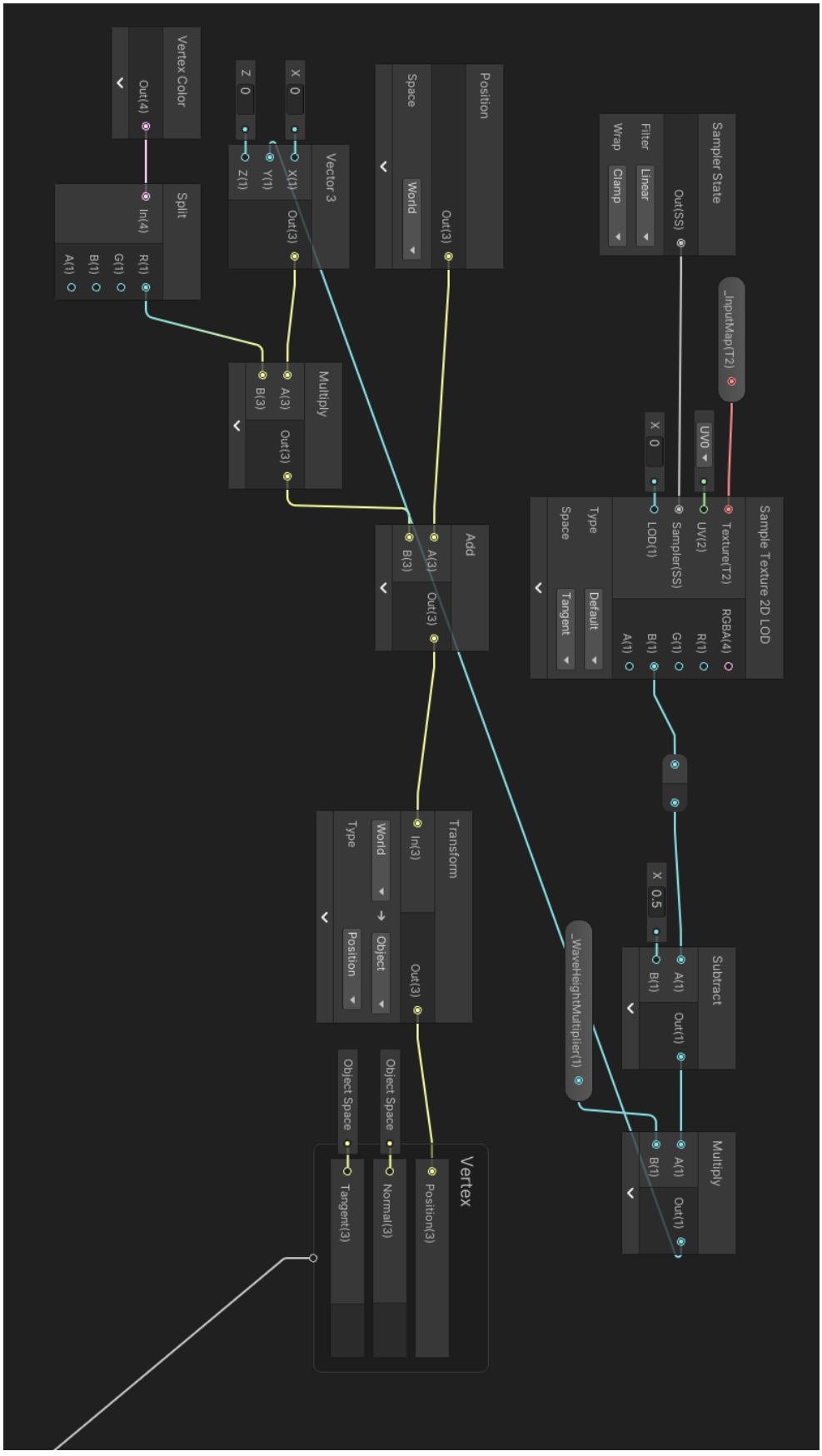

Figure 25: The vertex shader of the water shader made in the Unity shader graph

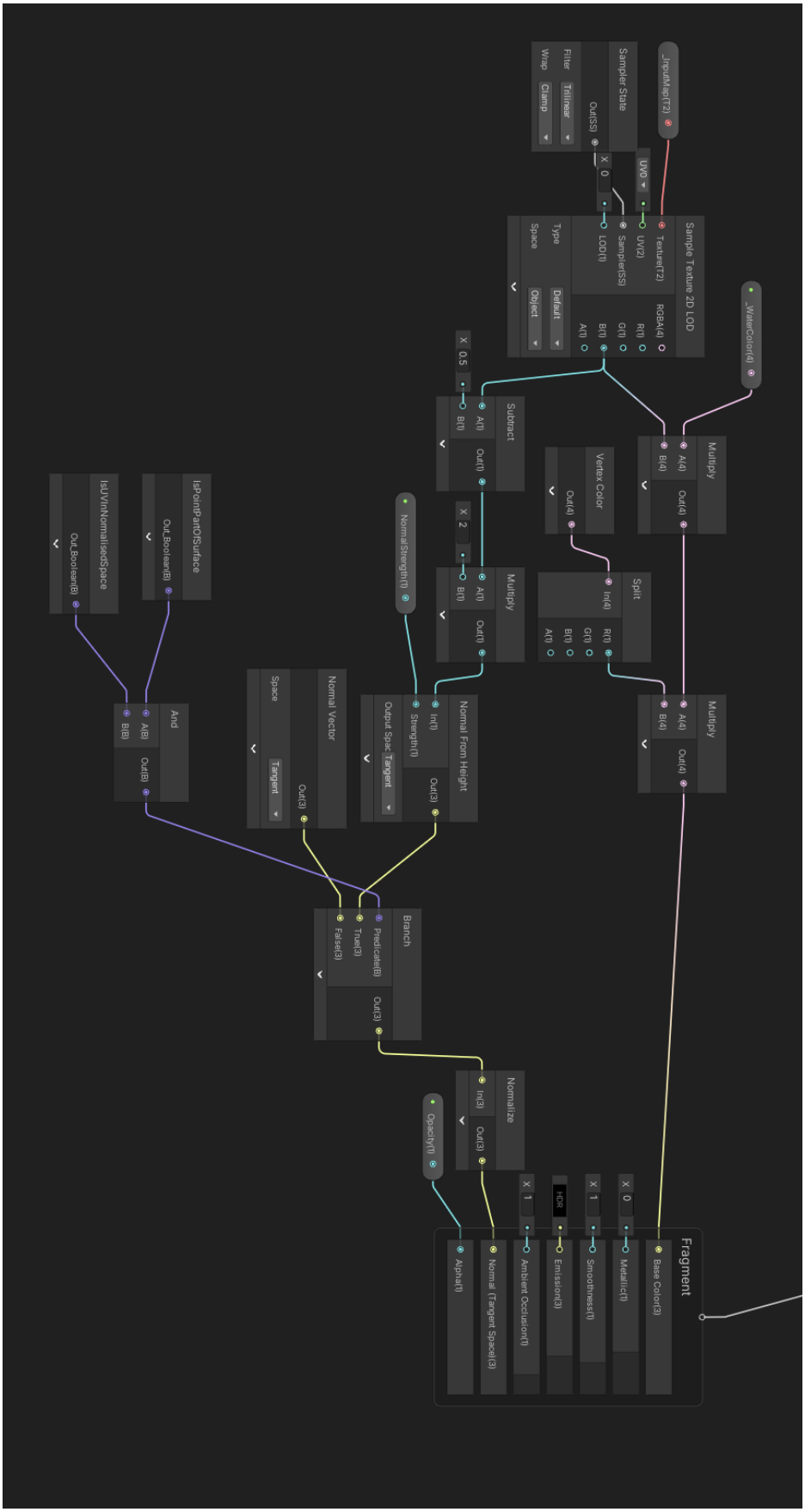

Figure 26: The fragment (pixel) shader of the water shader made with the Unity shader graph