Realtime Water Simulation using the Pipe Method

Sebastian Willenbrink, 19990601-7851, stwi@kth.se

August 21, 2024

Contents

1 Abstract

In this paper, we introduce a variant of the pipe method and implement it in the game engine Godot. The pipe method simulates water flow using a heightmap and allows actual shifts in water masses instead of visually adding waves to a static surface. We improve on the original method by using bi-directional pipes, allowing us to compute flow to eight neighbors per cell

instead of only four. We use compute shaders to implement the method in Godot 4.2. Despite Godots compute shader support being in its infancy, we achieve high performance and simulate a map consisting of 2048x2048 cells approximately 3x faster than realtime.

2 Introduction

Water is an element of many natural environments and as such, the simulation of its flow behavior has been the object of extensive research. Unfortunately, fluid simulations are incredibly complex and therefore require large amounts of computing resources for even tiny amounts of water. This problem is exacerbated when we wish to simulate large amounts of water, e.g. seas and oceans.

Faced with these circumstances, two distinct approaches crystalized. On the one hand, fluid simulations used for engineering projects focus on as accurate as possible simulations at the expense of time. Reasonably accurate simulations of water movement, density etc. are used to help design boat hulls and propellers. As time is of no concern, these simulations are often run for days or weeks to simulate a time window of seconds or minutes.

On the other hand, we have the video game industry. Here, accurate water movement is secondary. Instead, being able to show realistic looking water in realtime is the primary focus. As such, the actual physical simulation of fluids plays no role. A common approach is to visualize waves on an ocean or lake, resulting in a seemingly dynamic system with varying water levels while in reality the waves are only visual and layered on top of a static water level. While it is possible for these waves to affect floating objects such as boats, the other direction is not possible. The water level cannot be modified locally by boats displacing water, a dynamically built dam blocking water or a hole filling with water. Adding or removing water at a specific spot is not possible.

This lack of two-way interactions between water and physical objects is limiting. Simulating a ship accurately also requires simulating the displacement of water which results in waves around the bow which in turn affect other ships. While the wake of a ship can be faked by layering more waves on top of the existing waves, this method quickly fails for more complicated scenarios, such as a small channel where the water cannot easily flow around the ship, and is fundamentally limited. This issue is even more pressing if we consider a sandbox game that allows players to build custom structures interacting with the water such as dams, submarines, pipes etc. In this case,

the simulation needs to be reasonably accurate such that the player can see, for example, the water flow through gaps in the dam and observe the effect of their actions.

In recent years, the improved capabilities of GPUs and compute shaders made a third approach possible. Instead of limiting ourselves to only wave simulation on top of water, we can run a simplified simulation of the complete water body on the GPU, using their parallel compute power to simulate independent cells of the water body which combine to a reasonably accurate whole. Note that we have a significant margin of error and room for simplifications as the behavior of water is extremely complicated and intricate. Players will not perceive small inaccuracies in the simulation once a relatively low level of accuracy has been reached and visual fidelity can be improved by layering the visual-only waves on top of the actual simulation.

3 Related Work

The work is based on "Fast Water Simulation Methods for Games"[1]. This paper bases its implementation in turn on "Fast Hydraulic Erosion Simulation and Visualization on GPU"[2].

This technique hasn't been widely used yet, one reference claimed two large 3D games using simulated water as part of the core gameplay loop: From Dust and Cities: Skylines. The pipe method shares some similarities with the fluid simulation in Factorio, although their CPU implementation differs in both approach and goals.[3]

4 Implementation

In this paper, we implement a simple fluid simulation based on heightmaps that supports actual water flow. Waves are not implemented as a visual effect; instead they emerge organically from the water simulation. The focus is the efficient implementation of the simulation in the Godot game engine using compute shaders.

The simulation is based on the so-called pipe method which divides the whole simulated area into cells where each cell can be imagined as a column of water. Put differently, it operates on a heightmap for water. Each cell

[1] `https://dl.acm.org/doi/pdf/10.1145/2700533`

[2] `https://data.exppad.com/public/papers/Fast%20Hydraulic%20Erosion%20Simulation%20and%20Visualization%20on%20GPU.pdf`

[3] `https://www.factorio.com/blog/post/fff-274`

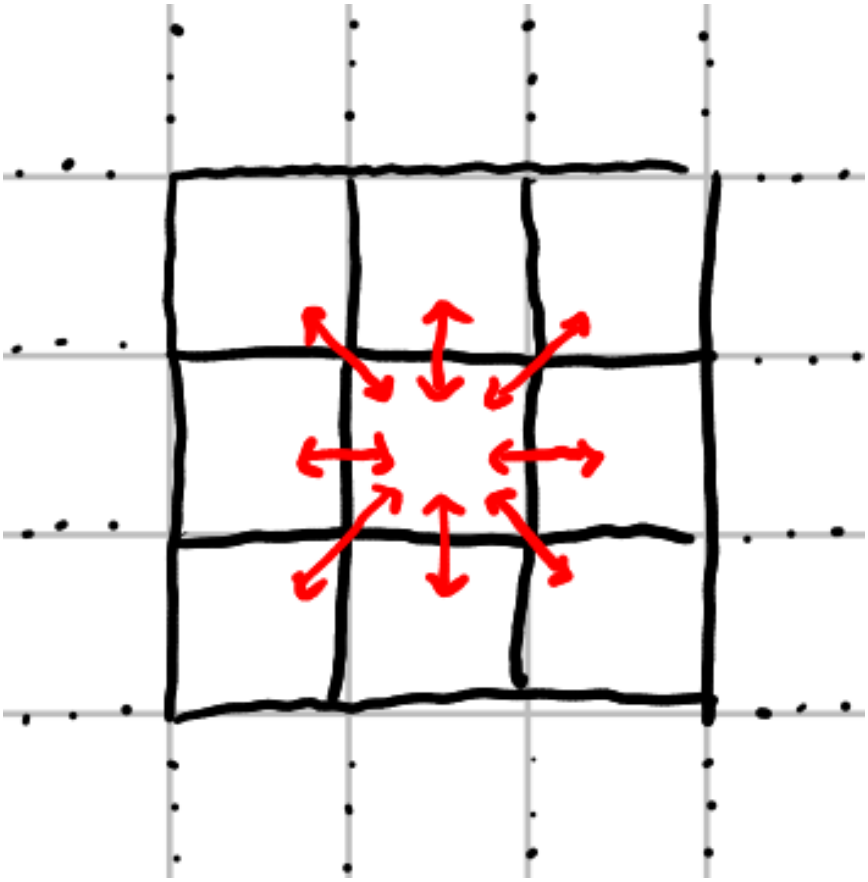

Figure 1: A top-down view of some 9 cells of the heightmap with the eight pipes involving the central cell highlighted in red.

interacts with its direct neighbors through pipes. Figure 1 shows one cell and the neighbors it interacts with.

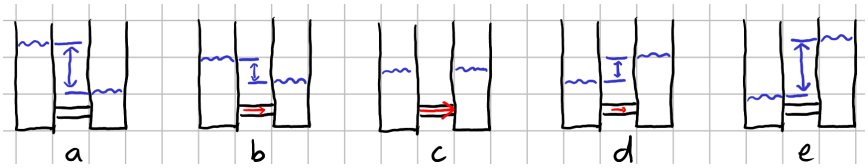

Figure 2: A side view of two cells connected by a pipe with water (blue) moving from one side to the other with the flux (red) increasing until c) before dropping to zero in e). Without dampening this system oscillates forever.

If the water height of two neighboring cells differs, the water from the higher cell will begin to flow through the pipe to the lower cell. Crucially, this flow has a momentum called flux and will slowly accelerate leading to an overshoot. Figure 2 shows an isolated example of just two cells, leading to a wave going back and forth similar to a cup that was bumped. With more cells with a single higher cell in the center, this leads to a wave propagating outwards.

Note that we do not specifically add any dampening to the simulation. Instead of the waves losing energy over time, waves naturally lose energy through destructive interference. This leads to a tumultuous pool slowly calming as waves reflect on the edge and interfere with each other. Adding a dampening factor is trivial and can be easily added if desired.

Despite this simulation being very simple, it is capable of modelling water flow with sufficient precision. Realistic flows from a high point to a lower one is achieved and waves are simulated convincingly. Nevertheless, the method has several shortcomings: Firstly, it cannot model vortices which occur when water flows not directly towards a lower point but with an angle, leading to it circling the low point as e.g. in a sink. Secondly, waves always behave the same irregardless of water depth. This makes breaking waves near coasts impossible and waves in shallow water slightly inaccurate. This mostly concerns visuals and can be neglected for the water flow simulation in larger scales like seas. It can also be convincingly faked through separate shaders if necessary.

Additionally, due to using heightmaps it fundamentally cannot encode caves or other environs with two unconnected columns of water above each other. This prevents the accurate simulation of submarines, water flowing into a sinking ship, waterfalls and many more scenarios. This is a fundamen-

tal shortcoming and can only be addressed by adding more information to each cells. One approach is to have multiple layers of water columns in each cell, with each water column being limited in height either by the amount of water or some kind of barrier like terrain or a ship. In that case submersion can be simulated by having a column of water below the ship and one above with water flowing to neighboring cells whenever the ship moves up or down.

4.1 Compute shader

To efficiently compute the water heights, I use compute shaders. Each cell is simulated in a separate thread in two steps before the results are returned to the main program. Firstly, the flux (flow momentum) of the water to the neighbors is calculated. After all cells are done, we update the water height by accumulating the flux for all neighbors. After the computation is done, we can either repeat the steps to compute more than one simulation step per frame or directly render the texture on the screen. In between frames, we can also add water according to user input.

One central issue with shaders is that each shader thread is run independently of the others. Synchronization is expensive and should be avoided. Thus, the data structures and algorithm must be designed in such a way to avoid memory races. This resulted in multiple changes.

Each pipe connects two neighboring cells. As each pipe has an associated flux, we need to ensure that this flux is only updated once by one cell, not both. In the original paper, this was ensured by having two uni-directional pipes for each neighbor-pair. In this way, each cell is only responsible for updating the pipe that removes water from this cell. This means that negative flux values are impossible and that the heights of neighboring cells are compared twice, once for each direction. Furthermore, we need to store twice the amount per cell as we have both flux-to and flux-from for each neighbor.

I used a more complicated scheme to combine both directions into one pipe, with positive and negative values representing the two directions. As a texture with four color channels can store four floats, we can have four pipes per cell. But due to this improved scheme, a cell only stores the flux for half of its pipes as the other half is updated by the other neighbor. This means that this scheme allows us to have eight neighbors per cell, i.e. not only orthogonally but also diagonally neighboring cells. This allows direct diagonal flow which would otherwise take two steps and makes some artifacts of the simulation harder to notice. In this concrete implementation, each cell is responsible for the right, bottom left, bottom, and bottom right neighbor.

Another critical issue is that the water height may not be negative. Imag-

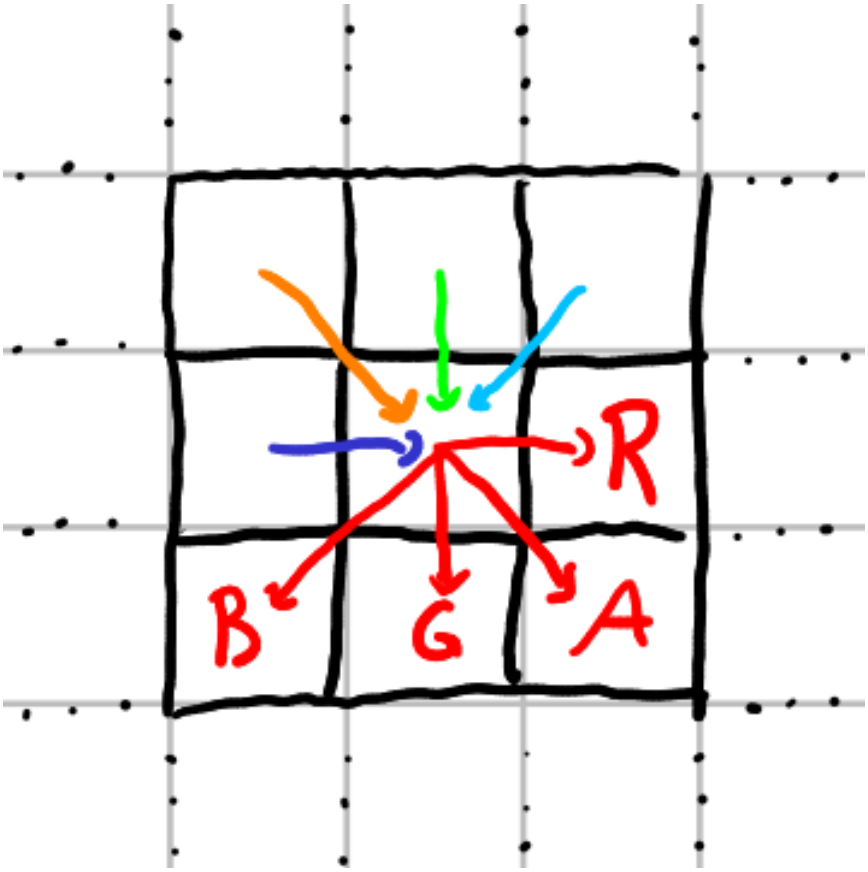

Figure 3: The pipes involving the central cell in detail. Red pipes are stored in the RGBA channels of the central pixel of the heightmap. The other pipes are stored in the respective channel of the upper, left, upperleft and upperright cells. One invocation of the compute shader updates the four red pipes. Through tiling all pipes are covered.

ine a single, very high column of water. The flux value in each of the eight pipes leading to neighbors will increase over several steps. At some point, the sum of fluxes will be larger than the remaining water height, i.e. the cell would be more than emptied completely in the next time step. As the flux and water height is simulated in separate steps and each cell updates independently from its neighbors, this means that the neighboring cells would add more water to their own height than can be drawn from the original cell, creating water from thin air (or lowering the water height below 0). This means that flux must be limited in the first step to ensure that all pipes in sum don't draw more water than the cell can supply.

In the original formulation with two one-directional pipes, each cell updates all outflowing pipes itself and can therefore limit this directly. In our formulation, half of the pipes are not available at the time of the computation (as each cell is updated independently in the shader and each cell is only responsible for half its pipes). As a result, we either introduce a third step specifically to limit the flux and prevent too much water being removed from a cell or we simply limit cells to always remove at most one eigth of the cells remaining water. While the latter is not ideal, it introduces no visible artifacts and is significantly cheaper to compute.

4.2 Rendering

After simulating the water, we also need to display it on the screen. Unfortunately, Godot does not support compute shaders seamlessly as of version 4.2. At the start of the project, using the compute texture directly as input to the vertex shader was not possible. Thankfully, support has improved since then and instead of expensively reading the texture from GPU to CPU and immediately transferring it back we can use the texture directly. While the support is still not perfect, it is good enough for this application. For more details see: [4]

4.3 Encountered Issues

4.3.1 Negative and small floats

Negative float values that should not be possible but occur anyway were a significant problem. Clamping the height and flux values solved the issue. Another issue is that the floats never reach absolute zero. Instead, the vary at some very low level (perhaps even the smallest possible float larger than

[4] `https://github.com/godotengine/godot-proposals/issues/6989`

zero). The water rendering shader has a cut-off to not render low water-levels at all. This in turn introduces issues with slopes where water slowly but steadily flows. So the cut-off respects both water levels and flux and only hides the water if very low water levels sit still.

4.3.2 Bad debugging ergonomics

Debugging shaders is quite tedious and error prone. I noticed only quite late that the simulation was fundamentally broken due to using the flux values from the last frame in the water height update. This cased quite a lot of instability, leading to numerous attempts to fix this despite the solution being quite simple. Setting up good debugging visualizations would have helped but specifically a one-frame difference in the flux values is hard to debug even with debug visualizations. Debugging is especially tedious with rarely occuring issues (such as negative water heights through float rounding).

4.3.3 Compute Shaders in Godot

Compute shaders in Godot are far from seamless. A lot of documentation is very bare-bones, examples only touch on basics and delving into Vulkan documentation to understand the Godot API costs time. Delaying the project for some time led to better support making the project highly performant. An area with 2048x2048 cells can be run at 3x acceleration at 60fps on a GTX 1080Ti, leaving significant margin for other aspects of a game.[5]

5 Future Work

5.1 Layering wave simulation on top

Unfortunately, this cell based method can only simulate waves over multiple cells. This means that realistic waves need an excessively large resolution. Another more performance-friendly approach is to use two systems. One for the water height changes and another for waves. Alternatively, this can also use a particle based method, see for example Uncharted wave particle system[6].

[5] Note that the acceleration does not render everything, it only computes the flow and water heights 3 times every single frame

[6] `https://www.researchgate.net/publication/333915560_Rendering_Rapids_in_Uncharted_4`

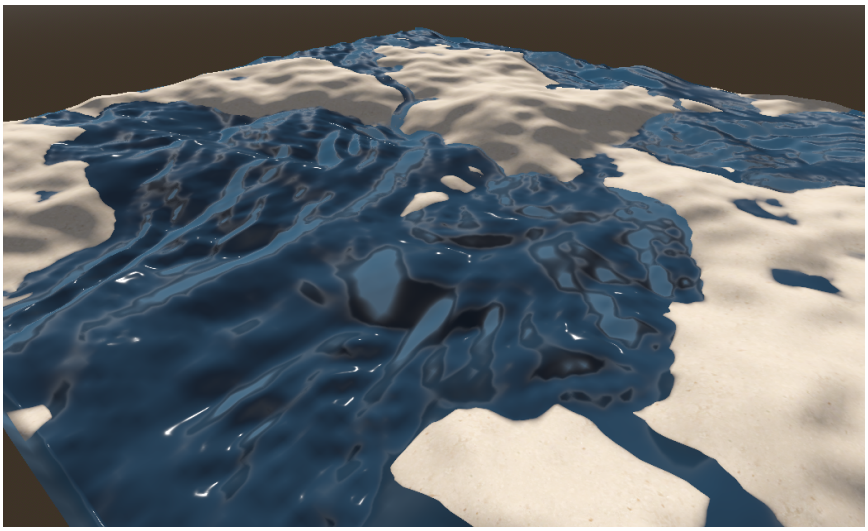

Figure 4: Rough seas shortly after the simulation start. The initial scenario is a terrain covered in an even layer of water.

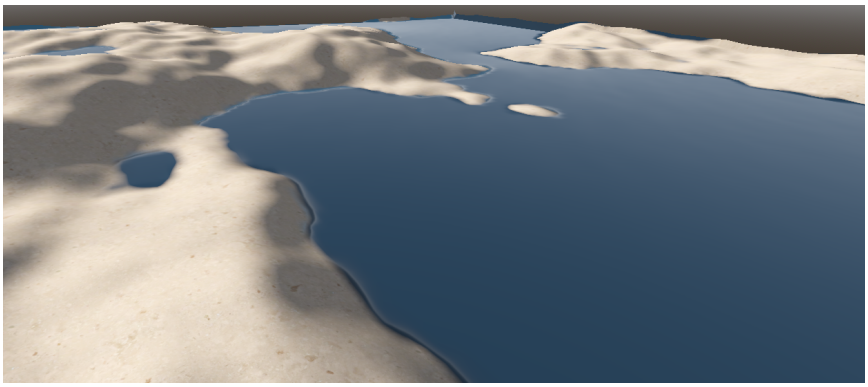

Figure 5: Calm seas long after the simulation start. Even with very low dampening, the sea is essentially mirrorlike.

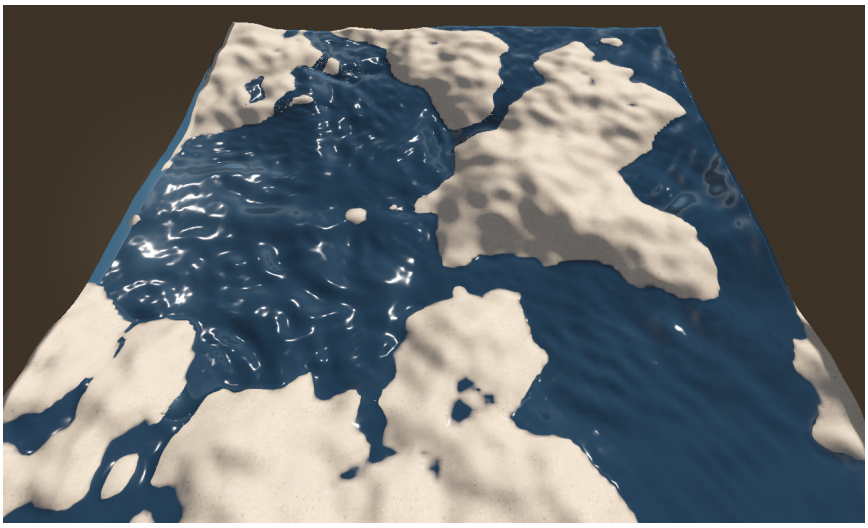

Figure 6: Rivers can be seen flowing from the top right to the left lake and to the bottom left which in turn feeds into the left lake. Rivers emerge as a natural phenomenon in the simulation when water flows from one spot to another. Their appearance is enhanced by highlighting areas with high flux. Otherwise, the river would barely be visible due to its low depth. Note that this scenario did not add water during the simulation, as such the rivers will naturally disappear once the top lakes have been emptied.

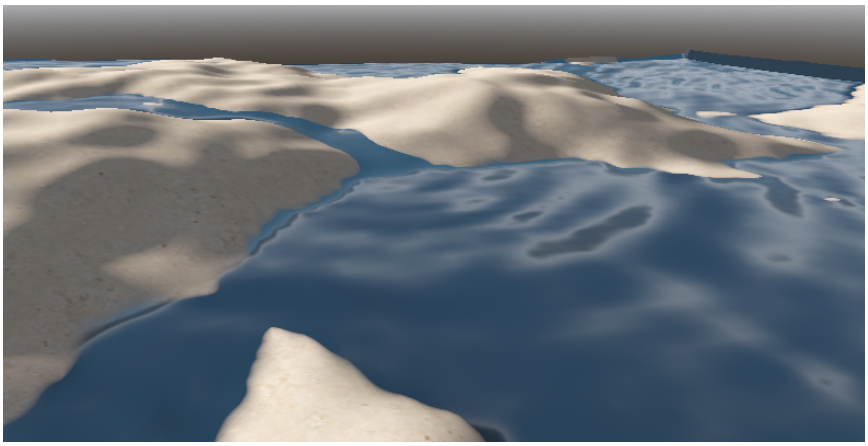

Figure 7: Here, small waves can be seen in the center of the image as a result of the river feeding water in to the lake.

5.2 Interaction with rigid bodies

Right now no interaction with rigid bodies is possible. Adding this is mostly an issue of interacting with Godot on an advanced level. As this was not the focus of this work, I did not investigate this in detail. Presumably, one needs to sample the texture (without loading the whole texture to the CPU to avoid bottlenecking) to get the height of water under a rigid body. This can then be used to add buoyant forces to the body. In addition and likely much harder to implement, the body needs to displace water, i.e. lower the water level below the body depending on the mass and volume of the body submerged in the fluid.

5.3 Beyond the pipe method

The pipe method is an approximation of the Navier-Stokes equation for 2D. For one, it assumes that the column of water doesn't have gaps and has constant pressure all the way through. The lack of gaps prevents us from having e.g. caves with water under a sea or a submerged submarine. Water below the submarine is displaced when it dives while water flows back over the submarine. This complex behavior is not possible to simulate with the pipe method.

Non-constant pressure and different flow directions is important to simulate more complicated ocean and wave behavior where different layers of water flow with differing speeds or even opposing directions. Breaking waves, for example, require this as they are caused by the bottom of the wave slowing while the top continues with static velocity, leading to it overtaking the bottom and breaking. This is likely less relevant for games but this is difficult to judge without investigating it.

The simplest way to solve both issues is by extending the pipes method to the third dimension. Instead of having one cell for each column of water, we could have each cell be a cube of volume. Such a cell would have at least 6 neighbors, for each orthogonal direction. Unfortunately, the naive approach would multiple the number of cells by potentially multiple orders of magnitude.

A more sophisticated approach would have a small number of cells with variable cell heights. This allows in the cave with water example to have one cell for the water in the cave, one for the gap between them and one for the water in the lake. This allows simulating pressure (if the cave is connected to the lake) and having air gaps in a water column without increasing computation time as significantly.

6 Possible Evaluation

Evaluating this method in regards to performance is simple as we can measure the execution time of the shader. Unfortunately, comparing the fidelity of the simulation is significantly harder. While most methods can be roughly ordered by increasing fidelity and realism, this becomes more difficult once we incorporate computation time. A method that is more realistic but takes 10x as long should be thought of as having only a 10th of the resolution (as we are constrained to at most 16ms per computation step for a reasonable frame rate). Comparing superior methods with lower resolution against a simpler but high-resolution method is difficult.

Another issue is that increased realism does not necessarily look better to a user. For example, water movements below the surface are generally invisible and their effects hard to observe, ergo irrelevant for games. Instead, we could use a simple simulation and fake additional features to enhance the visuals. Whether we can convincingly fool players and whether the additional fidelity is in fact necessary for a game is hard to judge.

For example, City Skylines uses a variation of the pipe method to simulate its water flows. Would it benefit from a more complicated method allowing submarines to be accurately simulated? Hardly. On the other hand, breaking waves would significantly improve tsunamis etc if they actually resemble real waves. Similarly, user perception is difficult to objectively study and depends largely on the expectations of the user. Furthermore, fine tuning parameters like wave height, viscosity etc. will have huge impacts on the user perception but are mostly independent of the method as almost all are able of being tuned in such ways.

We propose that a simple comparitive study to evaluate the different water simulation methods. We implement multiple methods (or the same method with different parameters) and present users with two scenarios that differ only in the method used. This way, a user can evaluate whether they prefer method A or B for e.g. tsunami, an ocean or a river. This can then be used to order the quality of each methods for each scenario, allowing a developer to choose the method best suited to their game.