Masaryk University
Faculty of Informatics

Real-time Water Simulation in Game Environment

Master's Thesis

Bc. Jakub Medvecký-Heretik

Brno, Spring 2018

Masaryk University
Faculty of Informatics

Real-time Water Simulation in Game Environment

Master's Thesis

Bc. Jakub Medvecký-Heretik

Brno, Spring 2018

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Jakub Medvecký-Heretik

Advisor: Mgr. Jiří Chmelík, Ph.D.

Acknowledgements

I would like to thank the biggest water lover I know, Natasha, for her infinite support and help. Special thanks goes also to my family and friends for their spiritual support throughout my studies.

Furthermore, I want to thank my advisor Mgr. Jiří Chmelík, Ph.D. and the Unity Grand Maester Pavel for their advice throughout this work.

Abstract

The aim of this master thesis is to investigate the existing fluid simulation techniques and select the ones that can be used in real-time applications such as games. Based on this research and following a set of predefined requirements, three different approaches to liquid simulation using cellular automata were implemented. They were compared in terms of performance, memory consumption, and how they tackle the existing challenges. Subsequently, the most suitable technique was chosen and further improved. The result of this thesis is a standalone application demonstrating the final solution.

Keywords

fluid, water, liquid, real-time, simulation, game, cellular automaton

Contents

List of Tables

List of Figures

List of Listings

Introduction

Fluids and our interactions with them are part of our everyday life – from air to oceans they are at the core of the most impressive phenomena we know. Due to our familiarity with the fluid movement, plausible simulation of it remains a challenging problem despite enormous advances in the recent years. The distinction between liquids and gases can influence the way we model the fluid, but both obey the same basic fluid formulae and share similar properties.

A *fluid* is any substance that flows and does not resist deformation. People often use the words fluid and liquid interchangeably, but from a physical point of view, fluid is a phase of matter and includes liquids, gases, plasmas, and to some extent, plastic solids. The branch of physics that deals with the governing forces and mechanics of fluids is called *fluid mechanics* and has a wide range of applications.

Fluid simulation is the topic of this thesis. We are focusing especially on real-time simulations usable in computer games.

In the first chapter the reader is introduced to the field of fluid simulation and the techniques which study the behaviour of fluids in a scientifically rigorous way are described.

This is put into contrast in the following chapter with the techniques used in movie and gaming industries, in which the qualitative visual behaviour of fluids is sought rather than physically correct results.

In the third chapter we introduce the game which influenced the development of the practical part of the thesis.

The next two chapters describe this development in bigger detail. It focuses mainly on the technique used, the existing similar solutions, the details of the implementation itself, and the additional improvements made to the original implementation.

The results of this work are presented in the sixth chapter, along with a comparison of the different stages in the process of development, the impact that the introduced improvements had, and the scaling of the final solution.

In the last chapter the resulting application is summarized and its potential future improvements are discussed.

1 Fluid simulation

1.1 Computational fluid dynamics

The modern discipline devoted to solving and analysing fluid mechanics problems with numerical methods and data structures is called *Computational fluid dynamics* (CFD). In CFD computers are used to perform the calculations required to simulate the interaction of fluids with rigid bodies, surfaces, or other fluids.

Navier-Stokes At the basis of solutions to almost all CFD problems are the famous *incompressible Navier-Stokes equations* [1, p. 3], which describe the motion of viscous fluid substances. These equations arise from applying Newton's second law to fluid motion, with the assumption that the stress in the fluid is the sum of a diffusing viscous term (proportional to the gradient of velocity) and a pressure term:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} + \nu \nabla \cdot \nabla \vec{u} \tag{1.1}$$

$$\nabla \cdot \vec{u} = 0 \tag{1.2}$$

where $\vec{u}$ is the velocity, ρ is the density and p is the pressure of the fluid. The symbol $\vec{g}$ is the familiar acceleration due to gravity, usually (0, -9.81, 0) m/s^2. The last symbol ν is kinematic viscosity. It measures how viscous the fluid is, i.e. how much the fluid resists deforming while it flows (or informally, how difficult it is to stir it).

The Naviers-Stokes equations are further modified to achieve various results, such as modeling the weather, the ocean currents, the water flow and the air movement around a wing. The equations, in their full and simplified forms, are found in various applications, such as the design of aircrafts and cars, the study of blood flow, the design of power plants, the analysis of pollution, and others.[1]

1. The Navier–Stokes equations are also of great interest from a purely mathematical point of view, since it has not yet been proven that in 3D solutions always exist, or that if they do, they do not contain any singularity. This is one of the seven most important open problems in mathematics and there is a prize of $1,000,000 for a solution or a counterexample [2] [3].

Different viewpoints Further, when we think about a moving continuum, there are two main viewpoints in tracking this motion - *Eulerian* and *Lagrangian*, as illustrated in Figure 1.1.

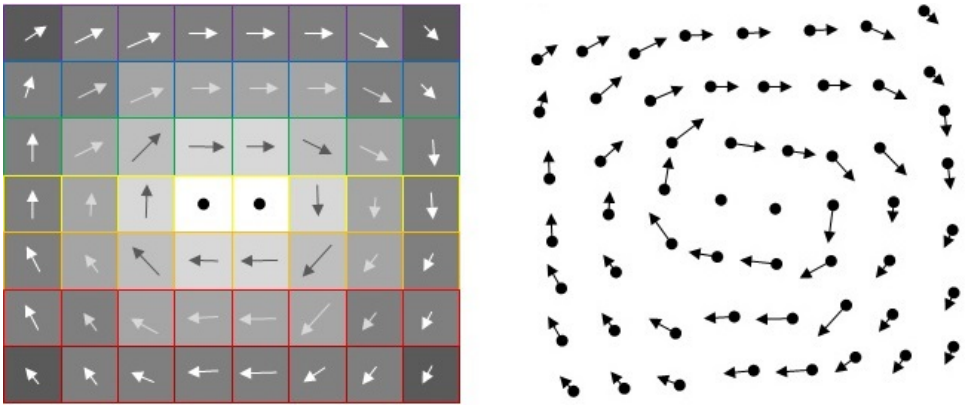

Figure 1.1: The Eulerian (left) and Lagrangian (right) viewpoints, taken from [4].

The following is a description from Robert Bridson's book *Fluid Simulation for Computer Graphics*:

> "One way to think of the two viewpoints is in doing a weather report. In the Lagrangian viewpoint you're in a balloon floating along with the wind, measuring the pressure, the temperature, the humidity, etc. of the air that's flowing alongside you. In the Eulerian viewpoint you're stuck on the ground, measuring the pressure, the temperature, the humidity, etc. of the air that's flowing past. Both measurements can create a graph of how conditions are changing, but the graphs can be completely different as they are measuring the rate of change in fundamentally different ways." [1, p. 7]

Both views are useful, each having its advantages and disadvantages, and we often switch between them or even combine them depending on the specific requirements of the application.

1.2 Eulerian

The Eulerian viewpoint tracks and stores fluid properties at fixed points in space. At each point, in a region containing fluid, we observe how measurements of quantities such as the velocity, the density or the pressure change in time as the fluid flows by [1, p. 7-9]. The positions of these points never move. These quantities are then updated according to some discretization of the Navier-Stokes equations. For example, as a warm fluid moves past followed by a cold fluid, the temperature at the fixed point in space will decrease – even though the temperature of any individual particle in the fluid is not changing! The Eulerian approach corresponds to grid based techniques.

Related work The simulation of a fluid can be split into three main steps: *advection*[2], *surface tracking*, and *pressure projection*. Each of these steps has been the focus of many research papers.

The pioneers in the field of Eulerian fluid simulation in computer graphics include Foster and Metaxas [6] who in 1996 introduced the first 3D grid based water simulation, using finite differences to solve the Navier-Stokes equations. Another innovator in the field is Stam [7] who added the semi-Lagrangian method for advection and made it unconditionally stable which allowed larger timesteps than before.

The eulerian methods do not implicitly define a free surface like particle based methods do [8]. To track the free surface of liquids, Foster and Fedkiw [9] combined Lagrangian particles with the level set method and Harlow and Welch [10] introduced the marker particles method.

Furthermore, a wide variety of methods have been proposed to accelerate fluid simulations in order to cope with large scenes. For example, usage of adaptive grids in order to focus the computational effort to important regions. Overview of these methods as well as more thorough overview of the related work can be found in a paper by Chentanez and Müller [11].

---

2. An operation which displaces the fluid in current quantity field, based on the velocity field over a time step [5]. However, finding a stable solution is problematic, as the Euler integration methods (using the forward finite difference) are unconditionally unstable [1, p. 28].

Rendering The rendering of simulations which are tracked on a grid can be done directly using *volume rendering* methods. Below are described the two main techniques:

1. The *Volume Ray Casting* [12, p. 21-26] method, also known as "ray marching", is illustrated in Figure 1.2. At first, a ray is generated for each pixel of an image. This ray is sampled at regular or adaptive intervals throughout the volume and clipped by the boundaries of the volume in order to save computational time. The colours of the samples are then shaded and eventually composited into the accumulated RGBA colour of the ray. The RGBA colour is finally converted to RGB colour and deposited in the corresponding image pixel.

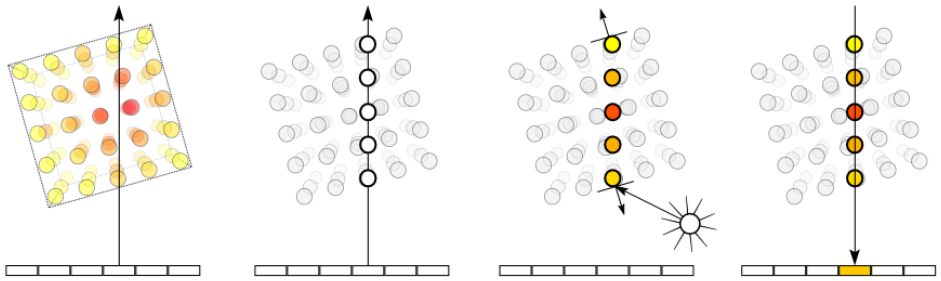

Figure 1.2: The individual steps of Volume Ray Casting, taken from [13].

2. The *Marching Cubes* algorithm, first presented by Lorensen and Cline in 1987 [14], is a very common algorithm for extracting a polygonal mesh of an isosurface from a voxel grid.

The algorithm works by sampling each voxel with a cube in its 8 corners. We check the corners of the cube if their value is bigger than a specified isovalue and determine whether the voxel contributes to the extracted surface or not. The knowledge of corners and edges contributing to the surface gives the information about the triangles in a voxel. There are in total 256 configurations, but if we exclude empty and full cube, and account for symmetries in the remaining 254 possibilities, we are left with only 14 unique configurations which are shown in Figure 1.3. After extracting the triangles from every voxel, an isosurface is obtained. In our case of a grid based fluid simulation, the

isosurface stands for a body of water and voxel stands for a single grid cell with fluid quantities out of which we can calculate the desired isovalue.

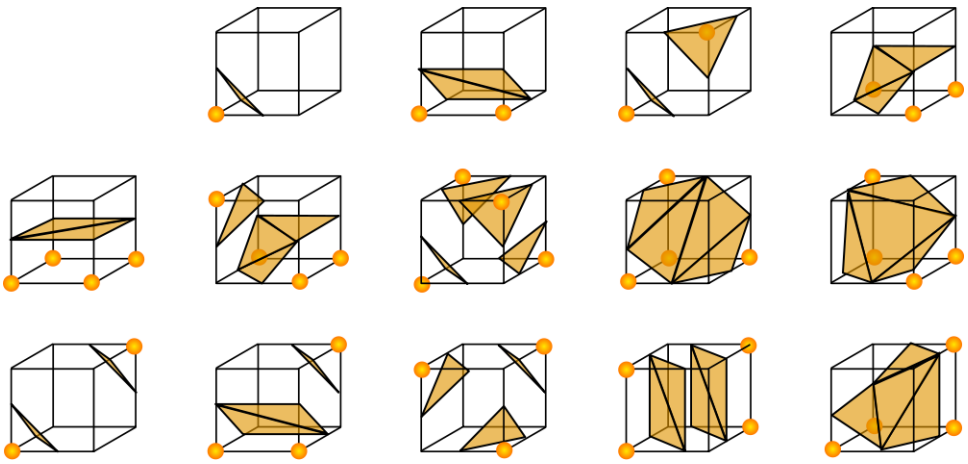

Figure 1.3: The configurations for Marching Cubes, taken from [15].

Summary Grid based techniques exhibit the advantage of having higher numerical accuracy (since it is easier to work with spatial derivatives on a fixed grid, as opposed to an unstructured cloud of particles) [1, p. 7], they are better at tracking smooth water surfaces [16], and incompressibility is also easily achieved [17].

On the other hand, they have the following weaknesses: a big resolution of the grid is required to capture finer details [18, p. 2], there are difficulties with advection [17], they are often computationally more demanding than particle based simulations [16], and pure Eulerian methods suffer from high numerical dissipation [1, p. 35-39] which manifests as an unphysical viscosity – this makes modeling low viscosity fluids like water and air difficult.

1.3 Lagrangian

Unlike Eulerian methods, the Lagrangian viewpoint treats the continuum just like a particle system. We can think of fluid in terms of a vast collection of particles that move around. Each point in the fluid or solid is labelled as a separate particle with various properties, such as mass, position, velocity, density, temperature, etc. Then the individual particles are observed as they move through space and time [19, p. 71-73] [20, §3-§7, §13-§16]. Methods utilising this approach are also called particle-based methods.

We can think of each particle as being one molecule of fluid. Solids are almost always simulated in a Lagrangian way, with a discrete set of particles usually connected up in a mesh [1, p. 7].

Related work A scheme that uses the Lagrangian viewpoint is, for example the mesh-free vortex method described by Yaeger et al. [21] as soon as in 1986 and Gamito et al. [22] or more recently by Angelidis et al. [23] and Park and Kim [24]. In them, large time steps are allowed and computational elements exist only where interesting flow occurs – used mostly for calculating aerodynamics. Another scheme is the Smoothed Particle Hydrodynamics (SPH) introduced by Desbrun and Cani [25] in 1996 and later improved by Premoze et al. [8], where particles move under the influence of the hydrodynamic and gravitational forces.

SPH method, being the most common example of particle-based method [1, p. 7-9], is flexible but it can only solve compressible fluid flow. Simulating incompressible fluids (like water) can become rather difficult and several extensions to SPH have been proposed to allow this. For example, weakly compressible SPH [26], Predictive-corrective incompressible SPH [27] or another gridless particle method called the Moving Particle Semi-Implicit method that solve the governing Navier-Stokes equations for incompressible fluids [28].

Rendering The particle-based simulations do not generate a surface and their results are usually stored as an unorganized fluid particles point cloud, where individual particles' positions and densities are known.

Probably the easiest, but computationally expensive technique, is sampling the particles onto a uniform 3D grid where we can step through the grid points and sample the fluid density on these points. Then, we can use this grid as input for volume rendering techniques, used for the Eulerian simulations mentioned on page 6, to extract the surface. While there are benefits to sampling particles onto a grid, there are other cheaper techniques that often produce comparable results.

Usually, these techniques use ray tracing and a function for each particle. The function typically used for this purpose was introduced by Blinn [29], who demonstrated surface reconstruction from native spherical particle geometry using implicit surfaces and is often referred to as *meta-balls* or "blobbies" because of its blobby-like look. Numerous improvements have been made to the function above throughout the years, the most important one presented by Williams in [30].

To prevent a jelly or blobby look, the *screen-space fluid rendering* techniques have been developed. For instance, a technique derived from Marching Squares (Marching Cubes in 2D) [31], or another method inspired by the work of Laan et al. [32] which uses ellipsoid splatting. This methods operates purely in screen-space without any meshes and only generates the visible surface closest to the camera. An overview of the process using screen-space rendering techniques can be seen in Figure 1.4.

At first, the particles are rendered as spherical point sprites and the depth of particles is computed. Then, the surface normals from the said depth are generated, the surface is shaded and reflection from the fresnel map and the cube map is added. Lastly, the transparency is approximated using additive blending through the volume to attenuate the colour and as the last step the image is merged back with the rendered scene [33].

As Foster and Fedkiw pointed out, a big number of particles is necessary and it is difficult to create a smooth surface out of particles [9]. More in-depth overview of the rendering techniques for particle based simulations can be found in a recent work by Reichl et al. [34].

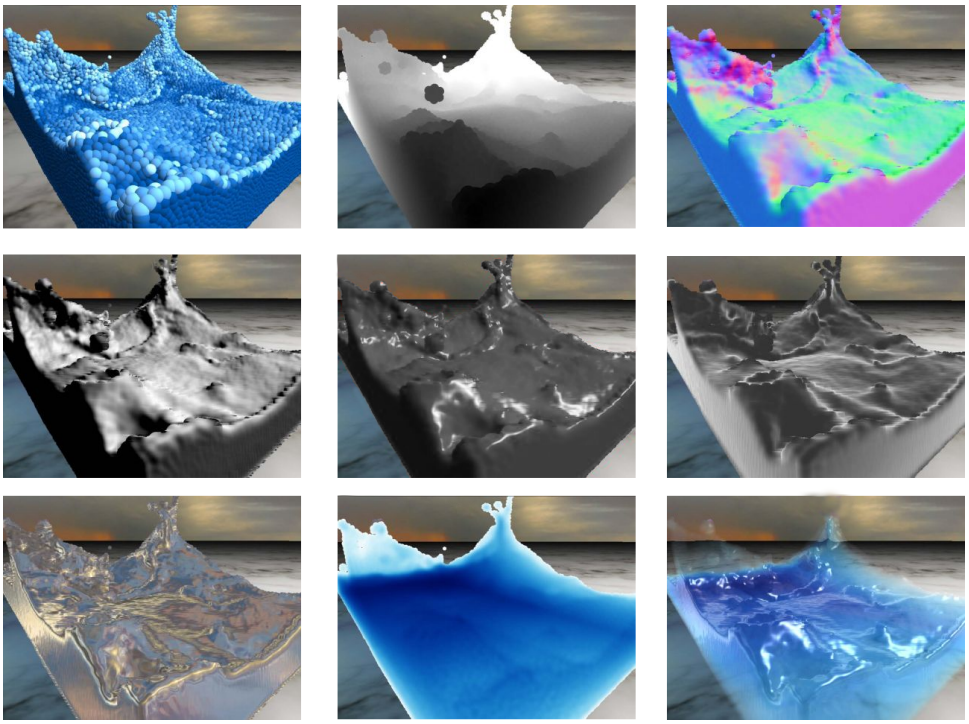

Figure 1.4: The steps of screen-space rendering, taken from [33].

Summary Particle based techniques can be considered intuitive (the motion of the fluid is the motion of particles themselves), easier to program and understand, they handle advection trivially [5], have a low numerical dissipation [35], handle free-surfaces implicitly [8], are not necessarily constrained to a finite grid (only perform computation where necessary), and are typically faster than the grid based techniques [16], therefore more suitable to be used in real-time applications.

However, they come with considerable downsides such as difficulties in dealing with spatial derivatives on an unstructured particle cloud [1, p. 7], worse accuracy when the density of particles is not sufficient (they require large number of particles for realistic results) [18], and have problems with incompressible condition stability when using bigger time steps [17]. Extracting smooth surfaces and rendering of particle based fluids, as already discussed, also presents a challenge.

1.4 Hybrid

Hybrid methods combining the Eulerian grid and Lagrangian particles also exist. Claiming the advantages of both worlds, the grid is used for the pressure solve (enforcing incompressibility), while the particles are used for advection (giving low numerical dissipation) [5] and enable us to track the free surface implicitly.

However, they have higher computational cost than the pure Eulerian methods because we need to maintain both particles and a grid, and are limited by the size of a grid. Furthermore, it is still necessary to find a way to render the particles.

Related work The first example of hybrid grid-particles approach is the Particle In Cell (PIC), introduced by Harlow [36], where compressible fluids are simulated. All forces are computed on the grid just like in an Eulerian solver, whereas advection is done purely on particles using interpolated grid velocities. The resulting velocities then get transferred from the particles back to the grid. Thanks to this averaging and interpolation, PIC suffers from excessive numerical dissipation [17].

This issue was addressed by the Fluid Implicit Particle (FLIP) method, introduced by Brackbill and Ruppel [37], in which particle velocities got updated with the change in grid velocity from the previous step instead. PIC and FLIP were later also adapted for incompressible flow [17].

These hybrid methods are still being researched as we can see in more recent works, such as the Affine Particle-In-Cell (APIC) method, proposed by Jiang et al. [38], which improves the accuracy of the transfers in PIC techniques by augmenting each particle with a locally affine rather than locally constant, description of the velocity. This reduced the dissipation of the original PIC without suffering from the noise present in the historic alternative FLIP.

In an even more recent work, a generalization of the APIC method, called PolyPIC [39] is presented. In this method each particle is augmented with a more general local function, which greatly improves the energy and vorticity conservation over the original APIC. It is also shown that the cost of this generalization is negligible over APIC,

when using a particular class of local polynomial functions, and it retains the filtering property of APIC and PIC and thus has similar robustness to noise.

2 Real-time fluid animation

Advances in computational fluid dynamics often cannot be directly applied to computer graphics, because they have vastly different goals. The characteristics of CFD and the film or gaming industry applications are quite different. In the film industry the simulated environments have to match the reality to the extent that a viewer cannot distinguish between them. We need very high resolution simulations with very realistic appearance and motion. The gaming industry, on the other hand, must compromise and aim for the highest realism possible while still maintaining an interactive frame rate.

Fluid animation refers to computer graphics techniques for generating realistic animations of fluids [1]. Fluid animation differs from CFD in that fluid animation is used primarily for visual effects – emulating the qualitative visual behaviour of a fluid, with less emphasis placed on rigorously correct physical results, whereas CFD is used to study the behaviour of fluids in a scientifically rigorous way. Nonetheless, in fluid animation we often rely on approximate solutions to the Navier-Stokes equations that govern real fluid physics.

Fluid animation can be performed with different levels of complexity, ranging from time-consuming, high-quality animations for films or visual effects, to simple and fast animations for real-time applications such as computer games.

2.1 Eulerian and Lagrangian

The development of fluid animation techniques based on the Navier-Stokes equations began with the already mentioned work by Foster and Metaxas [6] who implemented solutions to 3D Navier-Stokes equations in a computer graphics context. In 1999, Stam published the famous "Stable fluids" [7] method, which allowed for much larger time steps and therefore faster simulations.

Simulations of small-scale gaseous fluids, such as fire and smoke, are often based on Stam's seminal paper "Real-time fluid dynamics for games" [40]. The paper builds upon one of the methods proposed in his paper from 1999, which later was successfully implemented in 2D by Mick West [41] and 3D by Quentin Froemke [42]. This general

technique was extended by Ronald Fedkiw and co-authors to handle more realistic smoke [43] and fire [44] as well as complex 3D water simulations using variants of the level set method [9, 45].

So far, only a few researchers have shown 3D Eulerian liquid simulation at interactive rates [11]. To achieve real-time performance, Crane et al. confined the liquid to a relatively small rectangular domain without general fluid-solid interaction [46], while Long and Reinhard leveraged the discrete cosine transform to speed up their simulation [47]. Typically, when applying Eulerian techniques to a game environment, the computational cost scales with the size of the grid, as opposed to the actual amount of fluid so if we only have a small portion of a game world filled with fluid this may become prohibitive.

Usage of Lagrangian techniques for real-time applications is more suitable since, as we have already indicated, they are faster than Eulerian methods and having no grid, the fluid is free to flow anywhere within the game environment. For example, SPH has been adapted for animation purposes by Muller et al. [48]. Notwithstanding, rendering these kinds of fluids remains a challenge.

2.2 Shallow water

Alternatively, if only the fluid surface is required, real-time performance has also been achieved by using the pipe model [49] and lower dimensional techniques in 2D, such as height fields [50], the wave equation [51], and the Shallow Water Equations (SWE) [52, 53] to name a few.

The main limitation of methods based on the wave equation or the pipe model is that vortices which are responsible for many interesting water phenomena, such as swirling foam, whirlpools, and correct advection of floating objects, are not present in the model.

The Shallow Water Equations can capture these phenomena because in addition to the height field, they describe the evolution of a 2D velocity field normal to the water columns. Furthermore, the SWE have many important advantages over the wave equations as well: they take ground topography into account and they can simulate the flooding of previously dry areas [54].

The SWE were introduced to the graphics field by Layton et al. [55] and are an approximation of the full Navier-Stokes equations. This approximation is valid only when the vertical motion of the water is small.

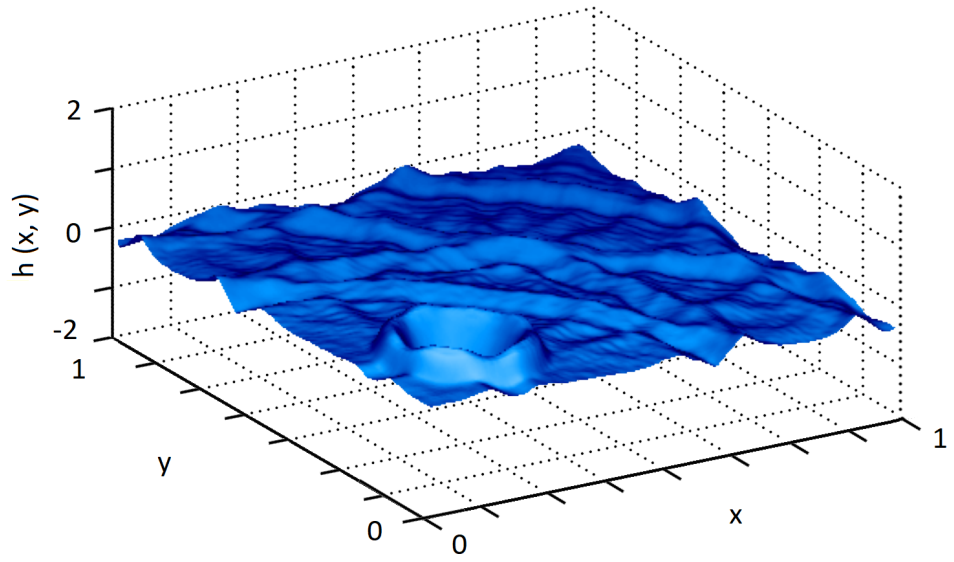

Figure 2.1: Output from a SWE model of water, taken from [56].

The fluid surface is represented as a height field, as illustrated in Figure 2.1, so phenomena such as breaking waves or waterfalls still cannot occur. But they are much less computationally intensive than any 3D method, therefore for large bodies of water, such as lakes or oceans or real-time applications where performance is important, the SWE may be the only option.

2.3 Cellular automata

Recently, the advantages of changing the viewpoint from a common approach, where we model the fluid dynamics by Navier-Stokes equations and apply numerical techniques for the simulation to the models of the Lattice Gas Automata (LGA) have been demonstrated [57, 58].

LGA is a type of *cellular automaton*[1] used to simulate fluid flows. These automata are discrete models based on point particles that move on a regular grid structure (lattice) according to suitable and simple rules in order to mimic a fully molecular dynamics [59]. Particles can only move along the edges of the lattice and their interactions are based on simple collision rules.

LGA does not require solution of complicated equations and combines the advantage of the low computational cost and the ability to mimic the realistic fluid dynamics. It has even been demonstrated that the Navier-Stokes model can be reproduced from LGA [60]. Also an animating framework for computer graphics applications based on LGA was developed by Xavier et al. [57] and further improved by Judice et al. [61].

Heintz et al. [62] have recently put forward a rather simplified approach to real-time fluid simulation based on cellular automata with the goal of demonstrating the performance and extensibility of such approach and a focus on short computation time, which could be interesting even for VR applications.

2.4 Lattice-Boltzmann

A relatively new simulation technique, which originated from the lattice gas automata, for complex fluid systems has attracted interest from researchers in computational physics called Lattice-Boltzmann model (LBM) [63, 64].

In LBM, the discrete Boltzmann equation (the analogue of the Navier-Stokes equation at a molecular level) is solved to simulate the flow of a Newtonian fluid with collision models instead. The number of physical phenomena covered by the model at this molecular level of description is larger than at the hydrodynamic level of the Navier-Stokes equation [65].

Due to its particulate nature and local dynamics, LBM has several advantages over other conventional CFD methods, especially in dealing with complex boundaries, incorporating microscopic interactions, and parallelization of the algorithm, since it was designed from scratch to run efficiently on massively parallel architectures [65]. This

1. Cellular automata are described in more detail in their own section on page 23.

efficiency leads to a qualitatively new level of understanding, since it allows solving problems that previously could not be approached (or only with insufficient accuracy) [65]. For a more in-depth review, history, development, and progress in LBM refer to a book by Succi [66] and a paper by Perumal and Dass [67].

Lattice gas automata and Lattice-Boltzmann method approaches for real-time applications, such as games, are highlighted and compared in terms of efficiency in a work by Judice et al. [68].

3 Ylands

An interest for a real-time water simulation has been shown by the game studio Bohemia Interactive for their upcoming game Ylands, which is already available for download in an early access[1] version on Steam[2] [69]. The practical part of this thesis was created in collaboration with, and guided by the specifications laid down by Bohemia Interactive, with the perspective of the solution being eventually integrated into Ylands.

Figure 3.1: A screenshot of Ylands, taken from [69].

1. a funding model in the video game industry where consumers can pay for a game which is still in development and obtain access to the pre-full release versions of the game, while allowing the developers to continue working on the game, helping them to debug the game and provide feedback and suggestions
2. the largest digital distribution platform for PC games

Developers describe Ylands as "sandbox[3] exploration adventure and a platform for making custom game" [69]. Players can create their own character and choose a geographical environment in which the action takes place. Environments include tropical, subtropical, temperate, winter and others types of climate. The game itself is similar to Minecraft but also features a built-in editor that allows the players to generate and share their own content with other players[4]. Players can unleash their imagination not only by creating visually distinct surrounding but also designing whole scenarios [70]. Thus, the gameplay may take place in various historical eras or fiction genres, such as Wild West, Medieval Europe, Steampunk and many others.

The game was written mostly in C# programming language using the Unity engine, while also utilising and integrating several third-party libraries. The procedural generation of the world, e.g. terrain, water and objects, as well as dynamic loading/unloading of the visible parts of this world, was initially based on the TerrainEngine framework [71], but has since been heavily pruned and customized.

World representation Ylands world is divided into an implicit regular 3D voxel grid aligned with the world coordinate system. The grid is divided as follows:

- *Voxel* is the minimal grid unit, size of a voxel being 0.75m. The terrain and fluid data are per voxel.
- *Chunk* is a 16x16x16 section of voxels. The terrain and fluid meshes are generated per chunk.
- *Block* is a 8x8x8 section of chunks, i.e. 128x128x128 section of voxels. The minimal unit by which Ylands world is loaded.

The world is currently only built in a 5x3x5 blocks locus centered at the player-controlled character. On server, this means one locus per player, whereas on client, it is just the one locus around the character

3. sandbox, or open world, is a term for a video game in which a player can roam a virtual world and approach objectives (such as building and creative activities) freely, as opposed to a game with more linear gameplay
4. `https://ylands.net/`

currently played by the client. These blocks are created on demand, typically as a player character explores the world and blocks in his vicinity are generated, loaded, and built for the first time.

3.1 Water problems

The current water simulation in Ylands is still based on a solution by original TerrainEngine framework with customized rendering which enhances the visual quality of the water. The actual simulation, however, uses a very simple cellular automata approach and its numerous drawbacks and deficiencies in the behaviour of the water have been discovered during the development. Some of them include:

- air bubbles are created in the places where parts of the underwater terrain are removed, either by digging or as a result of explosions
- big deep holes in underwater terrain which are created by terraforming[5] are not subsequently filled with water
- water "sticks" like jelly to objects and if those objects or supporting platforms are removed, the water does not fall
- water does not behave realistically in waterfalls – once it falls to the bottom it does not raise everywhere equally, but clumps on top of itself and takes a long time to spread
- pressure is completely ignored: if we have more communicating vessels or an U-bend shaped terrain and start pouring water on one end, the water will not raise up on the other end to an equal level but will get stuck at the bottom instead.

3.2 Requirements

Suggesting a new water simulation for Ylands a list of requirements was proposed by Bohemia Interactive with which the new simulation

5. modification (adding, removing, smoothing etc.) of terrain initiated by the player

should comply, ultimately allowing it to be integrated into the existing game.

Ideally, it should not suffer from the issues mentioned above. This means that, gravity should be implemented robustly and pressure at least for communicating vessels should be supported. Viscosity is another physical property of the water to be supported in the simulation, as this would allow for the differentiation between various types of liquids (e.g. lava, oil, jelly).

The whole simulation must be running on CPU because Ylands is a multiplayer game with an authoritative server and "thin" clients[6], and these authority servers do not posses GPU units. Since the CPU on the game server is already running the whole game, we are looking for a fast performance simulation with the lowest memory footprint possible.

The solution must also be able to be parallelized and run asynchronously in the background thread so that it does not block the main game thread.

In the current world representation in Ylands, a voxel can contain only terrain or water data exclusively. A new fluid simulation requires that both the terrain and the water compound are stored in a single voxel, for water to "surround" the terrain more nicely and to avoid artifacts in the transitions between water and terrain. For instance, when in the current solution we have 3/10 of voxel filled with terrain and next to it voxels full of water – no water is rendered in the terrain voxel and a sudden "wall" of water is rendered next to it. In the new solution the voxel could contain 3/10 terrain and 7/10 water so that both parts get rendered and we get a smoother transition.

6. Everything in the game happens in a central server and the clients are just privileged spectators of the game. In other words, the game client sends inputs (key presses, commands) to the server, the server runs the game, and the results are sent back to the clients.

4 Implementation

With regards to water simulations in games, the simplicity, speed and plausibility are prioritized over accuracy. Most of the time, working on implementation involves getting the general feel of the behaviour from the CFD literature and then programming a behaviour only vaguely approximating it. In almost every case, the simulation can be enormously simplified while still looking perfectly correct to most people [72].

For the practical part of this thesis, a fluid simulation using *cellular automaton* approach was chosen. Deciding between the real-time fluid simulation techniques described in the second chapter, this approach was selected because it requires the least memory and computation power while still satisfying all the company's requirements. It is also compatible with the world representation used in Ylands and existing terrain data. The already implemented rendering techniques (Marching Cubes) and visual effects (shaders) can be reused too. And lastly, since the algorithms using cellular automata are very easy to parallelize, as will become clear throughout the text, it should be possible to incorporate the final solution into an existing multi threaded job system in Ylands without too many modifications.

4.1 Cellular automaton

Cellular automata (CA) were originally introduced in the 1940s by John von Neumann who, attempting to understand biological evolution and self-reproduction, formalized the idea in order to create a theoretical model for a self-reproducing machine. The theory was complemented by his colleague, S. Ulam, who suggested to use a cell-based concept. Von Neumann describes this in his book: *Theory of Self-Reproducing Automata* [73].

After the book's publication in 1966, more scientists became interested in the subject and by the end of 1970s, John Conway's *Game of Life* made its first public appearance [74] – arguably the most popular two-dimensional cellular automaton. Nowadays cellular automata are studied in many scientific fields including computer science, mathe-

matics, physics, complexity science, theoretical biology, microstructure modeling as well as outside academia.

Cellular automata are the mathematical idealizations of physical systems in which space and time are discrete and physical quantities take on a finite set of discrete values [75].

A cellular automaton consists of a regular grid with a discrete variable at each site called a *cell*. The grid can be in any finite number of dimensions and each cell can be in one of a finite number of states. The state of such automaton is completely specified by the values of its cells [75]. For each cell, a set of cells called *neighbourhood* is defined relative to the specified cell. The two most common neighbourhoods are illustrated in Figure 4.1 in 2D as well as 3D: von Neumann (all immediately adjacent cells) on the left and Moore (all surrounding cells) on the right. Usually for water simulation in games which use CA, the von Neumann neighbourhood is considered, since its faster to work with (only 6 neighbours need to be considered for each cell) and the rules that are used for simulations give almost the same sort of results, whichever of the two definitions we use [72].

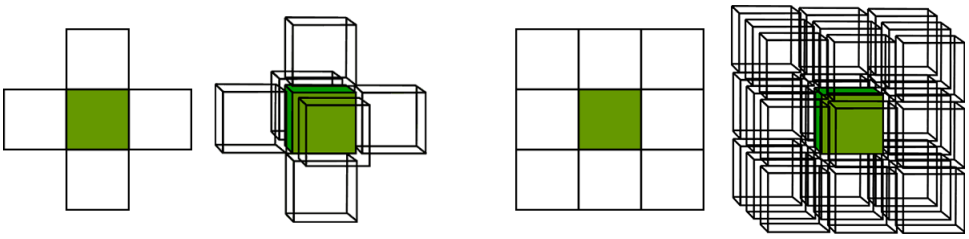

Figure 4.1: Von Neumann (left) and Moore (right) neighbourhood in 2D and 3D (3D taken from [76]).

An initial state (time $t = 0$) is selected by assigning a state for each cell. A cellular automaton evolves in discrete time steps (advancing t by 1) creating new generation of cells, according to a definite set of *local rules* (generally, mathematical functions) [77, p. 27] that determine the new state of each cell based on its value and the values of cells in its neighbourhood on the previous time step.

Typically, the rule for updating the state of cells is the same for each cell and does not change over time, and is applied to the whole grid simultaneously [78, p. 40].

CA is an usual choice for games using voxel grid for terrain representation. In these terrains, which are usually vast, a fast performance, low memory consumption, compatibility with the world representation, simple local rules, and ease to program and parallelize are crucial for a water system. CA is also very often used in games for procedural generation of content such as caves, biomes, items, monsters and so on.

There are more sophisticated cellular automata which work even on molecular levels, such as Lattice-Gas or Lattice-Boltzmann, as described in the second chapter. However, when taking into consideration all the other systems that are already running in a game (e.g. AI, pathfinding, combat, multiplayer synchronization, scene loading and many others), these techniques prove to require too much memory or processing power to be used in games. It is possible to run them at interactive rates by moving the simulation to the GPU, but the developers community has diverse opinions of that as the GPU should not be burdened with simulations because it is already fully occupied with rendering. Or the design of the application does not allow for the simulation to be run on GPU as it is in our case.

4.1.1 Existing solutions

Fluid simulation, or more precisely water simulation, has already been implemented by using cellular automata in various games and projects. Below are briefly described the most notable ones (sorted from the simplest to the more complicated ones), that served as an inspiration for my own implementation, and their corresponding CA rules which are applied to each cell[1] in the grid in each iteration.

Minecraft Minecraft is arguably the most popular sandbox voxel grid based game. According to the Minecraft's official wiki[2], all liquids in Minecraft originate from a source liquid cell, which is a liquid cell (as opposed to solid or air cell) that is completely full. Liquid cell has a value, which determines how "empty" it is. Flowing liquids have this value increased depending on their distance from the source. Solid

1. In this text, we will use a term 'cell' or 'voxel' for the minimal grid unit, but terms 'cube' or 'block' are also commonly used in various sources with the same meaning.
2. `https://minecraft.gamepedia.com/Liquid`

cells are unaffected by liquids. Source liquid cells that are completely confined by solid ones cannot start a flow. Otherwise they start to flow and the liquid spreads according to very basic fluid dynamics, implemented as the following CA rules:

- First, the source liquid considers the cell directly below. If that cell is air, it is replaced by the liquid.
- If the cell under the flow is deemed to be solid, the procedure continues by considering the four cells horizontally around. If any of the four cells are air, they are converted to that liquid and assigned an increased "emptiness" value.
- If all four surrounding cells are solid or if the liquid reaches its maximum "emptiness", the flow stops.
- Some kind of viscosity is also supported by having a speed value in flowing liquid cells that governs how fast the spreading effect takes place. For example lava is much slower than water, it moves at the speed of 4 cells per second.
- Liquids also take into account the steepness of the terrain by considering the shortest distance to the edge of a cliff and prioritizing flow in that direction.

Ylands/TerrainEngine In Ylands, powered by TerrainEngine, the solid cells do not take part in the simulation and the whole water simulation consists only of a few simple rules:

- First we check the bottom neighbour of current cell. If its empty or already contains water but is not full yet, we move there as much of the current's cell water as possible.
- Then we check if there is still some remaining volume in the cell being processed and we divide it equally among the current cell and its horizontal neighbours which contain less water than the current cell.
- We also keep track of the flow direction. The remainder after the division, if there is one, goes in favour of neighbouring cell with water flowing in the same direction as the current's cell flow.

Janis Elsts Another real-time CA water simulation, this time in 2D has been described by Janis Elsts on his blog[3] where an online demo written in Java can be found too. The same approach has been later ported to Unity engine and presented by Jon Gallant online also including a demo[4].

This approach uses very similar rules to the previously mentioned ones. The first two are almost identical – transfer as much water to the bottom neighbour as possible and distribute the remaining volume equally among horizontal neighbours and self.

However, one more rule is added which simulates pressure. The main idea is taken from a paper by Tom Forsyth [72] which proposes to treat water as a slightly compressible liquid.

Forsyth suggests to store pressure as a slight excess volume of water in a cell, above what the cell should normally be able to hold. If there are two or more water cells stacked vertically, the bottom cells is able to hold slightly more water than the one above.

> "In practice, the amount of compression needed is tiny allowing just 1% more water per cell per cube height is easily enough. In a static body of water whose cells can normally contain one litre of water each, the cells at the top will contain one litre, the ones under them will contain 1.01 litres, the cells under those will contain 1.02 litres, and so on to the bottom. This tiny amount of compression will be completely unnoticeable to the player, but has enough dynamic range to allow all the usual properties of liquids. For example, the levels of water in two containers joined by a submerged pipe will be the same, even if water is poured into one of them it will flow through the pipe to the other container." [72]

Eventually in the third rule of this simulation we look at how much excess water a cell has and move it upwards if required. We do not need to explicitly track pressure to make the water level equalize in communicating vessels.

---

3. `https://w-shadow.com/blog/2009/09/01/simple-fluid-simulation/`
4. `http://www.jgallant.com/2d-liquid-simulator-with-cellular-automaton-in-unity/`

Dwarf Fortress An alternative approach to simulating pressure has been implemented by Tarn Adams in the game Dwarf Fortress. The process has been described in an interview [79] for website Gamasutra.

The first two rules are again as follows: water falls down if it can, then spreads to the sides if it can. However, the pressure is handled differently and it is not a local operation.

When water cell cannot flow to the sides or fall down anymore (in Figure 4.2, on the left), it checks the cell below:

- if it is marked as "static", it stops
- if it is not marked as "static", it will start doing a flood-fill search downwards, through water cells which are full, never rising higher than its own level until it finds an open slot
  - if it finds one, it teleports there (in Figure 4.2, on the right)
  - otherwise it marks the cells below as "static" to avoid a recalculation later and to keep the simulation fast
- if the situation ever changes (like water is removed) it can just remove the static flags

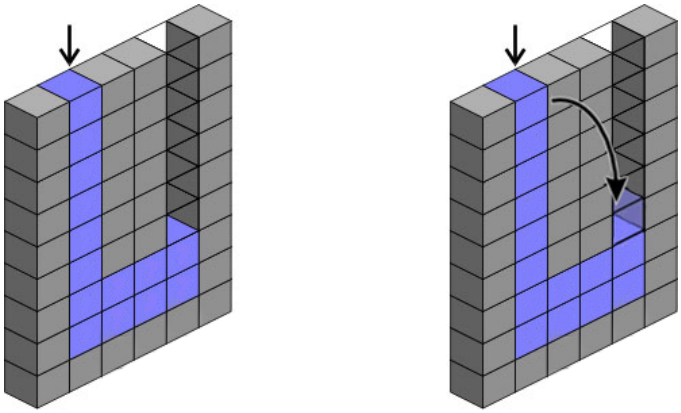

Figure 4.2: Teleport step in Dwarf Fortress, taken from [79].

Other examples In a game called Vox, the rules are the same as in TerrainEngine. In games DwarfCorp or Qubotron, the rules are the same but in the second step, the water from a cell which cannot fall down anymore, is distributed in random amounts to all its neighbours instead of equal amounts to neighbours which contain less volume than the current cell.

In Terraria, the water moves according to same set of rules as described in Janis Elsts's approach but does not support pressure.

Several more existing solutions of water simulation using CA can be found. Even though they can be visually different, it is because they handle rendering of the simulated water in distinct ways. After a more thorough observation, however, it can be noticed that the rules and the mechanics behind the simulation are usually just variations or modifications of the main concepts described above.

4.1.2 Drawbacks

The main problem when simulating water using CA comes from the very nature of CA. By having local operations applied only to the currently processed cell or considering at most the neighbourhood of that cell, some unnatural artifacts start to occur.

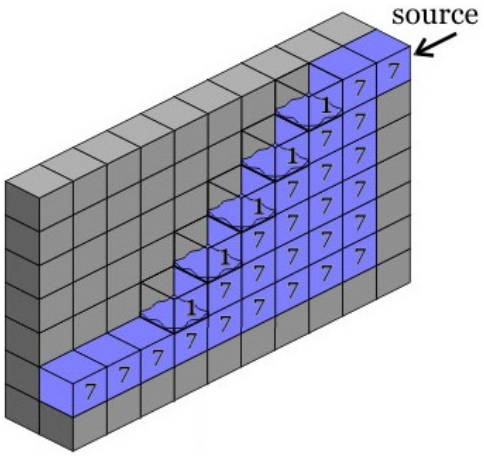

Figure 4.3: "Stairs" effect, taken from [79].

"The main problem is at waterfalls. The water falls on to the lake or river below, and just starts clumping, forming a pyramid. This is because of the local behaviour. It can't go down, so it goes over. So the water is flowing out, but too slowly." [79]

Because of this "stairs" effect, Figure 4.3, the water takes too long to spread and settle, and appears too viscous.

Out of all the concepts described, this has been battled to some extent only with the "teleport" approach in Dwarf Fortress, which already diverges from the usual usage of only local operations and provides the simulation with some kind of context.

Another issue is the pressure. When simulating water by CA in games, one of the two mechanisms described above is used for pressure, or the most common approach is to ignore it completely, as in the already mentioned U-bend shaped pipe problem. These kinds of simulations are called *falling-sand simulations*.

Or, while still ignoring the pressure, the game provides some other mechanism to push water upwards at least in a limited fashion. For example, by placing some "water pump" item at a fixed position in the game, which transfers water from one position to another, therefore even from lower to higher positions in the world.

4.2 Rules

My implementation is made up of 4 simple local rules, which are applied every iteration to each cell in the grid, as long as the simulation is running. A high-level algorithm of the simulation can be seen in Listing 4.1.

Listing 4.1: High-level fluid simulation algorithm

```
while (simulation is running) do
{
    foreach (cell in grid)
    {
        if (cell and its neighbours are settled)
            continue;

        Teleport;
        FlowUp;
        FlowDown;
        FlowSideways;

        if (fluid in the cell has not changed)
            DecreaseSettleCounter;

        if (cell has just settled)
            AddToFluidComponent;
    }

    UpdateGrid;
}
```

For each cell we consider the aforementioned von Neumann neighbourhood (page 24), i.e. only the immediately adjacent neighbouring cells (top, bottom, forward, backward, right, left). Also in each cell we track a few properties which define its current state and based on which we can calculate their next state. These properties include: the current *solid* (terrain) or *fluid* (water) volume in the cell, *world position* of this cell, several *boolean flags* needed for simulation, pointers to

all 6 of the cell *neighbours* and a *settle counter*, and a pointer to a *fluid component*, both of which will be explained later.

Update grid To avoid various side effects, such as oscillations or water spreading at different speeds depending on the order in which we update the cells, which would appear if we were writing directly to the grid from which we are reading the cells' properties, two grids are used. One, which contains the current generation of cells in CA and is used in a read-only fashion, and one for writing new values, which contains the next generation of cells. In the end of each iteration, the freshly calculated values from the writing grid are copied to the read-only grid so that in the next iteration we have the latest values available for reading.

This is also necessary if we want to run the simulation in parallel in multiple threads so that they all have the same data accessible for reading throughout an iteration. For the same purpose we always calculate inflow (*in*) and outflow (*out*) of fluid for a cell and modify the state only of the cell currently being processed, so that atomicity is assured and two different threads will never write to the same cell.

The new state of a cell after applying a rule is then calculated:

$$f^{t+1}_{x,y,z} = f^{t}_{x,y,z} + (in^{t}_{x,y,z} - out^{t}_{x,y,z}) \tag{4.1}$$

where f stands for the current fluid volume of a specific cell, the variable followed by subscript $_{x,y,z}$ denotes a property of the cell at position x,y,z in the 3D grid, t stands for the state of a cell in a current iteration, *in* and *out* mean how much fluid volume is flowing into this cell from its neighbours, and respectively, out of this cell to its neighbours, respectively.

In the following paragraphs a *clamp* function is used (widely recognised function in computer graphics community, which constrains a value to a given interval):

$$clamp(x, min, max) = \begin{cases} min & \text{if } x < min \\ max & \text{if } x > max \\ x & \text{otherwise} \end{cases} \tag{4.2}$$

Teleport This rule in my cellular automata will be just mentioned here for completeness but it will be explained along with the required apparatus in the next sections.

Flow Up The purpose of this rule is to take care of the excess fluid which can sometimes be found in a cell. For instance, if user adds fluid to already filled cells or raises some terrain underwater which in exchange has to push upwards the water laying on top of it. It corrects the state of such cell for the following simulation steps by pushing the excess fluid upwards. First we define the current volume of a cell:

$$curr_{x,y,z} = f_{x,y,z} + s_{x,y,z} \tag{4.3}$$

where s stands for the current solid volume in a cell (terrain), and also we define what is considered an excess fluid in a cell:

$$excess_{x,y,z} = \begin{cases} curr_{x,y,z} - MAX & \text{if } curr_{x,y,z} > MAX \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

where MAX is the maximum allowed volume (of both solid and fluid components combined) in a single cell – in my implementation this is set to 1.

Then we are able to calculate *in* and *out* for this step:

$$in_{x,y,z} = excess_{x,y-1,z} \tag{4.5}$$

$$out_{x,y,z} = excess_{x,y,z} \tag{4.6}$$

We simply push all the excess fluid upwards. In this context one aspect is worth noting. When the water is in a cave or covered by a terrain roof and the excess water pushed upwards hits the solid ceiling, to maintain simplicity of the algorithm, it will not look for the closest air cell but will be absorbed by the ceiling instead. However, if it gets above the water surface to an air cell, it will continue in a simulation normally.

Flow Down This is the basic rule which can be found across all the CA fluid simulations in games. Basically we transfer as much fluid as possible from the current cell to its bottom neighbour (see Figure 4.4). To compute a new state for a cell we have to find out how much free volume does a cell currently have:

$$free_{x,y,z} = \begin{cases} MAX - curr_{x,y,z} & \text{if } curr_{x,y,z} < MAX \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

Then we are ready to calculate the inflow and outflow needed for determining a new state:

$$in_{x,y,z} = clamp(f_{x,y+1,z}, 0, free_{x,y,z}) \tag{4.8}$$

$$out_{x,y,z} = clamp(f_{x,y,z}, 0, free_{x,y-1,z}) \tag{4.9}$$

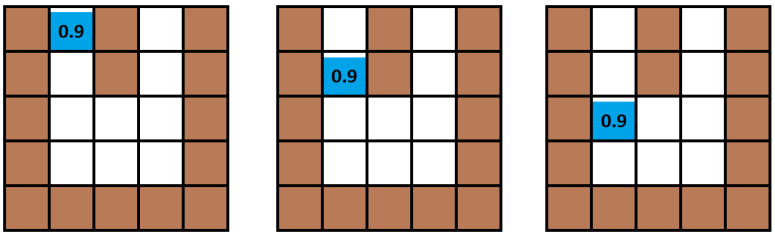

Figure 4.4: Fluid volume transfer according to Flow Down rule

Flow Sideways If transfer downwards fails, i.e. the bottom neighbour is not able to accept any more fluid from this cell, we apply this rule to spread the fluid sideways.

First we define a function *diff*. The function takes two cell positions, computes difference in their volume, and returns the proportional part of fluid volume which can be transferred from the first cell to the second if the second holds less volume than the first, otherwise returns 0.

$$diff((a,b,c),(d,e,f)) = clamp(\frac{curr_{a,b,c} - curr_{d,e,f}}{|N|+1}, 0, \frac{f_{a,b,c}}{|N|+1}) \tag{4.10}$$

$$in_{x,y,z} = \sum_{n \in N} diff(n, (x,y,z)) \tag{4.11}$$

$$out_{x,y,z} = \sum_{n \in N} diff((x,y,z), n) \tag{4.12}$$

where $N = \{(x+1,y,z), (x-1,y,z), (x,y,z+1), (x,y,z-1)\}$.

We distribute the fluid equally among the current cell and its horizontal neighbours, but we take only from those neighbours which contain more volume than the current cell and give to those that contain less – to avoid oscillations and ensure that the water flows only downwards and eventually equalizes (as illustrated in Figure 4.5). For each neighbour we compute the volume difference between it and the current cell and then we exchange as much fluid as possible divided proportionally by the number of neighbours.

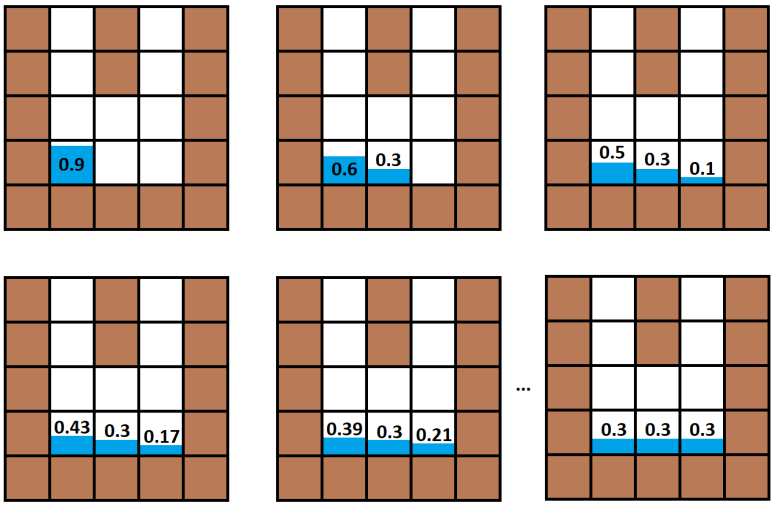

Figure 4.5: Fluid volume transfer according to Flow Sideways rule

4.3 Settling

After applying the rules, as a last thing we check whether the volume in a current cell has changed since the last iteration and if it has not, we decrease its *settle counter*. If the settle counter reaches 0 we mark this

cell with a *settled* flag, e.g. the water flow has settled. In the following iterations we can just skip processing of settled cells (deep underwater cells, terrain and air cells, settled water surface etc) to speed up the simulation.

Various events can "unsettle" the cells again resetting their settle counter, like adding or removing water in proximity of settled cells, terraforming, fluid coming from or to a neighbour, change in volume since the last iteration and so on.

4.4 Connected components

Simultaneously, while the fluid simulation is running and the cells are getting settled, there is another procedure taking place. This procedure iterates through settled cells of water, performs a flood fill search through these settled cells and dynamically constructs and maintains the connected components of water which exist in our game world.

Listing 4.2: High-level connected components algorithm

```
while (simulation is running) do
{
    foreach (existing connected component)
    {
        Cleanup;
        Grow;
        Merge;

        if (component is too small)
            Delete;
    }

    foreach (settled cell without a component)
    {
        Create;
    }

    wait for user defined time;
}
```

Procedure A high-level overview of this procedure is illustrated in Listing 4.2. At an user defined time interval all the components are updated:

- **Grow:** by trying to "grow", performing a flood search and reaching and adding new cells which became settled since the last update
- **Merge:** if a cell already belongs to another component and is reached while "growing", the two components are merged
- **Cleanup:** by removing cells which have become unsettled or do not contain fluid anymore
- **Delete:** if a component becomes too small, it gets removed
- **Create:** if there is still some settled fluid cell which does not belong to any component yet, a new component is created from it

Flood search We start with a settled cell, add it to a stack, then recursively pop a cell out of stack, mark it as "visited", add it to the current component and push its neighbours, which also contain fluid and are settled, and have not been visited yet to the stack.

Outlets Having introduced a concept of these connected fluid components in the game world, we now have some kind of "context" in our water simulation and we are no longer limited to local rules which "see" only as far as the cell's neighbourhood. We can extract some useful information from these components, which can help us implement more sophisticated features into our simulation.

One such information is a list of *outlets* that each component maintains. Outlets are generally the not yet full cells which are near the water surface, e.g. settled cells with the highest y world position or the ones having an air cell as their top neighbour, or unsettled cells which are near the component (their neighbour already belongs to one), and so on. This list is again dynamically maintained, and once the outlet fills up, it gets removed from the list.

Teleport Having these outlets, we can approximately track the current water level in each component (for example, the lowest y coordinate among its outlets). And this leads us to the explanation of the *teleport* rule in my CA simulation.

If during the simulation we detect that the cell currently being processed is just above the water surface of some component, we simply take all the cell's fluid volume and immediately spread it over the water surface, e.g. divide the volume by the number of the component's outlets and add it to all of them at once. This step represents the pressure solve in our simulation and also eliminates the "stairs" effect, which many of the simpler simulations described in the beginning of this chapter suffer from. The elimination is achieved by adding falling water immediately to the whole body of water making it look more realistic.

In Figure 4.6 we see the constructed component (yellow outline) and its outlets (red outline) and we can easily observe in the first row of pictures that the "stairs" effect is eliminated (compared to the Figure 4.5) and in the second row that once the full outlets are removed and new ones are found above the water surface, the illusion of pressure is achieved and the water added later raises on both ends of an U-bend.

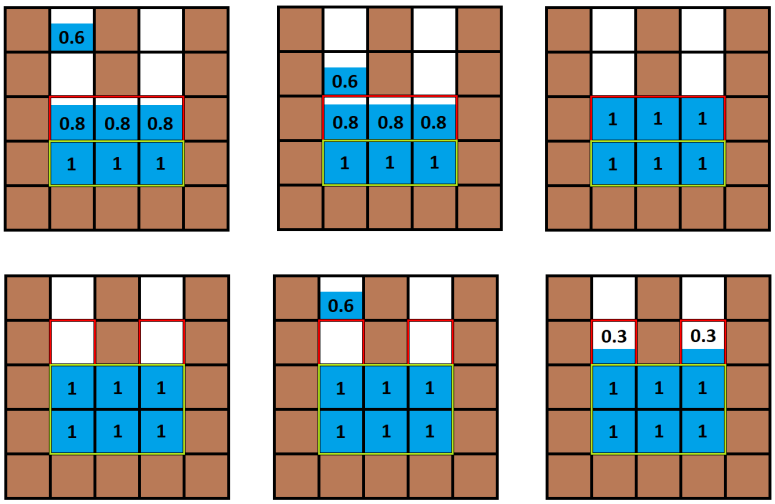

Figure 4.6: Fluid volume transfer according to Teleport rule

4.5 Viscosity

Viscosity is the property of the fluid which opposes the relative motion between the two surfaces of the fluid that are moving at different velocities. In simple terms, viscosity means the friction between the molecules of a fluid. For liquids, it corresponds to the informal concept of "thickness", for example honey has a higher viscosity than water.

As we mentioned in the beginning of the thesis, grid based techniques suffer from high numerical dissipation, which manifests as an unphysical viscosity - this makes it difficult to model low viscosity fluids like water and air. This is also the case with our simulation, even at the lowest possible viscosity, water appears too viscous and this is ultimately limited by the computational power available for the simulation while the game is running. The faster we run our simulation, the less viscous the water appears. Currently, this does not present an obstacle for our requirements and the visual style of the game.

Since we cannot directly influence the lowest viscosity of our water without purchasing a faster hardware or optimizing other systems in the game, we can at least model liquids of higher viscosities such as lava. In this way we can tell apart the different types of liquids in the game, not only by their visually distinct style but also by their behaviour.

In my implementation this has been done simply by adding a new property to the cell – viscosity. After applying the CA rules the final amount of water to be transferred into or from the current cell is scaled by the value stored in this property, therefore slowing the flow. The equation for calculating the new state of a cell then becomes:

$$f^{t+1}_{x,y,z} = f^{t}_{x,y,z} + (in^{t}_{x,y,z} - out^{t}_{x,y,z}) * (1 - v^{t}_{x,y,z}) \tag{4.13}$$

where v is the viscosity property of a cell and ranges from 0 (water) to the maximum of 1 (liquid will not flow).

Liquids of higher viscosities also take longer to "spread" horizontally. We take this into account by scaling the value by which we decrease the current cell's *settle counter* by its viscosity in the same fashion. This gives the liquids which flow slower more time to settle. Also for liquids of a very high viscosity, e.g. lava, we can turn off the *Teleport* rule so the "stairs" effect is preserved. Such a liquid then starts clumping on top of itself, thus making it look more realistic.

5 Integration

As mentioned in the third chapter, the simulation was implemented to be compatible with the requirements specified on page 21. However, in order to integrate the solution to Ylands, further modifications and optimizations had to be made to the original implementation (and to fully succeed, more are still yet to be done). In this chapter we describe the most significant ones that were already implemented by the time of writing this thesis.

5.1 Modifications

Blocks, chunks, voxels Up to this point my simulation run on a single 3D grid of voxels. For a potential integration it was of utmost importance to adapt it to the Ylands's world representation, i.e. world grid subdivided into blocks, chunks and voxels. After overcoming few initial implementation difficulties, it has brought several benefits over the original representation.

First we need to recall that the world is always loaded in a 5x3x5 locus of blocks centered around the player, which stands for approximately 157 millions of voxels. To iterate voxel-by-voxel over this loaded part of the world, while the simulation is running, only recognizing settled or unsettled voxels and skipping the settled ones is simply not feasible in real time.

Now that the world is divided into blocks and chunks of voxels, we can bring this "settling" mechanism to those chunks and blocks as well. For example, we are able to identify which chunks contain no liquids at all (air, terrain) or have all their voxels in a settled state (regions underwater, lakes, seas) and skip these chunks in a simulation altogether, which improves the performance to a big extent.

Parallelization and rendering Having designed the simulation with the eventual integration in mind throughout the whole process and by introducing this subdivision afterwards, only a small additional modifications in the code were required for running the simulation in a background thread and parallelizing it per block by the already existing custom job system in Ylands.

Rendering was almost straightforwardly adapted to Ylands's own rendering system, which also uses the Marching Cubes algorithm, along with the visual effects too. The benefits gained from the grid subdivision allowed us to only rebuild meshes of "dirty" chunks every time, instead of always rebuilding the water mesh for the whole grid, as was the case in my original simulation. "Dirty" chunks are the ones where something changed since the last mesh build, such as unsettled chunks which contain some fluid.

5.2 Optimizations

Spatial locality Since the simulation will run on CPU, we noticed an opportunity for a big optimization – to make better use of CPU cache memories by improving the spatial locality and therefore achieving faster performance overall.

Spatial locality refers to the use of data elements within relatively close storage locations. If a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future [80]. In this case it is common to attempt to guess the size and shape of the area around the current reference for which it is worthwhile to prepare faster access. Sequential locality, a special case of spatial locality, occurs when data elements are arranged and accessed linearly, such as, traversing the elements in a one-dimensional array.

In Ylands, chunks in blocks as well as voxels in chunks are already stored in such 1D arrays, even though they represent a 3D grid, in order to expose better sequential locality. Such arrays are called linearized arrays. The linearization in Ylands happens first over x then y and finally over z coordinate.

An example how an index to such 1D array for a voxel at 3D coordinate (x,y,z) within a chunk is calculated:

$$voxelId(x, y, z) = x * chunkSizeZ * chunkSizeY + y * chunkSizeZ + z$$

where *chunkSizeX*, *chunkSizeY* and *chunkSizeZ* are the dimensions of a chunk along the x,y and z axis respectively.

There are two kinds of types in C# programming language: reference types and value types. Variables of reference types (e.g. class,

object, string, list, etc.) store references to their data, while variables of value types (e.g. int, char, bool, float, struct, etc.) directly contain their data.

In my original code, a single cell was represented by a *class*[1] (reference type). The proposed optimization was to represent it as a *struct*[2] (value type).

The reference types are allocated on the heap, whereas value types are allocated on the stack, which makes value types much cheaper to allocate and deallocate. Also, arrays of reference types are allocated out-of-line, meaning the array elements are just pointers to instances of the reference type which reside somewhere else in the memory. Value type arrays are allocated inline, meaning that the array elements are the actual contents of the value type. Therefore, allocations and deallocations of arrays of value type are also much cheaper than of reference type. Moreover, in a majority of cases, value type arrays exhibit much better spatial locality [81].

Table 5.1: Memory access latency, taken from [82].

Type	Info	Latency
L1 cache	Usually built onto the CPU chip itself.	0.5 ns
L2 cache	Memory built on a separate chip.	7 ns
Main memory	The main module memory, i.e. RAM.	100 ns

When we access an element in an array of value types, the CPU loads a "chunk" of memory around that element into its caches (e.g. L1, L2). This means that when we access the next element in the array, it might already be in one of those caches (*cache-hit*). This can be an order of magnitude faster than reading from RAM, as we can see in the Table 5.1, but it varies by the type of CPU (the values are approximate and represent ratios between the orders of magnitude rather than the exact latency). On the other hand, when we loop over an array of reference types and access their instances, we are essentially visiting

1. A class is a construct that enables programmer to create their own custom types by grouping together variables of other types (data), methods and events (behaviour).
2. A struct type is a value type that is typically used to encapsulate small groups of related variables.

random locations in the RAM and the probability that the next element is going to be in one of the CPU's caches is very low, resulting in a *cache-miss*.

Memory footprint Another optimization worth mentioning that was applied to the original work, is minimizing the memory footprint per voxel. The future plans for Ylands include porting it to other devices, some of them having only a small amount of RAM. Therefore, we need to carefully identify and store only the absolutely necessary data in each cell. This is more important for this per-voxel data than per-chunk or per-block, because the memory consumption grows in a rather quick fashion with each additional variable stored.

For example, let us say we want to keep track of one more numeric property in every cell. In C# a number variable of a *float* type takes 4 bytes of memory[3]. At first this does not seem as much of a concern. But, to put it into perspective, during the gameplay there are always 75 blocks loaded in the main memory of the user's computer (i.e. 157 millions of voxels). Therefore, that one additional number stored per-voxel represents roughly 600 MB increase in memory consumed by the game during the gameplay.

As detailed in Table 5.2, we have been able to downscale the memory footprint of a single cell from the original (at least) 55 bytes to just 8 bytes. We identified that some properties, such as position of the cell in the world coordinates and the pointers to cell's neighbours, can be completely removed because they do not necessarily have to be stored. Instead, they can be quickly calculated if we need them. The only "edge cases", which require slightly more computational power, appear when it is needed to access a neighbouring cell of the current cell which resides in a different chunk or even a block.

One thing to be noted here is that the final size of a single cell in memory was targeted at 8 bytes on purpose. As we mentioned before, to speed up the access to data in the main memory of computer,

3. `https://docs.microsoft.com/en-us/dotnet/api/system.single?view=netframework-4.7.2`

4. Each reference is 4 bytes (on a 32-bit system) or 8 bytes (on a 64-bit system). To approximate the size of an array of references, the array length is multiplied by the reference size. There are also a few extra bytes for internal variables for the array class and memory management.

Table 5.2: Memory consumed by a single voxel.

Variable	Old type	Memory	New type	Memory
solid	float	4 bytes	byte	1 byte
fluid	float	4 bytes	byte	1 byte
viscosity	float	4 bytes	byte	1 byte
settleCounter	float	4 bytes	short	2 byte
settled	bool	1 byte	bool	1 byte
falling	bool	1 byte	bool	1 byte
visited	bool	1 byte	–	–
neighbour[6]	pointer[6]	> 24 bytes[4]	–	–
worldPosition	float[3]	12 bytes	–	–
valid	–	–	bool	1 byte
Total	class	**> 55 bytes**	struct	**8 bytes**

the CPU transfers "chunks" of memory around the element into its caches, from where they can be read in an order of magnitude faster. These "chunks" are called *cache lines* and are typically between 16 and 128 bytes in size (always power of 2) [83]. This is how we can gain additional memory speedup in our applications – if the individual elements of our data fit entirely into the cache line, e.g. their size is a multiple of the cache line size. Consequently, we can read the whole element quickly from the cache and do not have to fetch the rest of it from main memory in the next line.

Equalize outlets There is one more modification triggered by the storage of solid and fluid volume compounds of a cell as *byte* variables. Previously, each cell could contain in a volume ranging from 0 (empty) to 1 (full), including all the floating point values in between. Anything above 1 was recognized as the excess fluid. After transition to *byte*, we only have 127 integer values (from 0 to 127) to represent the volume. Even though a variable of the type *byte* can incorporate 255 discrete values, we still needed to somehow recognize the excess fluid without creating another variable. That is why a cell with volume 127

is considered full and any volume above that is considered as excess fluid.

This change has broken the *Teleport* step in our simulation. Originally, we would take the volume of a cell, which is just above the water surface of a fluid component, and divide it by the number of component's outlets. We cannot do it anymore because the cell can have at most 127 units of volume; as soon as the component has more than 127 outlets there are some outlets not getting a share of this fluid. We could distribute it only to some outlets (in a sequential or random fashion) and wait for another cell to approach the surface and distribute its fluid to the rest of them, but this would eventually create the undesired "stairs" effect again. Instead, we looked at this problem from another point of view. The *Teleport* step was removed and the *Equalize* method was added to the procedure which performs the maintenance of the fluid connected components (Listing 4.2).

While the simulation is running and the components are being constructed, the fluid is slowly filling up the component's outlets in an ordinary fashion (as if without the *Teleport* step). However, in the proposed *Equalize* method, we calculate what is the current average amount of fluid in all of the outlets and remove the fluid from the outlets which have more than the average, and we add to the ones that have less, eventually equalizing the water level in all outlets to that average amount. If we do this fast enough, we achieve both the illusion of the falling water being immediately added to the whole body of water as well as the illusion of pressure, just as we had previously achieved using the *Teleport* rule.

6 Results

The result of this thesis is a standalone application demonstrating the final implementation, written in C# programming language and using the Unity engine. The development of this application was an incremental process and 3 different methods for simulating the water were implemented. These methods are described below and afterwards measured and compared in terms of computation speed and functionality.

1. Basic This prototype was the result of the first stage of development. A grid of cells was created and the properties defined for each cell. The workings of cellular automaton were explored and different neighbourhoods tried out. The simulation was essentially made up only of the Flow Down and the Flow Sideways rule.

2. Pressure At this stage we tried to add support for pressure into the simulation. The attempted approach was inspired by previously mentioned Janis Elsts's simulation (page 27) and ported to 3D.

3. Components In the last prototype, a different approach was chosen, based on a technique used in Dwarf Fortress (on page 28). The "teleporting" and connected components, representing individual bodies of water, were introduced.

6.1 Benchmarks

The benchmarking was done on two computers, referred to as PC 1 and PC 2. The technical specifications of these computers are shown in Table 6.1. The benchmarks were recorded using the FRAPS software[1]. The resulting CSV files are included among the attachments of this thesis (page 69).

The resulting average FPS[2] gathered during benchmarking of the 3 prototypes are listed in Table 6.2. The simulations were run multiple

1. `http://www.fraps.com/`
2. number of frames generated per second

Table 6.1: Computers used for benchmarking.

	PC 1	**PC 2**
CPU	Intel Core i7-7700	Intel Core i5-6500
GPU	NVIDIA GeForce GTX 1070	AMD Radeon RX 470
RAM	16 GB	16 GB
OS	MS Windows 10 64-bit	MS Windows 10 64-bit

times for 20 seconds and the water was being added to and subtracted from the "world" repeatedly in the meantime. The "world" was a grid of 16x16x16 voxels in size and the rendering was done in resolution 1920x1080 using Marching Cubes algorithm on CPU. Graphics quality setting in Unity was set to Fastest.

Table 6.2: The average FPS of the 3 prototypes.

Prototype	**FPS on PC 1**	**FPS on PC 2**
1. Basic	205	163
2. Pressure	212	148
3. Components	216	163

However, as Kouřil pointed out in his work [5], when analyzing the performance and "smoothness" of the gameplay, the FPS metric is not conclusive and the application can still stutter even with stable FPS[3]. Therefore, the frame times[4] were also collected and can be seen in Figure 6.1.

We can see that these prototypes do not differ much performance-wise, however, there are big differences if we compare them in terms of functionality.

3. For example, if 60 frames take each 16ms to render, the result is 60 FPS. But if one frame takes 900ms and 59 frames take 1ms, the result is 60 FPS again, but the gameplay does not feel smooth.
4. the time required to render each frame

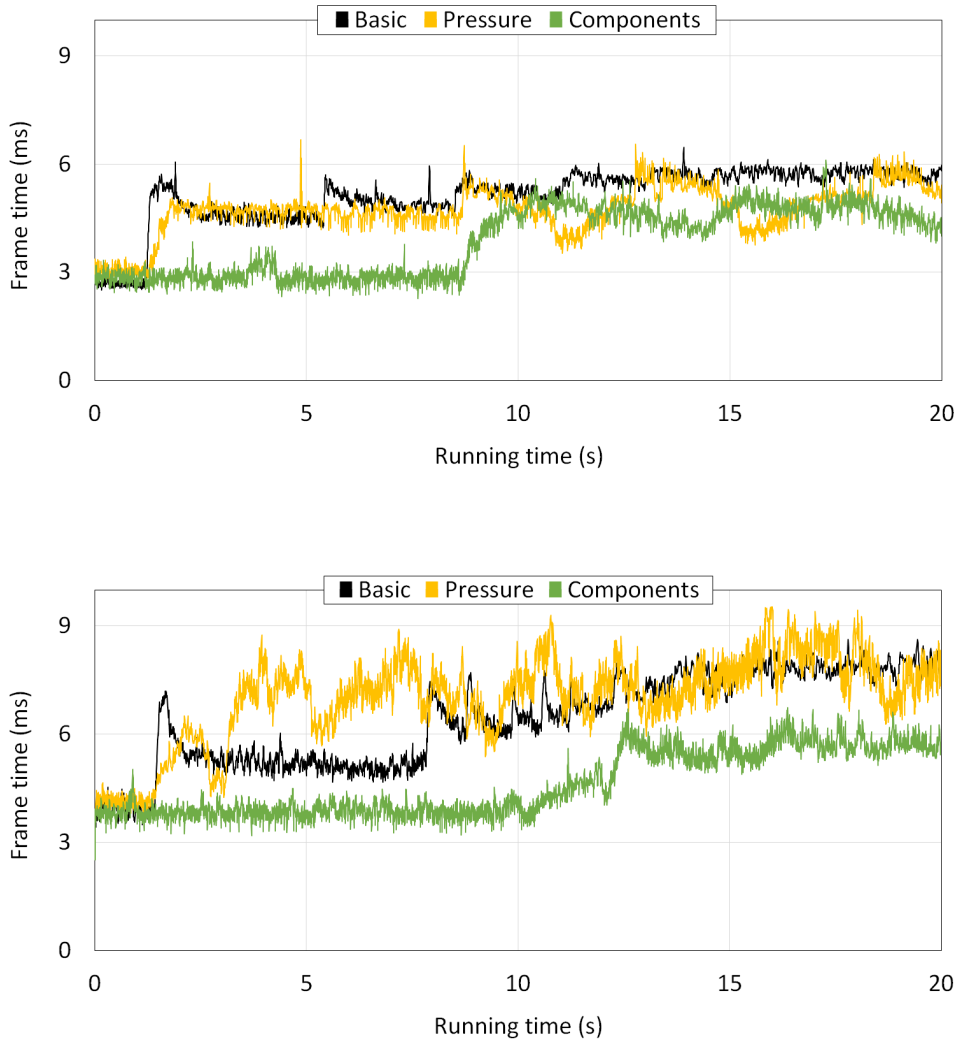

Figure 6.1: The frame times of the 3 prototypes measured on PC 1 (top) and PC 2 (bottom).

6.2 Comparison

In the *Basic* prototype, the water only flows downwards and horizontally, the issues of pressure and the "stairs" effect are not addressed at all.

The pressure was addressed in the next approach. The code written for the *Pressure* prototype is clear and simple, which makes it less likely to contain "bugs". However the "stairs" effect still persists and some scalability concerns were raised:

- The variables used to express the current water volume in a cell are of type *float* and we will not be able to convert them to *byte* to save memory. This is because the compression compound in each cell raises with the growing water level in the cells above the current cell - eventually reaching the maximum *byte* value 255, an aspect which becomes a potential limitation of this approach.

- Also, the compression compound has to be recalculated all the way to the bottom of the current body of water for the pressure to work correctly, each time there is some water added or removed from that body. This way we cannot employ the "settling" mechanism introduced earlier, and skip the settled cells while the simulation is running in order to save some processing power. This means that simulation will slow down rapidly as the bodies of water will grow.

The last prototype, *Components*, solves both the pressure and the "stairs" effect. However, the code here becomes more complicated, since we now perform non-local operations (i.e. Teleport) and have the connected components of fluid to maintain.

Since the prototypes were similar in the way they performed and from the points of view of functionality and scalability, the *Components* prototype proved to be the most promising one. It was decided that this prototype will be further improved and eventually integrated to the game, while the work on others will be discontinued.

6.3 Improvements

The Figure 6.2 shows two improvements worth mentioning and the numerical measurements of their impact. World grid size was again 16x16x16. The simulations were running for 60 seconds while the following actions were performed (in this order): 10 seconds of looking around in the scene with the camera, 20 seconds of adding and subtracting the water, 10 seconds of terraforming the world, which already contains water, and 20 more seconds of adding and subtracting water from the terraformed world.

As the first improvement, the Marching Cubes rendering algorithm was moved from CPU to GPU, step which brought a huge performance gain. The other improvement consisted in the implementation of all the optimizations mentioned in the Integration chapter (starting from page 41) as well as subdividing the world grid into blocks, chunks and voxels.

We can observe that the subdivision introduced some overhead. Furthermore, the data gives the impression that the version with optimizations is slower than the one without them. The variety of actions performed explains the spikes in the frame rates, e.g. we can see that terraforming the world with water unsettles it as it pushes it around, triggering more intense computations. Sometimes after performing an action, we waited for a small amount of time for water to settle down, which also explains the sudden jumps in the graphs.

Table 6.3: The average FPS of *Components* prototype improvements.

Version	**FPS on PC 1**	**FPS on PC 2**
CPU rendering	216	163
GPU rendering	1371	2543
GPU rendering + optimizations	1178	1267

Scalability is where we can finally see the advantage of these modifications. In the next graph, we perform the same actions again, but we increase the world grid size from original 16x16x16 to 32x32x32 and even 48x48x48. In the former we start to notice a slight advantage in frame times and in the latter the scaling superiority of optimized version is already obvious.

Table 6.4: The average FPS when scaling across different world sizes.

Grid size	PC 1	PC 2	PC 1	PC 2
16x16x16	1371	2543	1178	1267
32x32x32	997	958	560	576
48x48x48	26	3	502	433
	not optimized		**optimized**	

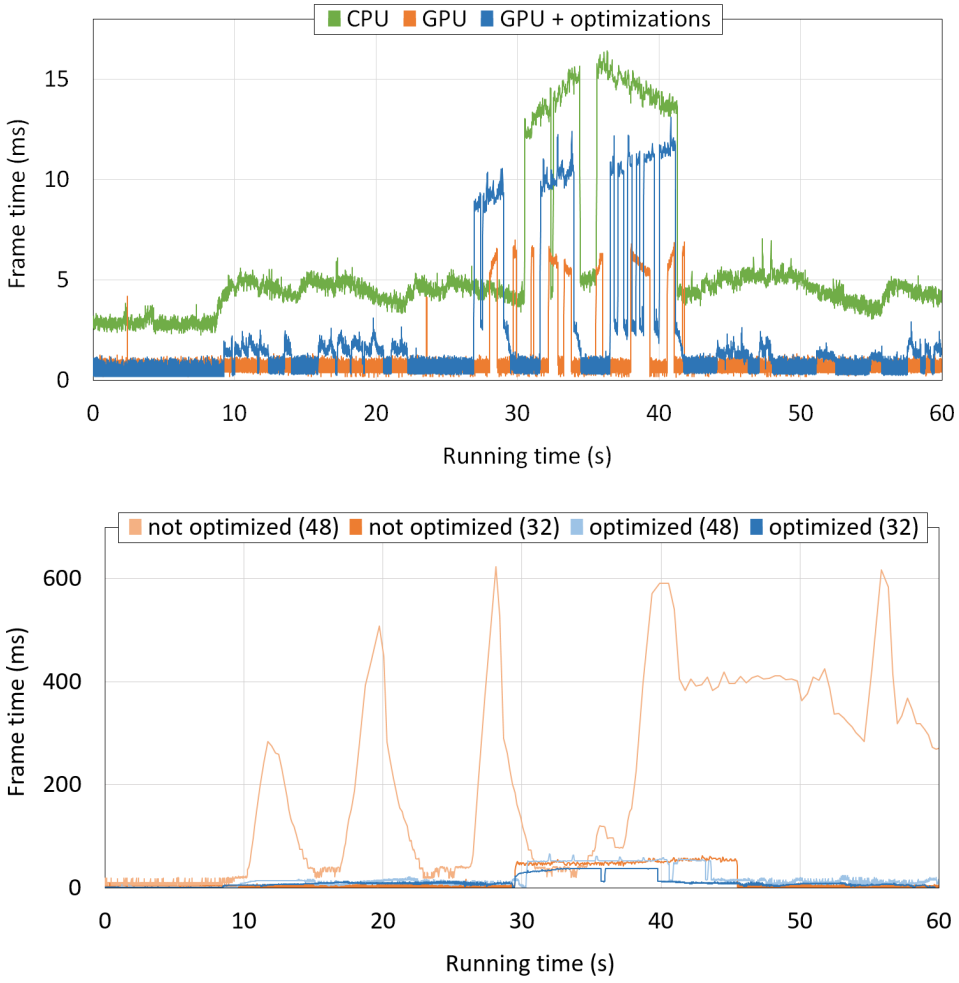

Figure 6.2: The frame times of *Components* prototype improvements (top) and scaling across different world sizes (bottom) run on PC 1.

The graphs of improvements mentioned in this section, as it was run on PC 2, can be found in the Appendix B of this thesis (page 71)[5].

6.4 Standalone application

For the final practical output of this thesis we took the *Components* prototype, added support for different viscosities, moved the rendering to GPU and introduced the modifications and optimizations required for integration to Ylands.

A simple user interface (UI) and a way to modify the terrain was also added to the prototype, so that we can demonstrate the resulting simulation through a standalone application. A screenshot is diplayed in the Figure 6.3 and more can be found in the Appendix C of this thesis (page 73). Here we need to stress that the UI, rendering and terrain modification is handled in a very simple fashion as these were not the goals of this thesis and are meant to be replaced by already existing alternatives in Ylands, once the integration is completed.

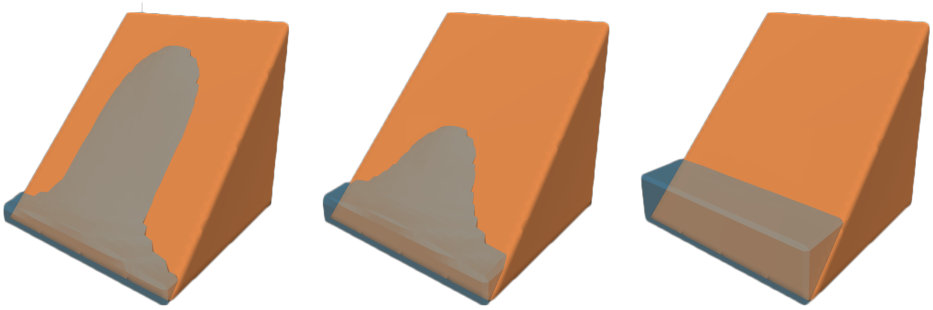

Figure 6.3: Water flowing dowhill in the application.

5. At the biggest world size in the unoptimized version we do not observe those occasional jumps back to small frame times (i.e. water has settled) on PC 2 as we did on PC 1. This can be explained either by the hardware on PC 1 being more powerful and therefore making the water settle faster. Or simply by unprecise timing of pauses after an action, since it became increasingly difficult to time the actions exactly at such high frame rates. This is also the case at approximately the 40 second mark, where we can observe the onset of terraforming actions 10 seconds later than planned.

7 Conclusion

The goal of this work was to investigate and describe liquid simulation techniques which can be used in real-time applications, such as games.

Specifications for a liquid simulation were collected from a consultant at Bohemia Interactive for their game Ylands. The practical part of this thesis consisted in the implementation and comparison of three different approaches to simulate liquids in real time using cellular automata. According to the results of the comparison and specified technical and functional requirements, the most suitable technique was chosen and further improved.

As a result, a standalone application which demonstrates the selected liquid simulation technique was created. The application allows users to freely interact with the liquids and the terrain; and the whole scene is being rendered in real time. In the implementation we have also focused on keeping the memory consumption low and maintaining high frame rates on a consumer hardware, regardless of the varying size of the scene.

This application can be used as a learning tool to better understand one of the many uses of cellular automata – water simulation. Furthermore, the eventual integration of this simulation to an existing game still remains among its main ambitions. A partial integration has already been done by the time of writing this thesis. The highly "work-in-progress" state in which it currently is, can be seen in a short video found with the other electronic attachments of the thesis (page 69).

7.1 Future work

This technique still has a lot of space for improvement. Below are proposed some suggestions for potential future work to make this simulation even more suitable for games.

Liquid mixing The application already supports liquid viscosity, which allows us to have liquids flowing at different speeds. However, right now the different kinds of liquids do not mix together.

One way to address this issue would be to implement a customizable behaviour (i.e. new rules) for situations when two different liquids meet. We could determine which specific cells met, look up a rule for this situation and resolve it by calculating a new state for those cells. For example, if lava met water, a part of water volume could evaporate from the concerned cells and a part could get converted into terrain.

An additional way to enhance the mixing behaviour would be to introduce one more liquid property – density, and adjust the existing rules to take it into account, e.g. denser liquids (water) would fall through the less dense ones (oil) and push them upwards.

Floating objects Another feature which could be added to the simulation would be for objects, such as rigid bodies, to be able to float on top of the water surface. In a basic manner this can be achieved simply by detecting the interface between water and air (e.g. from component water level and its *outlets*) and introducing a property in liquid cells where we would store and update the current flow direction. Then we can let the objects float on the water surface and move them according to the flow directions of the neighbouring cells.

Connected components representation Examining the code behind the simulation it becomes clear that one of its performance limitations is in the way how the connected components are represented. Currently, a component is simply represented by a set of all the cells it contains and through this set we iterate each update. This will most certainly slow down the simulation and take up increasingly more memory as the fluid component grows. A more efficient way would be to define and store only the boundaries of a component, for example, with a bounding box concept.

Once we are able to efficiently represent these large bodies of water, we can start keeping track of additional properties and recognize different types of components. For instance, we can have an "ocean" type and take it into account in our main simulation rules, such as: if there is an ocean below the current cell, it is going to absorb all the current cell's water volume; or the ocean can act as an infinite source of water, giving volume to all the empty cells nearby until they are at the same level.

Low level optimizations Many future improvements can also be in the form of low level optimizations. Since we are expressing volumes in cells as variables of type *byte*, we can make use of bitwise operations that programming languages provide which would lead to a faster performance.

As we have discussed in the Integration chapter (page 41), we have some variables of a *bool* type stored per-voxel. In most programming languages, a *bool* value takes up 1 or even 4 bytes of memory although it only stores a value (0 or 1) which can be expressed by a single bit. This is the case due to various reasons, such as memory padding or because the CPUs have their instruction sets optimized to operate on, and move around, whole bytes. However, if we wanted to further minimize the memory footprint of a single cell, we could group all *bool* properties in a cell or even a whole chunk into a more memory efficient data structures, such as BitArray or BitVector32 in C# where each *bool* takes up only 1 bit of memory.

We have written earlier that in the current implementation, chunks in blocks as well as voxels in chunks are stored in 1D linearized arrays. But to connect it up, each block stores a 2D array, where the index in the first dimension points to a chunk, and in the second – to a voxel. We have implemented it this way, the same way as it is in Ylands, in order to simplify the integration. However, a faster way would be to store all the voxels in 1D array directly at the block level in order to exploit better spatial locality.

Bibliography

1. BRIDSON, Robert. *Fluid Simulation for Computer Graphics*. 1st ed. Wellesley, Massachusetts: A K Peters, Ltd., 2008. ISBN 978-1-56881-326-4.
2. *Millennium Prize Problems — Navier–Stokes Equation* [online]. Clay Mathematics Institute [visited on 2018-03-11]. Available from: `http://www.claymath.org/millennium-problems/navier%E2%80%93stokes-equation`.
3. FEFFERMAN, Charles L. *Existence and smoothness of the Navier–Stokes equation* [online]. Clay Mathematics Institute [visited on 2018-03-11]. Available from: `http://www.claymath.org/sites/default/files/navierstokes.pdf`.
4. GOURLAY, Dr. Michael J. *Fluid Simulation for Video Games (part 1)* [online]. Intel, 2012 [visited on 2018-03-11]. Available from: `https://software.intel.com/en-us/articles/fluid-simulation-for-video-games-part-1`.
5. KOUŘIL, Pavel. *Underwater Excavation Serious Game*. 2017. Available also from: `https://is.muni.cz/th/qrfzw/`. Master's thesis. Masarykova univerzita, Fakulta informatiky, Brno. Supervised by Fotios LIAROKAPIS.
6. FOSTER, Nick; METAXAS, Dimitri. Realistic animation of liquids. *Graphical models and image processing*. 1996, vol. 58, no. 5, pp. 471–483.
7. STAM, Jos. Stable fluids. In: *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. 1999, pp. 121–128.
8. PREMOŽE, Simon; TASDIZEN, Tolga; BIGLER, James; LEFOHN, Aaron; WHITAKER, Ross T. Particle-Based Simulation of Fluids. *Computer Graphics Forum*. 2003, vol. 22, no. 3, pp. 401–410. Available from DOI: `10.1111/1467-8659.00687`.
9. FOSTER, Nick; FEDKIW, Ronald. Practical animation of liquids. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 2001, pp. 23–30.

10. HARLOW, Francis H; WELCH, J Eddie. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *The physics of fluids*. 1965, vol. 8, no. 12, pp. 2182–2189.

11. CHENTANEZ, Nuttapong; MÜLLER, Matthias. Real-time Eulerian Water Simulation Using a Restricted Tall Cell Grid. *ACM Trans. Graph.* 2011, vol. 30, no. 4, pp. 82:1–82:10. ISSN 0730-0301. Available from DOI: 10.1145/2010324.1964977.

12. WEISKOPF, Daniel. *GPU-Based Interactive Visualization Techniques (Mathematics and Visualization)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN 3540332626.

13. THETAWAVE [online]. 2006 [visited on 2018-03-25]. Available from: https://upload.wikimedia.org/wikipedia/commons/e/ec/Volume_ray_casting.png.

14. LORENSEN, William E.; CLINE, Harvey E. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *SIGGRAPH Comput. Graph.* 1987, vol. 21, no. 4, pp. 163–169. ISSN 0097-8930. Available from DOI: 10.1145/37402.37422.

15. JMTRIVIAL. *Marching cube cases* [online]. 2006 [visited on 2018-03-25]. Available from: https://upload.wikimedia.org/wikipedia/commons/archive/a/a7/20080131121020%21MarchingCubes.svg.

16. BRALEY, Colin; SANDU, Adrian. Fluid simulation for computer graphics: A tutorial in grid based and particle based methods. *Virginia Tech, Blacksburg*. 2010.

17. ZHU, Yongning; BRIDSON, Robert. Animating sand as a fluid. *ACM Transactions on Graphics (TOG)*. 2005, vol. 24, no. 3, pp. 965–972.

18. ENGLESSON, Dan; KILBY, Joakim; EK, Joel. *Fluid Simulation Using Implicit Particles* [online]. 2011 [visited on 2018-03-21]. Available from: http://www.danenglesson.com/images/portfolio/FLIP/%20rapport.pdf.

19. BATCHELOR, George Keith. *An introduction to fluid dynamics*. Cambridge, UK: Cambridge university press, 1973. ISBN 0-521-66396-2.

20. LAMB, Horace. *Hydrodynamics*. Cambridge university press, 1932.

21. YAEGER, Larry; UPSON, Craig; MYERS, Robert. Combining Physical and Visual Simulation–Creation of the Planet Jupiter for the Film "2010". *SIGGRAPH Comput. Graph.* 1986, vol. 20, no. 4, pp. 85–93. ISSN 0097-8930. Available from DOI: 10.1145/15886.15895.

22. GAMITO, Manuel Noronha; LOPES, Pedro Faria; GOMES, Mário Rui. Two-dimensional simulation of gaseous phenomena using vortex particles. In: TERZOPOULOS, Demetri; THALMANN, Daniel (eds.). *Computer Animation and Simulation '95*. Vienna: Springer Vienna, 1995, pp. 3–15. ISBN 978-3-7091-9435-5.

23. ANGELIDIS, Alexis; NEYRET, Fabrice. Simulation of smoke based on vortex filament primitives. In: *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 2005, pp. 87–96.

24. PARK, Sang Il; KIM, Myoung Jun. Vortex fluid for gaseous phenomena. In: *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 2005, pp. 261–270.

25. DESBRUN, Mathieu; CANI, Marie-Paule. Smoothed particles: A new paradigm for animating highly deformable bodies. In: *Computer Animation and Simulation'96*. Springer, 1996, pp. 61–76.

26. BECKER, Markus; TESCHNER, Matthias. Weakly compressible SPH for free surface flows. In: *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 2007, pp. 209–217.

27. SOLENTHALER, Barbara; PAJAROLA, Renato. Predictive-corrective incompressible SPH. In: *ACM transactions on graphics (TOG)*. 2009, vol. 28, p. 40. No. 3.

28. KOSHIZUKA, Seiichi; OKA, Yoshiaki. Moving-particle semi-implicit method for fragmentation of incompressible fluid. *Nuclear science and engineering*. 1996, vol. 123, no. 3, pp. 421–434.

29. BLINN, James F. A generalization of algebraic surface drawing. *ACM transactions on graphics (TOG)*. 1982, vol. 1, no. 3, pp. 235–256.

30. WILLIAMS, Brent Warren. *Fluid surface reconstruction from particles*. 2008. PhD thesis. University of British Columbia.

31. MÜLLER, Matthias; SCHIRM, Simon; DUTHALER, Stephan. Screen space meshes. In: *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 2007, pp. 9–15.

32. VAN DER LAAN, Wladimir J; GREEN, Simon; SAINZ, Miguel. Screen space fluid rendering with curvature flow. In: *Proceedings of the 2009 symposium on Interactive 3D graphics and games*. 2009, pp. 91–98.

33. GREEN, Simon. *Screen Space Fluid Rendering for Games* [online]. San Francisco, California, USA, 2010 [visited on 2018-03-26]. Available from: `http://developer.download.nvidia.com/presentations/2010/gdc/Direct3D_Effects.pdf`.

34. REICHL, Florian; CHAJDAS, Matthäus G.; SCHNEIDER, Jens; WESTERMANN, Rüdiger. Interactive Rendering of Giga-Particle Fluid Simulations. *Proceedings of High Performance Graphics 2014*. 2014.

35. DUKOWICZ, John K. A particle-fluid numerical model for liquid sprays. *Journal of Computational Physics*. 1980, vol. 35, no. 2, pp. 229–253. ISSN 0021-9991. Available from DOI: `10.1016/0021-9991(80)90087-X`.

36. HARLOW, Francis H. The particle-in-cell computing method for fluid dynamics. *Methods Comput. Phys.* 1964, vol. 3, pp. 319–343.

37. BRACKBILL, JU; RUPPEL, HM. FLIP: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. *Journal of Computational Physics*. 1986, vol. 65, no. 2, pp. 314–343.

38. JIANG, Chenfanfu; SCHROEDER, Craig; SELLE, Andrew; TERAN, Joseph; STOMAKHIN, Alexey. The affine particle-in-cell method. *ACM Transactions on Graphics (TOG)*. 2015, vol. 34, no. 4, pp. 51.

39. FU, Chuyuan; GUO, Qi; GAST, Theodore; JIANG, Chenfanfu; TERAN, Joseph. A polynomial particle-in-cell method. *ACM Transactions on Graphics (TOG)*. 2017, vol. 36, no. 6, pp. 222.

40. STAM, Jos. Real-time fluid dynamics for games. In: *Proceedings of the game developer conference*. 2003, vol. 18, p. 25.

41. WEST, Mick. Practical fluid dynamics. *Game Developer magazine* [online]. 2007 [visited on 2018-04-04]. Available from: `http://www.cowboyprogramming.com/2008/04/01/practical-fluid-mechanics/`.

42. FROEMKE, Quentin. *Multi-Threaded Fluid Simulation for Games* [online]. Intel, 2011 [visited on 2018-04-04]. Available from: `https://software.intel.com/en-us/articles/multi-threaded-fluid-simulation-for-games`.

43. FEDKIW, Ronald; STAM, Jos; JENSEN, Henrik Wann. Visual simulation of smoke. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 2001, pp. 15–22.

44. NGUYEN, Duc Quang; FEDKIW, Ronald; JENSEN, Henrik Wann. Physically based modeling and animation of fire. In: *ACM Transactions on Graphics (TOG)*. 2002, vol. 21, pp. 721–728. No. 3.

45. ENRIGHT, Douglas; MARSCHNER, Stephen; FEDKIW, Ronald. Animation and rendering of complex water surfaces. In: *ACM Transactions on Graphics (TOG)*. 2002, vol. 21, pp. 736–744. No. 3.

46. CRANE, Keenan; LLAMAS, Ignacio; TARIQ, Sarah. Real-time simulation and rendering of 3D fluids. *GPU gems*. 2007, vol. 3, no. 1.

47. LONG, Benjamin; REINHARD, Erik. Real-time fluid simulation using discrete sine/cosine transforms. In: *Proceedings of the 2009 symposium on Interactive 3D graphics and games*. 2009, pp. 99–106.

48. MÜLLER, Matthias; CHARYPAR, David; GROSS, Markus. Particle-based fluid simulation for interactive applications. In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 2003, pp. 154–159.

49. ŠŤAVA, Ondřej; BENEŠ, Bedřich; BRISBIN, Matthew; KŘIVÁNEK, Jaroslav. Interactive terrain modeling using hydraulic erosion. In: *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. 2008, pp. 201–210.

50. KASS, Michael; MILLER, Gavin. Rapid, stable fluid dynamics for computer graphics. In: *ACM SIGGRAPH Computer Graphics*. 1990, vol. 24, pp. 49–57. No. 4.

51. HOLMBERG, Nathan; WÜNSCHE, Burkhard C. Efficient modeling and rendering of turbulent water over natural terrain. In: *Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*. 2004, pp. 15–22.

52. THUREY, Nils; MULLER-FISCHER, Matthias; SCHIRM, Simon; GROSS, Markus. Real-time breaking waves for shallow water simulations. In: *Computer Graphics and Applications, 2007. PG'07. 15th Pacific Conference on*. 2007, pp. 39–46.

53. CHENTANEZ, Nuttapong; MÜLLER, Matthias. Real-time simulation of large bodies of water with small scale details. In: *Proceedings of the 2010 ACM SIGGRAPH/Eurographics symposium on computer animation*. 2010, pp. 197–206.

54. LEE, Richard; O'SULLIVAN, Carol. A Fast and Compact Solver for the Shallow Water Equations. In: *VRIPHYS*. 2007, pp. 51–57.

55. LAYTON, Anita T; PANNE, Michiel van de. A numerically efficient and stable algorithm for animating water waves. *The Visual Computer*. 2002, vol. 18, no. 1, pp. 41–53.

56. COPSEY, Dan. *Output from a (linearised) shallow water equation model of water in a bathtub.* [online]. 2007 [visited on 2018-04-08]. Available from: `https://upload.wikimedia.org/wikipedia/commons/3/37/Shallow_water_waves.gif`.

57. XAVIER, Adilson V; GIRALDI, Gilson A; APOLINARIO JR, Antonio L; RODRIGUES, Paulo S. Lattice gas cellular automata for computational fluid animation. In: *Proceedings of the 4th Workshop of Theses and Dissertations*. 2005, pp. 1–6.

58. BARCELLOS, B; GIRALDI, GA; SILVA, RL; APOLINÁRIO, AL; RODRIGUES, PSS. Surface flow animation in digital terrain models. In: *SVR 2007-IX Symposium on Virtual an Augmented Reality*. 2007, pp. 123–132.

59. FRISCH, Uriel; D'HUMIERES, Dominique; HASSLACHER, Brosl; LALLEMAND, Pierre; POMEAU, Yves; RIVET, Jean-Pierre. *Lattice gas hydrodynamics in two and three dimensions*. 1986. Technical report. Los Alamos National Lab., NM (USA); Observatoire de Nice, 06 (France); Ecole Normale Superieure, 75-Paris (France).

60. LI, J. Appendix: Chapman-Enskog Expansion in the Lattice Boltzmann Method. *ArXiv e-prints*. 2015. Available from arXiv: `1512.02599 [physics.flu-dyn]`.

61. JUDICE, Sicilia F; BARCELLOS, Bruno; GIRALDI, Gilson A. A cellular automata framework for real time fluid animation. In: *Proceedings of the Brazilian Symposium on Computer Games and Digital Entertainment*. 2008, pp. 169–176.

62. HEINTZ, Christian; GRUNWALD, Moritz; EDENHOFER, Sarah; HÄHNER, Jörg; MAMMEN, Sebastian von. The Game of Flow - Cellular Automaton-based Fluid Simulation for Realtime Interaction. In: *Proceedings of the 23rd ACM Symposium on Virtual Reality Software and Technology*. Gothenburg, Sweden: ACM, 2017, 76:1–76:2. VRST '17. ISBN 978-1-4503-5548-3. Available from DOI: 10.1145/3139131.3143415.

63. RÜDE, U; THÜREY, N. Free surface lattice-Boltzmann fluid simulations with and without level sets. In: *Proceedings of Vision, Modeling and Visualization*. 2004, pp. 199–208.

64. THÜREY, Nils; RÜDE, Ulrich. Stable free surface flows with the lattice Boltzmann method on adaptively coarsened grids. *Computing and Visualization in Science*. 2009, vol. 12, no. 5, pp. 247–263.

65. FLOWKIT LTD. *Lattice Boltzmann Method - the kernel of Palabos* [online]. Geneva, 2011 [visited on 2018-04-08]. Available from: http://www.palabos.org/software/lattice-boltzmann-method.

66. SUCCI, Sauro. *The lattice Boltzmann equation: for fluid dynamics and beyond*. Oxford university press, 2001.

67. PERUMAL, D. Arumuga; DASS, Anoop K. A Review on the development of lattice Boltzmann computation of macro fluid flows and heat transfer. *Alexandria Engineering Journal*. 2015, vol. 54, no. 4, pp. 955–971. ISSN 1110-0168. Available from DOI: 10.1016/j.aej.2015.07.015.

68. JUDICE, Sicilia F.; COUTINHO, Bruno Barcellos S.; GIRALDI, Gilson A. Lattice Methods for Fluid Animation in Games. *Comput. Entertain.* 2010, vol. 7, no. 4, pp. 56:1–56:29. ISSN 1544-3574. Available from DOI: 10.1145/1658866.1658875.

69. BOHEMIA INTERACTIVE. *Ylands* [online]. Steam, Valve Corporation [visited on 2018-04-22]. Available from: http://store.steampowered.com/app/298610/Ylands/.

70. O'CONNOR, Alice. Arma Devs Launch 5v5 FPS Project Argo's Free Prototype. *Rock, Paper, Shotgun*. 2016. Available also from: https://www.rockpapershotgun.com/2016/11/01/arma-developers-release-free-project-argo-prototype/.

71. DYOX. *[TerrainEngine] Voxels / Marching Cubes / Robust Architecture / FPS Control* [online] [visited on 2018-04-22]. Available from: https://forum.unity.com/threads/terrainengine-voxels-marching-cubes-robust-architecture-fps-control.158887/.

72. FORSYTH, Tom. *Game Programming Gems 3*. Cellular Automata for Physical Modelling. Ed. by TREGLIA, Dante. Charles River Media, 2002. Game Programming Gems Series. ISBN 1584502339. Available also from: https://tomforsyth1000.github.io/papers/cellular_automata_for_physical_modelling.html.

73. VON NEUMANN, John. *Theory of Self-Reproducing Automata*. Ed. by BURKS, Arthur W. Champaign, Illinois: University of Illinois Press, 1966.

74. GARDNER, Martin. Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game "life". *Scientific American*. 1970, vol. 223, no. 4, pp. 120–123.

75. WOLFRAM, Stephen. Statistical mechanics of cellular automata. *Reviews of Modern Physics*. 1983, vol. 55, no. 3, pp. 602.

76. TYLER, Tim. *Cellular Automata neighbourhood survey* [online] [visited on 2018-04-12]. Available from: http://www.cell-auto.com/neighbourhood/.

77. TOFFOLI, Tommaso; MARGOLUS, Norman. *Cellular Automata Machines: A New Environment for Modeling*. Cambridge, Massachusetts: The MIT Press, 1987. ISBN 0-262-20060-0.

78. SCHIFF, Joel L. *Cellular Automata: A Discrete View of the World*. Hoboken, New Jersey: John Wiley & Sons, Inc., 2007. ISBN 047016879X 9780470168790.

79. HARRIS, John. Interview: The Making Of Dwarf Fortress. *Gamasutra*. 2008, pp. 9. Available also from: https://www.gamasutra.com/view/feature/131954/interview_the_making_of_dwarf_.php.

80. BOLAND, Keith; DOLLAS, Apostolos. Predicting and precluding problems with memory latency. *IEEE micro*. 1994, vol. 14, no. 4, pp. 59–67.

81. CWALINA, Krzysztof; ABRAMS, Brad. *Framework design guidelines: conventions, idioms, and patterns for reusable. net libraries*. Pearson Education, 2008.

82. DEAN, Jeff. Designs, lessons and advice from building large distributed systems. *Keynote from LADIS*. 2009, vol. 1. Available also from: `http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf`.

83. SWANSON, Steven. *Memory and Caching - Introduction to Computer Architecture* [University Lecture]. University of California San Diego, 2013 [visited on 2018-05-13]. Available from: `https://cseweb.ucsd.edu/classes/sp13/cse141-a/Slides/10_Caches_detail.pdf`.

A Electronic attachments

- Executable file of the resulting standalone application
- Unity project files and source codes of the final application and discontinued prototypes
- Short video from the game demonstrating the partial integration of the simulation
- The results of benchmarking in CSV files

B Graphs

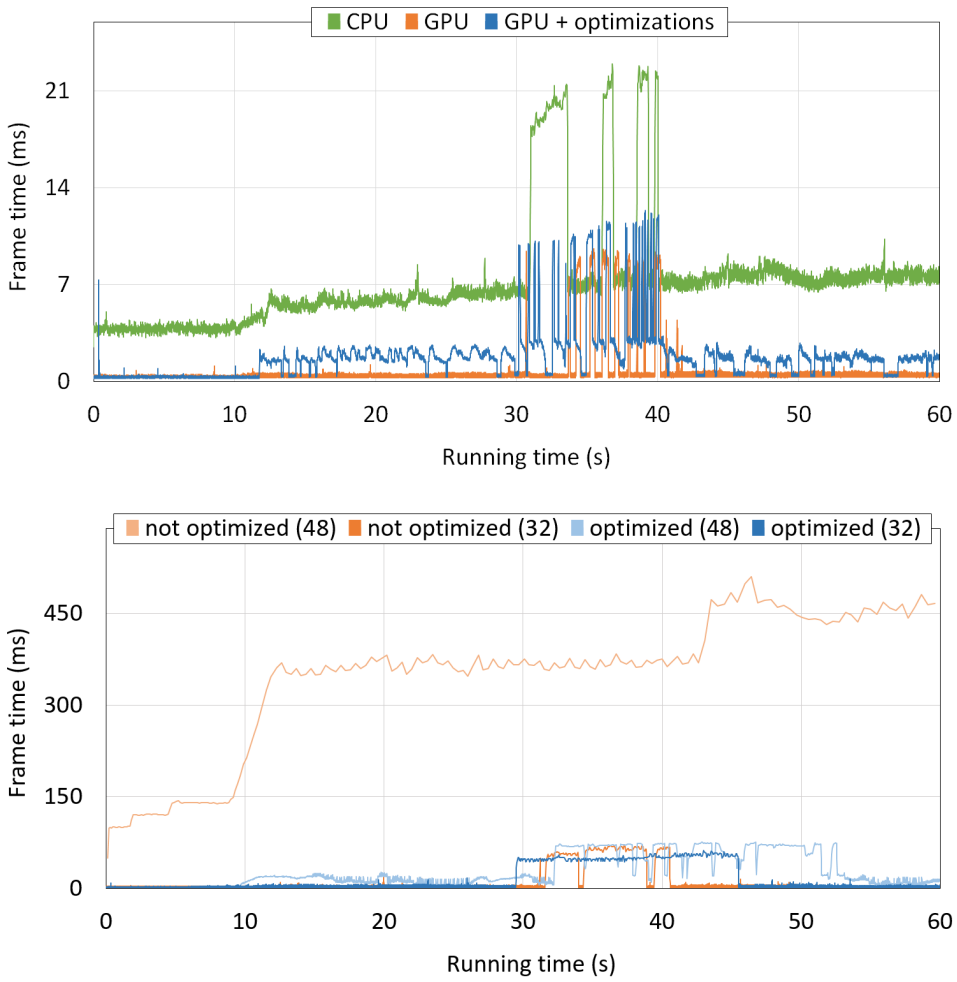

Figure B.1: The frame times of *Components* prototype improvements (top) and scaling across different world sizes (bottom) run on PC 2.

C Screenshots

User can choose from 4 pre-generated terrains in the application: ground, dowhill, u-bend and random. All of these terrains can user freely modify while the application is running. User is also free to add or remove water from the world at any time.

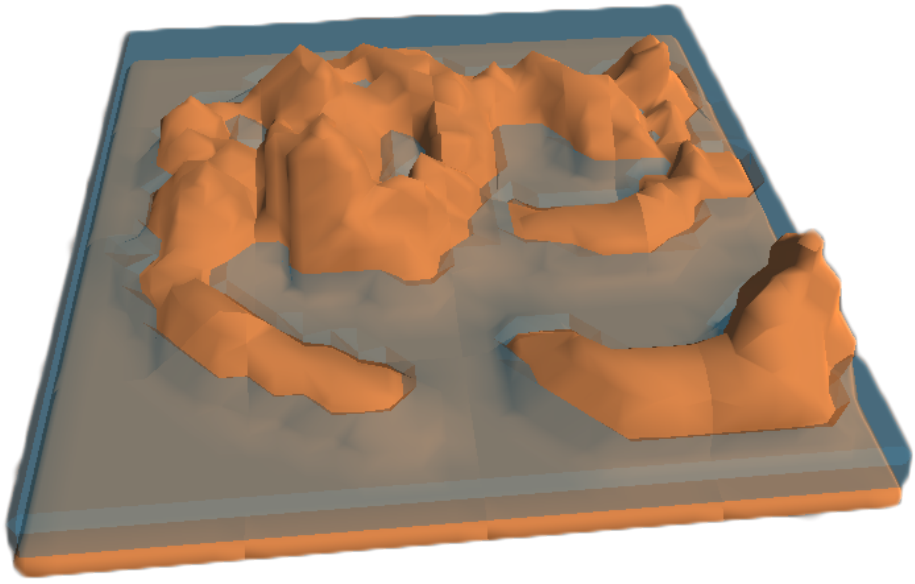

Figure C.1: "Ground" terrain already modified by the user.

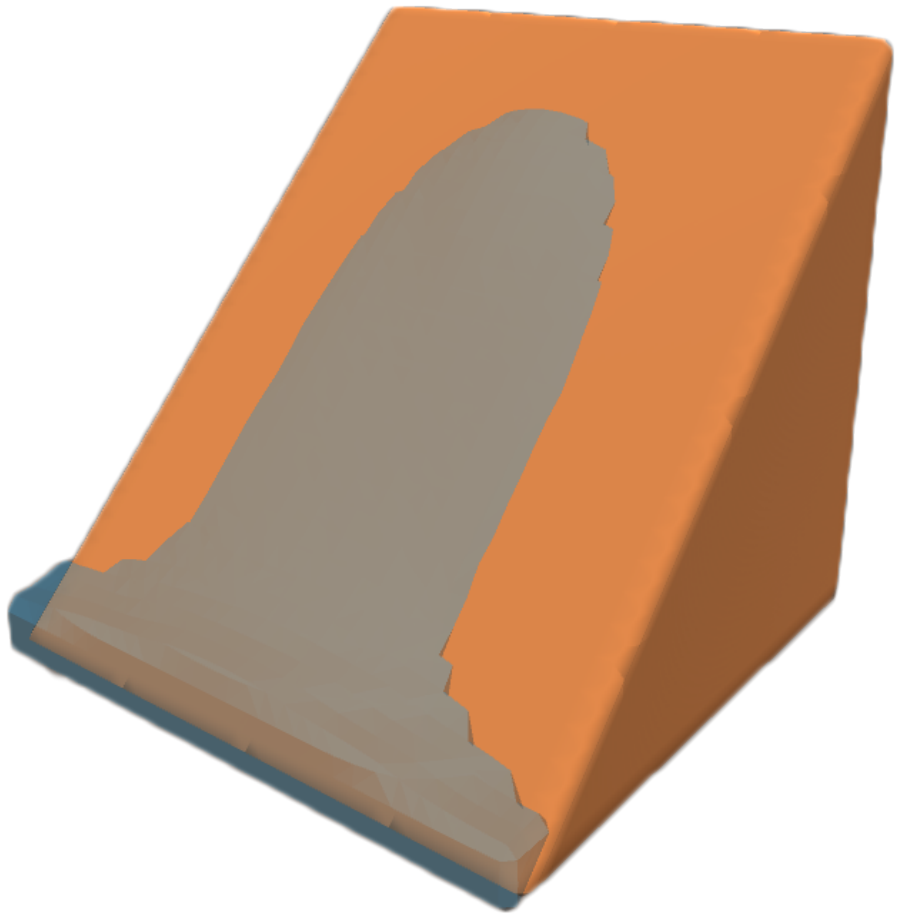

Figure C.2: Downhill terrain.

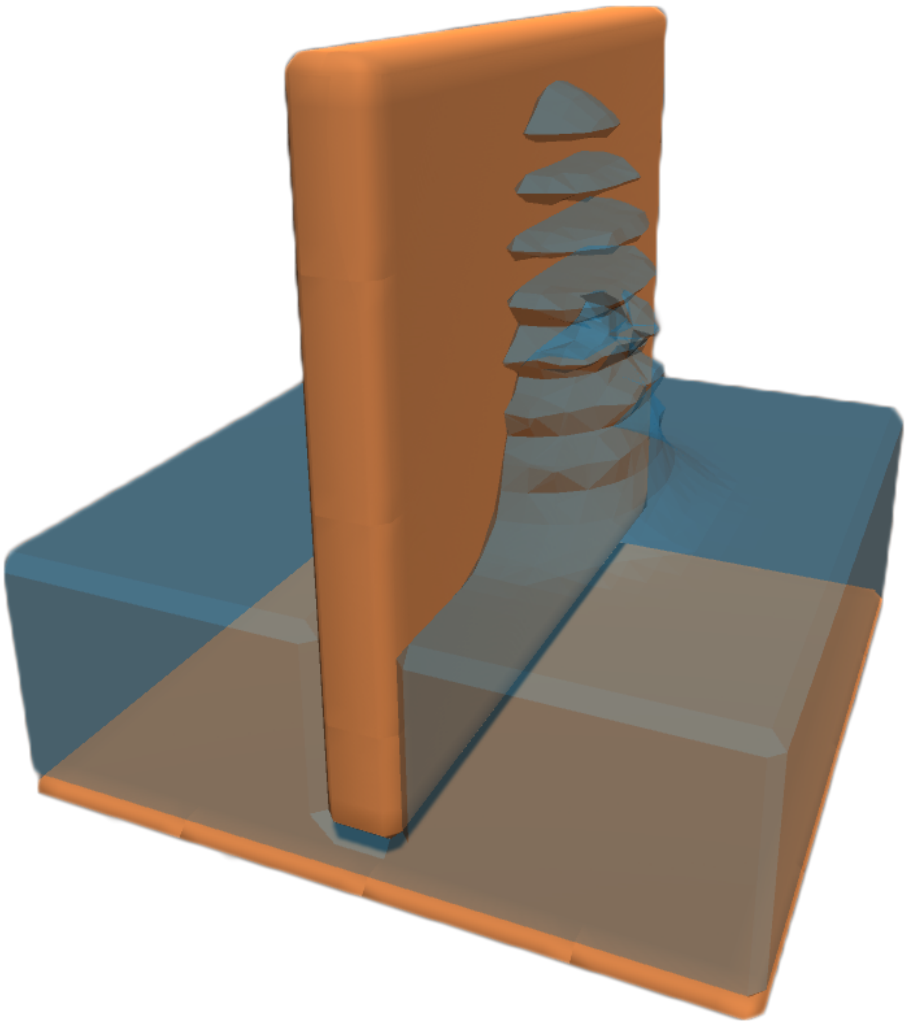

Figure C.3: U-Bend shaped terrain.

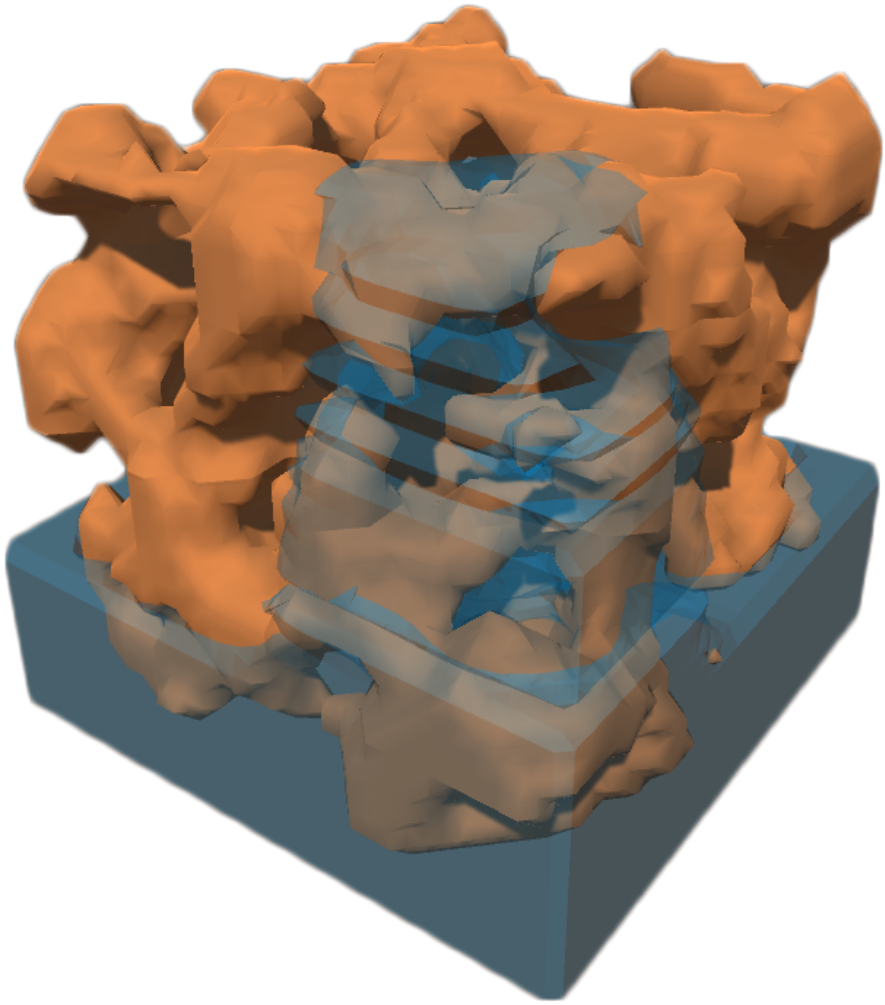

Figure C.4: Randomly generated terrain.