
SOL IUSTITIAE
Utrecht
University

CELLULAR AUTOMATA FLUID PHYSICS FOR VOXEL ENGINES

BY

WILLIAM VAN DER SCHEER

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Department of Information and Computing Sciences
in the Graduate College of the
Utrecht University, 2022

Supervisor:

Jacco Bikker

ICA-5988136

ABSTRACT

This master thesis investigates existing simulation techniques for continuum physics simulation and searches for a method that can be used for real-time simulation in a retro voxel engine. Consequently, a cellular automaton is developed that improves upon simple cellular automata for fluid simulation by modeling momentum advection and compression constraints, while maintaining the simplicity of a CA. The fluid simulation is implemented on a GPU with dynamic memory allocation and is capable of Real-Time simulation. A qualitative analysis of the visual quality of the simulation and quantitative analysis in terms of performance and memory consumption is performed. Finally, further extensions to the simulation framework and improvements are suggested.

ACKNOWLEDGMENTS

I would like to express my gratitude to my primary supervisor, Jacco Bikker, who guided me throughout this project. I would also like to thank my friends and family who supported me.

TABLE OF CONTENTS

1 Introduction . . . 1
1.1 Voxel World Template . . . 1
1.2 Research Motivation . . . 3
2 Voxel and continuum physics . . . 3
2.1 Representing Materials . . . 4
2.2 Governing Equations . . . 5
2.3 Finite Element Method . . . 6
2.4 Cellular automata . . . 7
2.5 Eulerian methods . . . 9
2.6 Lagrangian methods . . . 10
2.7 Semi-Lagrangian methods . . . 11
2.8 The Particle in Cell approach . . . 12
2.9 Real-time voxel physics . . . 13
3 Approach . . . 15
3.1 Scope . . . 15
3.2 Rule Neighbourhood . . . 16
3.3 Flow . . . 17
4 Implementation . . . 18
4.1 Buffers and Cell Definition . . . 18
4.2 Signing Flow . . . 20
4.3 Rules . . . 20
4.4 Flow clamping . . . 25
4.5 Data structure and memory management . . . 26
4.6 GPU implementation . . . 28
4.7 Rendering . . . 28
4.8 Required update rate . . . 29
4.9 Optimizations . . . 29
5 Results . . . 30
5.1 Water properties . . . 31
5.2 Comparison with Mantaflow . . . 32
5.3 Benchmark results . . . 36
5.4 Performance Comparison with a 3D Eulerian fluid simulator . . . 37
5.5 Simulation complexity . . . 38
6 Conclusion . . . 39
6.1 Future work . . . 39

1 Introduction

Voxels in computer graphics generally represent values associated with the cells on a three-dimensional regular grid. In that sense, they are similar to pixels on a 2D screen but instead with an extra dimension. Typically their value denotes a color, with which their cell is rendered as a solid cube. Clusters of voxels are then used to represent geometry. Polygons on the other hand represent 2D or 3D geometry using a collection of vertices and edges that efficiently represent their surface but do not implicitly model volume. Voxels are therefore an alternative and straightforward way to represent volumes in volumetric simulations and imaging.

Since voxels are simply values in a regular grid, they can be intuitively edited. Creating Voxel objects is very simple and in games they are therefore commonly used in procedural terrain generation, deformation, and destruction. Polygonal representations of objects on the other hand are often very difficult to modify on the fly without creating artifacts, as their geometry often needs to be carefully reconstructed. Handling voxels is often just a case of directly accessing the grid and setting or unsetting voxels individually. Although voxels can not model smooth surfaces since their locations and sizes are fixed on a discrete grid, this visual behavior is often desired stylistically and data structures such as the voxel octree [LK10] greatly increase the detail that can be achieved, where the individual voxels become so small they become nearly distinguishable from a smooth surface.

1.1 Voxel World Template

The goal of the Retro Voxel World template by Jacco Bikker, see Figure. 1, is to provide a simplistic Game Programming environment for novice *c++* game programmers that still provides, behind the scenes, a state-of-the-art ray tracing rendering system. That said, all low-end functionality is available for advanced users to delve into, allowing them to gain full control of their machine. To maintain its simplicity and enforce its retro feel, the world is comprised of a single grid without local transformations. All objects must exist in the main grid that represents the entire world and no sub-grids exist that have a relative position to the main grid. As of now, movement is simply achieved by setting clusters of voxels to a different location, i.e. all voxels of an object move to a neighbor next to them. Currently, no physics exists in this template, so if a user of the template wants to use physics then they must implement these themselves.

Physics is an essential element of many games, but programming physics is often not a simple task and therefore not something one would want to put onto novice programmers

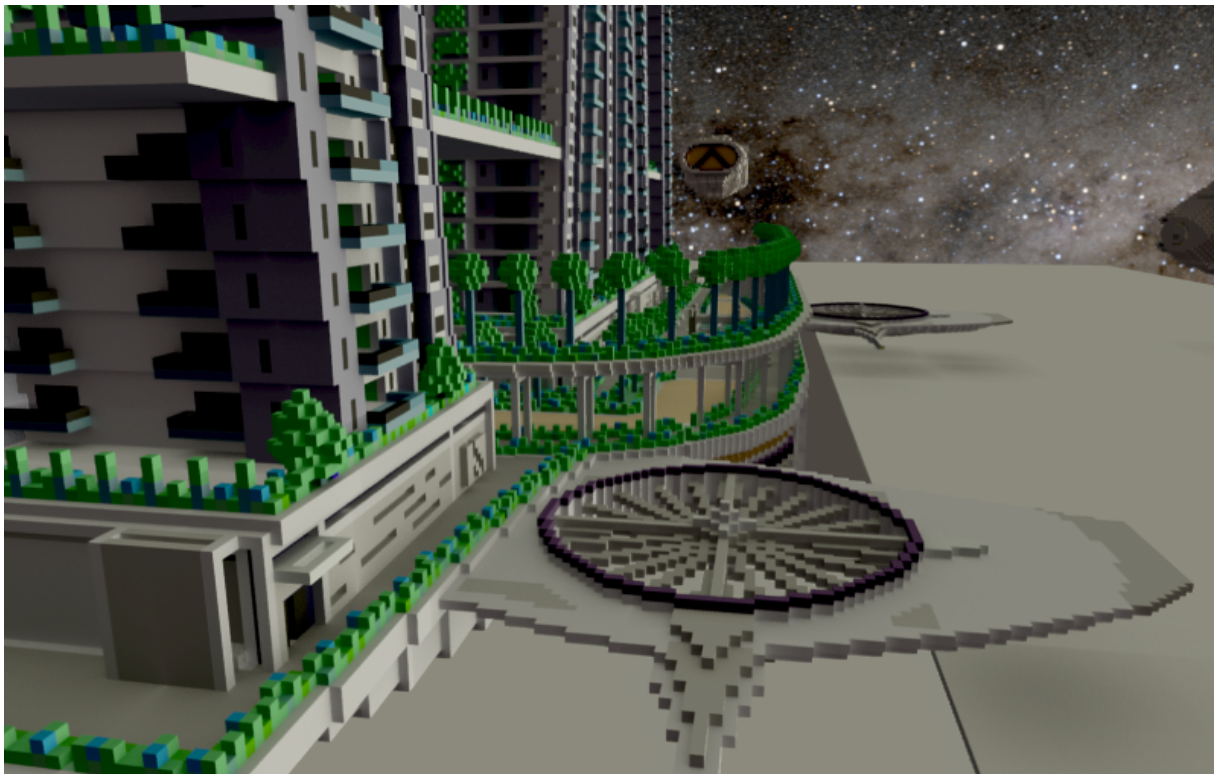


Figure 1: The voxel world template

to develop themselves. Providing interesting physics opens many more options for the novice programmer to create and experiment and therefore it would be beneficial to include some physics into this world template, primarily fluid flow, but potentially also heat, melting, smoke, and destruction. However, it is important to be specific on what kind of physics we are looking for because the available literature is enormous and clear boundaries need to be set to filter out irrelevant sources. Therefore a list of requirements is specified for our physics solution that fits the voxel world philosophy. The physics should:

- Work without local transformations and grids, i.e. no objects can exist in a local grid with a relative position or orientation to the world.
- Be robust and intuitive; The simulation should be stable, not easily blow up and the visual effects and interactions must make sense.
- Fast enough to be Real-time when used on a reasonable scale
- Add as little complexity to the template as possible; specifically, a simulation should be relatively easy to set up for a novice programmer.
- Look plausible and preferably cover as wide a variety of effects as possible.

1.2 Research Motivation

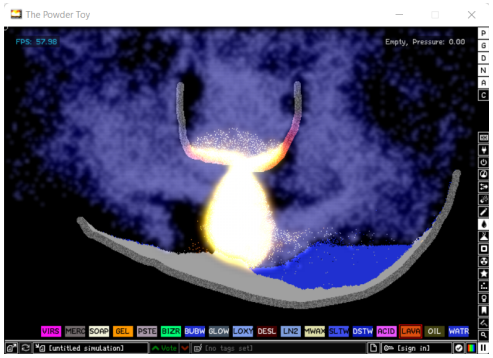
Because the voxel world utilises only a single coordinate system and therefore can not host typical rigid-body physics that use local transformations, we must look at other ways of simulating physics. Since there is a single grid that represents the volume of our world, one approach is to consider volumetric physics, in which one updates the state of the volume instead of points (e.g. locations of objects or vertices). To keep the system robust and intuitive, we would like to implement a unified system that can simulate multiple materials at the same time and model reasonably complex behavior implicitly. Ideally, a physics system that "just works" similar to how in a typical game engine an object can be marked as being a rigid body and it behaves as one would expect it to.

The goal is to provide real-time physics in a real-time rendering environment, but volumetric physics are generally far more expensive than rigid body simulation and the performance typically depends on the size of the simulation. Therefore we specify that we want to see a real-time simulation within reasonable limits. Additionally, the physics should add little complexity to the template, meaning a simulation does not require extensive configuration to get working and should deviate from the fundamental single grid premise as little as possible. Finally, we would like to simulate advanced effects following actual physical laws, however, it only has to look like it plausibly obeys those laws, additional accuracy is considered a bonus. There is no need to accurately solve engineering questions with the envisioned simulations so trading accuracy for performance and simplicity when appropriate is not an issue.

2 Voxel and continuum physics

Although games based on voxels remain popular in the game development community, there do not exist many large-scale voxel engines or AAA voxel-based games. Most voxel games are smaller indie games focused on building and destruction for which voxels are a straightforward choice. Many games use voxels only for representing terrain and do not include many physics besides gravity. Rigidbody physics are common but are generally implemented like in a typical game engine using local transformation of voxel objects. The upcoming Atomontage engine [Ato22] is an example of a powerful dedicated voxel engine that promises fast high resolution voxel rendering and editing. Although it uses local transformations for rigid-body movement it also includes "atom-atom" interactions between voxels and many interesting effects such as soft body voxels.

More examples of interesting volumetric voxel physics can be found in 2D games, even



(a) The Powder Toy

(b) Noita

Figure 2: Two falling sand games

though 2D cells are not voxels, many techniques and effects defined over 2D cells easily translate to 3D cells. Primary examples of this are falling-sand games such as The Powder Toy [tan08] and Noita [Nol20], see Figure. 2. These sandbox games extensively simulate different materials and interactions, providing advanced behavior such as fire, explosions, melting, and chemical reactions. Advanced real-time 3D material simulation physics are relatively uncommon in games, notable exceptions are the X-COM series [Fir12], Silent Storm [Niv03], and the fluid simulation in the recently successful game Teardown [Tux20], which simulates fire, water, and gases that interact realistically with a destructible environment. These games still use local rigid bodies for collisions and movement, however.

Typically computations over a grid of cells or voxels can be made massively parallel, and efficient methods for solving physics simulations over a grid can be found in [MST10] and [Set+14]. To keep memory usage down [Mus13] presents a data structure that allows high-resolution sparse grids with dynamic topology, for which [Hoe16] provides efficient ray traversal.

2.1 Representing Materials

Both fluids and solids alike are made up of numerous molecules whose local interactions ultimately cause the macroscopic behavior we are familiar with. Since it is computationally infeasible to simulate all the molecules present in real liquids, a suitable macroscopic representation for them that accurately represents their general behavior is required. There are two primary approaches to representing materials macroscopically.

The Lagrangian approach represents a fluid as a collection of individual particles and solves the movement for each particle separately to mimic the actual behavior of molecules in a material. This approach is simple and intuitive as it serves as a direct analogy to

the underlying phenomena, but such methods typically struggle whenever the density of particles is no longer sufficient and there is not enough information to accurately calculate the behavior of the particles.

In contrast, the Eulerian or Grid-based approach treats a fluid as a continuum and calculates the overall change of the state of the material across discretized pieces of space, typically a regular grid. It does not suffer from the same problem with low-density regions but instead has other drawbacks. Since the material is defined by a regular grid, the resulting surface will be similarly less detailed or requires a very large amount of cells. Eulerian methods are often also more complicated and mathematically less intuitive when used to model complex local behavior. In return, they are very good at enforcing constraints, like the conservation of mass and incompressibility. Eulerian methods are often used to simulate fluids but are less common for modeling solids.

A simulation method can also use a hybrid representation, in which the shortcomings of the purely Eulerian or Lagrangian approaches are balanced out. Typically this means solving the local movement using particles and enforcing the constraints with an auxiliary grid. The Eulerian or grid-based approach is of particular interest to us because incorporating it into a grid-based voxel engine is naturally very straightforward, but this research will not simply disregard Lagrangian methods because they have historically proven to be quite powerful.

2.2 Governing Equations

In general, when attempting to generate realistic-looking simulations of physics one tries to adhere to the governing equations that describe real-life physics. Since these often complex equations are of a continuous form that does not possess known analytical solutions, numerical methods must be employed to solve them. In nature, the behavior of Newtonian fluids such as water is modeled by the Navier-Stokes equations. These two partial derivative equations describe the flow of viscous fluids by mathematically ensuring the conservation of mass and momentum in the fluid. Most of computational fluid Dynamics (CFD) therefore revolves around numerically solving the Navier-Stokes equations to accurately simulate fluids.

$$\rho\frac{D\vec{V}}{Dt} = -\nabla p + \mu\nabla^2\vec{V} + \rho\vec{g}$$

$$\nabla \cdot \vec{V} = 0$$

The first equation is the momentum equation which describes how the velocity field of the fluid changes over time. Here ρ is density, t is time, V is velocity, p is pressure, μ is viscosity, and g is gravity. The first term is the pressure gradient, describing how fluids flow in the direction of the largest difference in pressure. The second term models diffusion of the momentum given a viscosity μ and the third term represents the external forces on the fluid, like gravity. The second equation is called the continuity equation and is a constraint that states that the total change in mass should be zero. For a more in-depth view of these equations see [Tem01].

Similar to solving the Navier-Stokes equations for fluid flow, other phenomena can be modeled by solving the corresponding governing equations, like for heat flow:

$$\rho \frac{Du}{Dt} = -\nabla \cdot Q$$

$$Q = -\kappa \nabla T$$

$$c = \frac{du}{dT}$$

Here Q is the heat flux, k is heat conductivity, T is temperature, u is heat energy and c is heat capacity per unit mass. More equations for different effects and interactions exist, like for elasto-plastic stress and phase changes. See [GS08] for an overview and more in-depth information on continuum mechanics.

Numerically solving these governing equations is typically a difficult task, especially when trying to maintain performance and simplicity. It is therefore common for simulation methods to use simplified models and make assumptions. The way the models are simplified and solved determines how realistic the simulation will ultimately be, and is therefore where different methods typically distinguish themselves and provide different advantages and disadvantages to simulation accuracy.

2.3 Finite Element Method

In many engineering and mathematical disciplines a popular method to numerically solve the differential equations like the governing equations mentioned above is the Finite Element Method (FEM). In FEM, a continuous system is subdivided into smaller discrete

parts called finite elements. This is typically done by creating a mesh of the object. The equations are then solved by formulating boundary conditions and solving a system of equations with minimum error. Mostly used in engineering, the focus of FEM lies with accuracy and is as such not very common in computer graphics although some approaches exist, typically for soft-body simulations [EKS03], [PO09], [Bar+07]. Additionally, real-time simulation using FEM is also becoming more common, see [MZ19] for an overview.

Because most of the literature around FEM has roots in engineering and falls outside the scope of this review we will not further focus on this domain, however, there is a high degree of similarity between FEM and many computer graphics and animation approaches. Discretizing a volume to finite elements and numerically solving the governing equations is something all simulation methods that attempt to be as realistic as possible must fundamentally do to facilitate computation. Therefore, since the definition of FEM is quite general, many simulation methods share similarities with FEM, even if they initially appear quite different or approach the problem from a different angle.

2.4 Cellular automata

Perhaps the most straightforward way to model complex behavior in a single grid is the cellular automaton. In traditional cellular automata (CA) as introduced by John von Neumann and Stanislaw Ulam in 1940, each cell in the grid has a finite set of possible states. A new state is determined based on the state of itself and its direct neighbors through a set of rules. Conway's infamous game of life [Gam70] is the prime example of this. In the game of life, each cell is either dead or alive. It is alive if two or three of its neighbors are alive. It is dead if 4 or more of its neighbors are alive. These are very simple rules but in actuality, this results in the emergence of very complex behavior and patterns that far eclipse the complexity of the underlying automaton. In fact, Conway's game of life is Turing complete [Ren16]. Cellular automata can be used to model many different phenomena, from crowd simulation [SHT10] to procedural generation [JYT10] and machine learning [LY02], [Sha+17]. It can in general be used to model complex systems and is therefore also a popular tool in biology literature and several other fields. For simulating more complex phenomena CA are typically extended in several ways. Extensions may include continuous state variables and corresponding continuous transition functions instead of discrete states and transition rules, stochastic transition functions, or asynchronous updating of the cells. However, it then begs the question at what point a CA with such extensions can still be appropriately called a CA, as it becomes more like a typical Eulerian method or a Finite Element Method. One may consider that using a CA for modeling physics is essentially the same as taking an Eulerian approach, as a CA discretizes the space into a grid and evolves the state as a function of the vol-

ume the cells represent over time. Although CA at first glance may seem too simple to model advanced physical behavior, theoretically they are not. Vichniac [Vic84] was one of the first to propose that all physics can be fully modeled as an automaton in theory.

Cellular automata for physics simulations are quite common in games, although they are mostly limited to a 2D environment and generally do not model very complex behavior. See [Gal17] for a simple implementation of a fluid with an interactive demo. Games like Terraria [Re-11] and Starbound [Chu16] likely use similar fluid simulations and other games in general use CA to model a wide variety of natural dynamic behavior like Far Cry 2 [Ubi08] which uses CA to model the spreading of fire over the terrain. The advanced physical effects present in Noita [Nol20] (Figure. 2b) are also the result of complex CA. Beyond 2D there is not much in games or research that presents fully-fledged physics modeling using CA. Notable examples are the previously mentioned Silent Storm [Niv03] and X-Com [Fir12], which model a broad collection of effects, although on a modest scale.

Overall the availability of thorough research on cellular automata for games appears quite thin or dated. One can find interesting practical examples and explanations for doing several unified physical effects such as fluids, heat, and gases by implementing the general ideas of the governing equations in [Cho09] and [TF02]. However, it is not immediately apparent how these techniques perform on a larger scale and how realistic they are as an actual modern implementation can not be found. Most academic research on physics simulations using CA come in the form of fluid simulations, see [JBG08], [JCG10], [MMH12], [MF15]. In [Med18] a CA method is presented for real-time fluid simulation in a game environment that improves upon methods like [Gal17], by more quickly simulating the effects of pressure by using connected components to determine the surface of the water and directly teleport fluid between cells that are far apart. While this solution solves some traditional problems with CA fluids, it does not model momentum and uses global operations over the cells. In [Hei+17] a fluid CA is presented that also models multi-mixture fluids and can model oil floating on water. In addition to fluid simulation, some work like [Don+13] also presents a CA method for simulating elastic solid material, and [XWK03] presents a CA that allows animation of melting solid objects.

There is also little research exploring and improving the performance of larger-scale CA in 3D [Dav09] for interactive applications, which is surprising since they are often used in 2D. Research on hardware acceleration on CA's exists in general [CSH11], [Gob+11], [Dou+15] but not really in the context of real-time physics simulations for games. When comparing CA's to other Eulerian physics simulations, one major disadvantage of traditional CA stems from the fact that they are limited in the type of computations that can

be done as they can only consider the state of direct neighbors in the update function. For many problems, this means they must update faster than the physical phenomena can traverse them. As an example, when they model an explosion that propagates rapidly, the speed of the explosion may not traverse faster than a single cell per update. Many simulation methods have similar constraints where the minimum timestep needed to guarantee a proper and stable simulation depends on the modeled phenomena. One way to deal with this is to use an extended CA that can dynamically set update rates for highly active cells, however, other Eulerian methods typically access data in an arbitrary location and circumvent this problem, see the way advection is solved by [Sta09] in Section 3.7 as an example. There exist extensions to CA that can extend the neighborhood beyond directly adjacent cells, like in [Eva01], or have dynamic topology allowing them to form links with non-local cells, which could allow them to mimic this behavior. There appears to be little information on how to deal with specifying the coarseness of CA as sometimes the amount of needed detail needed varies greatly and there is limited research on dynamic scaling of CA and level of detail adaptation for visual simulation, which seems like a logical solution to providing visually plausible physics efficiently. See [Wol19] and [Ila01] for comprehensive overviews of cellular automata.

2.5 Eulerian methods

Besides CA, fully Eulerian methods for continuum simulation mostly constitute fluid simulations such as presented in [CTG10] and [CM11] which presented a real-time simulation using tall cell grids. Eulerian methods typically require additional logic when a non-regular boundary is present, like the surface of water. This is typically done using level sets and particles, as in [CM11]. The marker-in-cell (MAC) approach simply adds particles to the grid which then simply follow the velocity field, but no fluid simulation computations are performed on the particles themselves, they are only used for rendering. See [GHD03] for a MAC approach to fluid-solid interaction.

As with CA, there is comparatively little work on representing non-rigid solids, at least in the domain of computer graphics. This may be because it is harder to translate objects defined by specific points (e.g. vertices) to an Eulerian continuous domain mathematically. Typically Eulerian approaches can be found in the physics literature, while in engineering Lagrangian methods are more prevalent. When looking for a unified framework of physics we find some unified Eulerian frameworks for multi-material continuum mechanics in the physics literature, see [GSL18] and [JN20], however, these methods are not designed for fast visual animation of physics.

Because of a lack of Eulerian approaches to solid mechanics, little progress has been made in creating a unified Eulerian approach. Relatively recently [Lev+11] introduces an Eulerian method for solid deformation and contact, and fluid-solid coupling was introduced in [TLK16]. However, these methods can not match the breadth of effects that have been simulated with Lagrangian and Hybrid methods, like multi-fluids and phase transitions.

2.6 Lagrangian methods

Purely Lagrangian methods are popular in computer graphics and animation because of their straightforward implementation, particularly because they play well with complicated meshes and boundary problems. Smoothed particle hydrodynamics (SPH) [Mon92] is a flexible method that is widely used for simulation of fluids [GBO04], [MCG03], deformable solids [Mül+04] and granular materials like sand [AO11]. Solenthaler [SSP07] introduced a unified SPH model for the simulation of liquids, deformable and rigid objects where the materials are defined simply by the attributes of the underlying particles and also includes phase changes. Other improvements to SPH were introduced in [BK15], which introduced a way to enforce incompressibility using a divergence-free solver which allowed SPH to run with higher timesteps without artifacts, and the method presented in [Rei+19] reduces noise by using a Shepard interpolation.

More recently acceleration techniques have been introduced. Variable resolution of SPH through dynamically splitting and coalescing particles is presented in [Vac+13] and [Vac+16] which results in a significant speedup. Similarly [Hu+17] and [Ji+19] introduce multi-resolution SPH while maintaining accuracy. Although Lagrangian methods are technically grid-free they often still utilize some data structure, like graphs or grids, to accelerate nearest neighbor queries, and a lot of research on increasing the performance of these algorithms therefore focuses on this area. Improvements to neighbour searches, like neighbour lists and neighbour particle identification can be found in [VBC08] and [Dom+10]. GPU acceleration methods are presented in [Cre+11] and [Che+20a], and a hierarchical strategy for SPH is presented by [Hua+20].

Many physics simulations solve forces and momentum on an object to calculate the new positions of the objects and substances as is physically accurate. However these methods often struggle to correctly fix position violations when the used time step is relatively large, and in computer animation and games, most manipulation is intuitively done by editing positions instead of velocities. The Position-Based Dynamics (PBD) approach which directly works with positions instead of forces and velocities was introduced

by Muller in [Mül+07]. In PBD the corresponding change in momentum on an object emerges by solving position constraints. The method is quite popular in games and real-time applications since it is stable and efficient, even if some accuracy is lost. In [MM13] Macklin and Muller introduce a similar approach for simulating fluids in Position-Based Fluids (PBF) and finally in [Mac+14] they present a unified Lagrangian approach position based fluids and constraint particle positions. Although performance is real-time, large amounts of particles still create a severe performance penalty and they suggest using variable particle sizes to allow the hierarchical simulation to improve scalability, similar to the method proposed by [Vac+13].

A way to model multiple fluids using a mixture model was presented in [Ren+14] and extended to handle more phenomena in [Yan+15]. Similarly, solids and multiphase-fluid SPH are provided in the unified framework presented by [Yan+16]. A unified particle framework that integrates the phase-field method with multi-material simulation and can model both liquids and solids with phase transitions was also presented in [Yan+17]. A recent moving least squares approach [Che+20b] to the problem can simulate advanced solid-fluid interactions but does struggle to maintain good performance when stiffer materials are involved.

2.7 Semi-Lagrangian methods

Likely the most historic paper in CFD is Jos Stam's Stable Fluids [Sta99], which presented the first method for an unconditionally stable fluid simulation by using the semi-Lagrangian method to approximate the Navier-Stokes equations. It is classified as semi-Lagrangian because, while it is mostly an Eulerian method, it solves the advection of the fluid across the grid as the linear movement of essentially "the average particle" but does not commit to an explicit particle definition. Although the results are still quite satisfying from a computer graphics point of view, the flow is not very detailed. This happens because the advection step linearly interpolates the in reality complex velocity of the implicit particle to the grid, which results in smoothening of the velocities and numerical dissipation of the flow. Many improvements and extensions to the method followed. In [FSJ01] the method is improved with vorticity confinement to display better vortice behavior, particularly for smoke effects, and [SRF05] introduces a hybrid Vortex particles approach to further improve the simulation of highly turbulent flows typically unachievable by grid-based methods. Other improvements to semi-Lagrangian advection can be made by using higher-order advection schemes (as opposed to a linear first-order scheme) such as the MacCormack Advection 2nd order scheme proposed in [Sel+07]. Although now quite old, Stable Fluids remains a popular approach and is often the initial go-to

method for game developers or researchers interested in learning about fluid simulations. The method is practically real-time in 2D, however, when extending the method to 3D the algorithmic complexity becomes a problem as there will simply be too many cells that need to be updated for larger simulation domains. This is in general a problem with (Eulerian) fluid simulations like with CA, as simulating large volumes quickly becomes too expensive memory-wise without efficient hardware utilization or careful level of detail LOD scaling like proposed in [LGF04]. A GPU implementation can be found in [Ngu07] and more recently [ZKM16] detailed a GPU implementation of Stable Fluids specifically for integration with a voxel engine.

2.8 The Particle in Cell approach

Taking the idea of the semi-Lagrangian method a bit further, many recent and successful methods use a hybrid Eulerian-Lagrangian approach, combining the good aspects from both approaches to fluid simulation; The advection is solved from a Lagrangian perspective and the mass and momentum preservation constraints are solved over a grid. In the Particle-In-Cell (PIC) method, the movement of particles is simulated as with SPH but then is interpolated to a grid where the conservation step is solved. The new velocities are then interpolated back to the particles which can then proceed to the next time step. This method is stable, but due to the double interpolation presents high numerical dissipation like the stable fluids method. A successor to PIC that aims to address this problem is FLIP. FLIP (Fluid-Implicit-Particle) was introduced in [BR86] in 2D and later extended by [ZB05] for animating sand in 3D. FLIP prevents the double interpolations and therefore allows for less dissipative fluids, however it becomes unstable because angular and shearing information is lost. Often PIC and FLIP are blended to balance the amount of dissipation and stability, but this requires manual tweaking and is not ideal. More recent developments such as Affine PIC (APIC) and Polynomial PIC [Jia+15], [Fu+17] further improve the translation from grid to particle. APIC [Jia+15] uses an affine transformation to make sure angular and shearing information is correctly transferred from the grid to the particles and similarly, Polynomial PIC [Fu+17] uses polynomial functions to do so. PIC techniques such as FLIP and APIC are currently state of the art for fluid simulation. Recent research has also provided several important performance increases, like the spatially adaptive FLIP [NB16] and sparse volume GPU implementation [Wu+18] which can handle tens of millions of particles within a sparse grid and on an almost unbounded simulation domain.

The Material Point Method (MPM) [SZS95] serves as a generalization of the PIC technique, meant to extend the simulations to the domain of solid mechanics that require

compressible materials, particularly for simulating the deformation of elastic-plastic materials. However, beyond this scope, MPM in graphics has recently proven to be a powerful and versatile way to simulate many complex phenomena. This is because the MPM framework, similar to the unified Lagrangian methods previously seen, universally handles particles, and as a hybrid technique is quite flexible in the type of calculations that are allowed. [Sto+13] was the first to introduce MPM to computer graphics and animation and used MPM to simulate the complex solid-fluid dynamics of snow. Many subsequent papers followed extending the capabilities of MPM. [Sto+14] introduces phase changes and heat transport of materials. CD-MPM [Wol+19] introduces continuum damage and dynamic fracture animation of materials. Visual simulation of baking bread, cookies, and the like is achieved using a thermomechanical MPM introduced by [Din+19]. Recently [Su+21] presents unified viscoelastic liquids with phase change using MPM. MLS-MPM presented in [Hu+18] takes a different approach to solving the MPM system using Moving Least Squares that naturally formulates the improvements specified in APIC[Jia+15] and the Polynomial PIC [Fu+17]. It also introduces material cutting and two-way coupling with rigid bodies while also providing performance benefits and being easy to implement. Several impressive Real-time simulations have recently been implemented using this method [Lin20], [Kot20]. MPM is currently an active domain of research and serves as the backbone of many animation solutions for studios such as Disney. It arguably delivers the most visually impressive and varied visual simulations of materials and fluids to date. Recent performance optimizations and GPU implementations such as presented by [Gao+18], [Hu+19] and [Wan+20] allow for MPM simulations with millions of particles. In general, the focus for MPM methods still lies with introducing new effects and improving stability and accuracy for offline rendering. Extensions to the simulation and coupling of interactions between different materials typically must be defined explicitly. Even with highly efficient GPU implementations, most MPM solutions are not intended to be real-time but for the amount of detail and realism provided the performance is quite impressive. For more information, see [Soł+21] for a contemporary overview of MPM.

2.9 Real-time voxel physics

To find a method for implementing interesting physics for a voxel environment this research has explored a number of techniques and developments. Although many solutions can theoretically be used in a large-scale voxel environment, it seems that there is little to no research that explicitly explores modeling real-time physics in a single grid voxel world, although some methods can likely be converted to suit this scenario. It may seem that "The Future Is Volumetric" [Ato22], but most relevant research has been developed in the context of offline computer simulation and with a large focus on accuracy, making

it not very suitable for use in interactive applications.

Most volumetric physics in grid-based games use cellular automata, which is a straightforward solution that can model interesting physical phenomena and can also be intuitively framed in the context of a voxel engine. However recent research on CA for physical modeling is limited and particularly there seems little research on what a dedicated real-time 3D physics solution would look like, especially for larger-scale grids and with more advanced and accurate effects. There is little elaboration in the literature on how more advanced transition functions that better approximate the Governing Equations can be created and how the nature of CA limits them from achieving higher accuracy in this regard. Furthermore, although several effects have been modeled using CA there is no real example of a unified CA framework that includes both solid and fluid mechanics in the same spirit as, for example, [Mac+14]. Particularly, modeling rigid body type movement is an interesting open question that is particularly relevant for our scenario. The performance of CA for physical simulation and acceleration through hardware and algorithmic improvements has also not seen much exploration, making it unclear how effective CA can really be.

Similar to cellular automata, pure Eulerian methods, in theory, are very suitable for representing physics in a grid world since the coordinate system of the simulation coincides with world. However, the purely Eulerian approach seems to be more complicated and less adept at representing solids, as there is little research on Eulerian solids for computer animation. This consequently means little research on a unified Eulerean system for material simulation, and the approach particularly lacks behind in modeling advanced phenomena such as multi-fluids and multi-phases.

On the other side of the spectrum, several Lagrangian methods seem very promising, such as the unified frameworks as proposed in [Mac+14] or [Yan+16], which can provide real-time unified physics with many interesting effects. Particularly the PBD based approach presented in [Mac+14] is attractive since it enforces positional constraints which are important in providing artifact-free real-time simulations. Unfortunately, it is not immediately apparent how these particle methods can be elegantly converted to a voxel world while remaining stable and intuitive.

Recently, hybrid simulation methods have introduced greatly improved performance and realism. Particularly particle in cell methods such as APIC and FLIP allows for high-quality fluid simulations and the more general MPM also enables advanced general material physics. Extended MPM methods, like presented in [Sto+14] that can model

phase changes and varied materials, are visually very impressive, but they appear to be too slow and unstable, while also generally being used to model specific scenarios. While MPM is an active research topic and its full potential has not been reached yet, it looks likely that the extra attention to detail and realism that MPM methods provide cannot compensate for its disadvantages in the context of the presented scenario. MPM methods have so far been mainly used in offline rendering and have generally not been explored much for use in interactive games. On top of that, as they heavily feature particles they are also further removed from use in a voxel environment as with the Lagrangian approach.

3 Approach

Since there does not seem to be a method that provides unified volumetric physics explicitly for use in a single grid voxel environment, someone looking for information on a physics simulation in that context is therefore left with many questions and little guidance on how to proceed. Additionally, few solutions exist that can provide Real-Time Eulerian simulation at interactive rates.

Although it is believed that a particle approach with PBD as described by [Mac+14], converted to work well in a voxel environment is a solid alternative, and recent MPM methods are becoming increasingly attractive, it was decided that cellular automata for physics modeling is the most interesting and suitable avenue to take in our unique scenario. The premise that CA can provide a stable, flexible, and intuitive system for a voxel environment, is valued highly, even with potentially limited physical accuracy. Additionally, with CA being common in the games industry it is somewhat surprising that little information is available on what a larger scale fast 3D physics implementation looks like. Therefore, this research set out to develop a modern large-scale CA physics solution that can robustly model the desired effects and interactions and try to accelerate the simulation with modern hardware and acceleration structures, so we may further determine and extend the potential of CA-based physics and see how it can be integrated intuitively into a real-time environment.

3.1 Scope

While the CA solution should ultimately include a broad set of physics, we limit our scope to develop improved CA fluids, while designing the system such that other physics can easily be added to it, like heat flow. The limitations of the type of logic and operations allowed in our solution are also defined. This is to prevent deviating from the pure CA

paradigm, which can be easy to do. The following constraints for the CA simulator are stipulated:

- The CA can only use local operations within range 2 neighborhoods.
- Similarly, when processing a cell, it updates only values associated with itself, as a function of its neighborhood.
- The CA does not maintain particles of any sort
- Cell state is directly used as input to the renderer
- Arbitrary logic must be able to be easily added to the framework, the fluid simulation will similarly be developed as a set of rules. Adding additional rules should be straightforward.

To improve on typical real-time CA fluid simulation, we aim for a solution that simulates fluids at a more advanced level like Stable Fluids [Sta99], while also sticking as much as possible to the CA format at the same time, so the associated benefits remain. In other words, this research tries to combine the good traits from both game-oriented CA simulation and Eulerian fluid simulation and find a new method and baseline high-performance CA framework for physics simulation.

Additionally, in contrast to typical Eulerian fluid approaches, the presented solution is designed to allow for mass compression, making it capable of simulating gases at different pressures and liquids at the same time. This is to make more advanced extensions like multi-material physics and phase changes easier in the future. To make sure the system actually supports such extensions, we test the ease with which new logic can be developed and added to the system, like heat conduction and mixture materials which are briefly mentioned as further work but this is otherwise not considered the focus of this research. Finally, we approach this problem with performance in mind and focus on determining an efficient data structure and GPU implementation so our implementation can actually perform in real-time, at reasonably large scales in 3D.

3.2 Rule Neighbourhood

We define our rules in a range 2 Von Neumann neighborhood, instead of a Moore neighborhood, see Figure. 3. The Von Neumann neighbourhood defines the cells in its range in terms of the Manhattan distance, while the Moore neighborhood uses the Chebyshev distance. Using a Von Neumann neighborhood means a significantly reduced number of

cells that need to be accessed to update a cell; 6 instead of 26 at range 1. However, this also means that there are few directions in which cells can propagate or receive information. This consequently implies that the simulation will display anisotropic behavior. A Moore neighborhood would also still have a limited set of directions, and thus also be anisotropic, but due to the inclusion of diagonal connections, likely much less so. Still, the Von Neumann neighborhood is considered worth this drawback from a standpoint of performance and simplicity, See 3 for a visual example.

In general, only direct neighbors are accessed and most rules operate in a range 1 neighborhood with the exception of the flow update rule (see Section 4.3).

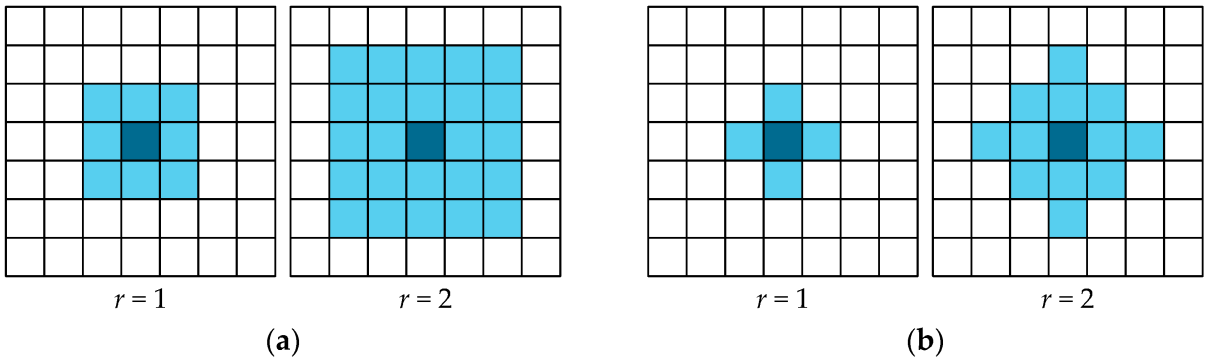


Figure 3: Range 1 and 2 Moore (a) and Von Neumann (b) neighbourhoods. Original image taken from [Evs+16]

3.3 Flow

In the CA there exist no explicit locations between cells, therefore state values are transferred discretely directly from one cell to another. While one can define the distance between cells, the CA does not use a semi-Lagrangian implicit particle to model advection and does not explicitly define time. Instead, advection is described in terms of the fraction of material that is transferred between cells in each update. This fraction will be called "flow" instead of velocity.

Since distance and time are not defined explicitly, material will essentially teleport instantly between cells. This means the fluid density is smeared out as it travels through the grid, similar to numerical dissipation and smoothing as present in many Eulerian methods. Because the system will render the fluid voxels based on material density this will be particularly visible. We will refer to this behavior as "dispersion" and visual examples can be seen in Section 5, particularly, in Figure. 9. While this definition of flow instead of continuous velocities moves us further away from accurately approximating the

Governing Equations, it fits the CA paradigm well, is intuitive, and it is still possible to model the momentum of flow in this way.

Staggered Grid

While a single flow is specified per cell, each flow value refers to the flow on the left side of each cell, i.e. on the border between the cell and its left neighbor. This offset or staggered arrangement of the flow component with respect to the pressure and density values is typically used in MAC methods [GHD03], which has some beneficial properties for modeling advection and determining the pressure gradient. Defining the flows for each cell in this way causes some awkward asymmetry in the way rules are defined, i.e. when updating cell flow we are more interested in the values of left neighbors than the right (see Section 4.3). However, the advantage is efficient storage of flow in all directions per cell, and also makes the definition of the pressure gradient between cells much more straightforward. See Figure. 4.

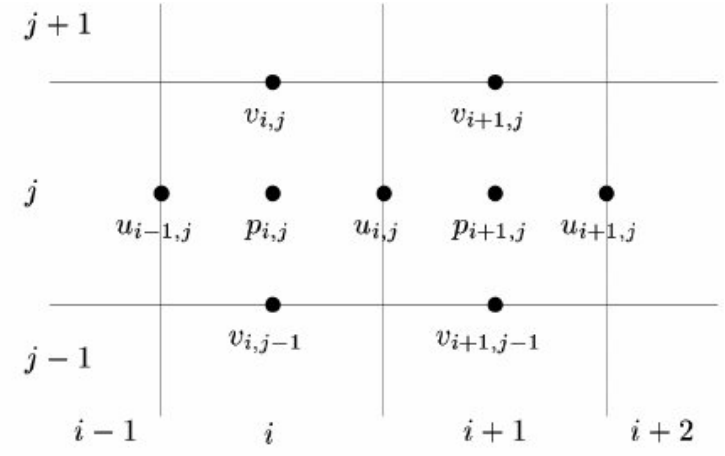


Figure 4: A 2D staggered grid arrangement, velocity or flow values u, v are located at the borders between cells. Original image taken from [SCC04]

4 Implementation

4.1 Buffers and Cell Definition

Often CA's are defined as a class or struct to which a number of data members belong, like mass and flow. We shall however consider a cell as the collection of data elements with the same index across buffers. Essentially adopting a Struct of Arrays approach (SoA), instead of an Array of Structs (AoS). This is mainly done to improve memory

performance, as typically rules do not require all cell information to be retrieved. For the fluid simulation the following buffers are allocated;

- pressure buffers P and $P0$
- flow buffers fx,fy,fz and $fx0$, $fy0$, $fz0$
- material buffers m and $m0$

Here $P0$, $fx0$, $fy0$, $fz0$ and $m0$ are typically the source buffers containing data from the current update, and P, fx, fy, fz and m are the buffers to which the values for the next update are written. These buffers are then swapped after each update. This double buffering is required so that cells do not read from already updated neighbors and use the appropriate old data.

Notation We will use the following notation to define the CA rules: Every rule is executed in the local neighborhood of a cell, therefore consider the location of the updated cell to be $(0, 0, 0)$ by definition. The range 1 neighborhood is then defined as $N = (-1, 0, 0), (1, 0, 0), (0, -1, 0), (0, 1, 0), (0, 0, -1), (0, 0, 1)$. Let u be the state of a buffer value in update u. We then express a cell property value as; $a^u_{(x,y,z)}$, which denotes the value of property a during update u at location (x, y, z) relative to the updated cell. Also let $a^u_{(0,0,0)} = a^u_c$ be a shorthand for the location of the updated cell itself and similarly a^u_i refers to value of neighbour i, where $i \in N$

To connect this notation with the previously mentioned data buffers in an example; for a cell with location in the world (x, y, z), $m^u(-1, 0, 0)$ refers to the value in buffer $m0$ at location $(x - 1, y, z)$, and $m^{u+1}_{(-1,0,0)}$ similarly refers to this location in buffer m.

Brick-Buffers

Additionally, for the top-level brick data structure (Section 4.5) the following buffers are maintained;

- brick mass buffer $brick_m$
- brick static buffer $brick_{static}$
- trash buffers $trash$ and $trash_{addr}$
- brick position buffers $brick_x$, $brick_y$, $brick_z$
- brick job buffers $brick_a$, $brick_{oa}$, $brick_{jobs}$

4.2 Signing Flow

A rule essentially expresses the change in state as a function of the state of a cell's neighbours and itself. For advection however, typically the state of a flow component is only interesting if it is pointed in the right direction. For example, when determining the outgoing mass of a cell, only flow components that point outwards from the cell are relevant. To prevent large-scale use of if statements and branching, the calculation is performed regardless of whether the flow component is actually pointed in the right direction. To make sure the correct final result is obtained, the contribution of the variable to the state is nullified if the flow component is not relevant. This essentially just comes down to setting the flow to 0 if it is in the wrong direction. For this purpose a helper function $s(f)$ is defined which returns 0 if the flow is outgoing or f if incoming:

$$s(fx_{(a,0,0)}) = \begin{cases} max(0, fx_{(a,0,0)}), & \text{if } a < 0 \\ min(0, fx_{(a,0,0)}), & \text{if } a > 0 \end{cases}$$

cases for $s(fy_{(0,a,0}$ and $s(fz_{(0,0,a)}$ are defined the same way.

4.3 Rules

The main pipeline of the simulation follows along the lines of traditional Eulerian fluid simulation, but where every step is implemented as a CA rule. These rules are executed sequentially for every cell, but the next rule is only executed after all cells have performed the previous rule.

Material advection

The material advection rule specifies that the new material of a cell is a function of the mass of it's direct neighbours, itself and the flows between it's neighbours. Specifically the sum of mass movement between each neighbour.

$$m^{u+1} = \sum_{i \in N} 0.5 * s(f_i) * m_i^u + \sum_{i \in N} 0.5 * s(-f_i) * m^u$$

In words, the new material of a cell is the sum of outgoing and incoming material across cell boundaries. The flow is multiplied by 0.5 to prevent oscillations. From this rule, it follows that as long as the mass of a cell does not become negative, the total mass in the system will be conserved. Such negative mass is prevented by restricting cell flow later on.

One reason for a MAC grid-style arrangement becomes apparent here; this advection scheme stores movement between cells instead of at the center of cells. The latter means that no movement gradient exists between two specific nodes. See [Sta99], which stores velocity at the center of nodes. In such a case the gradient is harder to allocate and diffusion can not be implicitly modeled. This solution, however, allows for compressible fluids, and can therefore also model gas at different pressures, and potentially be extended to phase changes.

Flow advection

The flow between cells moves matter along as described in the previous step, but in fluids, the velocity field also moves itself along, this is called self-advection and is essentially the momentum of the fluid travelling along with the material it belongs to, which is simply conservation of momentum.

To model this, flow advection is solved very similarly to the material advection, but includes some extra complexity. Momentum is equal to the movement of mass as per $p = mv$, therefore for every stored flow, start with the old momentum of that flow and subtract the outgoing momentum and add the incoming momentum. The new flow is therefore a function of all momentum flowing through the node and the change in mass of the cell:

$$fx^{u+1} = \frac{px^0 - px^o + px^x + px^y + px^z}{m^{u+1}}$$

where

$$px^0 = max(0, fx^u) * m^u_{-1,0,0} - min(0, fx^u) * m^u_{0,0,0}$$

is the old momentum across x,

$$p^o_x = \sum_{i=0}^{n} s(-f_i) * m^u$$

is the outgoing momentum, equal to movement of mass as in the material advection rule.

$$px^x = (s(f^u_{-1,0,0}))^2 * m^u_{-2,0,0} - (s(f^u_{1,0,0}))^2 * m^u_{1,0,0}$$

is the incoming colinear momentum across axis x from direction x,

$$px^y = fy^u_{-1,1,0} * m^u_{-1,1,0} * max(0, fx^u_{0,1,0})$$

$$+fy^u_{-1,0,0} * m^u_{-1,-1,0} * max(0, fx^u_{0,-1,0})$$
$$+fy^u_{0,1,0} * m^u_{0,1,0} * min(0, fx^u_{0,1,0})$$
$$+fy^u_{0,0,0} * m^u_{0,-1,0} * min(0, fx^u_{0,-1,0})$$

is the incoming orthogonal momentum in across axis x from direction y. p^z_x is obtained similarly but indexed in the z-component:

$$px^z = fz^u_{-1,0,1} * m^u_{-1,0,1} * max(0, fx^u_{0,0,1})$$
$$+fz^u_{-1,0,0} * m^u_{-1,0,-1} * max(0, fx^u_{0,0,-1})$$
$$+fz^u_{0,0,1} * m^u_{0,0,1} * min(0, fx^u_{0,0,1})$$
$$+fz^u_{0,0,0} * m^u_{0,0,-1} * min(0, fx^u_{0,0,-1})$$

Finding py^{u+1} and pz^{u+1} is then done in the same way, only with the relative coordinates swapped. Since there is a need to know the mass of source cells for incoming momentum, sometimes mass values beyond a direct neighbor must be accessed, i.e. $m_{-2,0,0}$ or $m_{-1,1,0}$, which is a Manhattan distance of 2. Therefore it is for this rule that the CA has a range 2 Von Neumann neighborhood.

To finish the new flow the global forces and acceleration can now be added. Since gravity g is an acceleration and not a force, it does not care about the mass of a cell and vertical flow can be increased as such:

$$fy^{u+1} = fy^{u+1} + g$$

Other accelerations in an arbitrary direction are similarly applied. When adding a force, simply additionally divide the force by the mass of the cell the force is moving out of as per $F = ma$.

Divergence

Divergence is the amount with which the simulation diverges from satisfying the density constraint and is found by taking the expected mass of a cell after applying the material advection rule and subtracting the maximum mass allowed in that cell. In other words, divergence is the amount of excess mass in a cell if we were to perform material advection with the current flow. At this point, cells have been updated to find m^{u+1} in the material advection step, in the next update with the current flows, m^{u+2} would therefore be cal-

culated. Therefore, simply apply the same material advection logic again to temporarily obtain the expected mass m^{u+2} and then find the divergence.

Often, Eulerian fluid simulators assume all cells are filled with liquid and only model movement of density through a liquid, in such a scenario cell density should always be 1, and a negative divergence, therefore, means a cell has less mass than it should have. However, in the presented solution we let a negative divergence simply denote available space, and do not try to solve for negative divergence. In other words the mass constraint is simply $m >= 0 \ \& \ m <= 1$, instead of $m == 1$. Divergence div is therefore:

$$div = m^{u+2} - 1.0$$

Pressure

To correct the flow values so that no cells can be compressed beyond maximum density, a pressure gradient is required such that any positive divergence is canceled out. Because pressure is the same as momentum over time, and the amount of mass needed to move to solve the divergence is equal to momentum, we require that the total pressure gradient for a cell with its neighbors cancels out the divergence:

$$\sum_{i \in N} \nabla P_i = \sum_{i \in N} \frac{P_i - P}{2} = -div$$

Typically here all values P form a matrix of unknowns which are acquired by solving a system of linear equations. However because this system is sparse (only 6 neighbors per cell), it is possible to keep calculations local and solve the pressure iteratively using a Gauss-Seidel approximation. This allows us to stick to the CA paradigm and only perform local operations. So for the pressure rule, the following rule is executed K times:

$$P^{k+1} = div + \frac{\sum_{i \in N} P^k}{\sum_{i \in N} solid(i)}$$

where P^k is the pressure in the current iteration k and $P^0 = 0$. It is not possible to have a pressure gradient to a solid cell, therefore to ensure this gradient is 0, it is simply removed from the equation. To this purpose $solid(i)$ is 1 if a neighbor i is marked solid, and 0 otherwise. The pressure of any solid cell is by definition always 0.

Note that since divergence can be negative, pressure can also be negative. This allows us to correctly and easily distribute pressure potential to empty cells across iterations, but this negative value must be set to zero before the pressure gradient step, therefore after finding the final pressure values, for every pressure value P_i:

$$P_i = max(0, P_i)$$

Pressure gradient

Now that each cell has determined its pressure, the change in flow using the pressure gradient rule can be determined. This rule simply takes the difference between two cell pressures to determine the force that is applied to the flow.

$$\nabla P_i = \frac{P_i - P}{2}$$

$$f_i^{u+1} = f^u + \nabla P_i$$

Finalising flow

Momentum is maintained, but small inaccuracies in solving divergence can build up over time and convert acceleration introduced by gravity into more energy, causing the simulation to build up more momentum over time in a positive feedback loop. To prevent this from occurring the flow is slightly dampened by multiplying with a parameter $0 \leq d \leq 1$. For our simulations, $d = 0.01$ has been empirically chosen, but this value can be experimented with:

$$f_i = f_i * d$$

To finalise the corrected flow, we make sure that it does exceed the maximum allowed and therefore clamp the flow to not exceed maximum mf (see Section 4.4);

$$f_i = clamp(f_i, -mf, mf)$$

World Set

The final rule simply converts the amount of material in a cell to a color and stores it in the datastructure used by the raytracer. This color is determined by linearly interpolating the color between full maximum density ($rgb(0, 0, 1)$) and minimum density ($rgb(0, 0.5, 1)$).

$$voxel = \begin{cases} rgb(0, \frac{(1-d)}{2}, 1), & \text{if } d \geq \epsilon \\ rgb(0, 0, 0), & \text{otherwise} \end{cases}$$

where d is the material density, $\epsilon = 0.0001$ is some threshold variable and $rgb(r, g, b)$ is a function converting r, g and b components to the 4-4-4 12-bit voxel color format of the template. The template's renderer considers 0 to denote an empty voxel instead of pure black.

Gas pressure

While the scope of this research has been restricted to liquids, gasses are also able to be simulated in the presented framework. Gasses, unlike liquids, are compressible and thus we do not find their pressure based on the amount of divergence. Instead, the pressure inside a cell filled with gas is a function of the material in the cell. Typically for non-extreme scenarios, the ideal gas law can be used;

$$P = nRT$$

where P is the absolute pressure of the gas, n is the amount of material, R is the gas constant and T is the temperature. Since n is known, and we can take some arbitrary temperature $T = 20$ (or implement heat as a rule), gas pressure can be easily introduced, and this is all that is needed to simulate compressible gas in this CA framework. However, for gasses, it is recommended to use coarser grids due to higher flows, like when modeling an explosion by creating an area with very high gas pressure. Such a scenario would cause high-pressure gradients to propagate outwards over long distances and in all directions, therefore requiring high update rates and or relatively large cells.

4.4 Flow clamping

The advection rules as described have two requirements:

- A single flow value can be no larger than 1 as flows above 1 are not defined because they would no longer index into a direct neighbor and create additional mass.
- The total sum of incoming and outgoing flow must be smaller than -1, as mass can not become negative.

If either scenario occurs then this is because there exists too much energy or pressure in the system that can be processed with the given update rate. A solution to this is to either increase the update rate or lower the forces acting on the fluid (e.g. gravity). To make sure an invalid scenario never occurs regardless of the update rate or gravity, the flow is clamped as part of the pressure gradient rule as described previously. To make sure the second scenario never occurs specifically, it is required that $mf \leq \frac{1}{3}$. This value arises from the fact that half of the flow in the material advection rule is applied, and the total sum of material flow can not eclipse 1, thus summed over all 6 neighbors at most $6 * \frac{1}{3} * 0.5 = 1$ material can exit a cell. While this is an easy fix and it ensures the simulation can not blow up, the maximum flow rate of the water is limited in many cases significantly below the maximum, which causes a high numerical dissipation or loss of

momentum, making the fluid appear more viscous, while also increasing the "dispersion" effect.

Another solution is to dynamically find an appropriate update rate or somehow prevent the second scenario from occurring without using clamping. Several attempts were made to prevent clamping below a flow of 1 but were unsuccessful. This is considered the main deficit in this simulator and the effects can be seen in Section 5.

4.5 Data structure and memory management

Each of the buffers contains 32-bit values with the exception being the $brick_{static}$ buffer, which contains 8-bit chars. Therefore each cell requires 320 bits for the 10 buffers containing its simulation data, and 296 bits for each brick. For 4^3 sized bricks this gives us $296/256 \approx 4.6$ bits per cell, or $296/256 \approx 0.57$ bits per cell for 8^3 bricks. As such, by distributing the brick data over each cell, we find a memory usage of 324.6 bits per cell when using 4^3 bricks and 320.57 bits per cell when using 8^3 bricks.

When working in 3 dimensions memory usage grows cubically with grid size (in terms of the longest axis). A 200^3 grid contains 8 million cells, and with each costing 324.6 bits, this results in a memory requirement of 324.6 Megabytes, a 400^3 grid requires about 2.6 GB, and a 1024^3 grid requires 43.5 GB. While modern GPUs have increasingly large amounts of memory, typically having at least 6 GB with up to 24GB on high-end devices, allocating memory for all cells in the 1024^3 voxel world is prohibitively expensive. While the memory requirement per cell could potentially be almost halved by using half floats, which appears to still provide sufficient accuracy for our purposes, feasibly one can not simulate much more than 200^3 cells at interactive rates (see Section 5) either way. Typically only a fraction of the full world will contain a fluid, the overall updates performed on cells not containing fluid are a waste of memory and computation time, so preventing storage and processing of such cells is desirable.

The fluid simulator dynamically allocates and processes bricks of voxels only for relevant locations. To determine whether a brick should be allocated, a simple brick-level cellular automaton is used with the following rules:

- A brick is alive if contains mass
- brick is alive if any of its neighbours contains mass

- Otherwise a brick is not alive

A brick that is not alive is not allocated in memory and is therefore also not visited during the update step. This system reliably allocates bricks when necessary and the second rule ensures cells never try to access a cell in a non-existing brick. To handle the boundary of the world, an additional layer of empty bricks is allocated around the borders of the world, but marked as static and skipped in the update loop. This means no additional boundary checking needs to be included during each cell update, at the cost of some extra memory.

Dynamic allocation of the bricks works as follows: Bricks are stored in memory as sequential numbers that represent the individual cell data. To index into a given cell, one therefore must multiply the index of the brick by the brick size. For every brick a status is maintained in $brick_{static}$, indicating whether it is alive or dead.

The number of bricks that are alive and the highest index in our brick buffer in which a brick has been allocated are tracked. Whenever a brick is killed, its location in the brick buffer is added to $trash$ and $trash_{addr}$, and the brick is skipped during iteration. When a new brick has become alive, it is either initialized into a location provided by the trash list (brick is recycled) or added to the end of the buffer. Whenever the buffer runs out of memory to allocate new bricks, the entire set of brick buffers doubles in size and is reallocated. Similarly, if the buffers become sparse, (only a 4th of the bricks are alive) it is compressed where all bricks are loaded sequentially into a buffer half the size. This reallocation is expensive, but a rare operation and reduces unnecessary brick visits and memory usage.

This straightforward approach allows a user to spawn fluid at any location in the world without having to manually define a fluid domain and it easily keeps track of only the relevant bricks for updating. However while the second rule guarantees relevant cells are always alive, especially for larger brick sizes, lots of empty bricks may still be initialized and updated. Larger brick sizes do have the advantage of better data locality inside the brick as relatively fewer cells are inside brick faces, but the 4^3 brick size was used in the presented simulations as this typically significantly reduces the number of updated cells, particularly in scenarios where the water has a lot of surface area relative to its volume.

4.6 GPU implementation

GPGPU (General-Purpose computing on Graphics Processing Units) allows us to utilize the fast computing power of graphics hardware to accelerate our CA. Therefore the simulator was originally developed on the CPU and then ported to the GPU using OpenCL [The13]. While being developed for the CPU, a GPU implementation was intended from the start, and as such, the CPU version as such was designed with that in mind and the conversion was relatively straightforward. Running a single kernel on the GPU is analogous to performing a single rule, where every thread solves the rule for its assigned cell. For each rule, a number of tasks equal to the number of active cells is assigned. We make sure each warp or wavefront, which is a group of typically 32 threads executing the same instruction at the same time, aligns neatly with a brick. This means that threads in a warp will always fall inside the same brick and when threads initially check if it is inside a static brick, all threads in the warp terminate at once, preventing branching costs.

Dynamic Memory allocation on the GPU is not available in OpenCL 2.0 and generally involves complex logic, therefore the brick update step is kept on the CPU. The brick update step makes sure that the bricks sent to the GPU don't end up being too sparse, i.e. most bricks in the buffers are alive, so not too many unnecessary brick visits are performed. To avoid unnecessary communication, all cell data is kept on the GPU, and data is only transferred between devices on the brick level. When bricks need to be initialized or freed, the CPU sends the jobs to the GPU using pinned job buffers in $brick_a$, $brick_oa$, and $brick_jobs$. These buffers contain all information (new address, old address, and job type) needed to initialize or free bricks. The GPU then executes these jobs to free and initialize the bricks and their cells in the reserved buffers device side. This eliminates large buffer copies. Full buffer transfers only occur when the brick buffers are reallocated, which is a relatively rare occurrence.

4.7 Rendering

The focus of this project is not to implement new rendering logic, therefore at the end of every simulation step, the material in cells is converted to a solid color and stored in the primary ray tracing voxel structure. At this point, the simulation has finished an update and returns. The rest of the voxel engine then does its work and later calls the renderer. That said, another advantage of the discussed brick data structure is that is very suitable for rendering and can double as an effective ray tracing structure. In fact, the voxel raytracer uses 8^3 bricks with a single top-level grid and an additional GPU only 32^3 ubergrid that is rebuilt each frame. This structure has been shown to be a powerful

tool for high speed raytracing and is similar to the data structure presented in [Hoe16]. While algorithmically the voxel octree, that has seen some popularity seems to be a superior data structure, in reality, the octree typically incurs too many traversal steps and makes updating too complicated.

Because the simulation data structure can be efficiently traversed and contains material densities for every cell, it should therefore be very suitable for volumetric ray casting, although for this project no volumetric ray tracing has been implemented. A volumetric ray tracer may prove to also greatly increase the visual quality of the simulation and may particularly help with the dispersion effect, as it should more intuitively be able to represent those areas as lower density.

4.8 Required update rate

In a local CA paradigm water can only move a single cell per update and typically does not flow at a maximal speed between cells, to simulate fast-flowing water a similarly fast update rate is required. It is somewhat subjective as to what is the appropriate update rate U, and this is left mostly as a parameter.

In general, if we were to define a cell to have a size of 1 meter, then to simulate water flowing at 10 meters per second, at least 10 updates per second are required, or similarly, if we define a cell as 0.1 meters in size, at least 100 updates per second would be required. In general, empirical observations find that for decently fast-flowing water in a grid large enough to provide enough detail, 100 updates per second is a good target to aim for, with 50 updates per second the minimum recommended amount. In the presented simulation results and benchmarks, the CA is simply updated every frame, which is alright since the voxel engine easily runs above 100 fps. However, it is possible to run multiple updates of the CA for each frame if a faster simulation is required. See Figure. 5 for an overview of the simulation pipeline.

4.9 Optimizations

Performance enhancements were kept in mind from the start of development and most have already been mentioned like the brick update structure and SoA approach. Additional optimizations were applied by reusing unused buffers for swapping to temporarily store information used in a sequential rule, reducing the need to make a dedicated buffer and improving memory usage.

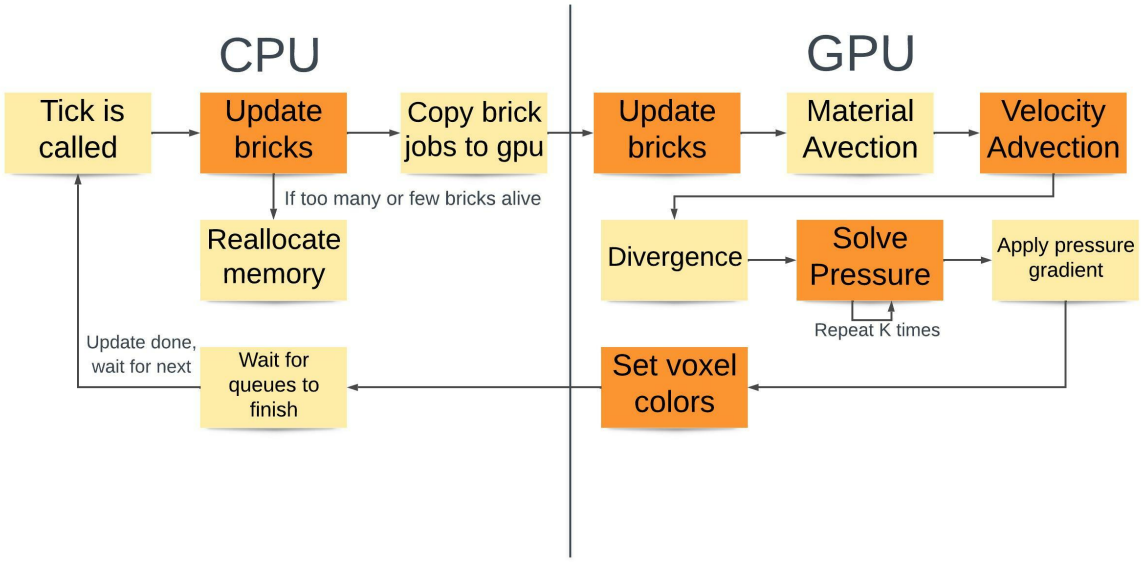


Figure 5: The simulation pipeline

While cell locations have been abstracted behind the brick data structure, whenever values associated with the updated cell itself are accessed, its global location is already known and direct indexing can be used when setting and getting values, in which case no additional indexing costs are incurred.

Furthermore small low-level optimizations to the rules have been performed, but ultimately the running times are typically dominated by memory accesses, as relatively little computation is performed per memory access. Latency hiding by means of context switches on the GPU alleviates this problem somewhat, but not much more room for improvement remains with regards to faster computation. Further improvements to runtime should therefore focus on improving memory utilization.

5 Results

In this section, the evaluation of the implemented CA system is presented in the context of our initial requirements. To this purpose, we check for supported visual features of the simulation, perform a benchmark and compare it with other fluid simulations. We also consider the complexity and stability of the framework. See 6 for a large scale simulation example.

Analysing the visual quality of a voxel fluid simulation is not entirely straightforward. The level of visual realism required is quite low for a real-time retro setting, and voxels

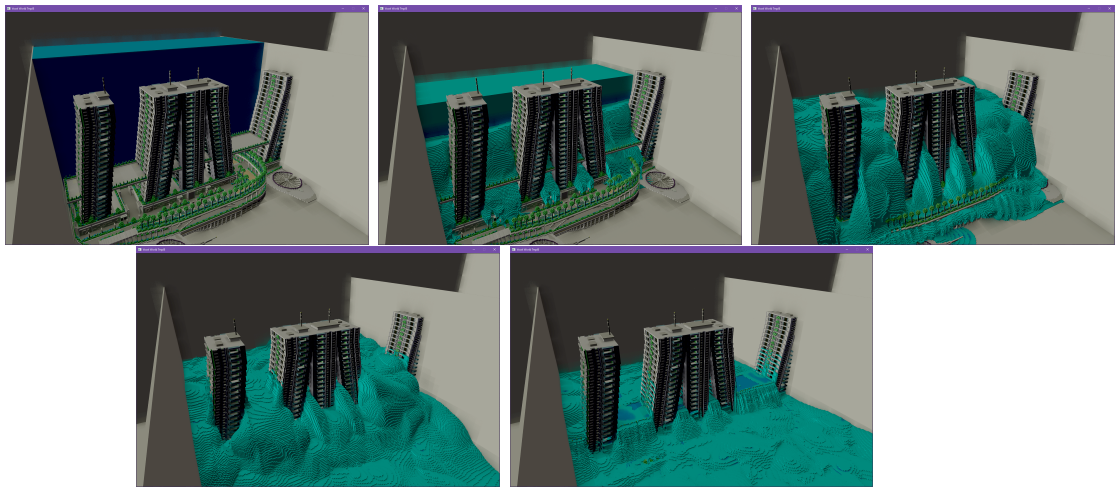


Figure 6: Tsunami scenario, a large block of water flows downstream and hits a few apartment buildings, see Table. 1 for benchmark numbers

themselves are not a natural visual presentation of water. Therefore, the solution is evaluated as follows:

- Property analysis: Test if the simulation expresses a certain feature of real water, e.g. pressure propagation, conservation of mass, and incompressibility.
- Visual comparison with a voxelised state of the art simulator to capture visual differences with regards to the conservation of momentum and viscosity.
- Benchmark performance over a number of scenarios and resolutions
- Compare performance against similar high-performance fluid simulator at the same resolution

5.1 Water properties

Two important properties of water are conservation of mass and incompressibility, i.e. divergence. We can calculate these values over the grid, as well as observe that the body of water maintains a consistent volume. We find that mass is perfectly preserved, aside from some fluctuations from floating point errors. Mass preservation can also be derived from our mass advection rule. More attention is paid to the divergence of the fluid, which may reach a couple of percent, depending on the height of the water mass, the strength of gravity g, and the number of iterations used in the pressure solver, see Figure. 7.

We find that there seems to be a linear relationship between divergence, water height, and K. While the observed percentages are low enough that it is quite hard to visually

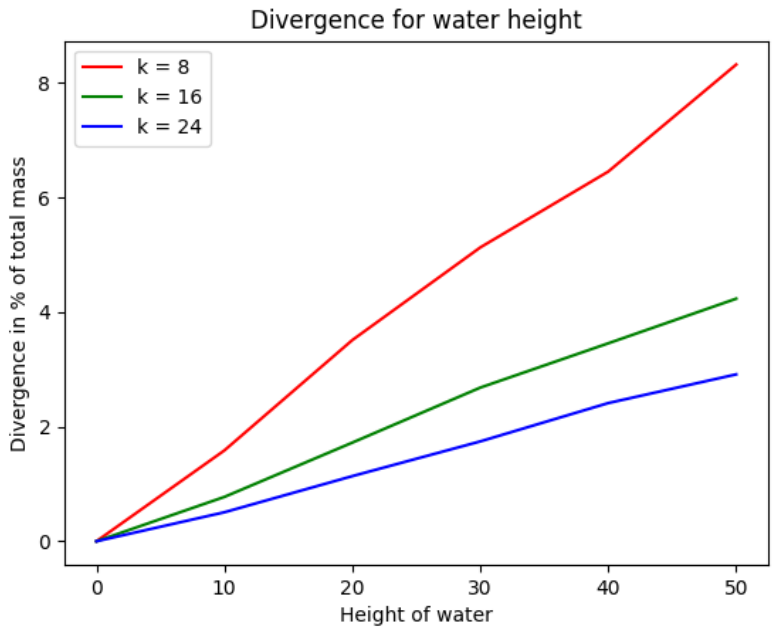


Figure 7: A graph showing the amount of divergence at different water levels at various K

observe that water has been compressed. In scenarios where a large body of water uniformly is being compressed, it is somewhat visible at a divergence $\geq 5\%$. Furthermore, allowing higher amounts of divergence can allow divergence errors to accumulate which essentially results in internal pressure waves inside the fluid that artificially create momentum, a divergence below 4% has empirically been determined to be an acceptable amount as compression is barely visible and the fluid settles appropriately.

Water pressure causes water to level out over time, even if it has to travel through an indirect path, like two boxes being connected by a pipe at the bottom. See 8 for an example of water leveling out after traveling through a tube. In this example, it is clear that our simulation does present this property of water.

5.2 Comparison with Mantaflow

In order to allow for a more visually distinct evaluation of the behavior of our water, we compare it with another fluid simulator. Mantaflow [TP18] is a state-of-the-art fluid simulator recently provided with Blender [Com18]. The simulator includes an implementation of FLIP as described in Section 2.8. Because this simulator is of high quality, particularly relative to our demands, it essentially serves as the ground truth for our evaluation purposes. To facilitate direct visual comparison, comparable simulation scenarios in the CA simulator and Mantaflow have been set up by duplicating the relative size of objects

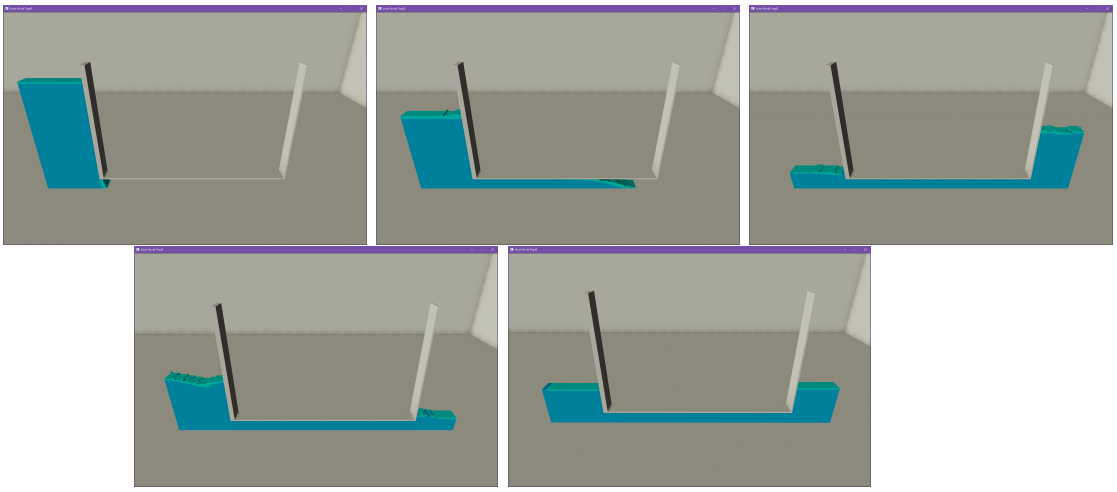


Figure 8: Water levels out after traveling to a tube at the bottom, initially the water oscillates due to build-up momentum, eventually settling down at equilibrium.

and the domain. Blender then interpolates the particles of the simulation to voxels, with approximately the same resolution as used in our simulation. The voxelisation process, on top of the high-resolution fluid simulation, is quite slow in Blender and has to be rendered offline, but allows us to perform decent a visual comparison.

Note that the simulations are still subject to many parameters that do not necessarily have a direct analog between the simulations and the chaotic nature of fluids (i.e. butterfly effect) makes it so that the scenarios will not be perfectly comparable, in addition to differences in rendering and camera settings. Therefore the following images serve as a comparison of the overall high-level behavior of the fluid, and these differences should be kept in mind.

We have picked 4 simulation scenarios that were found to demonstrate the breadth of similarities and dissimilarities between the methods, see Figures. 9, 10, 11 and 12.

We make the following observations regarding the first comparison (Figure. 9), in which a cube of water is dropped in a square box:

- Our simulation appears more viscous and loses momentum more quickly (due to high numerical dissipation), which is particularly visible in the difference in height that the fluid manages to climb.
- Our simulation is more smooth and displays symmetry, likely due to Von Neumann neighbourhood and a perfect initial scenario, in comparison to the chaotic nature of particles.
- Our simulation shows the dispersion effect as expected; the fluid has no discrete

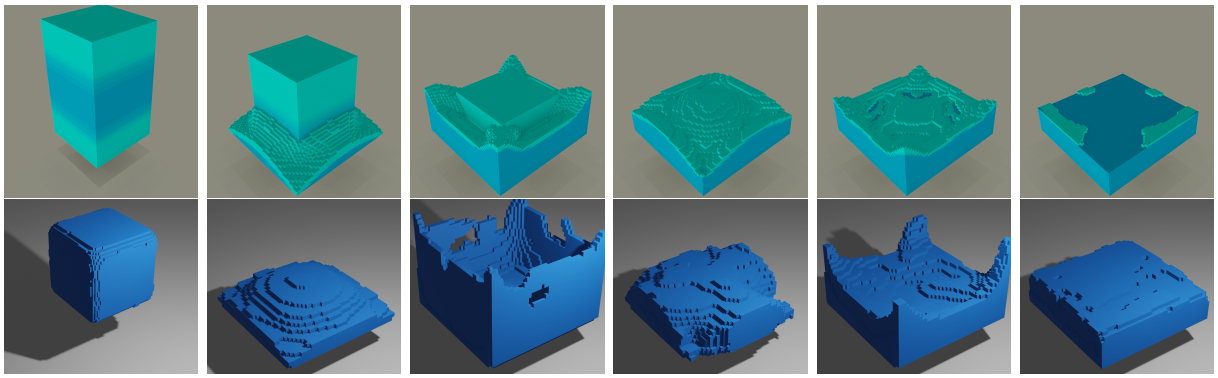


Figure 9: WaterCube: A comparison over time between our method (top 2) and voxelized Mantaflow (bottom) simulating a cube of water being dropped in an invisible box

location and gets stretched by global acceleration, while the Mantaflow simulation does not display this behaviour and the cube stays together before it hits the ground (see first image).

- Our simulation presents the same general behavior as the Mantaflow simulation, the formed waves follow along similar lines, other than the aforementioned differences.

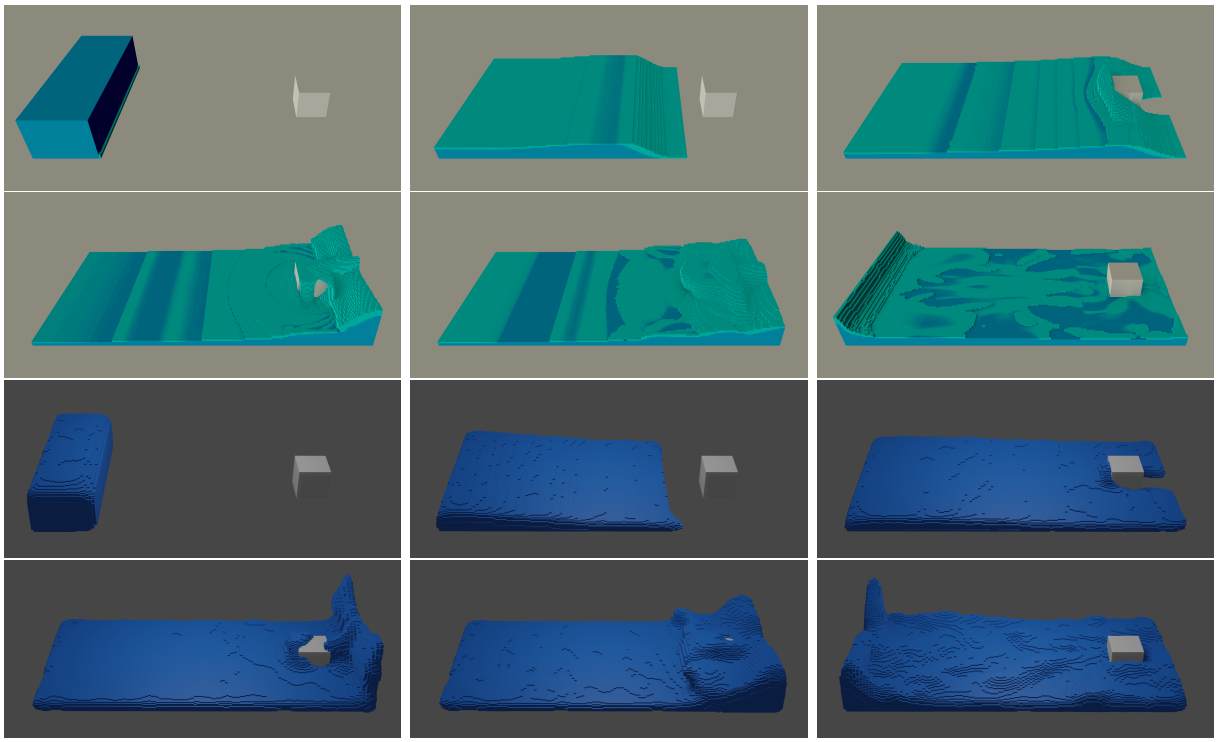


Figure 10: DamBreak: A comparison over time between our method (top 2 rows) and voxelized Mantaflow (bottom 2 rows) simulating a block of water suddenly being free to flow towards a cubic obstacle.

We make the following observations regarding the second comparison (Figure. 10), in

which a block of water crashes down a rectangular box and collides with a cube:

- While the water flows downhill, it does not spread as thin as the Mantaflow simulation, this is due to the restricted flow as the fluid moves at maximum speed regardless of the height of the wave.
- Again the CA simulation is more smooth and displays more symmetry than Mantaflow
- Our simulation presents the same general behavior as the Mantaflow simulation, the first contact with the cube (image 3), particularly how the water splashes against the back wall, then meets in the middle and is pushed towards the cube (image 4), and finally our simulation also fully travels to the left wall and climbs it (image 6).

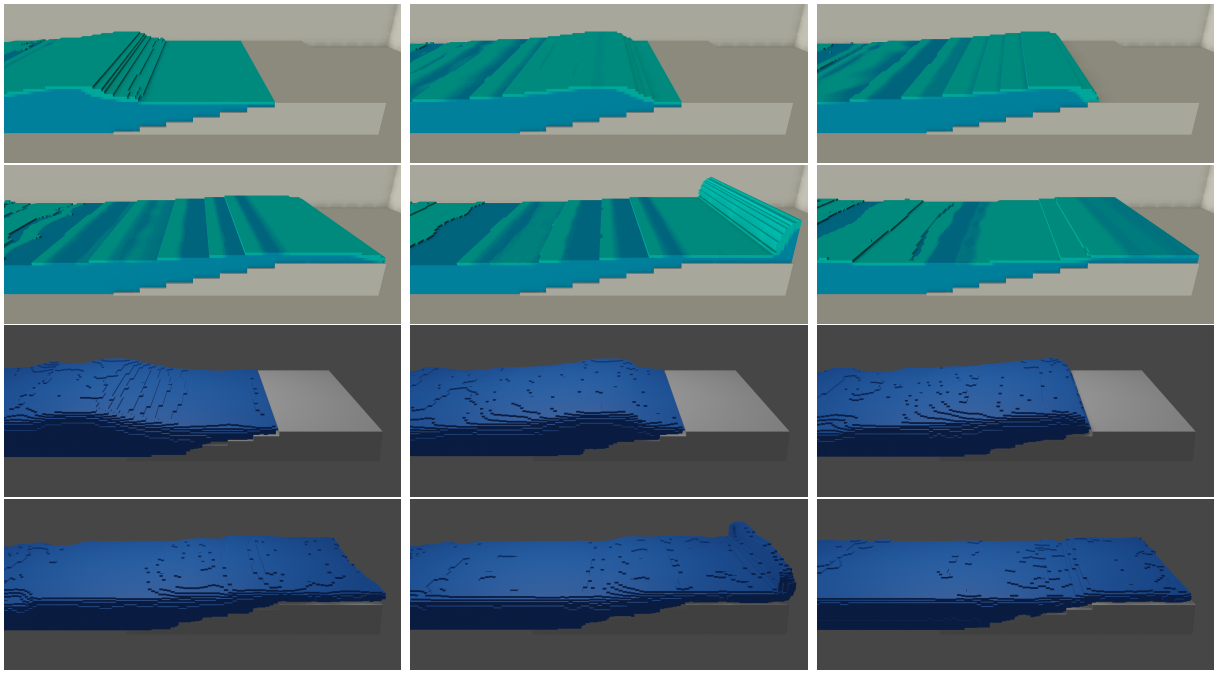


Figure 11: ShallowWave: A comparison over time between our method (top 2 rows) and voxelized Mantaflow (bottom 2 rows) simulating a wave moving into a shallow area

We make the following observations regarding the third comparison (Figure. 11), in which a wave traveling in shallow enters a shallow area:

- The wave in the Mantaflow simulation more smooth as it travels over the water in picture 1, the same phenomena as observed in the previous comparison.
- the behavior of the fluid is again comparable in both scenarios, as the water approaches the shallow area, the wave slows down and shortens (images 1-3) it then collapses (image 4) and hits the back wall and retreats (images 4 and 5).

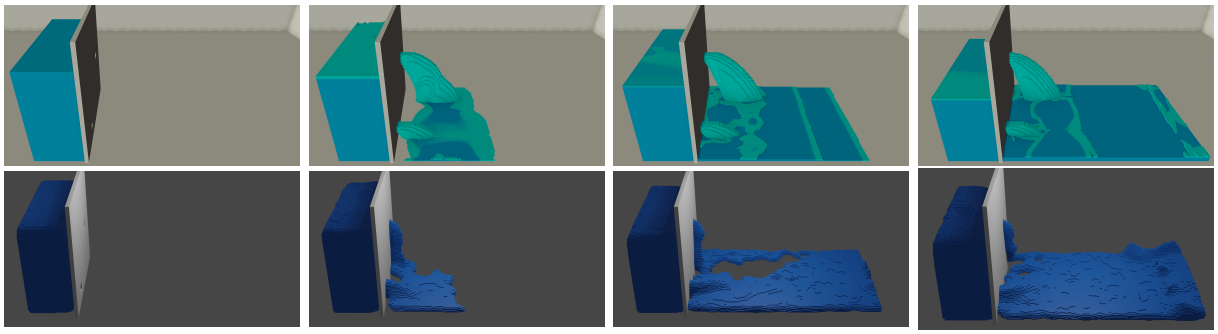


Figure 12: DamHole: A comparison over time between our method (top) and voxelized Mantaflow (bottom) simulating water flowing through two holes in a dam.

In the final comparison (Figure. 12) water exits a dam through two holes at different heights. Here the biggest difference between the methods can be observed, particularly due to the CA's flow clamping. Water flow through the holes quickly maxes out in the CA simulator for a relatively small water height. In contrast, the Mantaflow simulation more clearly shows that the top hole has a vastly reduced flow rate compared to the bottom hole. This consequently means there is little difference between the flow rate at the two different holes in the CA solution, and less momentum is carried to the far wall.

In general, the visual quality of the water is mostly satisfactory, particularly within the requirements of a retro voxel engine, however, it can not approach real fluids as closely as Mantaflow, and its viscous nature prevents chaotic and splashy effects to a certain degree.

5.3 Benchmark results

We perform a performance analysis by benchmarking a few simulation scenarios and also compare our system with another optimized GPU Grid-based fluid simulation run over a grid with the same resolution. See Table. 1 for benchmark numbers of the simulator as a whole and individual rules. The simulations are run on a system containing an AMD R9 6900XT GPU with 16GB of GDDR6 memory and Intel i7-6700k CPU with 16GB of RAM. Measured Execution times were averaged over 1000 updates. External costs of rendering and other template operations outside of the simulation were not included. The latter means that the number of updates per second presented is the theoretical updates per second if no other work is performed given the measured update time.

Performance relative to CPU

No extensive measurements were made of the original multi-threaded CPU implementation, particularly since it was not heavily optimized and outdated. However to give a rough idea of the performance difference; it was capable of updating around 5-15 million

	Water Cube	Dam Break	Dam Hole	Tsunami
Updates per second	2932	989	897	43
Peak Active Cells	130176	627248	730048	11294784
Peak Memory	5.28	25.45	29.62	458.29
Update time	341	1011	1114	23245
Brick Update	3	9	11	131
Material Advection	22	67	80	1466
Flow Advection	110	312	340	7177
Divergence	22	43	49	946
Pressure Solve	158	492	532	11433
Pressure Gradient	18	66	56	1288
World Voxel Set	8	22	45	804

Table 1: The Updates per second, memory usage (in MB), and average execution times (in nanoseconds) for all rules combined and individually.

cells every second. The presented GPU implementation on the other hand can update around 500 million cells per second as can be seen in Table 1. This would suggest a 30x performance increase at least. It is feasible that the CPU implementation could be optimized significantly, but this still indicates that the GPU implementation does provide a significant speedup.

5.4 Performance Comparison with a 3D Eulerian fluid simulator

The computational work performed in Eulerian fluid simulation typically scales linearly with the number of cells in the grid regardless of the simulated scenario, or in other words the solution. In that sense comparing performance between simulators is quite straightforward as we simply need to run them with the same resolution. As such the performance of this method can be loosely compared with other work that provides performance numbers for a given resolution like [ZKM16], [Hei+17].

However, to properly take hardware out of the equation we benchmark compare the performance of the presented CA fluid simulation with another optimized GPU Eulerian fluid simulator; GPU-GEMS-3D [Scr20] which is run on the same machine. It is a c# Unity [Uni22] implementation based on the GPU-GEMS article [Ngu07]. Which in itself is based on the semi-Lagrangian approach as previously explored in Section 2.7. While this simulation is visually not very comparable to the presented method, because it models smoke and uses volumetric ray tracing to render, the type of work performed is very similar. Like our CA solution, this simulator must perform advection over a grid and prevents divergence by solving pressure iteratively.

Because of the dynamic data structure in the CA simulator, to make sure that the same amount of cells is in fact being processed, we initialize a solid block of fluid so the simulation does not skip cells. Both simulation methods are run with 8 iterations for the pressure solver and again perform 1000 updates for each measurement. We only consider the simulation pipeline execution time and do not consider rendering time, see Table. 2.

	32^3	64^3	128^3	256^3
CA simulator	104	530.52	3400	27313
GPU-GEMS-3D	154	932	6221	44323

Table 2: The execution times in nanoseconds, for both simulators at given resolutions

We find that our simulation updates about twice as fast at the same resolution. Additionally, with the extra flexibility of dynamically adapting memory and resolution, in many scenarios, our method may be even faster as it typically does not update all cells within a given simulation domain. Keep in mind however that this comparison is meant to evaluate the per-cell performance and confirm our simulation keeps up with what is expected for an optimized GPU solution. Therefore, it can not be concluded that our simulation actually looks faster, as the modeled effects differ and the GPU-GEMS-3D solution can typically use a relatively high time step, only requiring around 10 updates per second, while the CA simulator prefers up to 100 as mentioned in Section 4.7.

5.5 Simulation complexity

From a user experience standpoint, where the user is a novice programmer that wants to initialize some water into the world the system is relatively easy to utilize. Because the CA dynamically allocates resources and handles cells analogously to the main 3D voxel-screen, the user can spawn fluid just like they would a voxel, at any location without having to worry about manually defining an appropriate simulation domain or collidable objects.

There are a number of parameters to the simulation that a user may want to tweak but is not required to. These are previously mentioned; pressure iterations K, update rate U, gravity strength g, and flow dampening d. The simulation will work out of the box with the default values just fine, and tweaking these parameters is relatively intuitive. The simulation is also robust in the sense that it can not be easily broken, or be made to blow up or crash. A user may however spend some time tweaking the simulation to perfectly fit their wants, as the update rate U and amount of flow dampening d can significantly

change the visual flow speed. Additionally, it is assumed that the user will be somewhat careful with how much and where they spawn the water with regard to performance.

6 Conclusion

The aim of this work was to research and develop real-time physics simulation for use in a retro voxel engine without local transformations or particles. The requirements of the system were that it should be robust and intuitive, fast enough to be real-time, add little complexity to the existing voxel template and look plausible, while ideally providing as many effects as possible.

A number of approaches were researched and ultimately, a CA approach was selected. A purely cell-based fluid simulation was implemented that is run on the GPU. The system simply fits as an add-on to the voxel template and requires little work to set up. It dynamically allocates memory and abstracts the underlying data structure, so that indexing retains the voxel template's promise of providing an intuitive 3D voxel 'screen', although tweaking the simulation may take some effort.

While the evaluation of the project was limited to water simulation, the provided framework still supports arbitrary extension and simplicity thanks to the CA paradigm. The visual quality and numerical accuracy of the simulation is inferior to state-of-the-art particle-in-cell simulation like FLIP, but provides satisfying effects considering a voxel environment, and the correct dynamics expected from a fluid, like pressure and momentum. It as such improves upon other CA fluid methods like [Med18] and [Hei+17] by introducing better flow and momentum modeling while sticking to a pure CA paradigm. This approach also allows the solution to serve as a base framework that can be easily extended with extra effects or rules, like heat conduction and multi-mixture materials.

6.1 Future work

There remains much room for extensions to our method, specifically since the CA paradigm should hopefully allow for easily specifying additional rules and logic. However, the current fluid simulation could also be further improved, particularly with regard to flow advection and clamping.

Improving advection

Clamping flows ensures that flow remains local to direct neighbors and mass can never

be negative, however, the most visual quality is achieved when flow nears 1. A flow of 1 means reduced dispersion as technically the flow will in such a case perfectly describe the movement of a particle from the center of one cell, to the center of another, and when no clamping is applied we do not dissipate momentum. Therefore finding a way to ensure clamping can be prevented or anticipated, can significantly reduce the viscous behavior of the fluid and provide a visually faster simulation for the same number of updates.

Additionally, it may be desirable to improve on the advection further using a non-linear scheme, like mac-Cormack advection, or introducing vorticity confinement to the simulation, which would help reduce the smoothness of the simulation. Alternatively, some randomness may be introduced to the system to prevent neatly aligned geometry from creating perfectly symmetric patterns, which is not technically incorrect behavior, but uncharacteristic for the chaotic nature of turbulent flows which arise from the smallest of imperfections in real-life scenarios. That said, such improvements may be wasted in the presented voxel template scenario as such details may not be very visually distinct.

Further optimization

As previously mentioned, memory accesses are the primary bottleneck in the simulation. Latency hiding on the GPU helps to alleviate high latency, but data locality is an issue on the borders of bricks and abstraction requires additional indexing operations. An interesting solution would be to split brick processing into two passes: A core pass operating only on internal cells (e.g. not on the faces of a brick) and a second pass for the faces of a brick. This allows for direct indexing for the core cells and good data locality. Similarly, apron voxels like those used in [Hoe16] may be an option to remove index mapping completely and could also decrease memory usage significantly by eliminating the need for double buffering, as lockstep GPU thread execution can give us the certainty that data is read before it is updated in the same buffer, given that the borders between warps are correctly handled. Similarly ordering bricks in memory, or using inverse, or neighbor mapping between bricks as described in [Hoe16] could further alleviate indexing costs and improve data locality.

Since solving the pressure is an expensive step, we could adopt an early-out scheme to avoid performing iterations that are not worthwhile. To accomplish this one would observe the maximum change in pressure after each iteration and when this falls below a certain threshold, cease iterating. An additional advantage would be that this removes the need for manually selecting and pressure iteration count K. However, it is not immediately apparent how kernel executions can be efficiently planned to account for this on the GPU, however recently introduced Dynamic Parallelism in GPGPU frameworks like

OpenCL 2.0, which allows kernels to be launched device side may be the solution.

Extending physics

The requirements stated that we wished to simulate as many effects as possible, this includes many effects beyond water, to that end, instead of a single pair of material buffers, any number of material buffers could be created, and a corresponding material information object would carry the relevant properties. Consequently, other rules can then model multi-material effects, like buoyancy and heat transport, like in [Hei+17]. Furthermore, phase changes and explosions, or even conduction of electricity are effects that could be added, to, for example, approach the breadth of effects modeled in [tan08] and previously seen Lagrangian solutions like [Mac+14].

BIBLIOGRAPHY

[Gam70] Mathematical Games. "The fantastic combinations of John Conway's new solitaire game "life" by Martin Gardner". In: *Scientific American* 223 (1970), pp. 120–123.

[Vic84] Gérard Y. Vichniac. "Simulating physics with cellular automata". Dutch. In: *Physica D: Nonlinear Phenomena* 10.1-2 (1984), pp. 96–116. DOI: 10.1016/0167-2789(84)90253-7.

[BR86] Jeremiah U Brackbill and Hans M Ruppel. "FLIP: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions". In: *Journal of Computational physics* 65.2 (1986), pp. 314–343.

[Mon92] J. J. Monaghan. "Smoothed Particle Hydrodynamics". Dutch. In: *Annual Review of Astronomy and Astrophysics* 30.1 (1992), pp. 543–574. DOI: 10.1146/annurev.aa.30.090192.002551.

[SZS95] Deborah Sulsky, Shi-Jian Zhou, and Howard L. Schreyer. "Application of a particle-in-cell method to solid mechanics". Dutch. In: *Computer Physics Communications* 87.1-2 (1995), pp. 236–252. DOI: 10.1016/0010-4655(94)00170-7.

[Sta99] Jos Stam. "Stable fluids". In: *Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH '99* (1999). DOI: 10.1145/311535.311548.

[Eva01] Kellie M Evans. "Larger than life: Digital creatures in a family of two-dimensional cellular automata". In: *Discrete Mathematics & Theoretical Computer Science* (2001).

[FSJ01] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. "Visual simulation of smoke". Dutch. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH '01* (2001). DOI: 10.1145/383259.383260.

[Ila01] Andrew Ilachinski. *Cellular automata: a discrete universe*. World Scientific Publishing Company, 2001.

[Tem01] Roger Temam. *Navier-Stokes equations: theory and numerical analysis*. Vol. 343. American Mathematical Soc., 2001.

[LY02] Xia Li and Anthony Gar-On Yeh. "Neural-network-based cellular automata for simulating multiple land use changes using GIS". In: *International Journal of Geographical Information Science* 16.4 (2002), pp. 323–343.

[TF02] Dante Treglia and Forsyth. *Game Programming Gems 3*. Culemborg, Nederland: Van Duuren Media, 2002.

[EKS03] Olaf Etzmuß, Michael Keckeisen, and Wolfgang Straßer. "A fast finite element solution for cloth modelling". In: *11th Pacific Conference onComputer Graphics and Applications, 2003. Proceedings.* IEEE. 2003, pp. 244–251.

[GHD03] Olivier Génevaux, Arash Habibi, and Jean-Michel Dischler. "Simulating Fluid-Solid Interaction." In: *Graphics interface*. Vol. 2003. Citeseer. 2003, pp. 31–38.

[MCG03] Matthias Müller, David Charypar, and Markus Gross. "Particle-based fluid simulation for interactive applications". In: *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Citeseer. 2003, pp. 154–159.

[Niv03] Nival and 1C Company. *Silent Storm*. 2003.

[XWK03] Xiaoming Wei, Wei Li, and A. Kaufman. "Melting and flowing of viscous volumes". Dutch. In: *Proceedings 11th IEEE International Workshop on Program Comprehension* (2003). DOI: 10.1109/casa.2003.1199304.

[GBO04] Tolga G. Goktekin, Adam W. Bargteil, and James F. O'Brien. "A method for animating viscoelastic fluids". Dutch. In: *ACM SIGGRAPH 2004 Papers on - SIGGRAPH '04* (2004). DOI: 10.1145/1186562.1015746.

[LGF04] Frank Losasso, Frédéric Gibou, and Ron Fedkiw. "Simulating water and smoke with an octree data structure". Dutch. In: *ACM Transactions on Graphics* 23.3 (2004), pp. 457–462. DOI: 10.1145/1015706.1015745.

[Mül+04] Matthias Müller et al. "Interaction of fluids with deformable solids". Dutch. In: *Computer Animation and Virtual Worlds* 15.34 (2004), pp. 159–171. DOI: 10.1002/cav.18.

[SCC04] Carlos Scheidegger, João Comba, and Rudnei Cunha. "Navier-Stokes on Programmable Graphics Hardware using SMAC." In: Jan. 2004, pp. 300–307.

[SRF05] Andrew Selle, Nick Rasmussen, and Ronald Fedkiw. "A vortex particle method for smoke, water and explosions". In: *ACM Transactions on Graphics* 24.3 (2005), pp. 910–914. DOI: 10.1145/1073204.1073282.

[ZB05] Yongning Zhu and Robert Bridson. "Animating sand as a fluid". In: *ACM Transactions on Graphics (TOG)* 24.3 (2005), pp. 965–972.

[Bar+07] Adam W Bargteil et al. "A finite element method for animating large viscoplastic flow". In: *ACM transactions on graphics (TOG)* 26.3 (2007), 16–es.

[Mül+07] Matthias Müller et al. "Position based dynamics". Dutch. In: *Journal of Visual Communication and Image Representation* 18.2 (2007), pp. 109–118. DOI: 10.1016/j.jvcir.2007.01.005.

[Ngu07] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 2007.

[Sel+07] Andrew Selle et al. "An Unconditionally Stable MacCormack Method". Dutch. In: *Journal of Scientific Computing* 35.2-3 (2007), pp. 350–371. DOI: 10.1007/s10915-007-9166-4.

[SSP07] Barbara Solenthaler, Jürg Schläfli, and Renato Pajarola. "A unified particle model for fluid–solid interactions". Dutch. In: *Computer Animation and Virtual Worlds* 18.1 (2007), pp. 69–82. DOI: 10.1002/cav.162.

[GS08] Oscar Gonzalez and Andrew M Stuart. *A first course in continuum mechanics*. Vol. 42. Cambridge University Press, 2008.

[JBG08] Sicilia F Judice, Bruno Barcellos, and Gilson A Giraldi. "A cellular automata framework for real time fluid animation". In: *Proceedings of the Brazilian Symposium on Computer Games and Digital Entertainment*. Citeseer. 2008, pp. 169–176.

[tan08] tanislaw K. Skowronek and jacob1 and Simon and LBPHacker and various other GitHub contributors. *ThePowderToy*. 2008.

[Ubi08] Ubisoft Montreal. *Far Cry 2*. 2008.

[VBC08] G Viccione, V Bovolin, and E Pugliese Carratelli. "Defining and optimizing algorithms for neighbouring particle identification in SPH fluid simulations". In: *International Journal for Numerical Methods in Fluids* 58.6 (2008), pp. 625–638.

[Cho09] Bastien Chopard. "Cellular Automata Modeling of Physical Systems". Dutch. In: *Encyclopedia of Complexity and Systems Science* (2009), pp. 865–892. DOI: 10.1007/978-0-387-30440-3_57.

[Dav09] Philip Davies. "3d cellular automata". In: *BPC Research Bulletin* 5 (2009).

[PO09] Eric G Parker and James F O'Brien. "Real-time deformation and fracture in a game environment". In: *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. 2009, pp. 165–175.

[Sta09] Jos Stam. “Nucleus: Towards a unified dynamics solver for computer graphics”. Dutch. In: *2009 11th IEEE International Conference on Computer-Aided Design and Computer Graphics* (2009). DOI: 10.1109/cadcg.2009.5246818.

[CTG10] Jonathan M. Cohen, Sarah Tariq, and Simon Green. “Interactive fluid-particle simulation using translating Eulerian grids”. Dutch. In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D 10* (2010). DOI: 10.1145/1730804.1730807.

[Dom+10] J. M. Domínguez et al. “Neighbour lists in smoothed particle hydrodynamics”. Dutch. In: *International Journal for Numerical Methods in Fluids* 67.12 (2010), pp. 2026–2042. DOI: 10.1002/fld.2481.

[JYT10] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. “Cellular automata for real-time generation of infinite cave levels”. Dutch. In: *Proceedings of the 2010 Workshop on Procedural Content Generation in Games - PCGames ’10* (2010). DOI: 10.1145/1814256.1814266.

[JCG10] Sicilia F Judice, Bruno Barcellos S Coutinho, and Gilson A Giraldi. “Lattice methods for fluid animation in games”. In: *Computers in Entertainment (CIE)* 7.4 (2010), pp. 1–29.

[LK10] Samuli Laine and Tero Karras. “Efficient sparse voxel octrees”. In: *IEEE Transactions on Visualization and Computer Graphics* 17.8 (2010), pp. 1048–1059.

[MST10] Aleka McAdams, Eftychios Sifakis, and Joseph Teran. “A Parallel Multigrid Poisson Solver for Fluids Simulation on Large Grids.” In: *Symposium on Computer Animation*. 2010, pp. 65–73.

[SHT10] Siamak Sarmady, Fazilah Haron, and Abdullah Zawawi Talib. “Simulating crowd movements using fine grid cellular automata”. In: *2010 12th International Conference on Computer Modelling and Simulation*. IEEE. 2010, pp. 428–433.

[AO11] Iván Alduán and Miguel A. Otaduy. “SPH granular flow with friction and cohesion”. Dutch. In: *Proceedings of the 2011 ACM SIGGRAPH/Eurographics Symposium on Computer Animation - SCA ’11* (2011). DOI: 10.1145/2019406.2019410.

[CSH11] Jonathan Caux, Pridi Siregar, and David Hill. “Accelerating 3D Cellular Automata Computation with GP GPU in the Context of Integrative Biology”. In: *Cellular Automata-Innovative Modelling for Science and Engineering* (2011), pp. 411–426.

[CM11] Nuttapong Chentanez and Matthias Müller. "Real-time Eulerian water simulation using a restricted tall cell grid". Dutch. In: *ACM SIGGRAPH 2011 papers on - SIGGRAPH '11* (2011). DOI: 10.1145/1964921.1964977.

[Cre+11] Alejandro C Crespo et al. "GPUs, a new tool of acceleration in CFD: efficiency and reliability on smoothed particle hydrodynamics methods". In: *PloS one* 6.6 (2011), e20685.

[Gob+11] Stéphane Gobron et al. "GPGPU computation and visualization of three-dimensional cellular automata". In: *The Visual Computer* 27.1 (2011), pp. 67–81.

[Lev+11] David I. W. Levin et al. "Eulerian solid simulation with contact". Dutch. In: *ACM Transactions on Graphics* 30.4 (2011), pp. 1–10. DOI: 10.1145/2010324.1964931.

[Re-11] Re-Logic and 505 Mobile S.r.l and Pipeworks Software and DR Studios and Codeglue and Engine Software. *Terraria*. 2011.

[Fir12] Firaxis Games and Feral Interactive and 2K Games. *X-COM: Enemy Unknown*. 2012.

[MMH12] Martin C. Marinack, Jeremiah N. Mpagazehe, and C. Fred Higgs. "An Eulerian, lattice-based cellular automata approach for modeling multiphase flows". Dutch. In: *Powder Technology* 221 (2012), pp. 47–56. DOI: 10.1016/j.powtec.2011.12.016.

[Don+13] Yin-Feng Dong et al. "Cellular Automata Model for Elastic Solid Material". Dutch. In: *Communications in Theoretical Physics* 59.1 (2013), pp. 59–67. DOI: 10.1088/0253-6102/59/1/12.

[MM13] Miles Macklin and Matthias Müller. "Position based fluids". Dutch. In: *ACM Transactions on Graphics* 32.4 (2013), pp. 1–12. DOI: 10.1145/2461912.2461984.

[Mus13] Ken Museth. "VDB: High-resolution sparse volumes with dynamic topology". In: *ACM transactions on graphics (TOG)* 32.3 (2013), pp. 1–22.

[Sto+13] Alexey Stomakhin et al. "A material point method for snow simulation". Dutch. In: *ACM Transactions on Graphics* 32.4 (2013), pp. 1–10. DOI: 10.1145/2461912.2461948.

[The13] The Khronos Group. *OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems*. July 2013. URL: https://www.khronos.org/opencl/.

[Vac+13] Renato Vacondio et al. “Variable resolution for SPH: a dynamic particle coalescing and splitting scheme”. In: *Computer Methods in Applied Mechanics and Engineering* 256 (2013), pp. 132–148.

[Mac+14] Miles Macklin et al. “Unified particle physics for real-time applications”. Dutch. In: *ACM Transactions on Graphics* 33.4 (2014), pp. 1–12. DOI: 10.1145/2601097.2601152.

[Ren+14] Bo Ren et al. “Multiple-Fluid SPH Simulation Using a Mixture Model”. Dutch. In: *ACM Transactions on Graphics* 33.5 (2014), pp. 1–11. DOI: 10.1145/2645703.

[Set+14] Rajsekhar Setaluri et al. “SPGrid: A sparse paged grid structure applied to adaptive smoke simulation”. In: *ACM Transactions on Graphics (TOG)* 33.6 (2014), pp. 1–12.

[Sto+14] Alexey Stomakhin et al. “Augmented MPM for phase-change and varied materials”. Dutch. In: *ACM Transactions on Graphics* 33.4 (2014), pp. 1–11. DOI: 10.1145/2601097.2601176.

[BK15] Jan Bender and Dan Koschier. “Divergence-free smoothed particle hydrodynamics”. In: *Proceedings of the 14th ACM SIGGRAPH/Eurographics symposium on computer animation*. 2015, pp. 147–155.

[Dou+15] Nikolaos Dourvas et al. “Hardware acceleration of cellular automata Physarum polycephalum model”. In: *Parallel Processing Letters* 25.01 (2015), p. 1540006.

[Jia+15] Chenfanfu Jiang et al. “The affine particle-in-cell method”. Dutch. In: *ACM Transactions on Graphics* 34.4 (2015), pp. 1–10. DOI: 10.1145/2766996.

[MF15] Martin C. Marinack and C. Fred Higgs. “Three-dimensional physics-based cellular automata model for granular shear flow”. Dutch. In: *Powder Technology* 277 (2015), pp. 287–302. DOI: 10.1016/j.powtec.2015.02.057.

[Yan+15] Tao Yang et al. “Fast multiple-fluid simulation using Helmholtz free energy”. In: *ACM Transactions on Graphics (TOG)* 34.6 (2015), pp. 1–11.

[Chu16] Chucklefish. *Starbound*. 2016.

[Evs+16] Oleg Evsutin et al. “The Algorithm of Continuous Optimization Based on the Modified Cellular Automaton”. In: *Symmetry* 8.9 (2016). ISSN: 2073-8994. DOI: 10.3390/sym8090084. URL: https://www.mdpi.com/2073-8994/8/9/84.

[Hoe16] Rama Karl Hoetzlein. “GVDB: Raytracing sparse voxel database structures on the GPU”. In: *Proceedings of High Performance Graphics*. 2016, pp. 109–117.

[NB16] Michael B Nielsen and Robert Bridson. "Spatially adaptive FLIP fluid simulations in bifrost". In: *ACM SIGGRAPH 2016 Talks*. 2016, pp. 1–2.

[Ren16] Paul Rendell. *Turing machine universality of the game of life*. Springer, 2016.

[TLK16] Yun Teng, David I. W. Levin, and Theodore Kim. "Eulerian solid-fluid coupling". Dutch. In: *ACM Transactions on Graphics* 35.6 (2016), pp. 1–8. DOI: 10.1145/2980179.2980229.

[Vac+16] Renato Vacondio et al. "Variable resolution for SPH in three dimensions: Towards optimal splitting and coalescing for dynamic adaptivity". In: *Computer Methods in Applied Mechanics and Engineering* 300 (2016), pp. 442–460.

[Yan+16] Xiao Yan et al. "Multiphase SPH simulation for interactive fluids and solids". Dutch. In: *ACM Transactions on Graphics* 35.4 (2016), pp. 1–11. DOI: 10.1145/2897824.2925897.

[ZKM16] Johanne Zadick, Benjamin Kenwright, and Kenny Mitchell. "Integrating Real-Time Fluid Simulation with a Voxel Engine". In: *The Computer Games Journal* 5.1-2 (2016), pp. 55–64. DOI: 10.1007/s40869-016-0020-5.

[Fu+17] Chuyuan Fu et al. "A polynomial particle-in-cell method". Dutch. In: *ACM Transactions on Graphics* 36.6 (2017), pp. 1–12. DOI: 10.1145/3130800.3130878.

[Gal17] Gallant. *2D Liquid Simulator With Cellular Automaton in Unity - jgallant's Indie Game Developer Homepage*. Jan. 30, 2017. URL: http://www.jgallant.com/2d-liquid-simulator-with-cellular-automaton-in-unity/ (visited on 01/27/2022).

[Hei+17] Christian Heintz et al. "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*. 2017, pp. 1–2.

[Hu+17] Wei Hu et al. "A consistent multi-resolution smoothed particle hydrodynamics method". In: *Computer Methods in Applied Mechanics and Engineering* 324 (2017), pp. 278–299.

[Sha+17] Hossein Shafizadeh-Moghadam et al. "Coupling machine learning, tree-based and statistical models with cellular automata to simulate urban growth". In: *Computers, Environment and Urban Systems* 64 (2017), pp. 297–308.

[Yan+17] Tao Yang et al. "A unified particle system framework for multi-phase, multi-material visual simulations". Dutch. In: *ACM Transactions on Graphics* 36.6 (2017), pp. 1–13. DOI: 10.1145/3130800.3130882.

[Com18] Blender Online Community. *Blender - a 3D modelling and rendering package*. Stichting Blender Foundation, Amsterdam: Blender Foundation, 2018. URL: http://www.blender.org.

[Gao+18] Ming Gao et al. "GPU optimization of material point methods". Dutch. In: *ACM Transactions on Graphics* 37.6 (2018), pp. 1–12. DOI: 10.1145/3272127.3275044.

[GSL18] Niranjan S. Ghaisas, Akshay Subramaniam, and Sanjiva K. Lele. "A unified high-order Eulerian method for continuum simulations of fluid flow and of elastic–plastic deformations in solids". Dutch. In: *Journal of Computational Physics* 371 (2018), pp. 452–482. DOI: 10.1016/j.jcp.2018.05.035.

[Hu+18] Yuanming Hu et al. "A moving least squares material point method with displacement discontinuity and two-way rigid body coupling". Dutch. In: *ACM Transactions on Graphics* 37.4 (2018), pp. 1–14. DOI: 10.1145/3197517.3201293.

[Med18] Jakub Medvecký-Heretik. *Real-time Water Simulation in Game Environment*. Tech. rep. Thesis. Apr. 2018. URL: https://is.muni.cz/th/mfkg2/master-thesis.pdf.

[TP18] Nils Thuerey and Tobias Pfaff. *MantaFlow. http://mantaflow.com*. 2018.

[Wu+18] Kui Wu et al. "Fast Fluid Simulations with Sparse Volumes on the GPU". In: *Computer Graphics Forum* 37.2 (2018), pp. 157–167. DOI: 10.1111/cgf.13350.

[Din+19] Mengyuan Ding et al. "A thermomechanical material point method for baking and cooking". Dutch. In: *ACM Transactions on Graphics* 38.6 (2019), pp. 1–14. DOI: 10.1145/3355089.3356537.

[Hu+19] Yuanming Hu et al. "Taichi". Dutch. In: *ACM Transactions on Graphics* 38.6 (2019), pp. 1–16. DOI: 10.1145/3355089.3356506.

[Ji+19] Zhe Ji et al. "A new multi-resolution parallel framework for SPH". In: *Computer Methods in Applied Mechanics and Engineering* 346 (2019), pp. 1156–1178.

[MZ19] Dragan Marinkovic and Manfred Zehn. "Survey of finite element method-based real-time simulations". In: *Applied Sciences* 9.14 (2019), p. 2775.

[Rei+19] Stefan Reinhardt et al. "Consistent shepard interpolation for SPH-based fluid animation". In: *ACM Transactions on Graphics (TOG)* 38.6 (2019), pp. 1–11.

[Wol19] Stephen Wolfram. *CELLULAR AUTOMATA COMPLEXITY*. Abingdon, Verenigd Koninkrijk: Taylor Francis, 2019.

[Wol+19] Joshuah Wolper et al. "CD-MPM". Dutch. In: *ACM Transactions on Graphics* 38.4 (2019), pp. 1–15. DOI: 10.1145/3306346.3322949.

[Che+20a] Jian-Yu Chen et al. "GPU-accelerated smoothed particle hydrodynamics modeling of granular flow". In: *Powder Technology* 359 (2020), pp. 94–106.

[Che+20b] Xiao-Song Chen et al. "A moving least square reproducing kernel particle method for unified multiphase continuum simulation". Dutch. In: *ACM Transactions on Graphics* 39.6 (2020), pp. 1–15. DOI: 10.1145/3414685.3417809.

[Hua+20] Kemeng Huang et al. "Novel hierarchical strategies for SPH-centric algorithms on GPGPU". In: *Graphical Models* 111 (2020), p. 101088.

[JN20] Haran Jackson and Nikos Nikiforakis. "A unified Eulerian framework for multimaterial continuum mechanics". Dutch. In: *Journal of Computational Physics* 401 (2020), p. 109022. DOI: 10.1016/j.jcp.2019.109022.

[Kot20] Grant Kot Kot. *Physically Based Rendering — Voxel Physics Devlog 5*. Dutch. Feb. 13, 2020. URL: https://www.youtube.com/watch?v=5F1oFcxXObU.

[Lin20] John Lin Lin. *This Real-Time 3D Fluid Sim Runs on One CPU Core*. Dutch. Aug. 21, 2020. URL: https://www.youtube.com/watch?v=4Y58Pg9tpSo.

[Nol20] Nolla Games. *Noita*. 2020.

[Scr20] title = GPU-GEMS-3D-Fluid-Simulation Scrawk. 2020. URL: https://github.com/Scrawk/GPU-GEMS-3D-Fluid-Simulation.

[Tux20] Tuxedo-Labs. *Teardown*. 2020.

[Wan+20] Xinlei Wang et al. "A massively parallel and scalable multi-GPU material point method". Dutch. In: *ACM Transactions on Graphics* 39.4 (2020). DOI: 10.1145/3386569.3392442.

[Soł+21] Wojciech T. Sołowski et al. "Material point method: Overview and challenges ahead". Dutch. In: *Advances in Applied Mechanics* (2021), pp. 113–204. DOI: 10.1016/bs.aams.2020.12.002.

[Su+21] Haozhe Su et al. "A unified second-order accurate in time MPM formulation for simulating viscoelastic liquids with phase change". Dutch. In: *ACM Transactions on Graphics* 40.4 (2021), pp. 1–18. DOI: 10.1145/3450626.3459820.

[Ato22] Atomontage Inc. *Atomontage Engine*. 2022. URL: https://www.atomontage.com/.

[Uni22] Unity Technologies. *Wondering what is? Find out who we are, where we've been and where we're going.* 2022. URL: https://unity.com/our-company.