The Game of Flow - Cellular Automaton-based Fluid Simulation for Realtime Interaction

Christian Heintz, Moritz Grunwald, Sarah Edenhofer, Jörg Hähner
University of Augsburg

Sebastian von Mammen
University of Würzburg

ABSTRACT

In this paper, we present a realtime fluid simulation based on cellular automata (CAs). The main goal is to demonstrate the performance and extensibility of this approach. To show this, we created a fluid simulation and extended it by simulating different kinds of fluids at the same time. With this fluid-to-fluid interaction we can simulate effects like oil floating on water with a focus on short computation time. This makes the simulation interesting for interactive simulations and games in VR. To show the potential of our simulation we created a small VR game with promising results: a high framerate of 100 FPS for a CA running on the CPU with 176^3 cells due to parallelization and optimization.

KEYWORDS

cellular automaton, fluid simulation, cell grid, realtime

1 FLUID SIMULATIONS

The simulation of fluids is a relevant part of physics engines and used by the movie and video games industry. The latter is faced by a great challenge: Games and interactive simulations must be realtime capable. For regular desktop computers, rates of 60 frames per second and more were deemed suitable [1]. For virtual reality (VR) applications which track the user's head orientation and movement, consistently achieving 90 frames per second is the current, broadly understood standard. Computations that involve a large number of interacting parts tend to significantly impact the latency. There are two general types of fluid simulations: grid-based, where you divide the simulation-space into cells and each cell has a fixed position and contains values such as fluid concentration or velocity (e.g. in [2]), and particle-based, where the fluid consists of large numbers of moving particles that influence each other (e.g. with the FleX Engine). In contrast to other realtime fluid engines, we utilised cellular automata (CAs), a computational representation originally conceived to retrace the foundational mechanisms of cause and effect to lead to the emergence of life [7]. The basic data structure of a CA is a multi-dimensional discrete grid whose cells can have different states. Changes are introduced to each cell based on its own and its neighbours' current states. An according set of state-changing rules that is applied at each simulation step and the initial configuration of a CA determine the pattern it produces over time. Adding new or changing existing rules of the set allow for an easy extension of the CA. We utilised a three-dimensional CA to create a fluid simulation that encodes physical and chemical properties in the cells' states.

Related to our approach are Smooth Particle Hydrodynamics (SPH), a widespread particle-based fluid simulation approach. Müller at al. presented an extension of the SPH method in 2005 [5]. It is based on the idea that numerous virtual particles pull and push each other to satisfy a density constraint, allowing for interaction between different fluids and temperature-dependent phase transitions of fluids. Tweaking SPH and pushing its computations to the GPU promises sufficient performance also for VR applications. However, discretisation of the underlying particles and resolving the multitude of neighbourhood-relations among them are costly processing steps. Another costly step is rendering SPH-driven fluids by reconstruction of surfaces. Screen Space Fluid Rendering [3], for instance, yields good visual results but is based on post-processing effects and, therefore, has to be executed twice in VR, one time for each eye. Another way is to simplify the simulation like in Height Field Fluids [6], where a 2D array of height values is used as the underlying representation to simulate the surface dynamics of large water bodies. Looking at realtime-capable, grid-based approaches to fluid simulation that also incorporate (imaginative) chemical reactions, a recent example has been implemented in the greatly successful Indie-game *Minecraft*[1]. It is a voxel-based game in which water, lava and heat is simulated by means of CAs. If lava comes into contact with water, it turns into obsidian or cobblestone. If there is some wood near a pond of lava, the wood will start to burn and fire will start spreading. In Section 2, we detail our simulation approach. In Section 3, our results are presented and the paper is concluded.

2 THE GAME OF FLOW MODEL

If one pours fluids with different densities into a single container, they arrange themselves in layers according to their specific densities: the fluid with the highest density drops to the bottom, whereas lower densities float on top. When simulating fluids, viscosity plays a vital role. It is a measure for tenacity and determines the speed of the fluids' flows. Both density and viscosity are basic factors of our simulation and both are influenced by temperature. In order to plausibly retrace the effects of these interdependent factors at an abstract level, we relinquish solving the physics equations and establish a simple and intuitive system of CA update rules. In order to account for different fluids and densities, the cells of our CA store a continuous state variable and an associated type. In order to manage the flow between cells, we provide storage space for different fluid types. Each fluid type has one of n unique identifiers (IDs), with 0 as the ID for the type of the lowest density and $n-1$ of the highest density. The properties (viscosity and density), of the fluid types are stored in global arrays and can be accessed via their IDs. Each cell of the CA saves an array of size n to store the local amount of units of each type. The maximal total amount of each type per cell is restricted by a certain value. The totalled state values of the CA need to stay constant (**Rule of Constancy**). Otherwise, matter would either get lost or appear out of nowhere. In

[1] https://minecraft.net/, accessed 2017-09-14

order to comply with this requirement, any state changes to a cell, whether additive or subtractive, need to keep a local equilibrium. Simply said, it is only allowed to transfer cell state' units instead of generating or deleting them. As a result, when a cell's update is calculated, it can either pull in some units from its neighbours or push some out. In our CA, fluids diffuse horizontally across the plane (**Rule of Diffusion**). This is realised by pushing equal amounts of fluid to a cell's neighbours. The actual amount is dependent on the viscosity of a fluid type. The update function, which computes the new amount of a fluid-type in a cell is computed as follows, whereas A is the total amount of the current element in both cells, V its current viscosity:

$$value = (A/2)\begin{cases} +V, & \text{if } \textit{cell stores more than its neighbour} \\ -V, & \text{otherwise} \end{cases}$$

In order to account for gravity, we transfer liquid vertically depending on the fluids' densities (**Rule of Descent**). We first merge the amounts of each type of fluid in two cells that lie on top of one another so we can distribute the amountaccording to their density. In order to do that, we iterate through the merged amounts from high to low density, adding as much as possible to the bottom cell without it overflowing. When the bottom cell has reached the maximal capacity we add the rest of the merged amounts to the top cell. In order to simulate gas, we allowed density values between *(0,1)*. Fluids with a density value within this range will cause its amount to favour placement in the top cell, ultimately making it move in an upwards direction. The amount of fluid of a given type (δ) stored in the bottom cell is: $\delta = Min(M - B, Min(D, 1) * A)$. Hereby, M is the maximum volume a cell can hold until it is full, B the combined volume of amounts of each fluid-type with higher density that are already in the bottom cell, D being the element density and A being the total amount of the current element in both cells.

The cellular automaton also calculates temperature spread. Each cell holds a temperature value that represents the combined temperature of each fluid in it in degree Celsius. A fraction of the temperature of one cell will mix with the temperatures of the surrounding cells depending on the amounts of fluids in each. The amounts of the fluid-types act as weights when mixing the values. Furthermore the flow direction also directly influences temperature spread in a fashion that makes the temperature seem to be linked to the cells content. If the cells temperature fulfils an specific element dependent criteria, the element will change its state accordingly, temperatures over 100°C for example will cause water to evaporate and turn into gas state.

CAs naturally work in parallel as each cell update is independent. Therefore, we implemented GPU compute shaders to calculate these parallel updates. For further optimization, we changed the update cycle: In a CA, every cell changes its state individually depending on the state of its neighbouring cells during each update. We changed this individual cell update to a dual-cell update cycle. During a dual-cell update, two neighbouring cells are updated in one step. By doing this, we can save about half of the calculations needed to update each cell, reusing the results calculated for one cell for the neighbouring cell instead of both having to calculate the same equation individually.

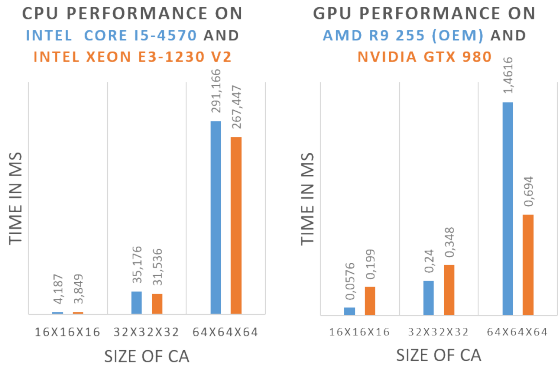

Figure 1: The time was measured with the Profiler of the *Unity Game Engine.*

3 EVALUATION

In addition to the GPU version of our CA, we also implemented a CPU version to underline the effect of parallelization on computation time. Besides a voxelised-visualisation, we used the marching cubes algorithm [4], because it's usually used for fluid rendering and needs the lattice shaped data we provide. It is not surprising that the simulation runs faster on the GPU, but it is still interesting to see the difference in performance. Figure 1 shows the results. One can see that the GPU version is not only much faster than the CPU version but it also scales better. The computation time on the GPU is so small that it scarcely affects the framerate and therefore guarantees realtime capability even for VR applications. Clearly, the computation costs depend on the size of the CA. The size, in turn, depends on the number of cells as well as the number of different elements and is limited by RAM (CPU) and VRAM (GPU). Depending on the type of visualisation and hardware, it is possible to have 176^3 cells at constant 100 FPS (on AMD R9 255 OEM). The project of our CA model for simulating plausible fluid dynamics in a three-dimensional lattice grid can be found on GitHub[2]).

REFERENCES

[1] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. 2011. *Real-time rendering.* CRC Press.

[2] Nuttapong Chentanez and Matthias Müller. 2011. Real-time Eulerian Water Simulation Using a Restricted Tall Cell Grid. In *ACM SIGGRAPH 2011 Papers.* ACM, New York, NY, USA, Article 82, 82:1–82:10 pages.

[3] Simon Green. 2010. Screen space fluid rendering for games. In *Proceedings for the Game Developers Conference.*

[4] William E. Lorensen and Harvey E. Cline. 1987. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *SIGGRAPH Comput. Graph.* 21, 4 (Aug. 1987), 163–169.

[5] Matthias Müller, Barbara Solenthaler, Richard Keiser, and Markus Gross. 2005. Particle-based Fluid-fluid Interaction. In *Proceedings of the 2005 ACM SIGGRAPH/SCA.* ACM, New York, NY, USA, 237–244.

[6] Matthias Müller-Fischer. 2008. Fast water simulation for games using height fields. In *Proceedings of the Game Developer's Conference.*

[7] John von Neumann and Arthur W. Burks. 1966. *Theory of self-reproducing automata.* University of Illinois Press, Urbana and London.

[2]https://github.com/Frager/CA-Fluid-Simulation