# A Framework of Artificial Matter

Christian G. Hahm

*Department of Computer and Information Sciences*
*Temple University, USA*
christian.hahm@temple.edu

October 10, 2023

Keywords: *Simulation, Voxel, Cell, Physics, Chemistry, Cellular Automata, World, Environment, Universe, Elements, Sand*

**Abstract**

This paper details a framework of granular artificial matter with which a user can interact as though in a macroscopic 3D environment. The framework is a particle simulation of varying elements interacting by a sort of digital physics and chemistry. This type of simulation is sometimes called a "falling-sand game" or "falling-sand simulator". Computation of the world state is implemented using a cellular automaton, since its computation is very fast by its parallel nature, and interesting phenomena could emerge from the local interactions of many voxels. Unlike most game/world simulations and cellular automata, this world simulation is "conservative" in that it strictly enforces conservation of matter.

## Contents



## 1 Introduction

Let us consider the problem of simulating a universe (i.e., world, environment, etc.) consisting of interacting granular matter, starting by considering our own universe.

Firstly, we can conceptualize the universe as a big empty (finite) space. Within this volume, some of the space is filled with physical substance (this is called *matter*) whereas some of the space remains empty (this is called *vacuum*). [LibreText, 2022, Section 3.2;3.E.2]

In the macroscale, matter appears to take an effectively limitless variety of forms. In technicality, all matter is built from a set of only 118 microscopic building blocks called *atoms* [LibreText, 2022, Section 3.2]. When multiple atoms are strongly connected to each other, they form a larger structure called a *molecule* [LibreText, 2022, Section 5].

These atomic and molecular building blocks contain *energy* which results in them physically moving around the space and colliding with each other. When building blocks collide, depending on their specific substance, they may chemically *react*; a *chemical reaction* is when the building blocks transform from one substance to another. Importantly, the quantities of both atoms and energy are always conserved, never created/increased or destroyed/decreased. In the case of atoms, this is called the Law of Conservation of Mass [LibreText, 2022, Section 3.7], and in this case of energy, it is called the Law of Conservation of Energy [LibreText, 2022, Section 3.9].

Sufficiently realistic and interactive simulated worlds can serve as challenging ecosystems for the life and evolution of artificial (AI) animals (aka animats) [Wilson, 1991]. In performing computer simulation of a world, on the one hand, we obviously cannot fully simulate the countless microscopic atoms and quantum strings of our own world, since we do not know all the technical details, and our computational hardware is too weak for any macroscale simulation. On the other hand, we desire a simulation which is very dynamic, flexible, and interactive. This means an animat should be able to physically change the world (e.g., build a structure, or tunnel through the earth) and chemically (e.g., moving water onto lava to make stone). If the atoms are autonomous particles, constantly moving around and colliding and interacting, the environment has the potential to exhibit interesting emergent phenomena, depending on the nature of the atoms.

So, we desire low-level granular particles, or *atomic building blocks* (atoms), which can move and locally interact in various ways. Importantly, the simulation should conserve matter (atoms) and energy to prevent resource duplication, infinite world growth/destruction, etc. [Hahm, 2022, p.3].

Starting from this information, we can examine and build on the current state-of-the-art in world simulation.

# 2 Related Works

## 2.1 Cellular Automata for Computing Matter

Is our universe digitally computable? At first glance it seems impossible: quantities in classical physics are continuous (and potentially infinite/irrational after the decimal), whereas digital computations can only operate over discrete (finite) quantities. This question was explored in Zuse's *Calculating Space* [Zuse, 1970], where he proposed that matter could be computed digitally. In this work, Zuse detailed a mechanism for the motion and interaction of digital particles throughout a 1D and 2D space using cellular automata. As noted by Zuse, it should be possible to simulate various types of particles: "the behavior depends on the size of the particles and the law of elasticity. Large particles collide more frequently than small ones. Hard particles behave differently than soft ones." [Zuse, 1970, p.17]

The cellular automaton (CA) is a very interesting way to consider the computation of a world, since it is in the fastest class of algorithm possible. The CA is a constant-time algorithm, having a time complexity of $O(1)$; that is to say, the time to complete the algorithm does not scale with the size of the CA. In this way, at least in terms of time limitations, a world of any size may be computed using CA, since changing the world's size does not change the time taken to compute the world.

Methods emerged by which to simulate 2D particle motion using two-state cellular automata, particularly the motion of gasses. One such class was the 2D *lattice gas automata* (LGA). These modeled the behaviors of gas particles on the microscale, such that the gas' behavior on the macroscale closely matched the Navier-Stokes equations. Unlike a cellular automaton, which proceeds in a single step (following the transition rule), each step of an LGA proceeds in two sub-steps: 1.) collision step (particle velocities are reoriented according to their collisions), 2.) translation step (particles move to a new position on the lattice according to their velocities). Examples of LGA include the deterministic Hardy-Pomeau-Pazzis (HPP) model (on a square lattice) [Hardy et al., 1976] and the non-deterministic Frisch-Hasslacher-Pomeau (FHP) model (on a hexagonal lattice) [Frisch et al., 1986]. The FHP model was extended to 3D in the face-centered-hypercubic (FCHC) model [d'Humieres et al., 1986].

A different 2D particle simulator of the CA class was the *billiard ball model* (BBM) [Fredkin and Toffoli, 1982, Margolus, 1984]. It was implemented using a partitioned CA which is *reversible*, in the sense that it can be run in reverse, and conservative, meaning the total number of particles on the grid is conserved at every transition step. By combining some reversible local interaction rules with "conservation of mass", the BBM models the motion of billiard balls bouncing off of each other using CA. It does so using a special type of CA neighborhood. Unlike the standard von Neumann neighborhood (4 neighbors) or Moore neighborhood (8 neighbors), it is "partitioned", using the Margolus neighborhood. At each timestep, a checkerboard pattern of 2x2 blocks alternates across space.

Reversibility is an important property to have in a physics simulator, because reversible systems produce more interesting behavior than non-reversible systems. Reversible systems are more "interesting" in the sense that they rarely repeat states or oscillate (at least globally). They will continue to do novel things as they run, unless the initial setup is very simple. When running a reversible system, it is not possible to repeat any global state $S$ computed so far, *unless* the initial global state $S_0$ is first repeated. If the system never returns to its initial state $S_0$, then it will never return to $S$ either, since every global state has a unique next global state. On the other hand, there is nothing preventing a non-reversible system (such as Conway's Game of Life [Gardner, 1970]) from oscillating between the same few states, since many varying global states can lead to the same global state. [Margolus, 1998, p.11]

The BBM conserves mass and energy, but not momentum. In a later work [Margolus, 2002], the rules of the billiard ball model CA (BBMCA) were changed into a 2D *soft spheres model* (SSM). This, by extension, was extended to 3D. By treating the particles as soft elastic spheres rather than rigid balls, the CA acts more like an LGA and conserves momentum.

The LGA and BBMCA models are two-state: at a given point in space there is either 1.) empty space, or 2.) a particle. If a particle is made immovable, it no longer acts like a gas particle but more like a solid "wall" or "mirror" off of which other gas particles can bounce. In this way, these models really simulate 2 substances: 1.) gas, and 2.) immovable solids.

Sandpile CAs (e.g., [Gruau and Tromp, 2000]) are two-state CAs which simulate sand particles. Instead of bouncing around eternally like elastic gas particles, each sand particle falls downward as though affected by gravity. When it reaches the flat ground, the particle stops; however, it is "unstable", such that if the particle directly below it (or below to the immediate left/right) disappears, the particle will resume falling into the empty space. When many of these sand particles are dropped into a simulation, they form structures that look like sandpiles.

## 2.2 Falling-Sand Simulators

A genre of 2D games called "falling-sand games" or "falling-sand simulators" appeared on the internet in 2005 [Wikipedia, 2023]. Similarly to sandpile CAs, these simulators allowed users to drop sand particles into a 2D space. However, these simulators went a step further by introducing various elements including stone, water, fire, oil, steam, and others. The elements moved around the cellular space in various ways, and interacted with each other in sort of "chemical reactions". Some of these 2D simulators apparently utilized cellular automata.

There are now very advanced modern day iterations of 2D falling-sand games (e.g., *Noita* [Purho, 2019]), and even 3D "voxel" versions (e.g., *Minecraft*). As these applications were designed to be more complex and oriented towards user gameplay, most no longer strictly adhered to a cellular automata paradigm. Thus, many modern falling-sand simulators do not take the full advantage of parallel processing in constant time ($O(1)$), but instead sequentially iterate over voxels ($O(n)$).

A sequential algorithm is too slow to iterate over every single voxel in a 3D world; it is not particularly performant for 2D simulations either. As such, these games require various optimization techniques to run in real time, e.g., splitting the world into smaller "chunks" then processing only some of the chunks, and various other optimization techniques (see discussions in [Purho, 2019] for the techniques used in *Noita*).

Although the falling-sand games are quite remarkable, with some of the 2D versions apparently utilizing cellular automata, there is sparse documentation of falling-sand simulators (with multiple elements) in the literature, especially for 3-dimensions.

# 3 A 3D Falling-Sand Simulator using Cellular Automata

The goal of this endeavor was to develop a framework of artificial matter in 3D, with granular particles that move and interact in various ways. After a review of the existing works, it was decided to accomplish this goal by implementing 1.) a falling sand simulator (for a granular world of many elements), 2.) using cellular automata (for the speed achieved via parallelism, and for its bottom-up interactions), 3.) in 3-dimensions (since that is our domain of interest, and it has yet to be accomplished).

First, the general algorithm of the 3D cellular automaton is described. Then, specific details of the simulator are introduced, including the specific voxel types, their physics, and their chemistry. Finally, there is supplemental technical discussion on efficiently rendering the voxel world to the camera, which is also important as we need to both compute *and* view the 3D world in real-time.

## 3.1 Computation

### 3.1.1 A Conservative Partitioned Cellular Automaton in 3D

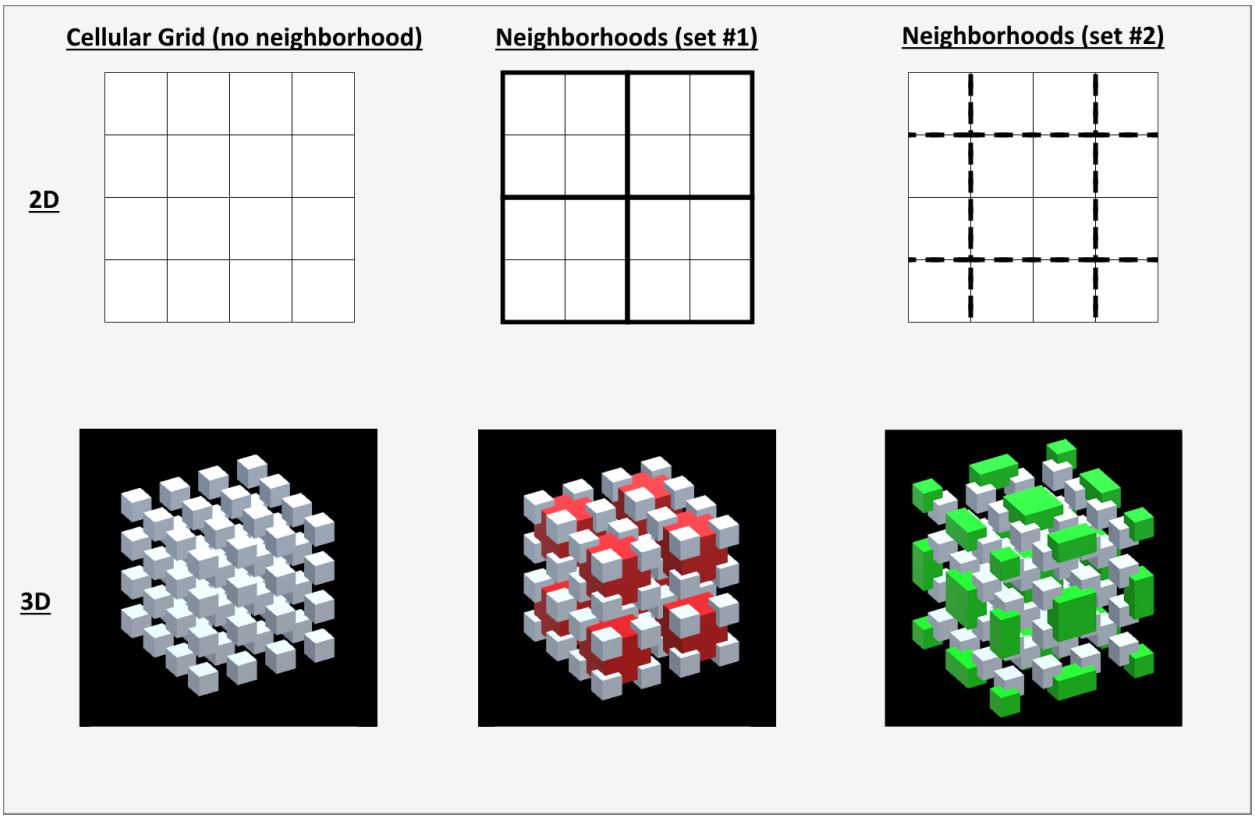



Figure 1: Visualizing the 2D (top) and 3D (bottom) Margolus neighborhood (note that neighborhoods in *set #2* are cut off).

First, consider the Margolus neighborhood. Its dimensions are $2x2$ in 2D. When the automaton is extended by adding a third dimension, the Margolus neighborhood can also be extended by adding a dimension, such that its dimensions are $2x2x2$. Just as in the 2D version, there are only 2 sets of neighborhoods. The sets overlap each other, though one is staggered by a single voxel in each dimension. The set of neighborhoods being used for computation alternates at each timestep (one set is processed on even-numbered timesteps, the other set is processed on odd-numbered timesteps). For a visualization of the 2D and 3D neighborhoods, see *Figure 1*.

Next, consider the rules of the 2D two-state BBMCA, as explained in [Margolus, 1984]. All configurations of neighborhoods are stable (unchanging), except for two cases: particle movement and collision. For the movement rule, when there is only one particle in the neighborhood, it moves diagonally. Visualized, this rule is  (and all rotations). This results in the particle flying in some unchanging direction. For the collision rule, when there are only two particles in the neighborhood, and they are directly diagonal to each other, they move to the unoccupied diagonals. Visualized, this rule is  (and all rotations). This results in the particles scattering away from each other, perpendicular to the directions at which they collided. Finally, certain stable configurations such as  and  are ambiguous. Depending on the larger picture, they may represent large stable walls/objects, or they may represent a particle bouncing off such an object. The bouncing takes a single timestep; after that timestep, the particle will shoot off in the direction opposite to which it collided with the object.

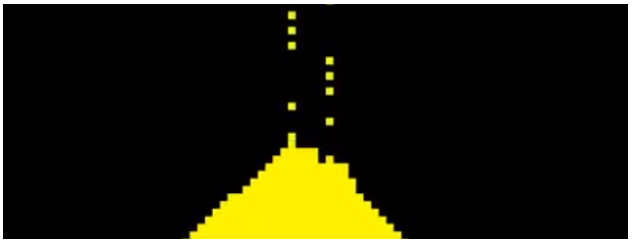
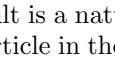


Figure 2: A 2D sandpile cellular automaton.

Then, consider the rules of the 2D sandpile CA. Though the exact method can vary, in essence it involves implementing gravity. "'Gravity" simply means moving the particle downward into an empty slot. If the empty slot is directly downward from the particle, the result is a natural settling, such as  or . However, if the empty slot is in the downward from the particle in the diagonal, as in , then the result is the particle toppling downward. This results in piles of particles.

From the rules and behavior of these cellular automata, we can observe an interesting property which allows us to extend them into a 3D falling-sand simulator with various elements. We can send a particle moving along a trajectory, simply by moving it in that direction within the local neighborhood at each timestep. This simple property allows us to simulate all 3 states of matter: solid, liquid, and gas.

We can simulate a *solid* as in the sandpile CA, by moving the particle to any empty slot which is *downward*; this makes the particle seem as though it is heavy and impacted by gravity. We can simulate a *liquid* by using gravity and additionally allowing the particle to move *sideways* whenever it is blocked from moving downward; this gives the impression that the liquid is flowing. To simulate a gas is a bit different. Unlike a solid or a liquid, a gas is not impacted by gravity, but is instead flying around its container. We can simulate a *gas* by moving the particle as in BBMCA, to its diagonal. For a more technical description, see *Section 3.1.2*.

The description of these rules are quite general, which leaves the specific cases still undefined. For example, what should a "solid" particle do when there is hilltop directly below it? Should it not move, like a flat stone which is balancing on the hilltop, or should it topple, rolling down the hill like a round grain of sand? We can define different "atoms" or "elements", such as sand, stone, water, lava, etc. by defining different behaviors and interactions of voxels in the more specific cases.

A table of the Elements is shown in *Figure 3*. The specifics of each Element are defined in *Section 3.1.2*. This set of Elements can be expanded in the future.

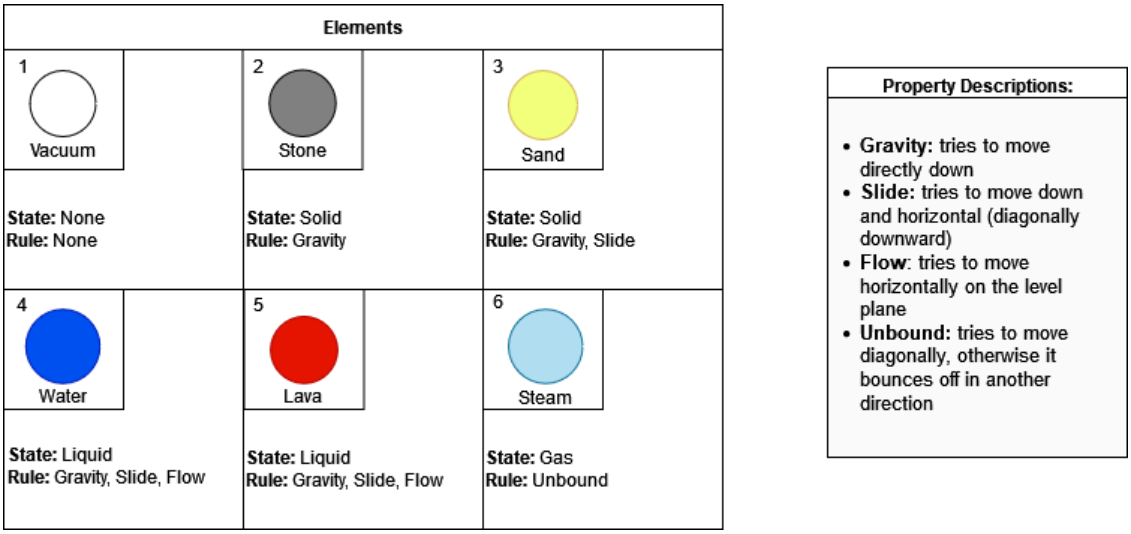



Figure 3: A simulation must define its irreducible physical components, called its "elements" or "atoms". This image shows a table of the Elements in this simulation and their physical rules.

### 3.1.2 Physicochemical Rules of the Elements

**The Physical Rules** At a given time step $t$, a voxel ($V$) has a *local position* in its neighborhood, specified by 3D coordinates $(p_x, p_y, p_z)$. Since the neighborhood dimensions are $2x2x2$, the local coordinates $p_x, p_y, p_z \in \{0, 1\}$. For example, a value $p_y = 0$ at a given timestep indicates that the voxel is on the *bottom* layer of the neighborhood, whereas $p_y = 1$ indicates the voxel is on the *top* layer of the neighborhood.

During the state transition phase, the voxel will attempt to move to a target cell at new position $(n_x, n_y, n_z)$ according to the voxel's *physical rules* (defined below). Each "physical rule" defines the behavior of a voxel's movement. These rules can be chained together to create solid, liquid, and gas voxels.

A voxel's movement succeeds if the target cell is empty. If the movement fails because $V$ is blocked by another voxel ($V_0$), then $V$ will attempt to displace $V_0$. A solid displaces liquids and gasses, whereas a liquid displaces only gasses. If the displacement fails, then the voxels will try to *chemically react*, converting them into different Elements. If the elements are not chemically reactive (as defined below), then the movement fails and $V$ will try to move to another position. The voxel $V$ will try to fulfill the new position $(n_x, n_y, n_z)$ on as many dimensions as possible, but will not always succeed due to being blocked by other voxels. Thus, the new position may retain part of the original position, e.g., $(n_x, p_y, n_z)$, $(p_x, n_y, p_z)$, etc. Finally, if a voxel cannot move at all, it must remain in its current position $(p_x, p_y, p_z)$.

Of course, a voxel is not permitted to move to any cell which is outside its local neighborhood, since that cell is by definition inaccessible.

**Gravity** A voxel moves directly down by 1 cell.

$$n_x = p_x$$

$$n_y = p_y - 1$$

$$n_z = p_z$$

Constraints: None.

**Slide** A voxel moves downs and off-center in any direction with a Chebyshev distance of 1 cell.

$$n_x = p_x + o_x$$

$$n_y = p_y - 1$$

$n_z = p_z + o_z$

**Constraints**: $(o_x \in \{-1, 0, 1\}) \wedge (o_z \in \{-1, 0, 1\}) \wedge \neg((o_x = 0) \wedge (o_z = 0))$

**Flow** A voxel moves in any direction on the level plane with a Chebyshev distance of 1 cell.

$n_x = p_x + o_x$

$n_y = p_y$

$n_z = p_z + o_z$

**Constraints**: $(o_x \in \{-1, 0, 1\}) \wedge (o_z \in \{-1, 0, 1\}) \wedge \neg((o_x = 0) \wedge (o_z = 0))$

**Unbound** A voxel tries to move to the opposing side on all dimensions.

$n_x = 1 - p_x$

$n_y = 1 - p_y$

$n_z = 1 - p_z$

**Element Rules** Each Element defines a sequence of the physical rules from *Section 3.1.2*. A voxel of an Element will behave according to the Element's rule sequence.

**Vacuum Rule**
Vacuum (empty) spaces do not themselves perform any actions. Instead, other voxels search for and move to them in order to traverse space.

**Stone Rule**

1. Gravity

Stone falls only straight down, and stops moving once it touches any ground. It does not slide down slopes, so dropping many Stone particles results in rigid and jagged pillars.

**Sand Rule**

1. Gravity
2. Slide

Sand falls straight down, and also slides down slopes. It stops moving once it touches stable ground. Dropping many Sand particles results in piles of Sand.

**Water Rule**

1. Gravity
2. Slide
3. Flow

Water falls straight downward, and also slides down slopes. It flows sideways when it is touching the ground. As a result of its flowing, Water refuses to pile up like sand, but instead levels out to fill its container.

**Lava Rule**

1. Gravity
2. Slide
3. Flow

Lava acts identically to Water.

**Steam Rule**

1. Unbound

When released, a Steam particle flies off in a certain direction. When many steam particles bounce off each other, they diffuse around the space.

**Chemical Rules** When a voxel is blocked from moving, it will try to displace or chemically react with the blocking voxel.

**Stone Formation** Lava + Water → 2 Stone

**Condensation** Steam + Water → 2 Water

## 3.2 Rendering

The algorithms for rendering the voxel world are described.

### 3.2.1 Meshing on CPU

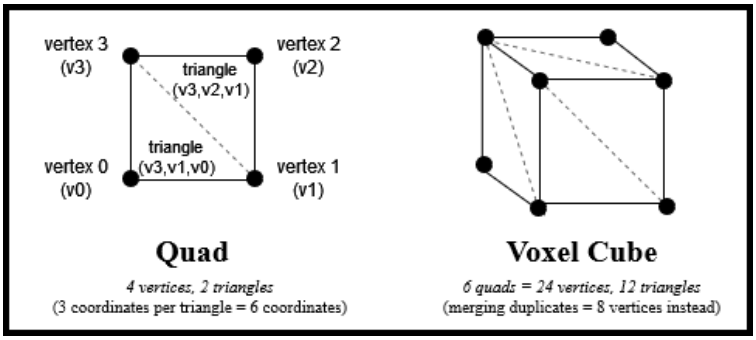



Figure 4: The mesh information required to render a quad (left) and a voxel cube composed of 6 quads (right).

**Meshing and meshes** The standard way to render objects is with "meshes". A "mesh" is a data structure specifying the coordinates of vertices and triangles in world space. These vertices and triangles form a surface, which is usually shaped to look like a 3D object (e.g., a cube, a sphere, etc.). See *Figure 5* for a screenshot.

Mesh information (vertices and triangles) is computed on the CPU. The goal of the meshing algorithm is to fill the mesh data structure with the proper vertices and triangles for the current timestep. It is then passed to the GPU for rendering. Here, the process for populating the voxel meshes on CPU is described.

For reference, a voxel is a cube, which consists of 6 quadrilateral/square faces (aka "quad"). Each quad consists of 4 vertices, one in each corner of the square. Despite a triangle needing 3 vertices each, 2 triangles (aka "tris") can be composed using these 4 vertices, by slicing the square along its diagonal. In that case, the triangles will share vertices. In total then, each voxel has 6 quads, times 4 vertices (or 2 tris), which is

24 vertices (or 12 tris) (see *Figure 4*). It is possible to optimize the voxel further, by merging vertices which are at the same position; that would result in 8 vertices in total.[1]

**Naive Meshing** In a naive meshing algorithm, we would sequentially iterate over all $n$ voxels. If the voxel is surrounded by other voxels, there is no need to render it (because it is hidden). Otherwise, we need to render at least part of the voxel. So, we append the voxel's mesh information to the mesh data structure. Sometimes, a voxel may be only partially surrounded by other voxels. In that case, there is no need to render the quads with neighboring voxels; only quads which neighbor a vacuum cell need to be rendered. Since each mesh can only render one material/color, one mesh is constructed for each Element.

**Parallel Meshing** The mesh calculation of one voxel is independent from all other voxel mesh calculations. Therefore, it is not necessary to calculate $n$ voxel meshes sequentially, instead they can all be computed and added to the mesh data structure simultaneously. This reduces the time complexity of the meshing algorithm from $O(n)$ to $O(1)$.

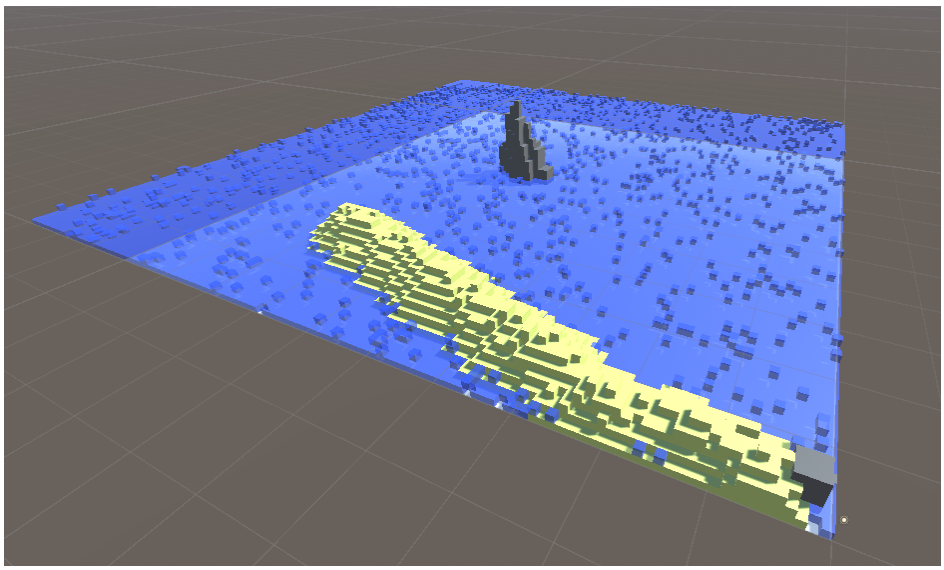


Figure 5: Rendering voxels with meshes.

### 3.2.2 Ray Tracing on GPU

The second rendering method bypasses the CPU, and instead performs rendering with the GPU alone. The color of each camera pixel is determined using the ray tracing algorithm described in [Amanatides et al., 1987]. It is efficient compared to regular ray tracing, because the 3D space is split into equal-sized cells. The ray can therefore travel voxel-by-voxel without risk of overshooting any objects. The algorithm was implemented in Unity's shader language (HLSL). See *Figure 6* for a screenshot.

## 4 Conclusion

By returning to the parallel method of cellular automaton for falling-sand simulator, the time complexity of the falling-sand algorithm is reduced from $O(n)$ to $O(1)$ (for $n$ voxels). When using sequential methods, only very small parts of the world ("chunks") could be computed at one time without a time-performance hit. However, the parallel cellular automaton method allows us to compute a world of any arbitrary size without any time-performance change, since a larger world takes the same amount of time to compute as a

[1] This optimization is not done here, since parallel processing is used, meaning data (such as vertices) cannot be shared across threads, and I wanted to simplify the process. However, it may be possible with some more thought.

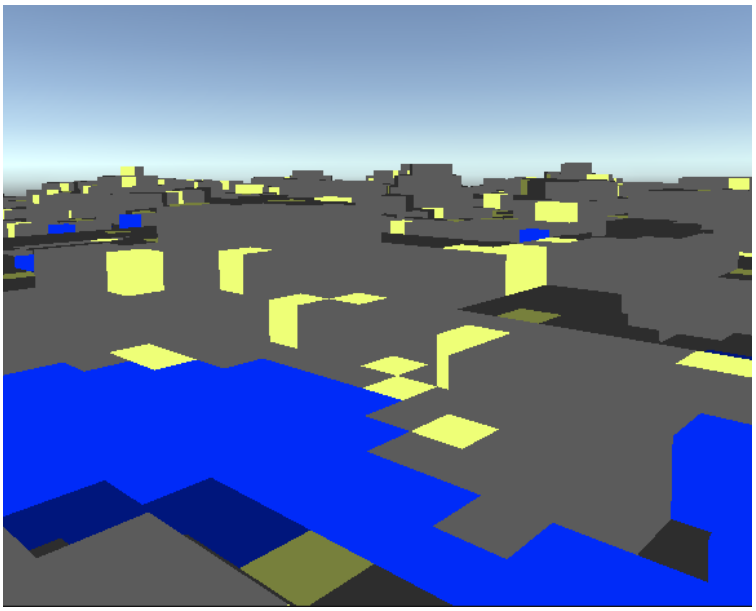


Figure 6: Rendering voxels with ray tracing.

smaller world. Furthermore, parallel algorithms are able to run directly on the GPU. Therefore, the cellular automaton's computation *and* rendering can be done on either the CPU *or* the GPU, depending on your preference.

In terms of complexity, the simulator is very simple. There are only a few Elements, and their reactions/behaviors are quite explicitly defined rather than emergent (such as the interesting behaviors of "gliders", "glider guns", etc. we see in Conway's Game of Life [Gardner, 1970]). We do see some emergent global structures, such as the sandpiles caused by sand falling, and the pools of liquid caused by water flowing. However, the complexity/function of these emergent structures is low, compared to patterns discovered in the Game of Life.

There is one fundamental issue with this simulator which should be addressed in the future works. Unlike the BBMCA, this simulator (like a sandpile CA) is not reversible. It is not possible to tell whether a given sand particle landed one timestep ago or 1000 timesteps ago; this is because the falling particle's energy is not conserved in the system once it hits the ground and stops moving. As such, once the particle's energy is lost, the simulation tends to settle into a loop of boring behaviors, rather than consistently exhibiting novel interesting behaviors as energy shifts around the system. To fix this simply requires conserving energy throughout the automaton, such as we see in the BBMCA.

It is certainly possible to at least add more Elements, such as those in the traditional falling-sand simulator games like oil, ice, etc. In the future, the simulation should exhibit higher complexity, with emergent behaviors, reversibility, and perhaps a more generalized atoms framework. However, this project already exhibits granular matter which acts like solids, liquids, and a gas, so it is now suitable to act as a very simple interactive 3D environment for human users and AI agents.

The code is open-source and available at `https://github.com/ccrock4t/3DCellularWorld`.

## Acknowledgments

This project was constructed in Unity game engine [Juliani et al., 2018]. The CPU code was in C#. The GPU code was in High-Level Shader Language (HLSL). The author thanks Professor Pei Wang for his comments on this report.

# References


[Amanatides et al., 1987] Amanatides, J., Woo, A., et al. (1987). A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, pages 3–10.

[d'Humieres et al., 1986] d'Humieres, D., Lallemand, P., and Frisch, U. (1986). Lattice gas models for 3d hydrodynamics. *EPL (Europhysics Letters)*, 2(4):291.

[Fredkin and Toffoli, 1982] Fredkin, E. and Toffoli, T. (1982). Conservative logic. *International Journal of theoretical physics*, 21(3):219–253.

[Frisch et al., 1986] Frisch, U., Hasslacher, B., and Pomeau, Y. (1986). Lattice-gas automata for the navier-stokes equation. *Physical Review Letters*, 56(14):1505–1508.

[Gardner, 1970] Gardner, M. (1970). The fantastic combinations of john conway's new solitaire game 'life'. *Sc. Am.*, 223:20–123.

[Gruau and Tromp, 2000] Gruau, F. and Tromp, J. (2000). Cellular gravity. *Parallel Processing Letters*, 10(04):383–393.

[Hahm, 2022] Hahm, C. (2022). Designing naturalistic simulations for evolving agi species. In *International Conference on Simulation and Modeling Methodologies, Technologies and Applications*. Springer.

[Hardy et al., 1976] Hardy, J., De Pazzis, O., and Pomeau, Y. (1976). Molecular dynamics of a classical lattice gas: Transport properties and time correlation functions. *Physical review A*, 13(5):1949.

[Juliani et al., 2018] Juliani, A., Berges, V.-P., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M., et al. (2018). Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*.

[LibreText, 2022] LibreText (2022). *Introductory Chemistry*. LibreText.

[Margolus, 1984] Margolus, N. (1984). Physics-like models of computation. *Physica D: Nonlinear Phenomena*, 10(1-2):81–95.

[Margolus, 1998] Margolus, N. (1998). Crystalline computation. *Feynman and Computation*, pages 267–305.

[Margolus, 2002] Margolus, N. (2002). Universal cellular automata based on the collisions of soft spheres. In *Collision-based computing*, pages 107–134. Springer.

[Purho, 2019] Purho, P. (2019). Noita: a game based on falling sand simulation. https://80.lv/articles/noita-a-game-based-on-falling-sand-simulation/ https://www.youtube.com/watch?v=prXuyMCgbTc.

[Wikipedia, 2023] Wikipedia (2023). Falling-sand game. https://en.wikipedia.org/w/index.php?title=Falling-sand_game&oldid=1170670140.

[Wilson, 1991] Wilson, S. W. (1991). The animat path to ai. In *From Animals to Animats: Proceedings of the first international conference on simulation of adaptive behavior*. MIT Press.

[Zuse, 1970] Zuse, K. (1970). *Calculating space*. Massachusetts Institute of Technology, Project MAC Cambridge, MA.