Performance impact of dynamically moving clouds using Cellular Automata in comparison to volumetric clouds.

Servaes Wouter

Graduation Work 2021-2022.
Howest University of Applied Sciences, Digital Arts and Entertainment, Game Development.

Abstract
A Cellular Automata cloud simulation model can provide complex, realistic-looking movement for clouds in a 3-dimensional environment. In this paper a comparison is made between the CA approach and the volumetric clouds approach. The CA model is explained, and both are implemented in Unity. The comparison is based on run-time performance, memory usage, and cloud quality. The measurements and quality of the clouds in the simulation are discussed. Cellular Automata provide more control over the motion of the clouds but perform worse than the volumetric clouds approach in runtime performance and scalability. The volumetric clouds approach is scalable, fast, and realistic enough. The model has a lot of room for future improvements in terms of performance and cloud quality, ideas for future work are proposed.

Table of Contents

1. Introduction

Clouds are an essential part of any scene set in an outdoor environment. This paper looks at clouds set in a 3-Dimensional (3D) environment for games, as the Unity Game Engine is used to create, showcase, and compare the proposed and explored method. Current methods for clouds in 3D environments include sky domes with clouds, moving cloud meshes, particle clouds and volumetric clouds. These methods are often performance friendly and look decent to good but lack the dynamic movement clouds have when interacting with the environment.

Cellular Automata (CA) simulations are interesting for many fields because they can represent emergent and dynamic behaviour on a large scale without too much computing power and are generally easy to implement. CA have been used for many applications; in games from world generation to crowd behaviour; in many scientific fields from meteorologic applications [1] (i.e. weather simulations) (Fig. 1) to brain dynamics research [2] (Fig. 2).

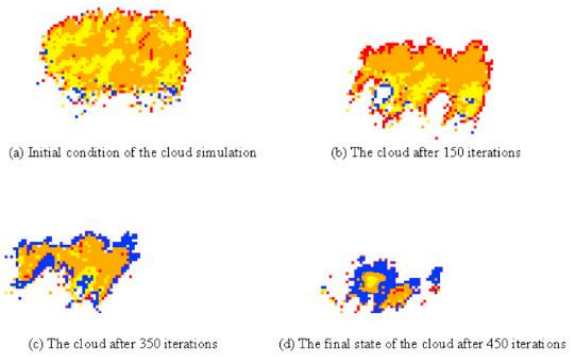

Figure 1. Meteorological cloud simulation using Cellular Automata [1, section 6].

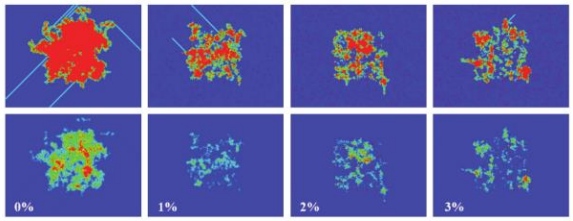

Figure 2. Artificial Brain Dynamics research, using the Game Of Life (3.1) as the base model [2, section 5].

In this paper a method is explored to create dynamically moving clouds using Cellular Automata. This can be used to create realistic dynamically moving clouds in mountainous terrain, high-rise city environments, “sky islands”, dust- and gas clouds and other game environments. The method is compared to 3D volumetric noise clouds from a performance and implementation feasibility standpoint. This paper does not go over the visuals of the clouds but does provide methods to implement visuals in future work. A section about the implementation of CA-based clouds is also provided at the end of the paper. To compare the performance results between the dynamically moving clouds using CA and the volumetric clouds using noise, volumetric clouds will also be needed in this paper, this will be done in Unity.

The paper is organized as follows: A brief introduction to CA followed by the description of the CA cloud model. A brief description of volumetric clouds. After implementing the CA cloud model and volumetric clouds, performance metrics are explained, measured, shown, and discussed. Finally, we conclude by summarizing the research and describing possible future work.

2. Related works

Considering clouds in 3D applications like games, there are two main approaches: the physically based approach and heuristics approach. The physically based approach

simulates the physical process of fluid dynamics. Most physically based methods need a large amount of computation time [3]. Stam developed a fast technique by simplifying the fluid dynamics [25]. A detailed physically based modelling technique was proposed by Overby et al [3] (Fig. 3), which uses a computational fluid solver and was a significant improvement over previous models of cloud formation in the field of Computer Graphics.

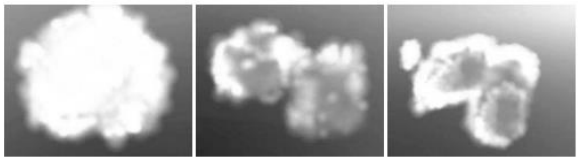

Figure 3. Example of cumulus clouds using Overby's computational fluid solver in [3, section 3.1].

The second approach is the heuristics approach, a simulation based on heuristics that use theoretical, non-mathematical descriptions to create a role-based system. The heuristics approach includes particle systems [4, 5], procedural noise [6], voxel-based simulating methods [7], volumetric clouds [20,21,22,24], Cellular Automata (CA), etc. The method researched in this paper falls under this heuristics approach.

Nagel et al proposed the use of CA to simulate the formation of clouds in 1992 [8] and found it a good approach. Dobashi et al demonstrate and improve the technique [9,10]; By adding Boolean variables and transition rules to simulate cloud formation, cloud extinction, simple wind effects, and controlling of cloud motion, they demonstrated its usefulness. In comparison to a physically based approach, CA greatly improve simulation efficiency because its transition rules are simple, and no complex Partial Differential Equations (PDEs) need to be solved.

Fan et al established a model of clouds based on CA, improving older state transition rules, and developed a method to deal with boundary grid points [11]. Christopher et al proposed a method for simulating the dynamic movement of clouds using CA [12] and provides a method to control the shape and appearance of clouds through custom shaping routines.

3. Cellular Automata clouds

3.1 Basic idea of Cellular Automata

A Cellular Automaton consists of a n-dimensional grid of cells, each in one of a finite number of states, which are either on or off states. Each cell has a neighbourhood that determines the state of said cell. The most common types of neighbourhoods are the Moore neighbourhood, the eight cells surrounding a central cell (Fig. 4), and the von Neumann neighbourhood, the four cells adjacent to a central cell (Fig. 5).

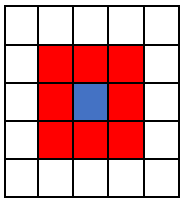

Figure 4. Moore neighbourhood, eight cells surrounding a central cell.

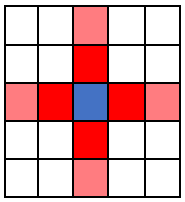

Figure 5. von Neumann neighbourhood, four cells adjacent to a central cell with a range of two cells.

The initial state, at $t = 0$, is selected by assigning a state for each cell. A new generation is created according to a specific fixed rule, or set of specific fixed rules, that determine(s) the new state of a cell in terms of the state of a cell and the states of its neighbours. The rule for updating the state of each cell is the same for every cell, does not change over time and is applied to the whole grid simultaneously [13, 14].

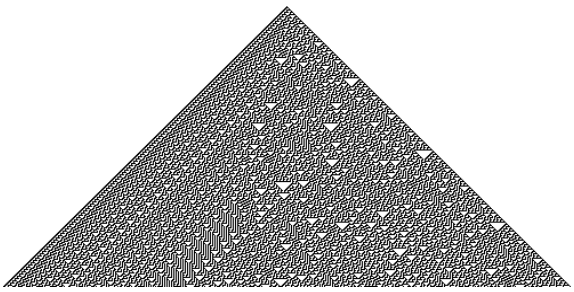

Figure 6. Famous 2D Cellular Automaton Rule 30, simple input patterns lead to chaotic, seemingly random histories [13].

Perhaps the most well-known Cellular Automaton is Conway's Game of Life (GoL) [16]. In this zero-player game an initial state is given to let the CA evolve without further user-input. With only four rules (Fig. 7) and two states the GoL simulates the dynamic evolution of a society. Many types of patterns occur in the GoL which are classified according to their behaviour, such as "still lives", "oscillators" and "spaceships". GoL is Turing complete and can simulate a universal constructor or any other Turing machine.

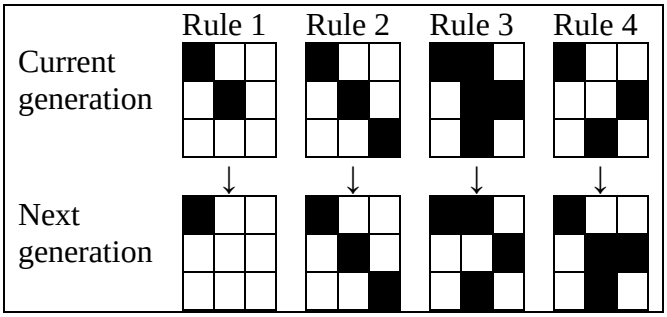

Figure 7. Conway's Game of Life rules [16]:

1. Any live cell with fewer than two live neighbours die, as if by underpopulation.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

3.2 Cellular Automata Clouds

3.2.1 Basic idea

According to the method proposed by Nagel et [8] al, which was later improved by Dobashi et al [9,10] and numerous other papers on the subject [5,11,12, 15] a 3D simulating space is subdivided into a grid of cells, each cell has three logical state variables: humidity/vapor (*hum*), clouds (*cld*) and activation/phase transition (*act*) (Fig. 8). The state of each variable is either 0 or 1. *Hum = 1* means vapor exists to form the cloud, *act = 1* means the phase transition from vapor to cloud is ready, and *cld = 1* means the cloud is formed.

Figure 8. Representation of *hum*, *cld* and *act* in examples

The cloud evolution is simulated by applying multiple methods which represent formation, growth, extinction, advection by winds, and environment interaction. Since the state variables are either 0 or 1, these methods can be expressed by Boolean operations and the variables can be stored as one bit, saving memory cost, and decreasing process time of the operations. The model uses a von Neumann neighbourhood with a range of two.

3.2.2 Cloud initial formation

The initial state of the CA is done before the first generation, at $t = 0$. The states hum and *act* are randomly set to 1 for each cell according to given probabilities. *Act* can only be set to 1 if *hum* is 0, and *cld* starts at 0 for every cell, described in equations 1 & 2 below this paragraph. This method does not make the best results: it is not realistic; a lot of single cloud cells are made because the randomness is too random, which don't grow into full clouds but instead stay active until they go extinct (section 3.2.4). Setting the initial cloud using a noise function or something similar might be a better idea.

$$hum(i,j,k,t_{i=0}) = rnd \leq p_{hum} \quad \text{E.1}$$

$$act(i,j,k,t_{i=0}) = \neg hum(i,j,k,t_{i=0}) \wedge rnd \leq p_{act} \quad \text{E.2}$$

Initial formation of clouds ($t = 0$).

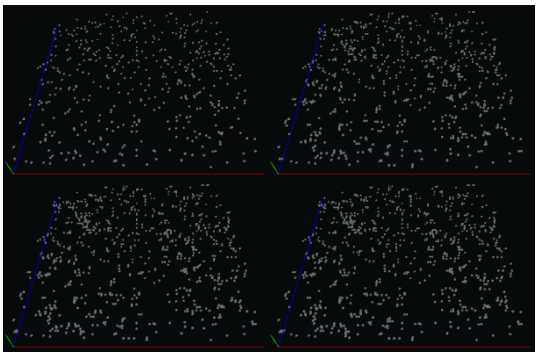

Figure 9. Initial formation and growth of clouds, first four generations

3.2.3 Cloud dynamic growth

The dynamic growth of clouds using CA was first described by Nagel et al [8]. Clouds form when warm air from the surface rises to colder regions in the atmosphere, cooling the warm air down by expanding the warm air bubble in the cold air. This causes the relative humidity inside the warm air bubble to increase, resulting in a phenomenon called "phase transition", which changes the water vapor in the bubble to water droplets. This can be described by three state transition functions, equations 3 to 5. The method forms the clouds in a realistic way, but once the clouds are formed, the animation stops, and the clouds don't disappear. Cloud extinction, introduced by Dobashi et al [9], is described in the next section of this paper (3.2.4).

$$hum(i,j,k,t_{i+1}) = hum(i,j,k,t_i) \wedge \neg act(i,j,k,t_i) \quad \text{E.3}$$

$$cld(i,j,k,t_{i+1}) = cld(i,j,k,t_i) \vee act(i,j,k,t_i) \quad \text{E.4}$$

$$act(i,j,k,t_{i+1}) = \neg act(i,j,k,t_i) \wedge hum(i,j,k,t_i) \wedge f_{act}(i,j,k) \quad \text{E.5}$$

Dynamic growth of clouds.

In equation five, $f_{act}(i,j,k)$ is a Boolean function that gets its result by looking at the status of surrounding cells. This Boolean function has many different variations, but according to Debashi et al [10] there is no significant difference between them. The function used is the function described in the original paper by Nagel et al [8] (E.6).

The rules described in this section are summarized in fig. 10.

$$f_{act}(i,j,k) = \text{act}(\text{i}+1,\text{j},\text{k},\text{t}_i) \vee \text{act}(\text{i},\text{j}+1,\text{k},\text{t}_i) \vee \text{act}(\text{i},\text{j},\text{k}+1,\text{t}_i) \vee \text{act}(\text{i}-1,\text{j},\text{k},\text{t}_i) \vee \text{act}(\text{i},\text{j}-1,\text{k},\text{t}_i) \vee \text{act}(\text{i},\text{j},\text{k}-1,\text{t}_i) \vee \text{act}(\text{i}+2,\text{j},\text{k},\text{t}_i) \vee \text{act}(\text{i}-2,\text{j},\text{k},\text{t}_i) \vee \text{act}(\text{i},\text{j}+2,\text{k},\text{t}_i) \vee \text{act}(\text{i},\text{j}-2,\text{k},\text{t}_i) \vee \text{act}(\text{i},\text{j},\text{k}-2,\text{t}_i) \quad \text{E.6}$$

Figure 10. Representation of growth rules. Humid cell goes through phase transition and becomes a cloud.

3.2.4 Cloud Extinction

After a cloud is formed it will eventually disappear. The method Nagel et al [8] proposed did not include cloud extinction, after *cld* was set to 1, it would remain 1. Dobashi et al [9] introduced a method to simulate cloud extinction. This method repeats formation and extinction frequently, resulting in unnatural animations. In their next paper about the subject, Dobashi et al improve the cloud extinction method as follows [10]: When the *cld* variable of a cell equals 1, a random number is generated ($0 \leq rnd \leq 1$). If this random number is smaller than a given extinction probability (p_{ext}), *cld* is set to 0, the cloud disappears. p_{ext} can be changed at different times and can be different for each cell, thus allowing the animator to specify regions of less and more frequent cloud extinction.

To allow the cloud to regenerate, vapor (*hum*) and phase transition (*act*) probabilities (p_{hum} and p_{act}) are given at specified time intervals and once again compared to *rnd*. If $rnd < p_{hum}$ then *hum* is set to 1, else if $rnd < p_{act}$ then *act* is set to 1. By controlling the three probability variables, p_{ext}, p_{hum} and p_{act}, cloud motion can be controlled at each cell at each time step. The random number *rnd* is only generated once per time step per cell, all three probabilities are compared to this

random number that step. The above-described methods can be summarized as three transition rules:

$$hum(i,j,k,t_{i+1}) = hum(i,j,k,t_i) \vee (rnd < p_{hum}) \quad \text{E.7}$$
$$cld(i,j,k,t_{i+1}) = cld(i,j,k,t_i) \wedge (rnd > p_{ext}) \quad \text{E.8}$$
$$act(i,j,k,t_{i+1}) = act(i,j,k,t_i) \vee (rnd < p_{act}) \quad \text{E.9}$$

Cloud extinction and regeneration.

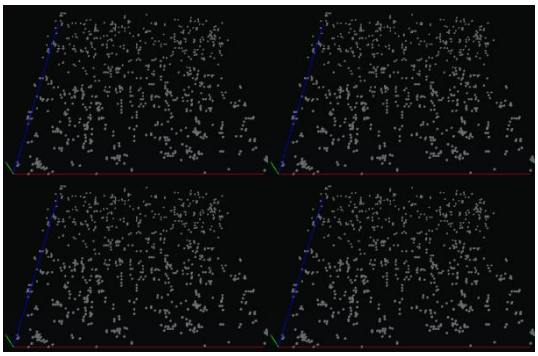

Figure 11. Cloud extinction and generation, four generations.

A different method for cloud extinction was proposed in [11], this new method builds upon the method by Dobashi et al [10]. It adds an extra state variable *ext* to the CA, when *ext = 1* and *cld* = 1 a disappearing timer is introduced and *cld* is set to *0*. During this disappearing time, the cloud cannot be regeneration, avoiding frequent switching between extinction and generation.

3.2.5 Advection by wind

Clouds move in one direction, following the wind. As Dobashi et al described it [10], to add the effect of wind to the simulation, we move all the variables of each cell to its neighbouring cell following the x-axis. To control the direction of the wind, the whole CA grid is rotated to a given direction.
The velocity of wind changes depending on the height in the sky, it increases with increasing height from the ground (due to the no-slip condition [17]). Therefore, the wind velocity of each cell (i,j,k) is specified as a function using the z coordinate of the cell, $v(z_k)$ (*z* describing the z coordinate of the cell in the world, and *k* the row number of the cell in the grid), this is a piecewise linear function. To implement this wind velocity to our CA, new transition rules are added, described by Dobashi et al [10]. These set the state of each cell to the state of a cell whose position is calculation by $(i - v(z_k))$, described as follows:

$$hum(i,j,k,t_{i+1}) = (i - v(z_k) \geq 0) \wedge hum(i - v(z_k),j,k,t_i) \quad \text{E.10}$$
$$cld(i,j,k,t_{i+1}) = (i - v(z_k) \geq 0) \wedge cld(i - v(z_k),j,k,t_i) \quad \text{E.11}$$
$$act(i,j,k,t_{i+1}) = (i - v(z_k) \geq 0) \wedge act(i - v(z_k),j,k,t_i) \quad \text{E.12}$$

Advection by wind.

Depending on the implementation of the CA, these rules may not affect the cells at *i=0*. This will make the states of the cells at *i=1* to *i=n* (*n* representing the column count following the x-axis of the CA grid) copy the states of cells at *i=0*, to avoid this, the states of cells at *i=0* are set to 0 when processing the advection by wind rules.

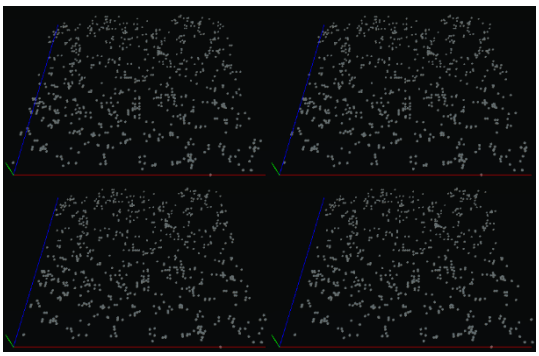

Figure 12. Cells moving along the x-axis in the wind, four generations.

3.2.6 Boundary grid cells

When processing the transition rules of boundary grid cells, calculations will exceed the grid. As previously mentioned, Fan et al proposed a method on how to handle these boundary grid cells [11], earlier papers do not mention it. In the calculation function (E.6) we confirm that the referenced cells around the current cell are in the grid space and valid. If a cell around the current cell is not valid, that part of the calculation

function is simply removed, not affecting the current cell in any way.

4. Volumetric clouds

The volumetric clouds modelling approach has been popular in current- and next-generation games. What would have previously been done using sky domes and billboarding panes, methods that leave little to no control over the clouds at runtime, is now done using 3D animated noise. In this paper, the implementation of volumetric clouds by Sebastian Lague is used, which can be found on Youtube [23] and is under MIT Licence. Lague's implementation is qualitative and provides customization for the clouds. Their approach is based on multiple papers [22, 24], most notably Guerrilla Games' Horizon Zero Dawn SIGGRAPH 2015 presentation [24]. Andrew Schneider and Nathan Vos, the authors of the presentation, provide a detailed account on the implementation of volumetric clouds in the game. The approach from the presentation was a first of its kind, using a combination of Worley and Perlin noise to get the cloud shapes and animation.

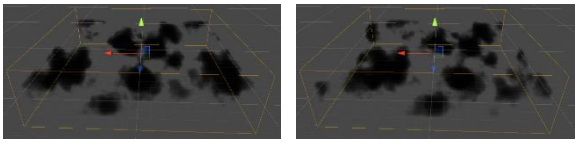

Figure 13. Volumetric clouds using a combination of Worley and Perlin noise, from S. Lague's implementation [23].

5. Implementation

When first implementing the CA model for this paper, all calculations were done on the CPU, in Unity with c#. The obvious disadvantage, the lack of performance, quickly became clear, so it was decided to implement the CA model on the GPU using a compute shader. In the following section, the implementation for the model on both CPU and GPU is explained.

5.1 CA clouds implementation CPU

At the start of the application, a grid is defined with a number of rows (*r*), columns (*c*) and depth (*d*), this was done using the *SciptableObject* type of Unity for easy access across the whole application. After defining the grid, all containers are initialized: two containers of type *BitArray*, for each state of the CA. In our case, six containers. *BitArray* is a reference type native to the .NET library, it stores bits that can be read as booleans. Two containers per state are needed, as one will be the current container used in the calculation and the other is used to save the calculation results, as explained in 3.1.

```
private BitArray _Act, _Cld, _Hum,
_NextAct, _NextCld, _NextHum;
```

Then, the initial state of the CA is made by going over each cell of the grid and following the method described in 3.2.2, E.1 and E.2. The probabilities are set by the user using a *ScriptableObject*.

```
bool humAtStart = Random.Range(0f, 1f) <=
_CASettings.HumProbabilityAtStart;
_Hum[idx] = humAtStart;
if (!humAtStart)
   _Act[idx] = Random.Range(0f, 1f) <=
_CASettings.ActProbabilityAtStart;
```

Now, everything is set for the clouds to start growing, go extinct, regenerate, and get carried away by the wind. These things are done in the *Update()*, by calling a function for each method with the *cellIdx* as parameter. The first function to be called is the one handling the growth of the cloud, as described in 3.2.3, E.3 to E.5. See code in appendix F.

To set *act* of the cell in this growth function, a Boolean function is required that looks at the states of the surrounding cell, described in E.6. This also holds boundary grid cells

into account, described in 3.2.6. *cellIdxI*, *cellIdxJ* and *cellIdxK* represent the row-, column- and depth index of the cell, these can be calculated from the *cellIdx*. The function *ThreeDToOneDIndex()* returns the *cellIdx* of the cell given the *i, j,* and *k* of the cell. Code in appendix G.

The next rules to process in *Update()* are the extinction and regeneration rules. First, a random value is generated using Unity's Random generator. Then, the rules described in 3.2.4, E.7 to E.9, are processed as follows:

```
float rand = Random.Range(0f, 1f);
//hum
_NextHum[cellIdx] = _Hum[cellIdx] || rand < 
_CASettings.HumProbability;
//cld
_NextCld[cellIdx] = _Cld[cellIdx] && rand > 
_CASettings.ExtProbability;
//act
_NextAct[cellIdx] = _Act[cellIdx] || rand < 
_CASettings.ActProbability;
```

Again, the probabilities are set by the user using a *ScriptableObject*.

The last function called in the loop processes the advection by wind, described in section 3.2.5, E.10 to E.12. First, the height of the cell in the world is calculated; This was done by calculating the row of the cell, multiplying it with the height of the cells and adding the bottom position of the grid in the world to this. Then the displacement of the cell in the grid is calculated using a piecewise linear function that takes the height of the cell using a *WindHelper* static class that holds the *WindSpeed*. Finally, the states of the cell are shifted in the grid according to the displacement if the displacement of the cell results in a valid position on the grid. As described at the end of 3.2.5, the states of the cells in the first column (*cellIdxI = 0*) are set to false. See code in appendix H.

Now all the rules are processed, and the next states of the cells are known, the *next* containers become the *current* container, as explained in 3.1.

```
_Act = (BitArray)_NextAct.Clone();
_Cld = (BitArray)_NextCld.Clone();
_Hum = (BitArray)_NextHum.Clone();
```

To visualize the cloud cells, at the end of the loop if *_cld[cellIdx]* is true, the local position of the cell is saved to a ComputeBuffer which is used to draw instanced meshes on those locations through a point shader.

5.2 CA clouds implementation GPU

The implementation of the Cellular Automata on GPU happens via a compute shader. One of the main differences is the lack of a bit type in the shader language, so the bits in integers are used. This adds a layer of complexity to the method but is well worth it considering the amount of memory saved by doing this instead of saving the states as normal Boolean types.
Shaders don't support generating random values. This is a major issue for the cloud simulation, as a different random number is required for every cell, every update, at the cloud initialization and cloud extinction and regeneration (section 3.2.2 and 3.2.4). Different pseudo random number generation methods were explored, multiple noise and hash functions, but none gave the randomness required for the project. The main reason for this lack of randomness are the bad input parameters given to these functions. The low grid resolution and no difference in kernel (as everything was done on just one kernel), or thread, meant that most cells would give close to the same values to the noise and hash functions. However, this does not mean that this

method could not work and further research on the topic should be conducted. As a result, the clouds were initialized on the CPU, and cloud extinction and regeneration were excluded from the GPU project.
At the start of the project, six ComputeBuffers are made with a stride of *sizeof(int)*, four bytes, and a length of *TotalInts*.

```
private ComputeBuffer _ActBuffer, _CldBuffer, _HumBuffer, _ActNextBuffer, _CldNextBuffer, _HumNextBuffer;
```

TotalInts represents the total amount of integers needed to store every cell as a bit:

```
TotalInts = Mathf.CeilToInt(TotalCells / 32f);
```

These buffers are filled with zero, so every state would be false. Next, the initial state of the grid is generated on the CPU (explained in a previous paragraph), local *act* and *hum* containers are made and filled in the same way as explained in section 9.1, paragraph 2. Then, the *_ActBuffer* and *_HumBuffer* buffers are set with the values of the local containers.

After the initial state of the grid has been set, the *Update()* function will set the required variables and buffers of the ComputeShader and will dispatch the kernel of the shader. As explained in section 8, Future Work, only one kernel without multithreading is used for the cloud simulation.

In the shader, everything is virtually the same as the CPU implementation. A loop goes over every cell, *_CellCount* was given to the GPU before the dispatch. In the loop, the growth rules are first processed, then the advection by wind and at last the position of the cell is stored in a position buffer if *cld* of the cell is true.

As previously mentioned, the buffers representing the states of the cells are made up of integers but used as bit containers. To get and set the state of a cell helper functions were made, shown after this paragraph. To access a specific bit in the buffer, a function is called that returns the index of the integer in the buffer and the index of the bit in this integer:

```
void GetBitPosition(const uint cellIdx, out uint intIdx, out uint bitIdx)
{
    const float IntBits = 32.f;
    intIdx = floor(cellIdx / IntBits);
    bitIdx = cellIdx - (intIdx * (int)IntBits);
}
```

To get and set the state, one of the following functions is called. These functions require the integer containing the cell from the buffer, *data*, and the index of the bit, *bitPosition*. The *Set()* and *Reset()* functions return the new integer for the buffer containing the updated state of the cell. *CheckBit()* simply returns true or false according to the state of the bit. See code in appendix I.

After processing the rules, the growth and advection by wind rules, in the exact same way as the CPU implementation (slightly modified to use the bit functions), a function is called to possibly add the cell's position to visualize later. The cell's position is saved to a *float3* buffer *_CloudPositions* if the *cld* of the cell is true. See code in appendix J.

An additional buffer is used for variables that are accessed in both script and shader, *_IntVariables*. These variables are the generation counter and the amount of cloud cells this generation.

After going over every cell of the grid, when all the states are set to the *next* buffer, the

current buffer becomes the *next* buffer, as explained in section 3.1.

```
for (uint intIdx = 0; intIdx < _CellCount * 32; intIdx++){
        _Cld[intIdx] = _CldNext[intIdx];
        _Hum[intIdx] = _HumNext[intIdx];
        _Act[intIdx] = _ActNext[intIdx];
    }
```

Now all the operations on the GPU are complete and the cells can be drawn. This is done using a point shader. The *_CloudPositions* buffer is passed to this point shader and the cells are drawn using Unity's *Graphics.DrawMeshInstancedProcedural*.

6. Metrics

In this section of the paper, the metrics that are measured are described, in section 6.1 the results are shown and in section 6.2 a conclusion is made. The metrics measured are initial start duration for the simulation, calculation/process duration per tick (with and without wind), and memory usage. Both CPU and GPU applications for the CA cloud simulation will be measured, in both cases the visualization of the cells is done via the GPU. S. Lague's volumetric cloud application will be used for the volumetric cloud metrics (Vol. Clouds), which uses the GPU [23].

Three grid sizes for the CA Simulations will be used, size 2 is the size and attributes used in the papers of Dobashi et al [9,10]:

	resolution	cells
Size 0	10x10x10	1000
Size 1	100x10x100	100000
Size 2	256x20x128	655360

For each size, different probability values are used, these are set to give approximately the same result in the grid (*pActStart* and *pHumStart* refer to *pAct* and *pHum* from E.1 and E.2):

	pActStart	*pHumStart*	*pExt*	*pHum*	*pAct*
Size 0	.01	.1	.1	.8	.1
Size 1	.005	.05	.05	.5	.05
Size 2	.001	.01	.01	.1	.01

For the volumetric clouds, the settings are set to get the best-looking clouds. The resolution of the Worley noise maps is 132x132 and the resolution of the Perlin noise maps is 64x64. The size of the container in which the clouds are rendered, shown in figure 13, will be the same, as this did not impact the performance in any way while testing.

The applications are played through Unity 2020.3.19f1, a Long-Term Support (LTS) version. On a Windows 10 laptop with an Intel Core I5-9300H Quad Core CPU at 2.4GHz, 16.0 GB Dual Channel RAM, NVIDIA GeForce GTX 1650 graphics card.

6.1 Results

6.1.1 Initial start duration

The simulation is started 10 different, consecutive times, the time is measured to generate the first generation, $t = 0$. For the CA simulation this means the cloud initial formation, section 3.2.2. The GPU application was not included as the initial states are set on the CPU, the same as the CPU application. For the volumetric clouds this means generating the noise maps used in the simulation. The results can be seen in appendix A.

6.1.2 Calculation duration without wind

The simulation is played three different times, five consecutive generations are measured (not included initialization), $t = n$ to $t = n + 5$. The results can be seen in appendix B. The simulations include cloud

growth, extinction, and regeneration. Advection by wind is NOT included here, it is in the next section. Size 2 for the GPU CA applications are not shown, the project was unplayable with < 1 fps.

6.1.3 Calculation duration with wind

As in the previous section, the simulation is played three different times, five consecutive generations are measured (not included initialization), $t = n$ to $t = n + 5$. The results can be seen in appendix C. The simulations include cloud growth, extinction, regeneration, and advection by wind. Size 2 for the GPU CA applications are not shown, the project was unplayable with < 1 fps.

6.1.4 Memory usage

The memory needed for the simulation is measured. The amount doesn't chance while playing for both the CA and volumetric cloud simulations. The results for the CA cloud applications can be seen in appendix D. Appendix E for the volumetric cloud memory usage.

6.2 Metrics discussion

The metrics measured in the previous sections clearly show that, in this paper with the implementation used, the volumetric clouds performed better. The Cellular Automata is not scalable without taking a big performance hit, any one added cell means an extra cycle must be done in the calculation loop and extra memory is used (at least 6 bits for each cell). The "big O" notation of the method is $O(n)$, n representing the cell count.

Currently, the difference between the GPU and CPU applications are big. Using only one kernel without multithreading and having to set all the buffers and variables every tick before the dispatch leaves no room for performance, this should be addressed in further research, see section 8 Future Work.

The loss of performance when wind is added to the simulation is also significant, as multiple operations take place for every cell un regardless of the state of this cell. The first couple of generations in the CA cloud simulation take more time, as a lot of cloud extinction happens in these first generations after initialisation. In this paper, the visuals of the clouds are also not considered, which would add additional strain on the system.

The volumetric clouds approach leaves us with optimal runtime performance but uses more memory, to store the different noise maps, it also is scalable. The CA cloud application's memory usage scales with the size of the grid. The volumetric cloud's memory usage is the same, it depends on the resolution of the noise maps that don't affect the size of the cloud container in the world.

7. Cloud Quality discussion

As written in the metrics discussion above, from a performance standpoint the CA approach loses against the volumetric clouds approach. But is this also true for the quality of the clouds?

The CA approach currently gives more control over the clouds' generation and motion, and could add even more control which is talked about in Future Work, section 8. But the results that can be acquired from volumetric clouds are surprisingly realistic in looks and behaviour, as seen in multiple games including Horizon: Zero Dawn [24]. True, the CA approach gives more random, complex, and realistic cloud behaviour which might be interesting for certain applications, for games the volumetric clouds approach is preferable.

Now, more about the CA approach. The cloud generation and motion leave realistic results, which is also true for the extinction of the clouds. The regeneration of clouds in the project made alongside this paper leaves a lot to be desired. The parameters of these properties are difficult to get right and need to be changed according to the size of the grid. In the project, the wind often blows faster than the generation of the clouds and thus leaves big open gaps in the world.

8. Conclusion

In conclusion, the CA approach gives more control over the behaviour of the clouds, this however is hard to get right and needs further research. The performance of the approach is an issue, as said in section 5.2. Processing the grid needs to be parallelised and optimized, the little added realism is not worth the loss in performance for real time applications. Volumetric clouds give us enough realism in terms of cloud motion, can leave us with beautiful pictures [24] and perform great, so this approach is preferable for real time applications like games.

9. Future work

Cellular Automata cloud simulations have been and are actively researched, in this section possible future work is talked about.

One of the original ideas in this paper, established in the introduction, is to make the clouds flow through the air around high-rise building, mountains, trees, and other environmental obstacles. A method for this has not been established and thus needs further research. The dynamic movement of clouds interacting with the environment has a lot of similarities with fluid simulation, methods for 3D CA-based fluid simulations exist [18], so combining the two might be possible and could give interesting results. Another method that shares similarities to the movement of clouds is oil spill simulations [19], which is also closely related to fluid simulations. Perhaps a simpler method would be making the wind direction dynamic and affecting each cell instead of the whole grid; Instead of shifting every cell in the same direction (as described in section 3.2.5, E.10 to E.12), every cell moves along the direction of the wind at that cell. Having a different wind direction for each cell would allow for even more dynamic movement and the direction of the wind could steer the clouds away from the environment.

The CA currently lacks in scalability, it would not be possible to have a CA cloud simulation for a big scene, which would be possible with volumetric clouds [24]. Scalability could be solved by improving the performance of the method, for example by parallelising the 3D CA to allow better runtime performance and bigger scenes.

This paper also did not take visuals of the clouds into account, methods for this do exist (particles systems with meta-balls) and should be used in future research [5,8,9,10,11,12]. The initial state of the cloud simulation is currently done with random numbers, using a noise function or similar methods might give more interesting results to start the simulation with.

References

1. Silva, A. R., Silva, A. R., & Gouvêa, M. M. (2019). A novel model to simulate cloud dynamics with cellular automaton. https://doi.org/10.1016/j.envsoft.2019.104537
2. Fraile, A., Panagiotakis, E., Christakis, N., & Acedo, L. (2018). Cellular Automata and Artificial Brain Dynamics. https://doi.org/10.3390/mca23040075

3. Overby, D., Keyser, J., & Melek, Z. (2002). *Interactive Physically-Based Cloud Simulation*. https://doi.org/10.1109/PCCGA.2002.1167904
4. Reeves, W. T. (1983). Particle Systems - Technique for Modeling a Class of Fuzzy Objects. https://doi.org/10.1145/964967.801167
5. Bi, S., Bi, S., Zeng, X., Lu, Y., & Zhou, H. (2016). 3-Dimensional modeling and simulation of the cloud based on cellular automata and particle system. https://doi.org/10.3390/ijgi5060086
6. Man, P. (2006). Generating and Real-Time Rendering of Clouds. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.103.3172
7. Neyret, F. (1997). *Qualitative Simulation of Convective Cloud Formation and Evolution*. https://doi.org/10.1007/978-3-7091-6874-5_8
8. Nagel, K., & Raschke, E. (1992). Self-organizing criticality in cloud formation? https://doi.org/10.1016/0378-4371(92)90018-L
9. Dobashi, Y., Nishita, T., & Okita, T. (1998). Animation of clouds using cellular automaton. https://www.researchgate.net/publication/244453100_ANIMATION_OF_CLOUDS_USING_CELLULAR_AUTOMATON
10. Dobashi, Y., Kaneda, K., Yamashita, H., Okita, T., & Nishita, T. (2000). A simple, efficient method for realistic animation of clouds. https://dl.acm.org/doi/10.1145/344779.344795
11. Fan, X. (2014). *Real-time Rendering of Dynamic Clouds.* https://www.researchgate.net/publication/288816683
12. Christopher Immanuel, W., Paul Mary Deborrah, S., & Samuel Selvaraj, R. (2014). Application of cellular automata approach for cloud simulation and rendering. https://doi.org/10.1063/1.4866854
13. Cellular Automaton. (2021). In *Wikipedia*. https://en.wikipedia.org/wiki/Cellular_automaton
14. Wolfram, S. (2002). A New Kind of Science. https://www.wolframscience.com/
15. Da Silva, A. R., & Gouvêa, M. M. (2010). Cloud dynamics simulation with cellular automata. https://www.researchgate.net/publication/221113065_Cloud_dynamics_simulation_with_cellular_automata
16. Conway's Game of Life. (2021). In *Wikipedia*. https://en.wikipedia.org/wiki/Conway's_Game_of_Life
17. Wind gradient. (2021). In *Wikipedia*. https://en.wikipedia.org/wiki/Wind_gradient
18. Cattaneo, G., & Jocher, U. (2005). *Cellular Automata for 2D and 3D fluid- dynamics simulations*.
19. Shyue, Shiahn-wern & Sung, H.-C & Chiu, Y.-F. (2007). *Oil spill modeling using 3D cellular automata for coastal waters*. https://www.researchgate.net/publication/286804292_Oil_spill_modeling_using_3D_cellular_automata_for_coastal_waters
20. Olajos, R. (2016). *Real-Time Rendering of Volumetric Clouds*. https://lup.lub.lu.se/student-papers/search/publication/8893256
21. Liao, H. S., Ho, T. C., Chuang, J. H., & Lin, C. C. (2005). Fast rendering of dynamic clouds. https://doi.org/10.1016/j.cag.2004.11.005
22. Häggström, F. (2018). *Real-time rendering of volumetric clouds*. https://www.semanticscholar.org/paper/Real-time-rendering-of-volumetric-clouds-H%C3%A4ggstr%C3%B6m/89e9153a091889c584df034a953a0eff4de45ee9

23. Lague, S. (2019). *Coding adventure: clouds*. https://www.youtube.com/watch?v=4QOcCGI6xOU
24. Schneider, A., & Vos, N. (2015). *The Real-time Volumetric Cloudscapes of Horizon:Zero Dawn*. Guerrilla Games. https://www.guerrilla-games.com/read/the-real-time-volumetric-cloudscapes-of-horizon-zero-dawn
25. Stam, J. (1999). Stable fluids. https://doi.org/10.1145/311535.311548

Appendices

Appendix A

	size 0		size 1		size 2		
Test nr	CPU CA	GPU CA	CPU CA	GPU CA	CPU CA	GPU CA	Vol Clouds
1	0.5	0.6	5.0	5.2	37.3	36.7	18.0
2	0.5	0.7	8.5	9.9	40.4	62.0	14.0
3	0.5	0.7	6.3	5.2	42.8	50.6	11.0
4	0.8	0.6	7.4	5.8	41.6	42.5	15.0
5	0.5	0.6	5.8	5.0	37.3	34.7	10.0
6	0.5	0.7	6.1	5.0	44.1	40.0	13.0
7	0.5	0.6	10.1	8.8	40.3	35.9	14.0
8	0.5	0.6	8.7	6.8	45.5	37.0	18.0
9	0.5	0.6	5.6	5.5	38.2	41.7	13.0
10	0.6	0.6	6.2	5.7	48.3	41.9	14.0
avg	0.5	0.6	7.0	6.3	41.6	42.3	14.0

Metrics results for initial start duration, section 5.1.1. Times are in Milliseconds. Note, the GPU CA initialisation is done completely on the CPU, the exact same way as the CPU CA.

Appendix B

		size 0		size 1		size 2		
Test nr	gen nr	CPU CA	GPU CA	CPU CA	GPU CA	CPU CA	GPU CA	Vol Clouds
1	1	1.07	7.11	5.98	531.67	5.53		2.04
1	2	0.29	7.11	13.64	530.26	10.26		2.04
1	3	0.32	7.11	14.66	533.67	14.93		2.05
1	4	0.15	9.08	16.04	529.97	14.21		1.95
1	5	0.14	7.10	17.09	527.97	14.07		1.97
2	7	0.13	7.49	13.66	534.05	19.60		4.05
2	8	0.13	7.16	13.38	529.69	14.21		2.18
2	9	0.23	9.15	13.92	529.72	18.02		2.02
2	10	0.11	7.16	12.89	533.84	13.58		1.98
2	11	0.20	7.15	12.26	533.62	13.07		2.07
3	15	0.20	5.32	11.23	527.87	11.97		1.95

3	16	0.10	5.32	11.36	533.54	11.35	2.03
3	17	0.18	5.32	12.90	533.77	11.49	1.96
3	18	0.10	7.19	11.32	533.63	10.96	1.95
3	19	0.10	5.32	11.94	531.66	11.81	1.99
avg		0.2	6.9	12.8	531.7	13.0	2.1

Metrics results for calculation duration, section 5.1.2. Simulation includes cloud growth, extinction, and regeneration. Times are in Milliseconds. Size 2 for the GPU CA is not included, the application ran too slow to measure (<1 FPS).

Appendix C

		size 0		size 1		size 2		
Test nr	gen nr	CPU CA	GPU CA	CPU CA	GPU CA	CPU CA	GPU CA	Vol Clouds
1	1	0.97	6.02	5.88	546.48	35.38		2.09
1	2	0.73	5.64	31.86	544.75	187.74		2.08
1	3	0.27	7.23	28.81	543.03	194.83		2.46
1	4	0.26	5.51	27.51	546.72	182.30		1.76
1	5	0.27	6.02	29.05	544.70	188.45		1.85
2	7	0.50	5.37	29.48	546.65	174.92		2.16
2	8	0.33	5.48	29.68	544.57	213.69		1.94
2	9	0.32	7.26	29.85	546.46	188.13		3.00
2	10	0.28	5.48	31.12	541.04	188.59		3.93
2	11	0.38	5.62	26.20	546.35	185.75		2.61
3	15	0.27	5.64	28.91	544.75	184.42		2.16
3	16	0.25	5.48	28.77	543.03	179.29		4.39
3	17	0.32	7.23	26.53	546.72	195.71		3.51
3	18	0.26	5.48	25.66	544.70	189.21		3.21
3	19	0.33	5.62	30.23	545.65	219.07		3.01
avg		0.4	5.9	27.3	545.0	180.5		2.7

Metrics results for calculation duration, section 5.1.3. Simulation includes cloud growth, extinction, regeneration, and advection by wind. Times are in Milliseconds. Size 2 for the GPU CA is not included, the application ran too slow to measure (<1 FPS).

Appendix D

	size 0		size 1		size 2	
	CPU CA	GPU CA	CPU CA	GPU CA	CPU CA	GPU CA
bytes	750.0	750.0	75000.0	75000.0	491520.0	491520.0
kilobytes	0.75	0.75	75.0	75.0	491.5	491.5

Metrics results for memory usage for the CA applications, section 5.1.4. In bytes and kilobytes.

Appendix E

Vol Clouds

104.6 MB

Metrics results for memory usage for the volumetric cloud application, section 5.1.4. In MegaBytes.

Appendix F

```
bool currentAct = _Act[cellIdx], currentCld = _Cld[cellIdx], currentHum =_Hum[cellIdx];
//hum
_NextHum[cellIdx] = currentHum && !currentAct;
//cld
_NextCld[cellIdx] = currentCld || currentAct;
//act
_NextAct[cellIdx] = !currentAct && currentHum && GetActFromSurrounding(cellIdx);
```

Implementation of cloud growth rules, explained in Section 9.1

Appendix G

```
return (cellIdxI + 1 < _CAGridSettings.Columns && _Act[ThreeDToOneDIndex(cellIdxI + 1,
cellIdxJ, cellIdxK)])
|| (cellIdxI > 0 && _Act[ThreeDToOneDIndex(cellIdxI - 1, cellIdxJ, cellIdxK)])
|| (cellIdxJ + 1 < _CAGridSettings.Rows && _Act[ThreeDToOneDIndex(cellIdxI, cellIdxJ + 1,
cellIdxK)])
|| (cellIdxJ > 0 && _Act[ThreeDToOneDIndex(cellIdxI, cellIdxJ - 1, cellIdxK)])
|| (cellIdxK + 1 < _CAGridSettings.Depth && _Act[ThreeDToOneDIndex(cellIdxI, cellIdxJ,
cellIdxK + 1)])
|| (cellIdxK > 0 && _Act[ThreeDToOneDIndex(cellIdxI, cellIdxJ, cellIdxK - 1)])
|| (cellIdxI + 2 < _CAGridSettings.Columns && _Act[ThreeDToOneDIndex(cellIdxI + 2, cellIdxJ,
cellIdxK)])
|| (cellIdxI > 1 && _Act[ThreeDToOneDIndex(cellIdxI - 2, cellIdxJ, cellIdxK)])
|| (cellIdxJ + 2 < _CAGridSettings.Rows && _Act[ThreeDToOneDIndex(cellIdxI, cellIdxJ + 2,
cellIdxK)])
|| (cellIdxJ > 1 && _Act[ThreeDToOneDIndex(cellIdxI, cellIdxJ - 2, cellIdxK)])
|| (cellIdxK > 1 && _Act[ThreeDToOneDIndex(cellIdxI, cellIdxJ, cellIdxK - 2)]);
```

Implementation of *act* Boolean function, looking at the surrounding cells, explained in Section 9.1.

Appendix H

```
float cellHeight = GetCellHeightInWorld(cellIdxK);
int cellDisplacementByWind = WindHelper.GetWindSpeedCellDisplacementAtHeight();

if(cellIdxI - cellDisplacementByWind >= 0)
{
	int cellIdxDisplacementByWind = ThreeDToOneDIndex(cellIdxI -cellDisplacementByWind,
	cellIdxJ, cellIdxK);
	//hum
	_NextHum[cellIdx] = _Hum[cellIdxDisplacementByWind];
	//cld
	_NextCld[cellIdx] = _Cld[cellIdxDisplacementByWind];
	//act
	_NextAct[cellIdx] = _Act[cellIdxDisplacementByWind];
}
else if(cellIdxI == 0)
{
	_NextHum[cellIdx] = false;
	_NextCld[cellIdx] = false;
	_NextAct[cellIdx] = false;
}
```

Implementation of wind, explained in Section 9.1.

Appendix I

```
bool CheckBit(const int data, const uint bitPosition)
{
    const int n = data >> bitPosition;
    return n & 1;
}
int SetBit(int data, const uint bitPosition)
{
    data |= 1 << bitPosition;
    return data;
}
int ResetBit(int data, const uint bitPosition)
{
    data &= ~(1 << bitPosition);
    return data;
}
```

Implementation of bit checks for GPU CA, explained in Section 9.2.

Appendix J

```
void CloudPosition(const uint cellIdx)
{
    uint intIdx, bitIdx;
       GetBitPosition(cellIdx, intIdx, bitIdx);
       if (CheckBit(_Cld[intIdx], bitIdx))
       {
              _IntVariables[1]++;
              const float cellHalfHeight = _CellHeight / 2.f;
              uint cellIdxI, cellIdxJ, cellIdxK;
              OneDToThreeDIndex(cellIdx, cellIdxI, cellIdxJ, cellIdxK);
              const float3 cloudPos = float3( (float)cellIdxI + cellHalfHeight,
              (float)cellIdxJ + cellHalfHeight,  (float)cellIdxK + cellHalfHeight);
              _CloudPositions[_IntVariables[1]] = cloudPos;
       }
}
```

Implementation of cloud position saving for GPU CA, explained in Section 9.2.