NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BACHELOR THESIS

Two and Three-Dimensional Cellular Automata for the Generation of Objects in Computer Graphics

Panagiotis A. Mikedakis

Supervisors: **Theoharis Theoharis,** Professor NKUA
Vasileios Drakopoulos, Assistant Professor UTh

ATHENS

MARCH 2017

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Διδιάστατα και Τριδιάστατα Κυψελικά Αυτόματα για τη Γένεση Αντικειμένων στη Γραφική Υπολογιστών

Παναγιώτης Α. Μικεδάκης

Επιβλέποντες: **Θεοχάρης Θεοχάρης,** Καθηγητής Ε.Κ.Π.Α.
Βασίλειος Δρακόπουλος, Επίκουρος Καθηγητής Π.Θ.

ΑΘΗΝΑ

ΜΑΡΤΙΟΣ 2017

BACHELOR THESIS

Two and Three-Dimensional Cellular Automata for the Generation of Objects in Computer Graphics

Panagiotis A. Mikedakis
R.N.: 1115201000091

SUPERVISORS: **Theoharis Theoharis,** Professor NKUA
Vasileios Drakopoulos, Assistant Professor UTh

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Διδιάστατα και Τριδιάστατα Κυψελικά Αυτόματα για τη Γένεση Αντικειμένων στη Γραφική Υπολογιστών

Παναγιώτης Α. Μικεδάκης
Α.Μ.: 1115201000091

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Θεοχάρης Θεοχάρης,** Καθηγητής Ε.Κ.Π.Α.
Βασίλειος Δρακόπουλος, Επίκουρος Καθηγητής Π.Θ.

ABSTRACT

Cellular automata comprise a family of discrete models with locality constrains.The concept was originally discovered in the 1940s by Stanislaw Ulam and John von Neumann while they were contemporaries at Los Alamos National Laboratory. While studied by some throughout the 1950s and 1960s, it was not until the 1970s and Conway's Game of Life, a two-dimensional cellular automaton, that interest in the subject expanded beyond academia, coinciding with the rise in computer processing power and accessibility. There are several applications of CA in simulating natural microcosms, chemical systems or the spread of viruses and forest fires.

The simplest type of cellular automaton is a binary, nearest-neighbour, one-dimensional automaton. Such automata were called "elementary cellular automata" by S. Wolfram, who has extensively studied their amazing properties. In two dimensions, the best-known cellular automaton is Conway's game of life, discovered by J. H. Conway in 1970 and popularized in Martin Gardner's Scientific American columns.

In this thesis we study some novel applications of cellular automata in the field of computer graphics, especially in generating three-dimensional polygon meshes. Emphasis will be given on formations found in nature, but examples of artificial and abstract forms are also included. We discuss ways to effectively use the generative product of automata in a real-time graphics application context and compare it with other popular methods for mesh generation. We populate virtual worlds with foliage and props, organically placed by automata that simulate its growth and spread. A versatile image generator was created and dissected to reach certain conclusions for the complicated behaviour that cellular automata exhibit. Lastly we introduce ways to compress given structures to simple CA rules using a genetic algorithm. For all the above, a software application was developed and implemented taking advantage of modern computational techniques on the GPU.

SUBJECT AREA: Cellular Automata

KEYWORDS: cellular automaton, computer graphics, object modelling, procedural generation, procedural texturing, genetic algorithm

ΠΕΡΙΛΗΨΗ

Τα κυψελικά ή κυτταρικά αυτόματα αποτελούν μια οικογένεια διακριτών προτύπων με τοπικούς περιορισμούς. Η έννοια ανακαλύφθηκε την δεκαετία του 1940 από τον Stanislaw Ulam και John von Neumann, συνεργάτες στο Los Alamos National Laboratory. Ενώ μελετήθηκε από αρκετούς καθ' όλη τη δεκαετία του 1950 και του 1960, δεν ήταν μέχρι τη δεκαετία του 1970 και το παίγνιο της Ζωής του Conway, ένα διδιάστατο κυψελικό αυτόματο, όπου το ενδιαφέρον για το θέμα επεκτάθηκε πέρα από την ακαδημαϊκή κοινότητα, συμπίπτοντας με την άνοδο επεξεργαστικής ισχύος και προσβασιμότητας των υπολογιστών. Υπάρχουν πολλές εφαρμογές των ΚΑ στην προσομοίωση φυσικών μικροκόσμων, χημικών συστημάτων ή την εξάπλωση ιών και πυρκαγιών.

Ο απλούστερος τύπος κυψελικού αυτομάτου είναι ένα δυαδικό, πλησιέστερου γείτονα, μονοδιάστατο αυτόματο. Τέτοια αυτόματα ονομάστηκαν “στοιχειώδη” ΚΑ από τον S. Wolfram, ο οποίος έχει μελετήσει εκτενώς τις εκπληκτικές ιδιότητές τους. Σε δύο διαστάσεις, το πιο γνωστό κυψελικό αυτόματο είναι το παίγνιο της ζωής, όπου ανακαλύφθηκε από τον J. H. Conway το 1970 και διαδόθηκε από τις στήλες του Martin Gardner στο Scientific American.

Στην εργασία αυτή μελετούμε κάποιες νέες εφαρμογές κυψελικών αυτομάτων στον τομέα των γραφικών, ειδικώς στην παραγωγή τριδιάστατων αντικειμένων. Έμφαση δίνεται σε σχηματισμούς που παρατηρούνται στη φύση, αλλά περιλαμβάνονται επίσης παραδείγματα τεχνητών και αφηρημένων δομών. Συζητάμε τρόπους για την αποτελεσματική χρήση του προϊόντος των ΚΑ στα πλαίσια εφαρμογών γραφικών πραγματικού χρόνου και τους συγκρίνουμε με άλλες δημοφιλείς μεθόδους για παραγωγή πολυγωνικών προτύπων. Οικούμε δυνάμει κόσμους με χλωρίδα οργανικώς τοποθετημένη από αυτόματα που προσομοιώνουν την ανάπτυξη και εξάπλωσή της. Δημιουργούμε και αναλύουμε μία ευέλικτη γεννήτρια εικόνων, ώστε να φτάσουμε σε ορισμένα συμπεράσματα για την περιπλεγμένη συμπεριφορά που εμφανίζουν τα ΚΑ. Τέλος, παρουσιάζουμε τρόπους για την συμπίεση δομών σε απλούς κανόνες ΚΑ χρησιμοποιώντας γενετικούς αλγορίθμους. Για όλα τα παραπάνω, αναπτύχθηκε και υλοποιήθηκε μια γραφική εφαρμογή, αξιοποιώντας σύγχρονες υπολογιστικές τεχνικές στην κάρτα γραφικών.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Κυψελικά ή Κυτταρικά Αυτόματα

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: κυψελικό ή κυτταρικό αυτόματον, γραφική Η/Υ, προτύπωση αντικειμένων, διαδικαστική γένεση, διαδικαστική ύφανση, γενετικοί αλγόριθμοι

CONTENTS

LIST OF FIGURES

1. PRELIMINARIES

1.1 Cellular automaton

A *cellular automaton*, or CA for short, is comprised of a lattice with a regular grid of cells. Each cell is assigned one of a finite number of states. In every iteration the cells change state synchronously as a function of their current state and that of their neighbours. Commonly, the cell state is represented as an integer or boolean value. The grid can be of any finite dimension, but we mostly delve into three-dimensional automata, as they have immediate application in computer graphics. The neighbourhood of a cell in a one-dimensional grid is illustrated in Figure 1. Cells in the immediate neighbourhood are shown in green, the ones in yellow and red constitute the extended neighbourhood in radius 2 and 3 from the cell.

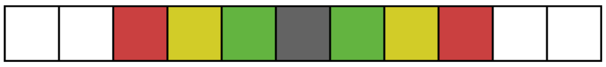


Figure 1: Neighbourhood of the grey cell in a one dimensional grid.

In each iteration of the automaton, a rule is synchronously evaluated for each cell, changing the grid states accordingly. The behaviour of a simple rule is shown in Figure 2. A finite one-dimensional grid is initialized with a single grey cell (state 1) in the middle. For each iteration, a cell becomes grey if one or two grey cells exist in its immediate neighbourhood. Otherwise, it turns or remains white (state 0). Note that the cell being evaluated is included in the neighbourhood.

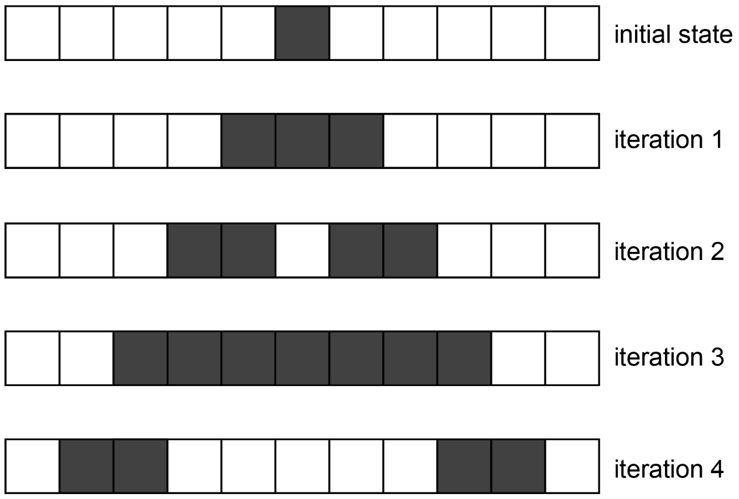


Figure 2: Four iterations of an elementary cellular automaton rule on a one-dimensional grid.

Expanding this concept to two or three dimensions presents more choices for the selection of the neighbourhood. The most common are the *von Neumann* and *Moore* neighbourhoods and their extensions, illustrated in Figures 3 and 4.

For the purpose of this thesis we consider the immediate Moore neighbourhood, consisting of 26 cells (including diagonals) and its extensions. The number of cells in an extended Moore neighbourhood of radius r in d dimensions is $(2 \times radius)^d - 1$. In fact, *radius* represents the *Chebyshev* or *chessboard distance*, meaning all neighbouring cells in positions np, of cell in position cp, where $max(|np_i - cp_i|) <= radius$, are included in the rule evaluation.

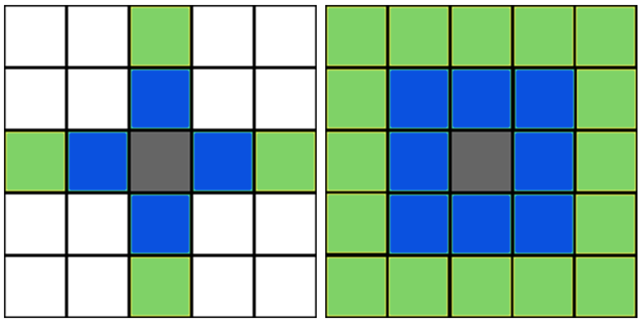


Figure 3: Neighbourhoods in a two-dimensional grid. Left: von Neumann, right: Moore. The immediate neighbourhood is shown in blue and the extended in green.

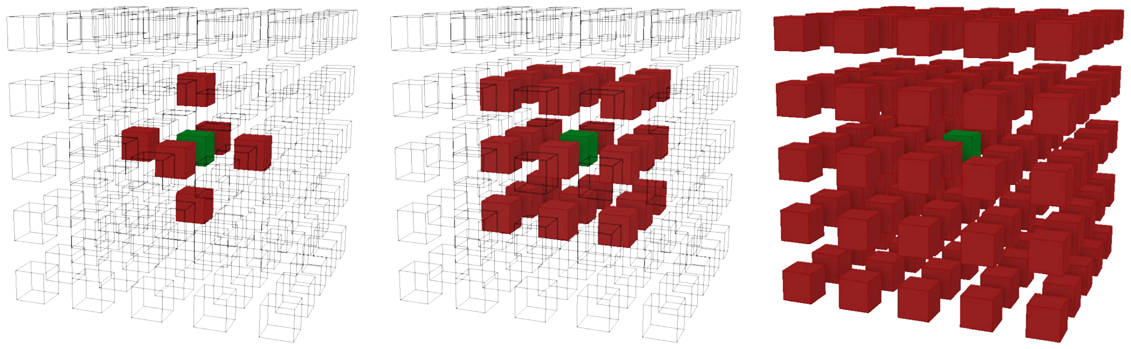


Figure 4: Three-dimensional neighbourhoods of the green cell shown in red. From left to right: von Neumann, Moore, Moore extended.

CA rules may take into account both the number of neighbours in a certain state and the specific states of each neighbour. The types of constrains we set for expressing rules in this thesis can be grouped in the following categories:

1.1.1 Totalistic cellular automata

Totalistic cellular automata are one of the simplest and most extensively studied automata [22]. As the name implies, the cells of these automata consider a summation (or average) of the neighbouring cell values to decide on their future state. Commonly they have two possible states, alive or dead (0,1) and the rules are stated as such:

1,2,3/2,5

Numbers before the slash denote the count of alive neighbours for which an alive cell may survive, while the ones after, the count for triggering a dead cell to be born. In the above case, an alive cell survives if it has 1, 2 or 3 alive neighbours, while a dead one is born if it has 2 or 5 alive neighbours. In any other case the cell dies, or remains dead. The behaviour of the totalistic rule $1/1$ is shown in Figure 5.

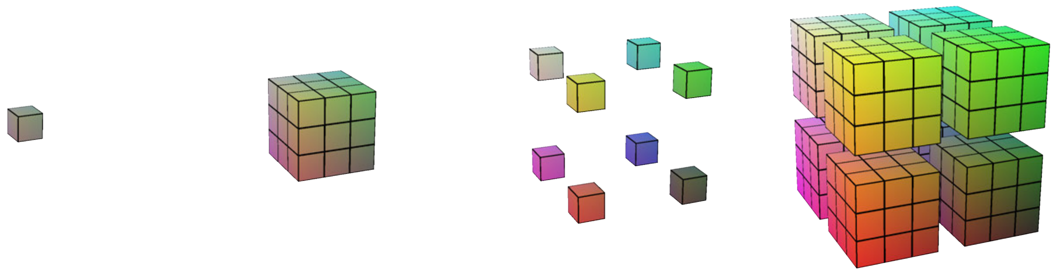


Figure 5: The first three iterations of the totalistic automaton rule $1/1$, starting from a single alive cell. Dead cells are not rendered.

A cellular automaton (CA) is *Life-like* (in the sense of being similar to Conway's Game of Life [4]) if it meets the following criteria:

- The array of cells of the automaton has two dimensions.
- Each cell of the automaton has two states (conventionally referred to as "alive" and "dead", or alternatively "on" and "off")
- The neighbourhood of each cell is the Moore neighbourhood; it consists of the eight adjacent cells to the one under consideration and (possibly) the cell itself.
- In each time step of the automaton, the new state of a cell can be expressed as a function of the number of adjacent cells that are in the alive state and of the cell's own state; that is, the rule is outer totalistic (sometimes called *semi-totalistic*).

```
Rule: Game of Life [4]
Output: newCellState
cellState ← current cell state;
aliveCellCount ← number of ALIVE cells in the immediate Moore neighbourhood;
if (cellState = ALIVE and aliveCellCount ∈ [2, 3])
   or (cellState = DEAD and aliveCellCount = 3) then
    return ALIVE;
else
    return DEAD;
```

Life-like cellular automata exhibit bounded growth and have the ability to form a translating oscillator, also referred to as "glider". The glider is a pattern that travels across the board in Conway's Game of Life. A life-like rule once discovered only behaves as such on limited initial structures. The following three-dimensional rules, discovered by Bayes [3] exhibit this behaviour:

- 5,7/6
- 5,6,7/6
- 2,7/5
- 3,5/5
- 4,7/5

The full period of life-like rule 4,7/5 is illustrated in Figure 6.

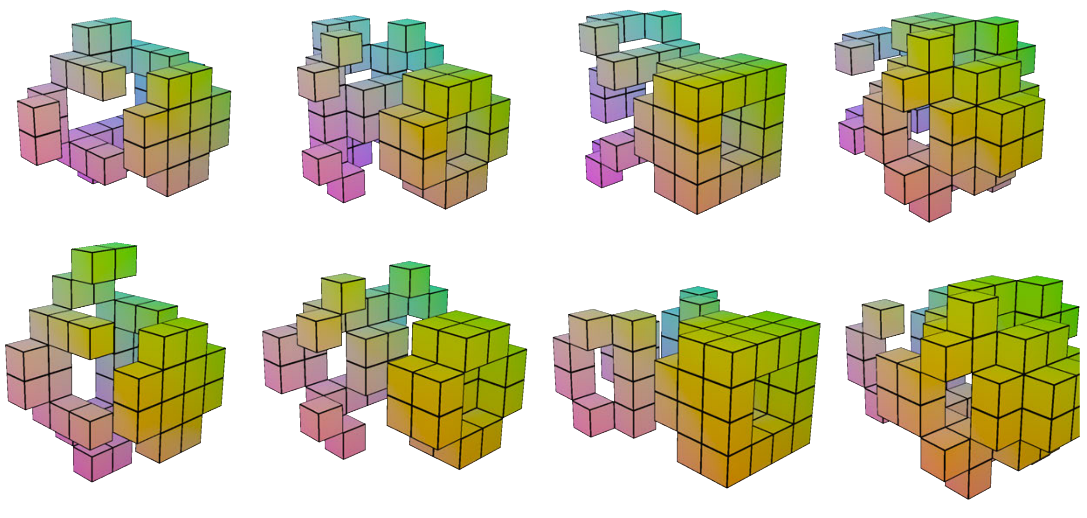


Figure 6: The full period of rule B47/S5, indexed left to right, top to bottom.

1.1.2 Life-based cellular automata

Life-based cellular automata are *multi-state CA* and a generalisation of the “Brain rules” [20]. Let N be the count of possible states, $N - 1$ the alive state index and 0 the dead state. All other states are not considered in a cell’s survival computation. A number of cells are assigned the maximum state (alive) upon initialization, if a cell does not survive an iteration their state gets reduced by 1.

These automata exhibit seemingly alive behaviour, thus are better observed in motion. Brian Silverman’s “Brian’s Brain” ported to three dimensions is depicted in Figure 7. When in motion, the structures glide and interact in complicated ways.

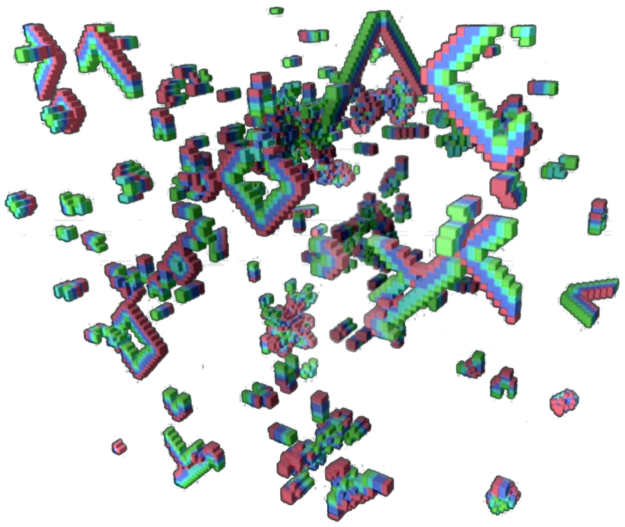


Figure 7: 50th iteration of Brian Silverman’s “Brian’s Brain” [20] 2D rule ported to three dimensions. Each colour represents a cell state.

1.1.3 Cellular automata with memory

In *cellular automata with memory* [2], the current state is evaluated as a function of m previous states, with some weight distribution.

$$State_i = \frac{\sum_{n=i-m}^{i} State_n \times w_n}{\sum_{n=i-m}^{i} w_n}$$

1.1.4 Cellular automata with directionality

Cellular automata with directionality are totalistic CA, but the evolution function has access to directionality predicates when neighbour states are evaluated, e.g counting neighbouring cells on the X-axis with state 1. As seen in Figure 8, the neighbourhood placement is altered according to the automaton rule.

1.1.5 Stochastic cellular automata

In *stochastic cellular automata* [6], a cell may change state according to some probability distribution.

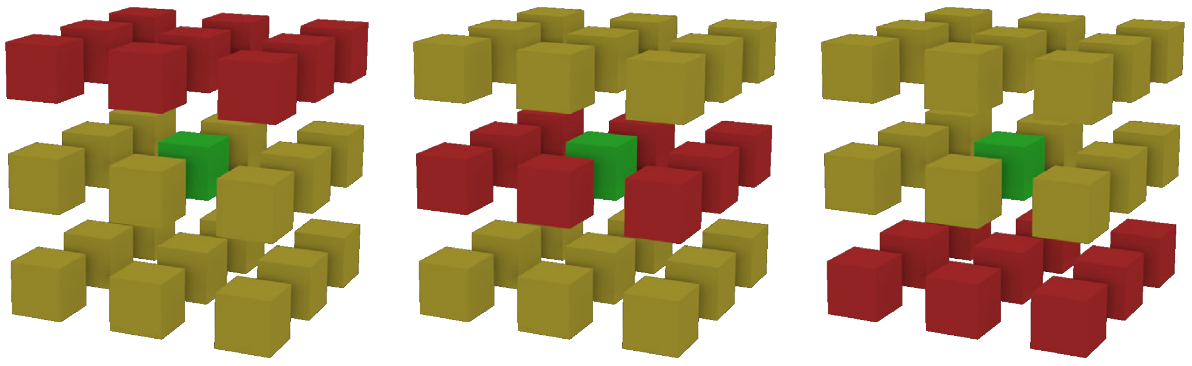


Figure 8: Automata with directionality. Top, around and bottom neighbourhoods of green cell in radius 1, shown in red.

1.2 Conventions and methodology

Commonly, all cells are given a neutral initial state (dead, 0, air), while a small number of them another (solid, 1, alive), to better discern the evolutionary process of the automaton. Some automata rules are better observed when the initial states are random. For some automata, different initial states result in different behaviour, while others always converge to a certain form. In our case, grid states are carefully constructed for synthetic and geometric shapes, while random initial volumes with controlled initial state distribution will expand and contract to form organic shapes found in nature.

We handle cells on the edge by wrapping the neighbour search around the border of the simulated configuration. This ensures the end result is tile-able on all axes, and interesting behaviour can happen at the edges. The topology of the grid and connections between the cells is illustrated in Figure 9.

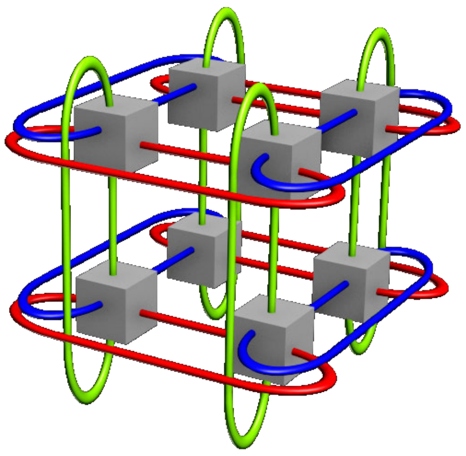


Figure 9: A cellular space of $2 \times 2 \times 2$. Cells form an interconnected toroidal network.

During the development of the ideas presented in this thesis, we found that relying on a single rule to produce the desired structure doesn't allow much creative freedom. Instead, we provide means to successively apply multiple rules and perform boolean operations between volumes of different states. A structure is described by its initial state, a list of automata rules and the count of steps to apply them, and optionally a list of boolean operations. These operations are also expressed in simple CA rules but only need to run once to produce the desired result.

Our investigation focused on simple, lightweight queries run on a cubic grid. A single rule is executed per time frame and affects the entirety of the cell grid. To meet our real-time generation goals, all rule configurations obey the following requirements: A single rule iteration should take place in a single frame of a real-time graphics application when run on modern hardware. If multiple rules and iterations are required to produce the desired result, we set a maximum duration of 15 seconds as if it runs during the pre-processing stage of a level.

Cells change state synchronously, so there is a need for two buffers, one for the current simulation state and one for the next. When a CA rule is evaluated the neighbour lookup is done on the current buffer while the new cell state is saved in the next. Finally, the buffers are swapped. Rule algorithms presented in the next chapters are run once per cell to take advantage of the GPU architecture, but are not thoroughly optimized in exchange for readability. Calculations independent of the current simulation state should be sent to the GPU by the host application and expensive neighbourhood queries should be hidden behind conditionals if they can be avoided.

2. OBJECT GENERATION WITH CELLULAR AUTOMATA

2.1 Cave structures

Cellular automata are extensively used for the generation of cave levels [9]. Implementations in recent games mostly focus on creating two-dimensional level layouts, efficient in the context of that game. For this thesis, we will port the two-dimensional cave rule to three dimensions and investigate ways to improve and parameterise it.

For this, we make use of a two-state automaton of solid(1) and air(0). According to this automaton's rule, every cell changes state based on the ratio of air and solid cells around it, resulting in a clustering of same-state cells in space. The *connectedness* parameter determines how substantial the ratio in favour of different states around the cell needs to be, in order for it to change state. The default value is 0.5. The percentage of solid cells in the primitive structure is referred to as *density*.

Rule: Cave structure

```
Parameters: connectedness, radius
Output: cell
cellCount ← number of same state cells in radius;
ratio ← cellCount / $(2 \cdot radius + 1)^3$;
if ratio < connectedness then
    cell ← majority state;
```

Structures displayed in the next pages are not hollow on the inside. They simply need to be inverted to act as intended. In case we wanted to extract a surface from this volume, the final result would be the same. In Figure 10 and subsequent figures, we show the original product of the cave rule for visibility.

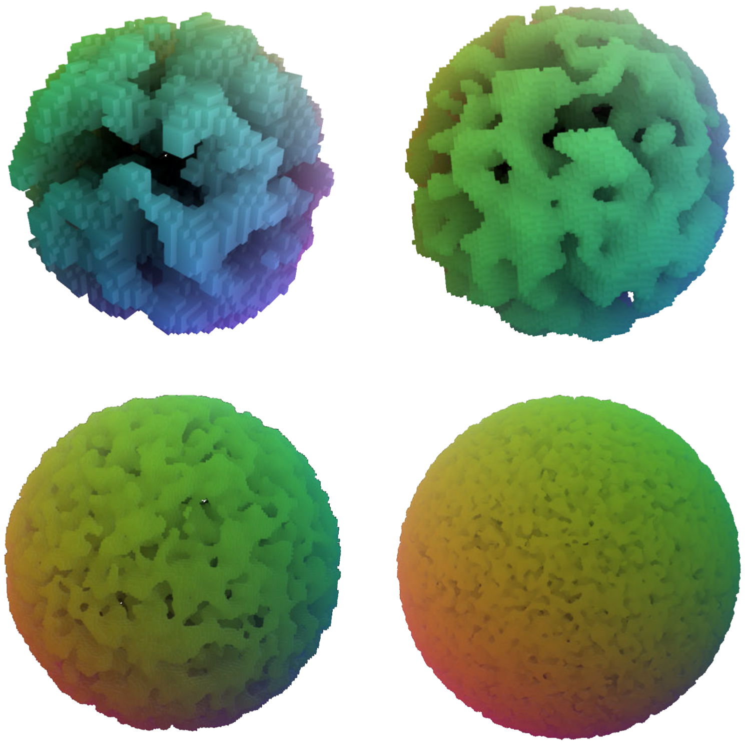


Figure 10: The cave structure rule applied for 7 iterations to a sphere with 0.5 density of varying resolution. Search radius set to 1 unit. From left to right, top to bottom: 32, 64, 128, 256 grid resolution. The connectedness parameter is set to 0.5.

We observe that increasing the simulation resolution results in the repetition of the cave features, which is the desired behaviour in case we wished to expand our world while retaining the same per-voxel resolution for the cave structures. There are of course ways to scale the structures or increase the cave fidelity. We will look into some of those.

2.1.1 Iteration count

The cave structure rule needs a small number of iterations to produce the desired result, in most cases 4 are enough. More iterations result in a smoother mass with less cave connections. We observed that as the iteration count increases, successive iterations have lesser effect on the volume. This behaviour is observed in Figure 11.

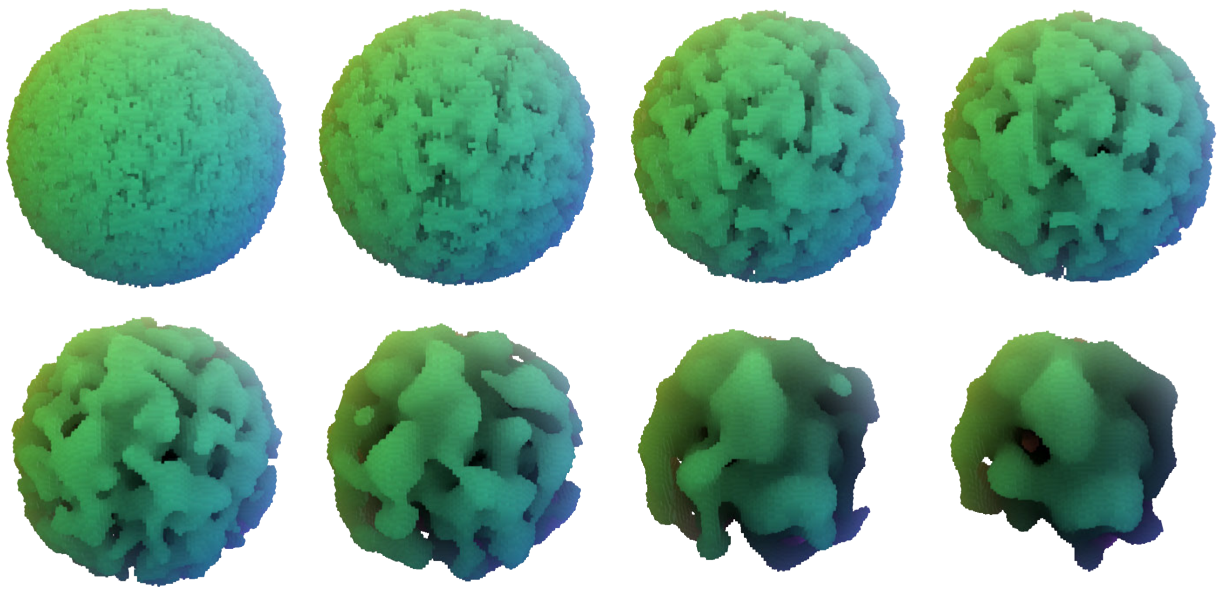


Figure 11: Selective iterations of the cave rule as applied to a 64^3 resolution sphere with density 0.5, radius 1, connectedness of 0.5. From left to right, top to bottom: 1,2,3,4,5,13,20,37 iterations.

2.1.2 Connectedness parameter

The smaller the *connectedness* value is, the more compact the volume generated, also, the cave network is less prone to collapse under successive iterations. Values under 0.3 provide mostly unchanging structures while ones greater then 0.6 result in expanding uniform structures. The effect of *connectedness* is shown in Figure 12

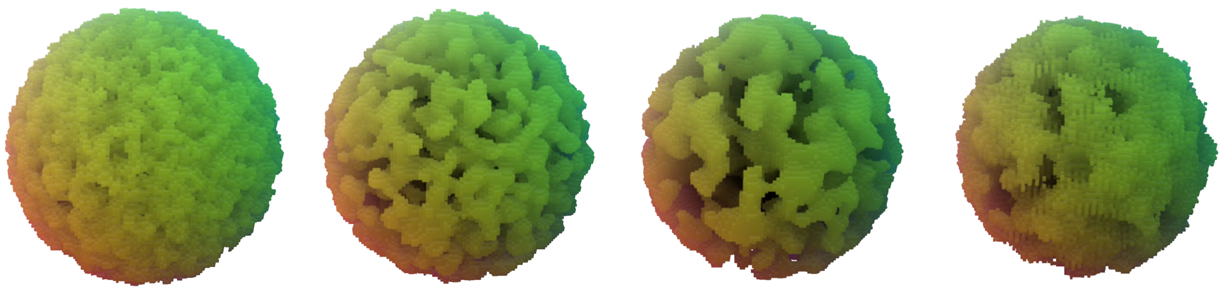


Figure 12: Modifying the connectedness parameter of the cave rule as applied to a 64^3 resolution sphere with density 0.5, radius 1 for 10 iterations. From left to right: 0.3, 0.4, 0.5, 0.6 connectedness.

2.1.3 Initial density

The initial density is the *solid* to $(air + solid)$ cells ratio inside the primitive structure upon initialisation. This ratio greatly affects the end result, evident in Figure 13. If either state significantly outnumbers the other, we will quickly see it dominating the simulation space after some iterations. Therefore, the density values for our primitive must fall in the [0.45, 0.55] range.

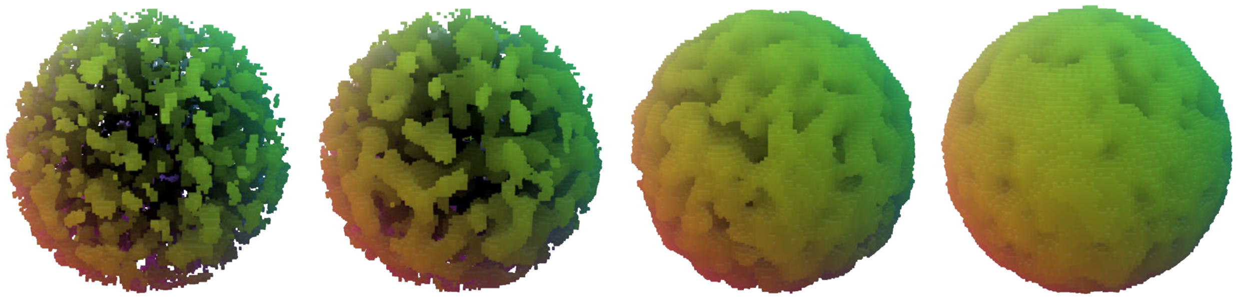


Figure 13: The cave rule as applied to a 64^3 resolution sphere with varying density, 1 search radius, for 2-7 iterations. From left to right: 0.45, 0.48, 0.52, 0.55 initial density.

2.1.4 Neighbourhood radius

Expanding the radius scales the features accordingly but demands a more expensive computation. We saw that 4 iterations using a neighbourhood radius of 1 produce similar results with these of two iterations by using a radius of 2. However, expanding the radius allows the evolution of structures beyond that state they would otherwise become stable. The results of radius expansion can be seen in Figure 14.

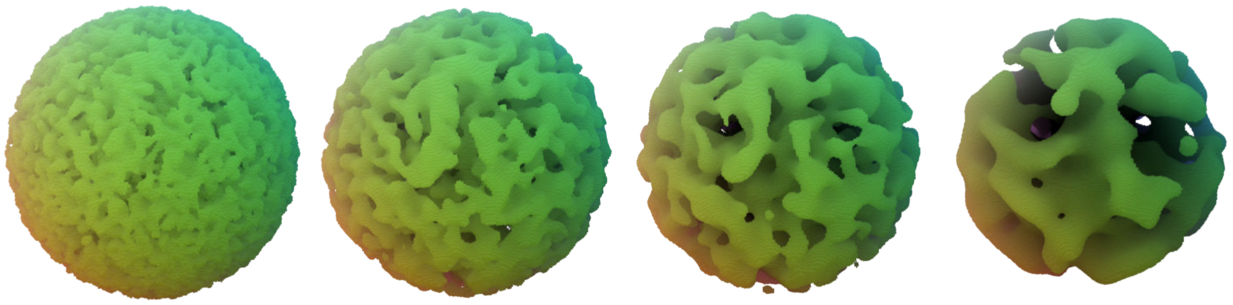


Figure 14: The cave rule as applied to a 128^3 resolution sphere with 0.5 density, with varying neighbourhood radius, for 4 iterations. From left to right: 1,2,3,5 search radius.

2.1.5 Augmenting the rule with life-based properties

Starting from a sphere with cell states 0 or *N*, equally distributed, if not enough cells of state *N* exist in the neighbourhood subtract 1 from the current state. In this case, states greater that 0 are considered solid. We observed that more states smooth and expand the volume while less cause it to shrink and become coarse. The cave rule with life based properties is shown in Figure 15.

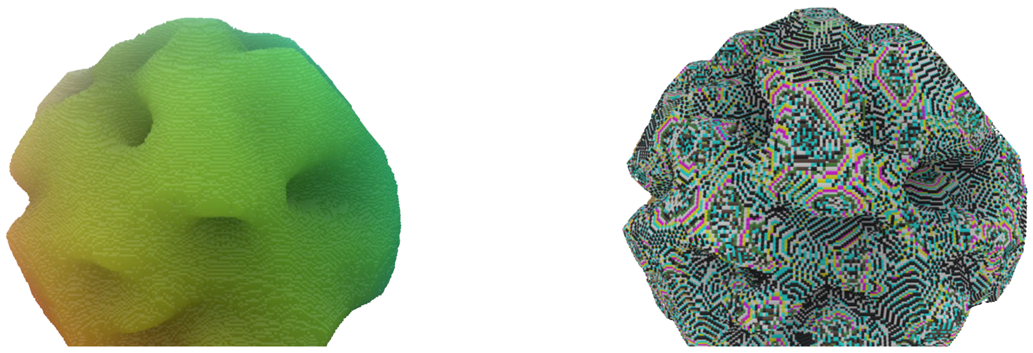


Figure 15: The modified cave rule as applied to a 64^3 sphere for 7 iterations. On the right we see a visualization of the different states. All states with index > 0 are rendered as solid.

2.1.6 Generating the cave structure

Our goal was to create a small and usable level, with well-structured cave connections a player could easily pass through and explore. We used a cube of 256^3 cells and 0.49 initial density, and applied the cave rule for 40 iterations, using a *radius* of 3 and *connectedness* of 0.5. The resulting volume is shown in Figure 16. The surface was extracted using methods that will be discussed in Subsection 2.4.2.

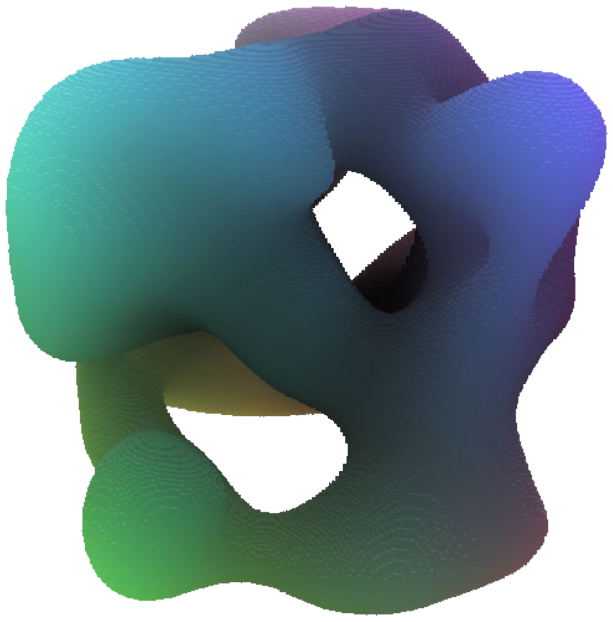


Figure 16: Generated cave structure.

Texturing poses a challenge for CA and all procedural generated meshes for that matter. Commonly, texture is mapped on a model through UV coordinates stored in each vertex of the polygon mesh. This is generally a manual process, aided by unwrapping algorithms and various tools.

For visualizing the cave structure's interior, shown in Figure 17, we used a different technique: A diffuse texture was projected using triplanar mapping and modulated by slope and normal orientation. In *triplanar mapping*, the textures are sampled using the vertices' world space coordinates and then blended based on the vertex normals. Multiple textures can be combined based on the projection axis. To provide further variation, the relative slope of the polygon faces and height information was used to modulate the texture colour.

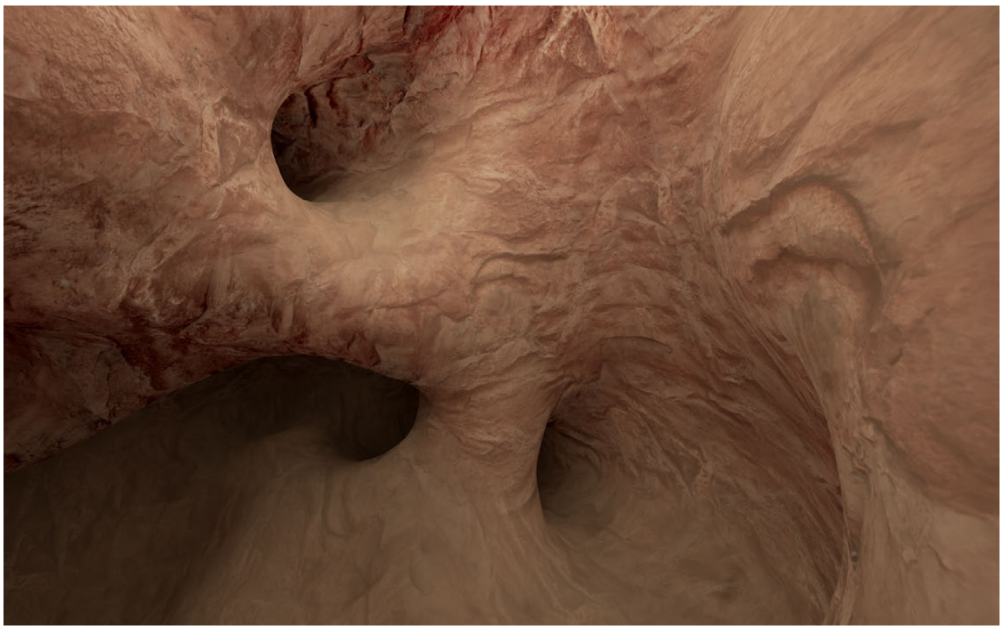


Figure 17: Inside the cave structure after surface extraction and texture mapping.

2.2 Terrain

We will look into simple CA rules that elevate and shape a thin voxel volume, eventually forming a natural looking terrain. To achieve a believable result we combined multiple rules and boolean operations.

2.2.1 Elevation rule

The elevation rule, as the name implies, displaces the top voxels on the y axis while expanding the ones under on the *x* and *z*. It aims to imitate the terrain elevation tool found in game engines. We will apply the rule on a thin volume shaped by the cave rule, as shown in Figure 18.

```
Rule: Elevation
Parameters: expandValue, radius
Output: cell
cell ← current cell state;
if cell = Air then
    bottomCell ← cell state directly under the cell;
    aroundCount ← solid cell count in radius around the cell;
    if bottomCell = Solid or aroundCount = expandValue then
        cell ← Solid;
```

2.2.2 Selective height displacement rule

The selective height displacement rule generates wide and sharp peaks. Unlike the elevation rule, it does not require specific initial conditions to generate interesting results, as it creates height variation from a flat volume. It may not produce realistic results by itself but is the backbone of our mountain range generation method. As with the cave rule we can scale the mountainous features across the landscape by extending the neighbourhood radius. The rule works by creating solid cells only when enough bottom cells exist to support them. We can parameterise the overall displacement by changing the required

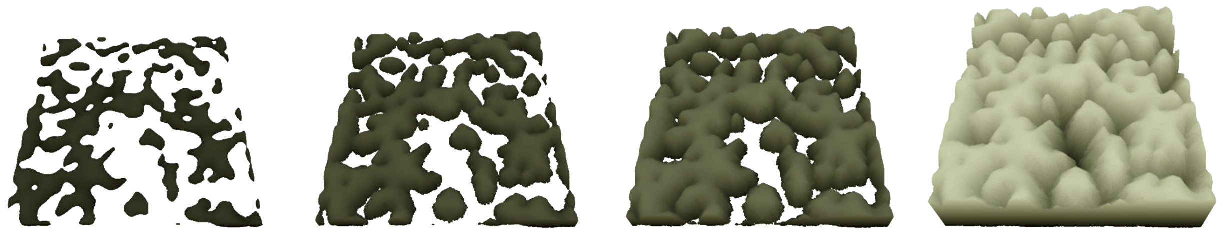


Figure 18: From left to right, top to bottom: 0, 10, 20, 55 iterations of the Elevation Rule on a $256 \times 256 \times 10$ **voxel mass.** *Radius* $= 1$**,** *expandValue* $= 3$

supporting cells count, relative to the maximum bottom neighbour count. The number of neighbouring cells in radius r where $height_{neighbour} < height_{cell}$ is $r \cdot (2r + 1)^2$.

```
Rule: Selective height displacement
Parameters: threshold, radius
Output: cell
cell ← current cell state;
bottomCell ← cell directly under the cell;
if $cell = Air$ and $bottomCell = Solid$ then
    bottomCount ← solid cell count in radius under the cell;
    if $bottomCount > radius \cdot (2 \cdot radius + 1)^2 \cdot threshold$ then
        cell ← Solid;
```

The initial volume density correlates with the number of displaced voxels and therefore mountain peaks. When applied on a thin voxel mass with 0.5 density, the rule becomes stable after about 50 iterations. When expanding the radius the threshold must be lowered accordingly to prevent exaggerated displacement. This behaviour is illustrated in Figure 19.

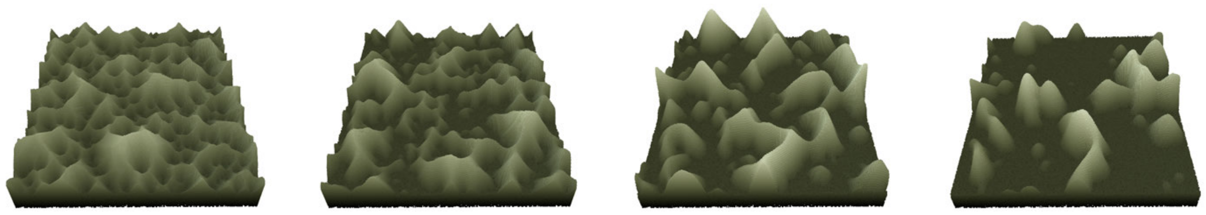


Figure 19: The selective height displacement rule applied on a $256 \times 256 \times 10$ **voxel mass with 0.5 density for 50 iterations. From left to right: 1, 2, 3, 4 neighbourhood radius, 0.77, 0.65, 0.6, 0.575 threshold respectively.**

Another interesting feature of the selective height displacement rule is the ability to combine small and large features produced by the above configurations. We found that by applying smaller features first, the resulting grid state prevents larger ones from seeding. Therefore, we applied sequentially starting from the highest neighbourhood radius going downwards. The behaviour of this combination resembles an addition of heights rather than a maximum. When combining a small and a large feature configuration, high frequency noise from the first appears intact where flat areas and valleys existed in the second configuration. Where there are peaks and uneven terrain, the noise builds upon them, extending the elevation but getting lost in the process. This is evident on the first and last image of Figure 20.

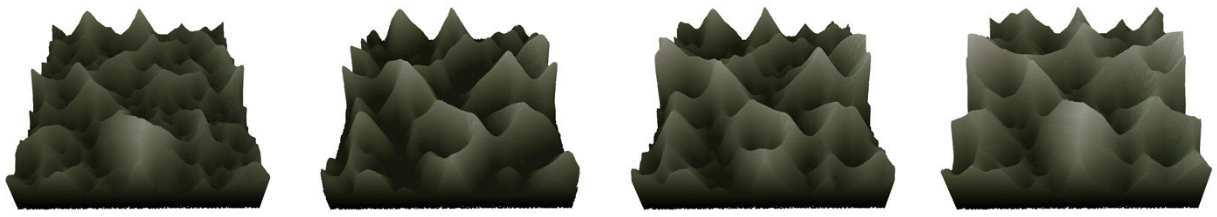


Figure 20: Different configurations of the selective height displacement rule, additively applied on a $256 \times 256 \times 10$ voxel mass with 0.5 density for 50 iterations each. From left to right: 1+3, 3+4, 2+3+4, 1+2+3+4.

2.2.3 Selective displacement rule variation

This variation generates wide, high altitude peaks and is more suited for alien terrain. It shares all the features of the original rule discussed above, as seen in Figures 21 and 22.

```
**Rule:** Selective height displacement variation
**Parameters:** *threshold*, *radius*
**Output:** *cell*
$cell \leftarrow$ current cell state;
if $cell = Air$ then
    $topCount \leftarrow$ solid cell count in $radius$ over the cell;
    $bottomCount \leftarrow$ solid cell count in $radius$ under the cell;
    if $bottomCount > topCount + 2 \cdot radius \cdot (2 \cdot radius + 1)^2 \cdot threshold$ then
        $cell \leftarrow Solid$;
```

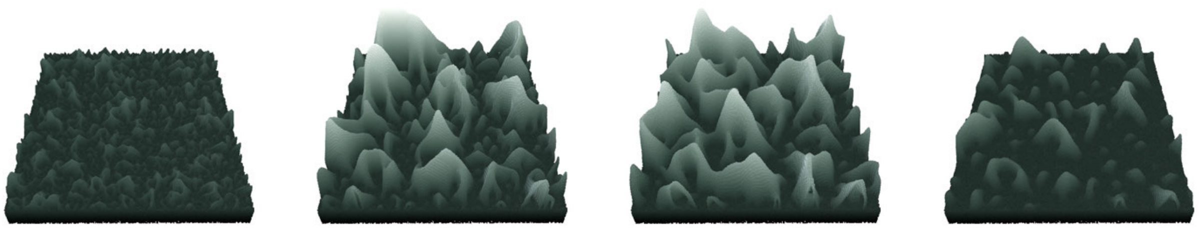


Figure 21: Variation of the selective height displacement rule as applied on a $256 \times 256 \times 10$ voxel mass with 0.5 density for 50 iterations. From left to right: 1, 2, 3, 4 neighbourhood radius, 0.278, 0.278, 0.265, 0.265 threshold respectively.

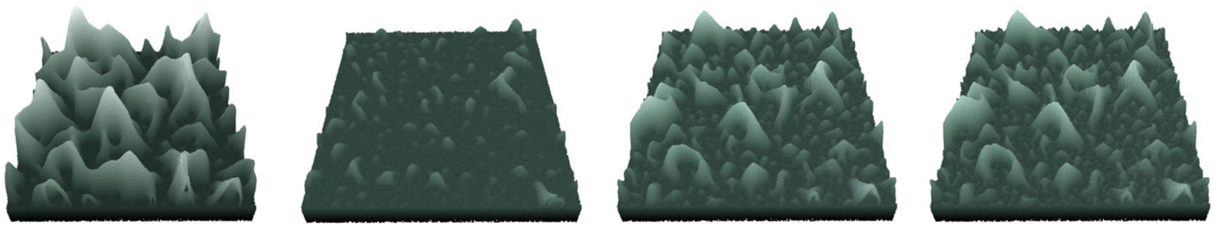


Figure 22: Variation of the selective height displacement rule as applied on a $256 \times 256 \times 10$ voxel mass with 0.5 density for 50 iterations each. From left to right: radius 3 0.278 threshold, radius 3 0.265 threshold, 4+2 radius, 4+2+1 radius.

2.2.4 Sediment carry rule

This rule aims to simulate the effect of hydraulic erosion and sediment deposition on rock formations. It is of course a very simplified model and the material is created rather than displaced, but still manages to generate believable results. It is parameterised by the slope threshold, an integer ranging from -3 to 3 that constrains the creation of solid cells based on the slope of adjacent voxel groups. Expanding the neighbourhood radius alleviates the harshness on the surface of the deposed material, but in most of the cases this is not a desired effect. The behaviour of the rule is shown in detail in Figures 23 and 24.

```
Rule: Sediment carry
Parameters: slopeThreshold, radius
Output: cell
cell ← current cell state;
bottomCell ← cell state directly under the cell;
if cell = Air and bottomCell = Solid then
    bottomCount ← solid cell count in radius under the cell;
    aroundCount ← solid cell count in radius around the cell;
    topCount ← solid cell count in radius over the cell;
    if bottomCount − aroundCount < topCount − slopeThreshold then
        cell ← Solid;
```

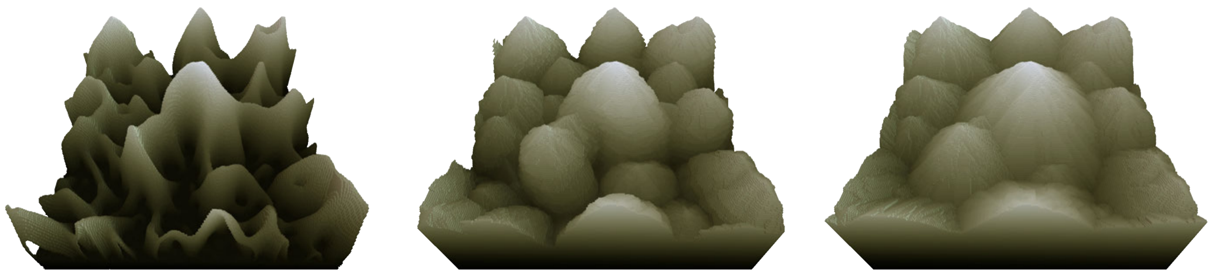


Figure 23: The sediment carry rule as applied on a 256^3 volume generated by the mountain variation rule. From left to right: initial, 35, 70 iterations. Slope threshold = 1, radius = 1.

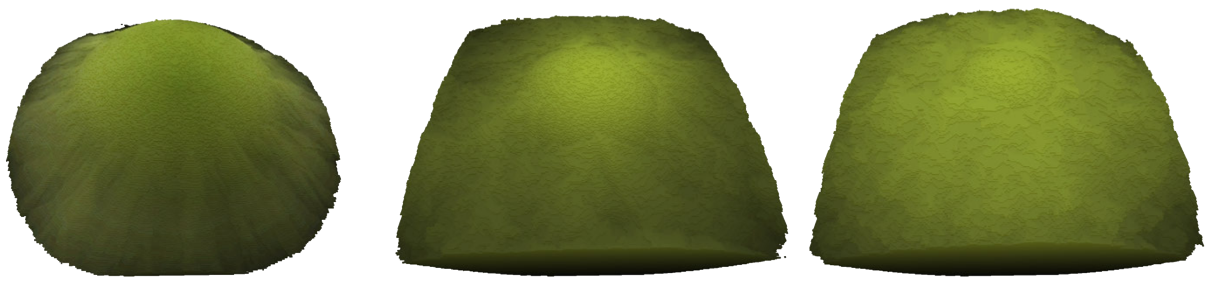


Figure 24: The sediment carry rule as applied on a 128 voxel diameter hemisphere with 0.5 density, neighbourhood radius of 1. From left to right: 1, 0, -1 slope threshold.

Using a different *slopeThreshold* in successive iterations allows us to control the slope, peaks and base of our mountains. When a desired sequence of values is found, the process can be automated, as different initial conditions provide similar results.

2.2.5 Space colonization rule

This rule was inspired by growth patterns found in nature, and proved extremely versatile. In order for a cell to grow, some solid neighbours are required to grow from, and enough empty cells to grow to. Furthermore, by considering only the bottom neighbourhood when evaluating this condition, the growth is guided towards the top of simulation. Applying this rule on a thin volume with an extended neighbourhood radius results in the creation of canyons consisting of stacked layers of solid cells, shown in Figure 25. The *threshold* and *limit* values control the amount and size variation of these layers.

```
Rule: Space colonization
Parameters: threshold, limit, radius
Output: cell
cell ← current cell state;
if cell = Air then
    solidBottomCount ← solid cell count in radius under the cell;
    solidRatio ← solidBottomCount / $(radius \cdot (2 \cdot radius + 1)^2)$;
    if solidRatio ∈ [threshold, limit] then
        cell ← Solid;
```

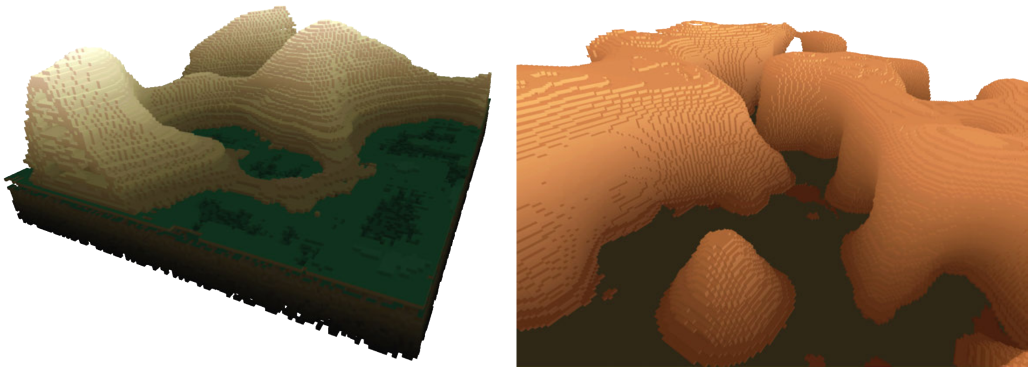


Figure 25: Voxel canyons generated with the Space colonization rule. Parameters from left to right: $(0.38, 0.46, 2)128^3$ **resolution grid,** $(0.4, 0.48, 6)256^3$ **resolution grid**

If we were to consider the entire neighbourhood for the upper threshold calculation the growth is more organic similar to underwater features, as seen in Figure 26.

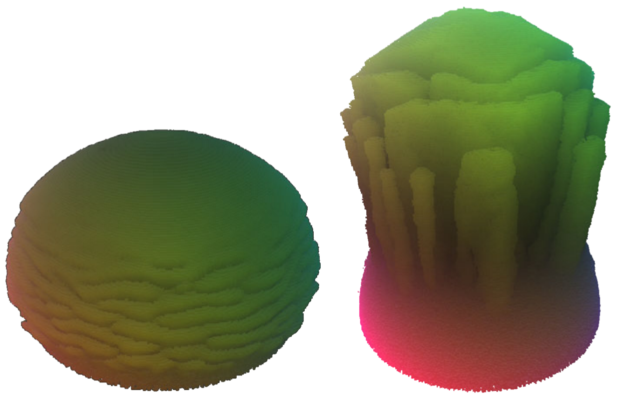


Figure 26: Modified space colonization rule as applied on a 128 **voxel diameter hemisphere of 0.5 density. Parameters from left to right:** $(0.2, 0.2, 3)$**,** $(0.2, 0.13, 3)$

2.2.6 Forming a mountain range

We discovered several viable rule combinations that produce realistic mountain ranges. The result shown in Figure 28 was achieved by sequentially applying the following rule configurations:

- grid initialization with a thin voxel volume of 0.5 density.
- 40 iterations of the selective height displacement rule, $\mathit{threshold} = 0.6$, $\mathit{radius} = 3$.
- 30 iterations of the selective height displacement rule, $\mathit{threshold} = 0.65$, $\mathit{radius} = 2$.
- 4 iterations of the sediment carry rule, $\mathit{slopeThreshold} = 0$, $\mathit{radius} = 1$
- 30 iterations of the sediment carry rule, $\mathit{slopeThreshold} = 1$, $\mathit{radius} = 1$
- invert solid-air volume.

The steps are shown in Figure 27.

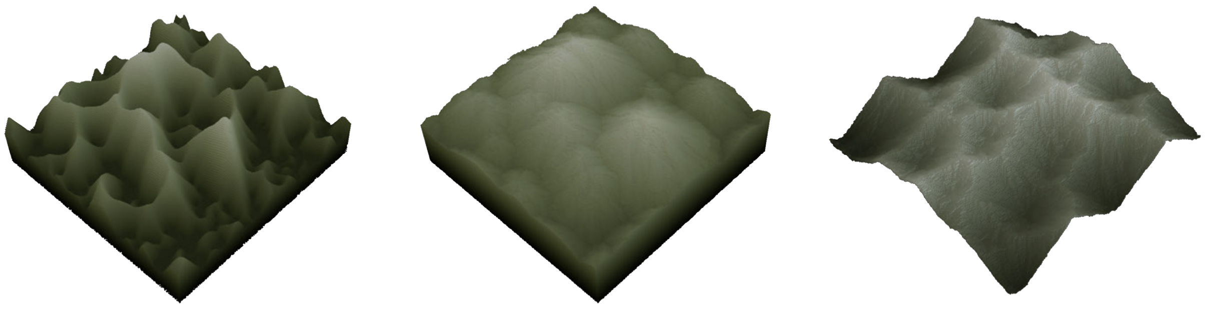


Figure 27: Step by step creation of a mountain range from a $256x256x16$ thin voxel volume. From left to right: displacement, sediment carry, inversion.

To reduce the voxel artefacts apparent in Figure 28, we could increase the simulation resolution or use a polygonization algorithm that will be discussed in Subsection 2.4.2. A higher resolution terrain is shown in Figure 29 rendered using a *tessellation shader* driven by a heightmap. The extraction of heightmaps will be discussed in Subsection 2.4.1. The second terrain was generated using the same steps, starting with a negative value for the *slopeThreshold*.

Creating terrain with the above means has certain advantages. In each rerun of the generation process using the same settings, the features are similar but visually distinctive enough to be placed in a nearby location. The final voxel structure is tile-able enabling our terrain to seamlessly repeat, infinitely or otherwise. Rule parameter values and the shape of the generated features are correlated: We control the scale of our landscape through the selective height displacement radius relative to the simulation space, initial density translates to the ratio of mountains to plateaus, and the height is affected by the threshold parameter. The mountains are shaped by the sediment carry parameter *slopeThreshold* and the number of iterations. Although this relationship could prove complicated for the end user, a program acting as the middle man using real world units as input could further simplify the process.

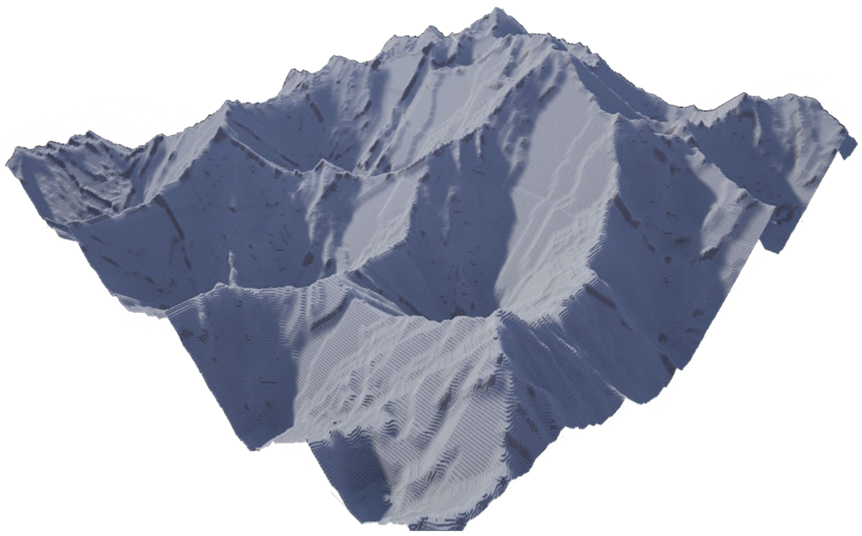


Figure 28: A mountain range generated from a combination of the rules discussed in this chapter with snow/rock colour applied based on slope steepness. Rendered in Unreal Engine 4.

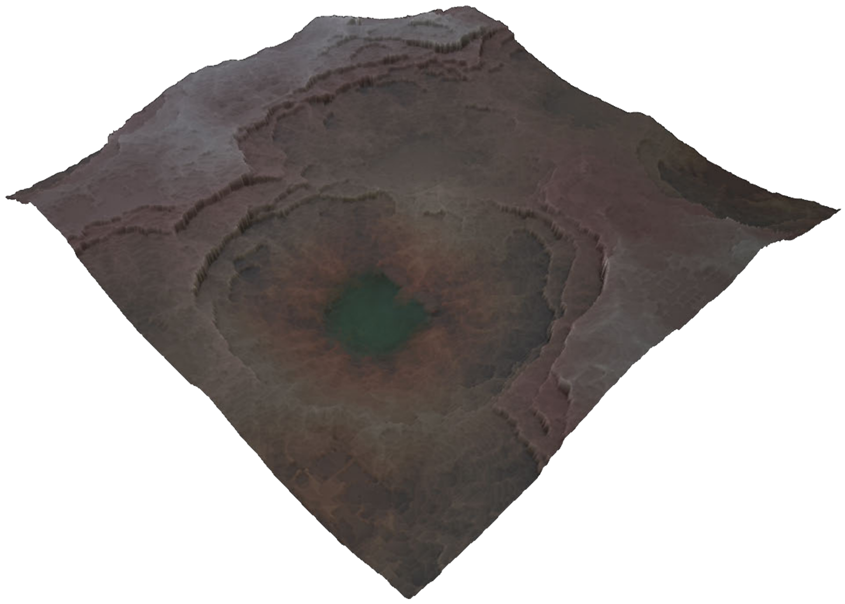


Figure 29: An alien terrain featuring an impact crater generated using a similar technique, colourization applied by height using a lookup table (LUT). Rendered in Unreal Engine 4.

2.3 Cave interior

Common cave interior features include stalagmites, stalactites and large supporting columns. These proved a challenge to generate in the same simulation space as they demand different levels of detail. Previously we focused on generating low resolution large scale structures, granting that high frequency detail would be provided by other means. There exist many ways to achieve this, either through textures that guide surface tessellation and normals or superimposed objects. Thus, we used two automata rules, one for the generation of the large cave interior, and another for the stalagmite growth and rock surface.

2.3.1 Large scale features

A significant obstacle when studying three-dimensional automata is the inability of complete visualization. Many times interesting behaviour unfolds behind the rendered surface, with no practical way to visualize it. Thus, in the process of evaluating the significance of a discovered rule we used a number of boolean operations and simple rules that discarded surface -or otherwise unimportant- cells to reveal the behaviour underneath. One of these operations, expressed as an automaton rule is the following:

```
Rule: Cull
Parameters: cullThreshold, radius
Output: cell
solidCount ← solid cell count in radius;
if solidCount > cullThreshold then
    cell ← current cell state;
else
    cell ← Air;
```

The process we used for generating the cave interior is:

- initialize the grid with a thin volume of 0.5 density.
- apply 20 iterations of the selective displacement rule, *radius* $= 3$, *threshold* $= 0.6$.
- apply the cull rule for a single iteration, *cullThreshold* $= 26$

The final result could benefit from the application of additional rules to smooth or further diversify the cave features. Another approach would be the space colonization rule, although the surface generated is rather aggressive and would require post processing in a 3D application for practical usage. The two are shown in Figure 30.

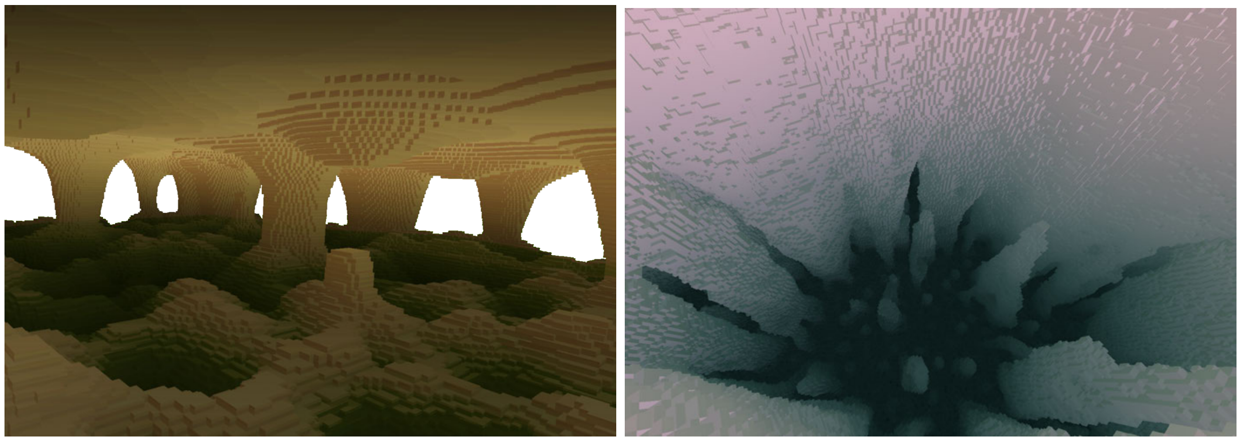


Figure 30: Left: voxel cave interior generated by the height displacement and cull rule, on a 256^3 simulation space. Right: top down view of a voxel cave interior generated with the space colonization rule on a 256^3 simulation space (*threshold* $= 0.4$, *limit* $= 0.6$, *radius* $= 2$)

2.3.2 Polygon Meshes

The space colonization rule is ideal for stalagmite and stalactite generation. Experimenting with different configurations allows us to create multiple visually distinctive formations. Starting with *threshold* $= 0.4$, *limit* $= 0.57$, *radius* $= 2$, structures raise from the voxel volume. To stop the stalagmite growth prematurely, we lower the *limit* accordingly (in this case to 0.55) after some iterations. This allows us to control the height of the formation. Modifying the initial density enables greater size variation in the stalagmites. The generated voxel structures are shown in Figure 31. In case we wanted to dynamically populate our world, this procedure, as with all the rules discussed previously, is predictable in each repetition. Once the desired parameter values have been found, there is no need for supervising the generative process.

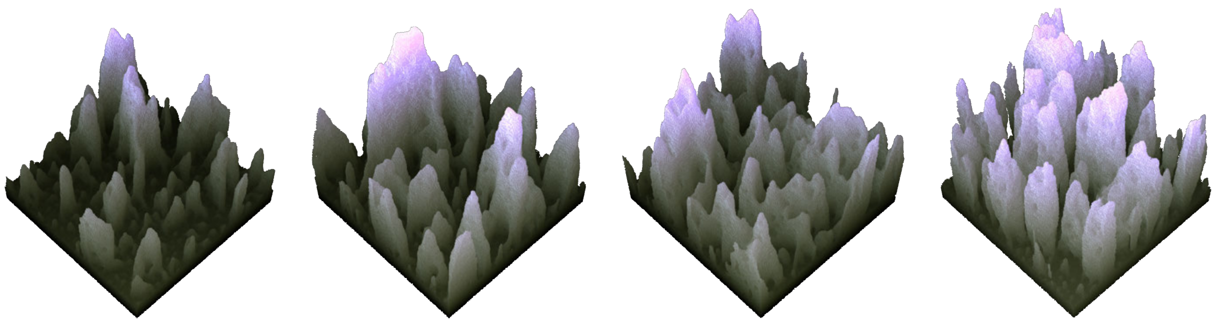


Figure 31: Stalagmites generated with the space colonization rule on a 256^3 simulation space. (*density*, *threshold*, *limit*, *radius*) parameters, from left to right: $(0.6, 0.4, 0.57, 2), (0.45, 0.4, 0.57, 2), (0.5, 0.4, 0.59, 2), (0.5, 0.33, 0.44, 2)$

As seen in Figure 31 the structures wrap around the simulation borders. Generally, there is no need for organic meshes to be tile-able, as intersections are hardly noticeable and sometimes encouraged to create new structures from a limited asset pool. Therefore, a disc or planar volume that doesn't reach the borders, is better suited for initialization. Another method, is to regard all cells outside the simulation border as solid or air, depending on the desired behaviour. The cells in the bottom can be discarded by clipping based on height, if we wished to attach the stalagmites on a different surface.

Processing the generated voxels depends on the usage. For real-time graphics applications, we propose hand-picking interesting stalagmite structures in a 3D package and discard other polygons. A high resolution mesh used for baking additional maps (normal, ambient occlusion) is produced by subdividing and relaxing the extracted surface. This is only needed in low resolution simulations as it alleviates any underlying voxelization or surface extraction artefacts. Afterwards, the mesh is decimated, unwrapped and textured through traditional means. For use in offline rendering, the surface could be rendered as exported from the CA simulation, providing a high resolution was set ($>= 256$). For a game using real-time procedural generation, we could extract the surface using a lower resolution polygonization grid. We tested all the above scenarios with positive results. Some of the steps are shown in Figure 32.

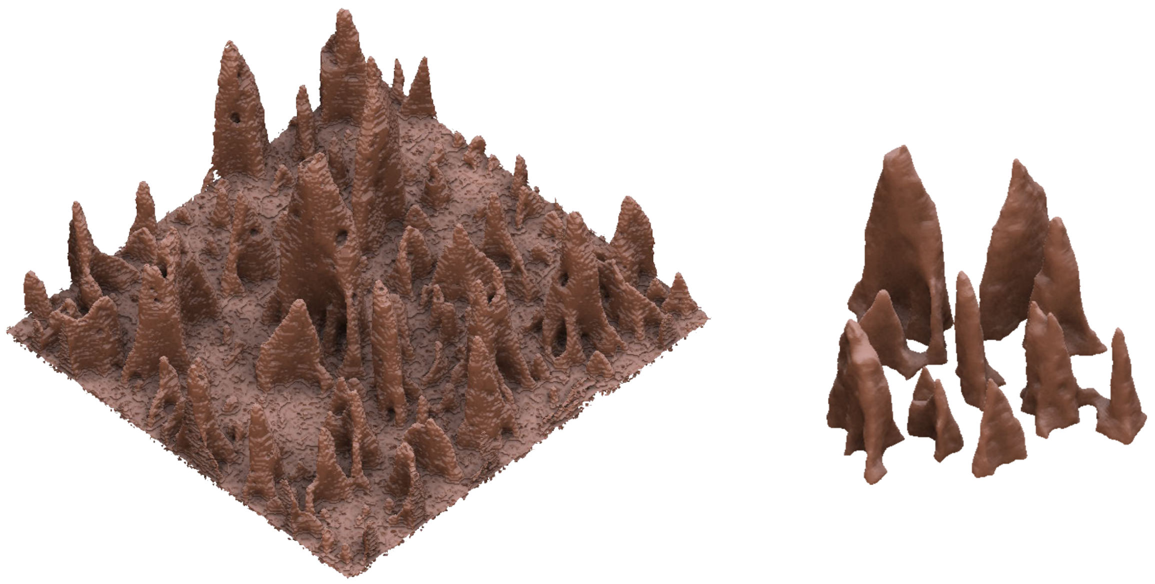


Figure 32: Left: original stalagmite extracted surface, right: hand-picked and simplified meshes.

2.3.3 Surface detail

Using triplanar mapping, we can project a texture on the cave walls that allows us to create and displace polygons through tessellation, and thus provide much higher resolution structures. Again, using the space colonization rule with different threshold values, resulted in some interesting texture maps for displacement or bump. The selective height displacement variation rule also proved effective. The maps shown in Figure 33 were created using the heightmap extraction method discussed in Subsection 2.4.1.

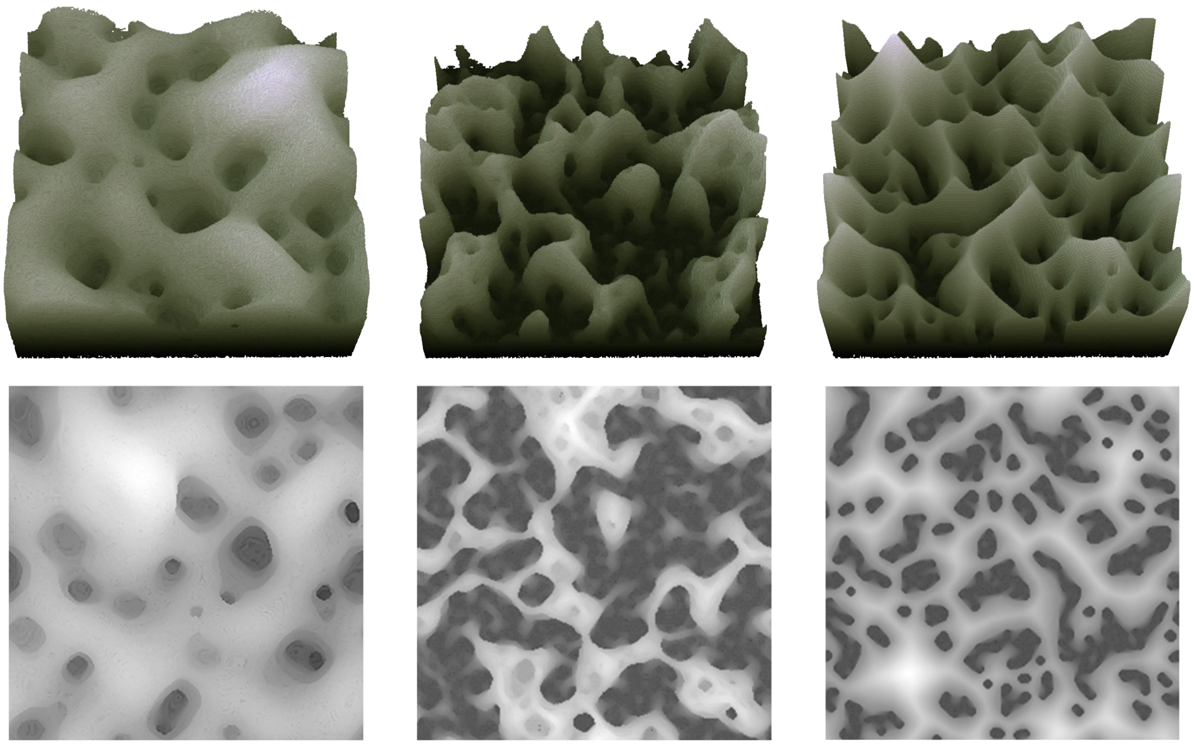


Figure 33: Volume (top row) and extracted height map (bottom row). Rules applied on a 256^3 **thick plane, left to right, space colonization rule** ($threshold: 0.4$**,** $limit: 0.46$**,** $radius: 6$)**, space colonization rule** ($threshold: 0.38$**,** $limit: 0.46$**,** $radius: 3$)**, combination of selective height displacement rule configurations.**

2.4 Surface extraction

2.4.1 Heightmaps

The most prominent method of storing terrain data in computer graphics is through a *heightmap* or *heightfield*. Heightmaps are single-channel textures with each pixel's value representing the relative elevation of the target terrain. Black values represent the minimum height and white ones the maximum. A plane equally subdivided with as many vertices as the heightmap pixels is conformed accordingly to produce the terrain mesh.

This technique although practical and fast to render has some drawbacks, the most obvious being the inability to represent overhangs and caves. Also for an 8-bit image only 256 height levels can be stored in a single channel, so 16-bit images or using the green, blue and alpha channels in addition, are common practices.

The way to convert our generated voxel terrain to polygon meshes is straightforward. First we measure the highest and lowest voxel. Then, iterating from the top of each column in our simulation we record the height of the first solid voxel encountered. The following formula is used to calculate the corresponding pixel's luminance:

$$heightmap_{x,z} = \frac{voxelHeight_{x,z} - minHeight}{maxHeight - minHeight} \times 255$$

Note that we remap the voxel height to cover the full 256 range the 8-bit image supports, but if our grid dimensions are under 256 it would not make a difference. In case we needed a higher resolution terrain than our voxel grid width, a *Gaussian blur* after rescaling the heightmap should provide an acceptable result.

2.4.2 Polygonization

There is a vast body of literature on the efficient rendering of volumetric structures, much richer in properties than the discrete-state voxels generated in our simulations. Data structures used for storing and raycasting the voxel data, like sparse voxel octrees [11] could replace the toroidal network we use for accessing the neighbourhood of a cell.

We also explored some of the most popular surface extraction algorithms, namely *marching cubes* [13] and *Dual Contouring* [10], both of which greatly benefit from parallel processing on the GPU. A concept these two techniques share is the *density function*. This function takes a point in 3D space as input and returns a value representing the signed distance from that point to the surface, with positive values signifying that the point lies inside the volume. By means introduced by these methods, vertex positions are selected and properly connected to form the polygon surface. Also, normal vectors are calculated by sampling the density function on the polygonization grid and calculating a gradient. Here lies a problem, as we cannot provide an accurate floating point density function, or it would be impractical and time consuming to do so while simulating the automaton. This results in faceted, jaggy surfaces with intersection artefacts in low density volumes. Calculating smooth normals for each vertex from the normalized cross product of the adjacent faces, alleviates the faceted appearance of the generated mesh, but does not improve the fidelity.

Simulations in this thesis sometimes involve close to a billion of cells, changing state almost every frame. Extracting a smooth isosurface from binary volumes of comparable size was investigated by Lempitsky [12] showing great results, but certain compromises on the real-time generation frequency of the surfaces are needed, depending on the usage.

2.5 2D Noise generation

2.5.1 Widely used methods

To measure the viability of CA techniques to produce terrains seen in modern CGI, a comparison is drawn with common two and three-dimensional noise generation methods.

Perlin noise [19] is a proven method for fractal terrain generation. Efficient for cave structures and is commonly applied on terrains at a medium/high frequency. A visual comparison between CA-generated noise and Perlin noise is shown in Figure 34.

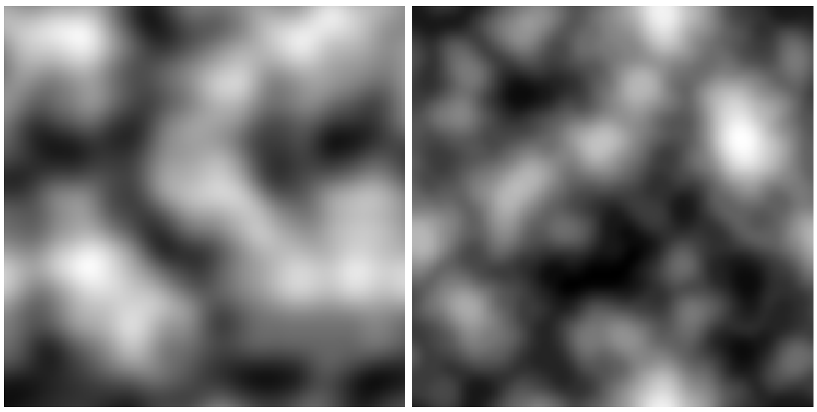


Figure 34: Left: 5 octaves of perlin noise with 0.3 persistence. Right: perlin-like noise generated with the height displacement and cluster rule.

Worley noise [23], is used to simulate the texture of waves and stone crevices. It is applied on a lower frequency to form mountain ranges and other large features. A visual comparison between CA-generated noise and Worley noise can be seen in Figure 35.

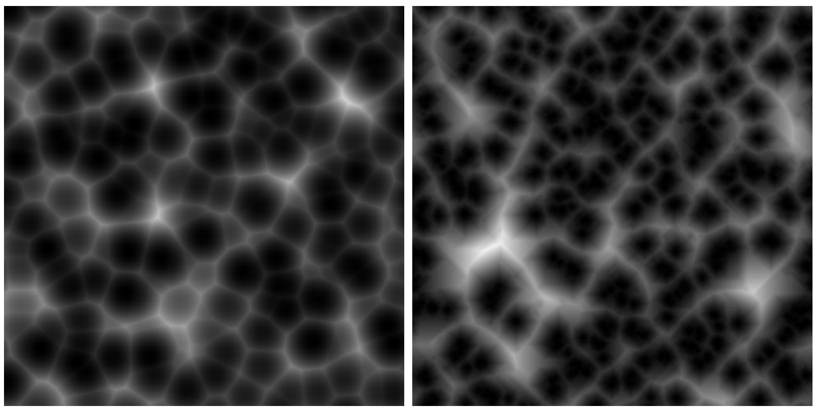


Figure 35: Left: Worley noise. Right: Worley-like noise generated with the height displacement rule $(threshold : 0.77, radius : 2)$**.**

Of course the CA-based technique is lacking both in speed and scalability. This comparison mostly aims to establish 3D CA as a viable framework for producing interesting two-dimensional noise for texturing and polygon mesh manipulation. Imitating the algorithms used for Perlin and Worley noise generation with a CA or hybrid method could be possible, but we are mostly interested in the emergence of such structures from simple rules.

2.5.2 CA generated noise and patterns

In Figure 36 we see show some interesting examples of CA generated noise patterns.

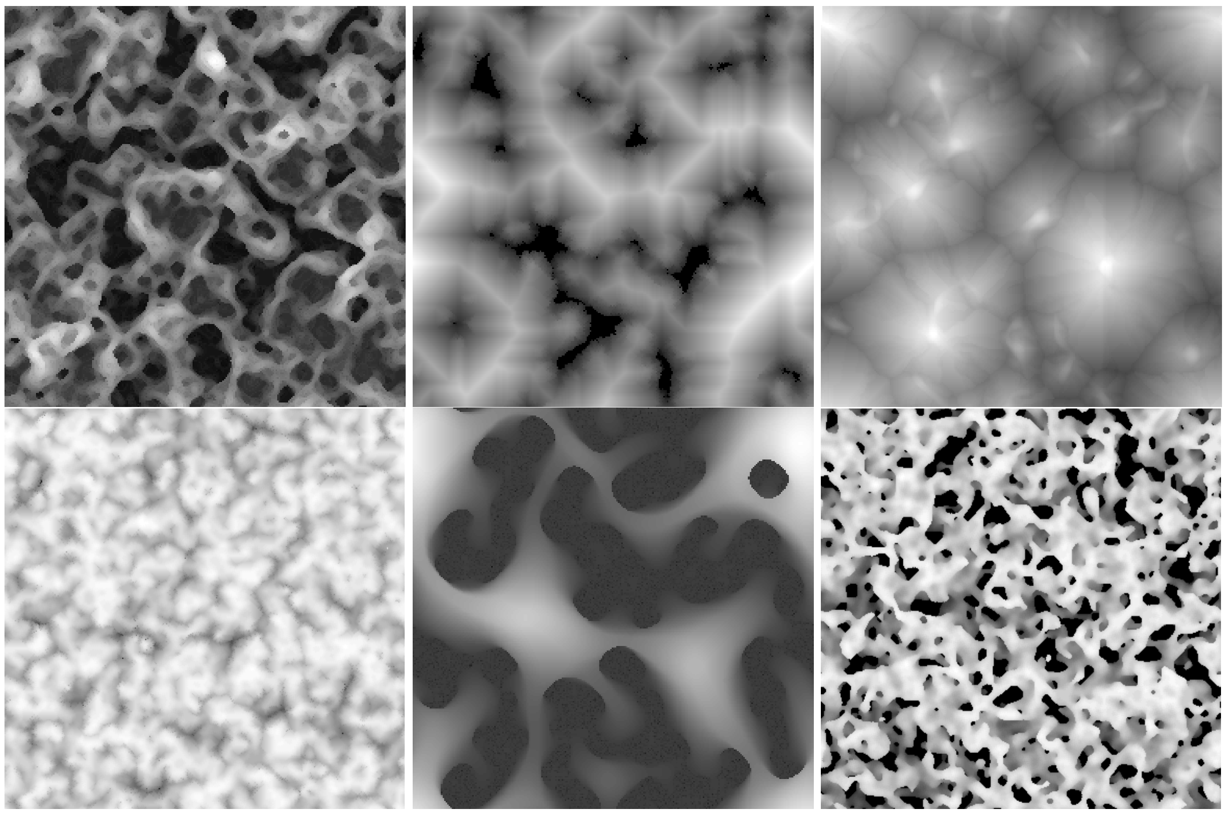


Figure 36: 3D CA generated noise.

2.6 Abstract results

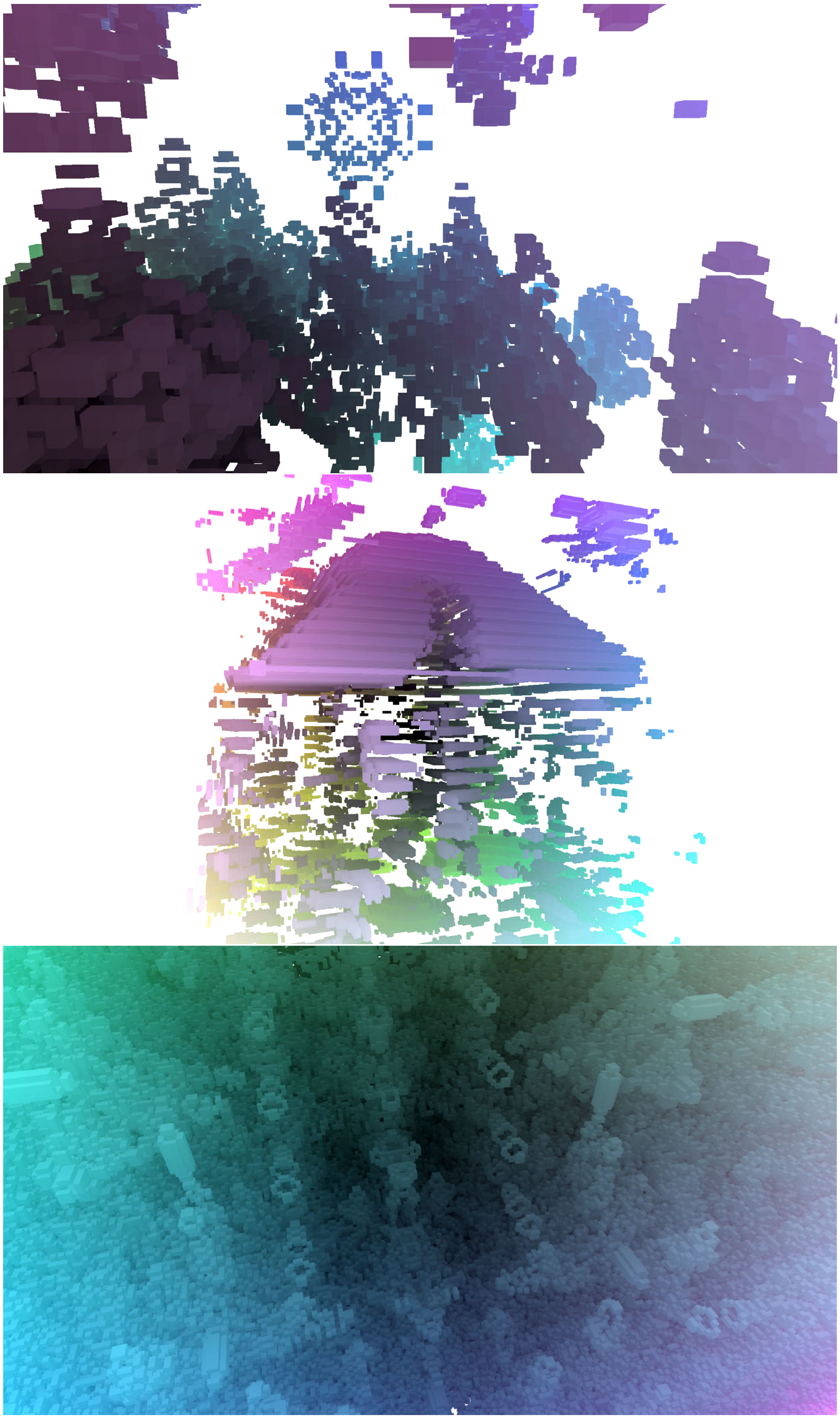


Figure 37: Some abstract results.

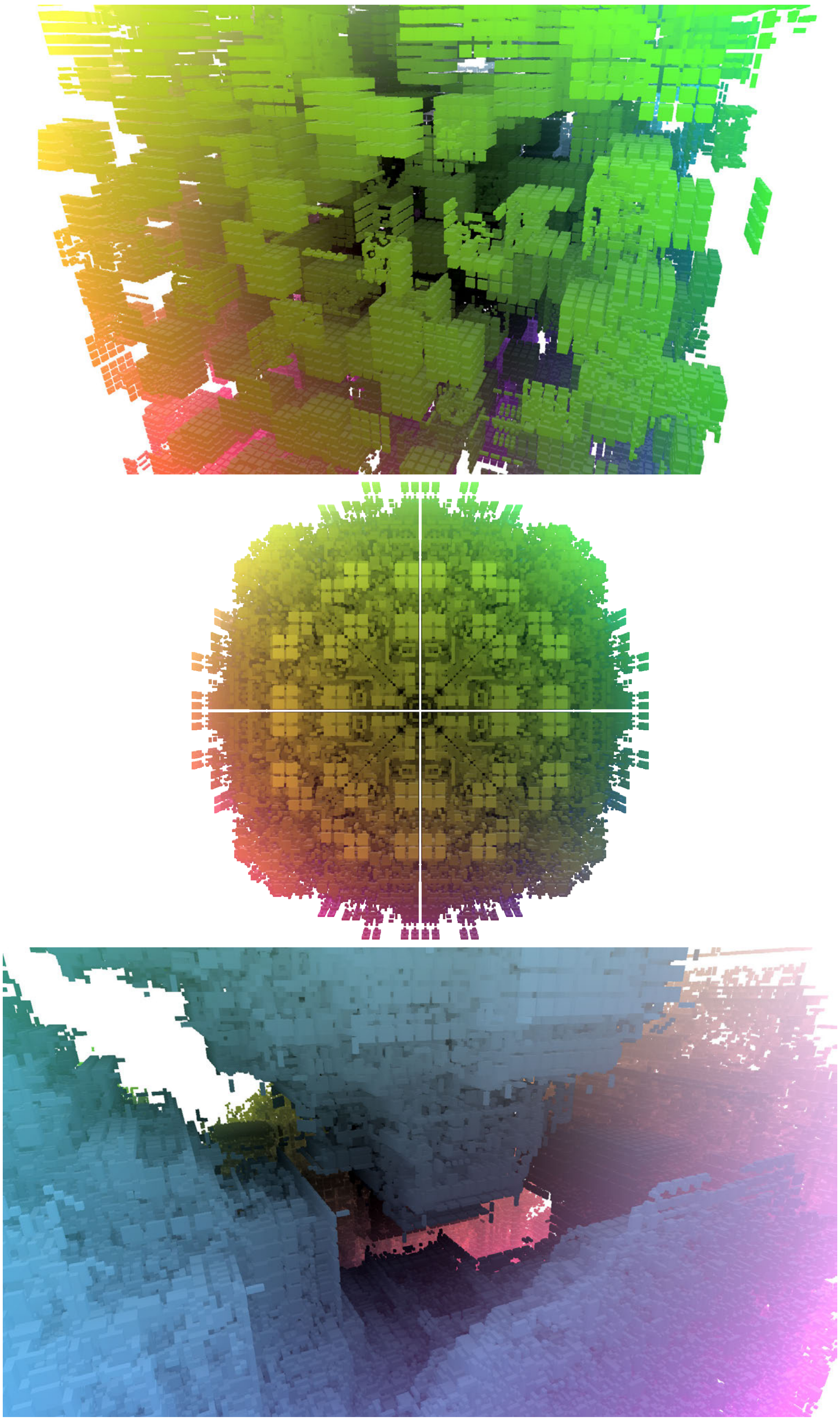


Figure 38: More abstract results.

3. POPULATING THE WORLD WITH CELLULAR AUTOMATA

3.1 Goals and requirements

In this chapter we discuss simple automata rules that populate our generated worlds with additional objects and surface materials. Although these methods are inspired by natural processes, our goal is not to produce accurate physical models but to generate interesting and believable patterns with easily parameterized behaviour for practical usage in game levels.

A prerequisite of this process is that all existing objects in our level are either in the form of voxels having an integer coordinate vector and discrete state, or a preprocessing stage partitioned space in evenly sized cells containing the necessary data on the enclosed models. These rules affect only empty cells, leaving underline voxels intact.

3.2 Population rules

3.2.1 Moss spread

This rule guides the spreading of moss on a variety of surfaces. Could be used for highly detailed voxel structures or entire terrains. This is a texture effect, thus we are not interested in the placing of the moss voxels on the y axis. For another CA method that simulates the propagation of green patina on copper see [8]. The moss spread rule in action is shown in Figure 39.

```
Rule: Moss spread
Parameters: spreadFactor
Output: cell
cell ← current cell state;
bottomCell ← cell state directly under the cell;
topCell ← cell state directly over the cell;
bottomSolidCount ← Solid cell count under the cell;
topSolidCount ← Solid cell count over the cell;
// has solid under it and room to grow
if $topCell = Air$ and $bottomCell = Solid$ and $bottomSolidCount > 6$ and
  $topSolidCount < 6$ then
    return Moss;
// otherwise, spread as if by neighbouring moss influence
mossSpread ← Moss cell count around the cell;
aroundAirCount ← Air cell count around the cell;
if $mossSpread > 2$ and $aroundAirCount < spreadFactor$ then
    return Moss;
return cell;
```

3.2.2 Forest spread

In this simple model, trees grow by means of wind carrying the seeds from afar, and seeds falling from nearby trees. Since we would like to apply this process on a volume with no inherent randomness in density or elevation, a stochastic process was used uniformly distributing the seeds over the simulation space. The environment conditions are evaluated (not enough soil, too many trees competing for resources) aiding the growth or premature

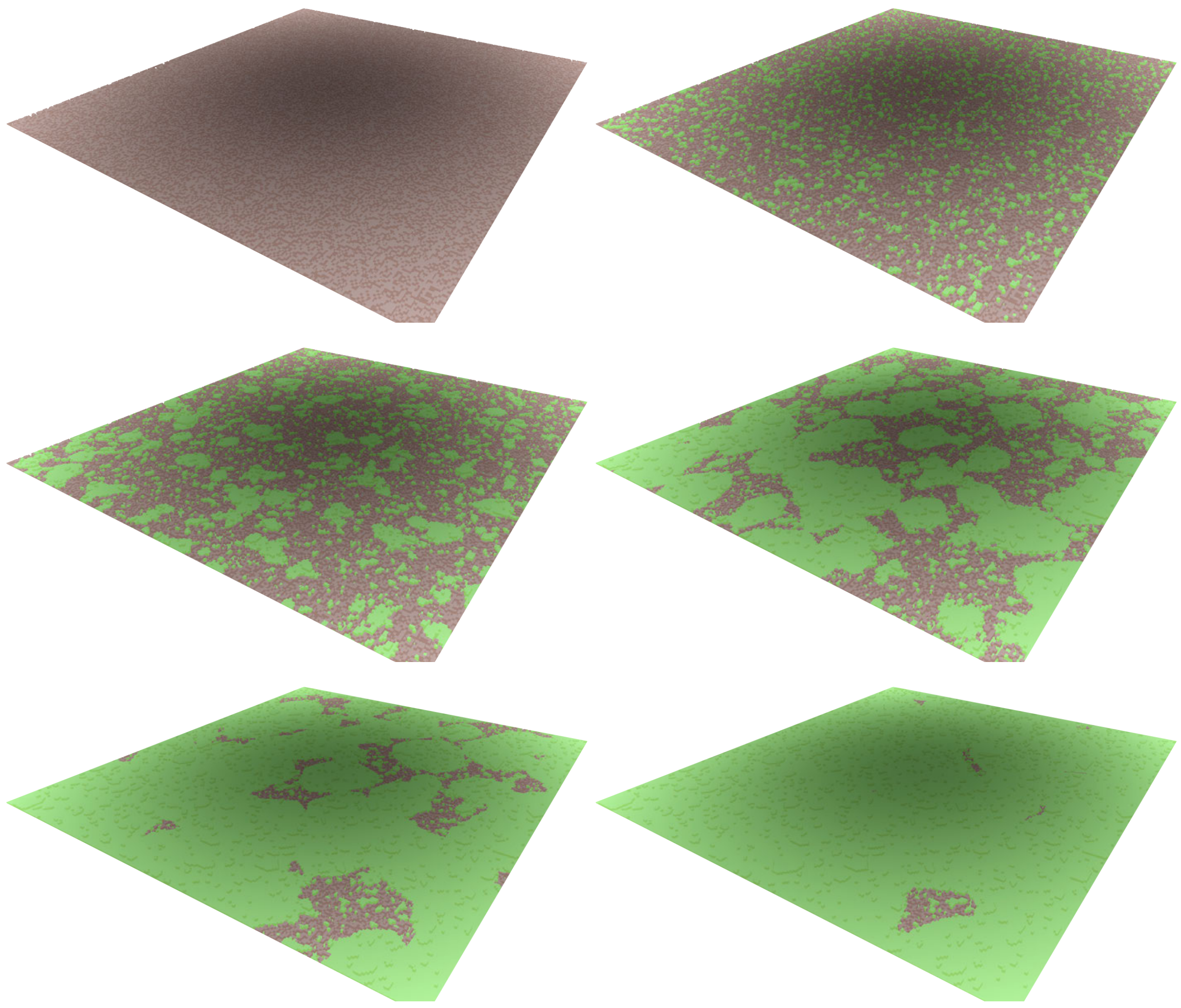


Figure 39: Moss spread rule applied on a 256^3 plane, *spreadFactor* $= 5$. From top to bottom, left to right: 0, 1, 5, 10, 15, 20 iterations.

death of a tree. Multiple foliage species are supported with controlled spacial diversity. Note, these conditions aim to produce a forest usable in a game level, with variation in density and large patches of terrain unaffected by the rule, separating the different forest sections.

Some information on the rule parameters:

- *treeVariations* is the count of unique foliage species.
- *blendFactor* is a floating point value $\in [0, 1]$ that affects the separation of the tree variations.
- *windSeedProbability* $\in [0, 1]$, affects the number of seeds placed by the wind in proportion to the simulation space. Can be set to zero after a tree group is formed.
- *limit* denotes the maximum allowed count of trees in a single neighbourhood.
- *soilThreshold* is the number of *Soil* cells in the neighbourhood required for a tree to grow. A low *soilThreshold* enables tree growth on uneven terrains and mountain peaks.

Immediate Moore neighbourhood was used, and each cell has the capacity for a single tree. All foliage variations have cell state indices $\geq$ *Tree*, where *Air* $= 0$, *Soil* $= 1$, *Tree* $= 2$.

The effect of each parameter in the forest spread rule is shown in Figures 40, 41 and 42.

```
Rule: Forest spread
Parameters: $treeVariations$, $blendFactor$, $windSeedProbability$, $limit$, $soilThreshold$
Output: $cell$
$cell \leftarrow$ current state;
$neighbouringTrees \leftarrow$ tree count in neighbourhood;
// prune lone trees or trees in overcrowded patches
if $state \geq Tree$ and $neighbouringTrees \notin [1, limit]$ then
    return $Air$;
$bottomCell \leftarrow$ cell state directly under the cell;
$topCell \leftarrow$ cell state directly over the cell;
$soilCount \leftarrow$ count of soil cells in neighbourhood;
// ideal growing conditions are met
if $cell = Air$ and $topCell = Air$ and $bottomCell = Soil$ and $soilCount \geq soilThreshold$ then
    // seed carried by wind
    if $uniform\ random \in [0, 1] < windSeedProbability$ then
        return $Tree + uniform\ random \in [0, treeVariations - 1]$;
    // seed by nearby trees
    if $neighbouringTrees = 3$ then
        if $uniform\ random \in [0, 1] < blendFactor$ then
            return a random $Tree$ variation from the neighbourhood;
        else
            return most common $Tree$ variation in neighbourhood;
return $cell$;
```

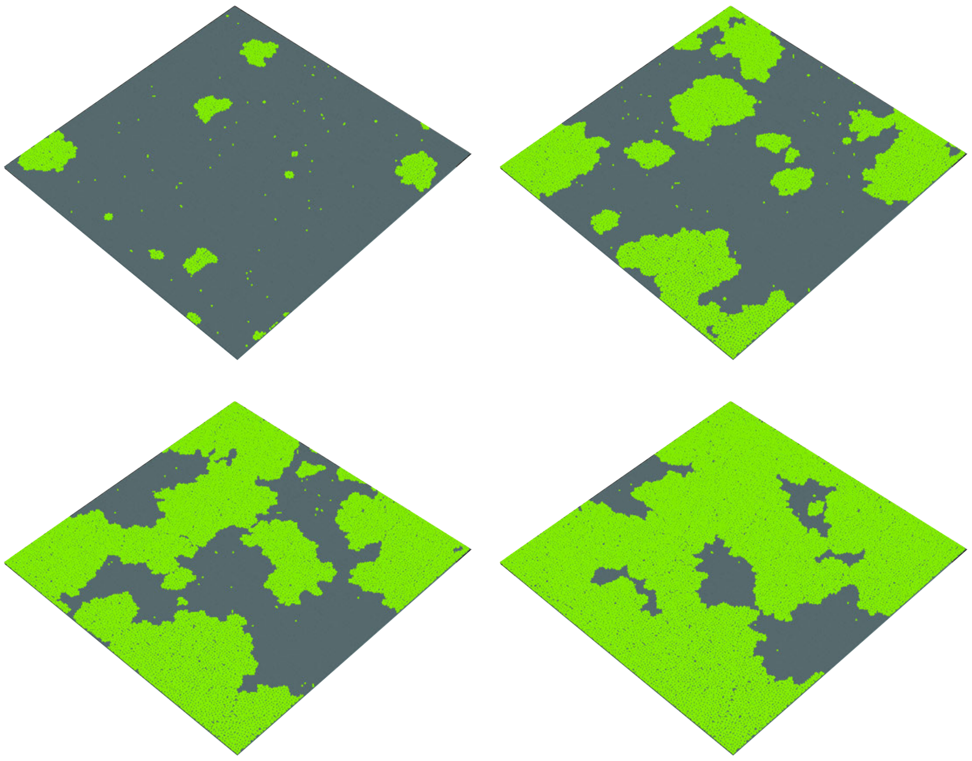


Figure 40: Select iterations of the forest spread rule applied on a 256^2 plane, $treeVariations = 1, windSeedProbability = 0.001, limit = 7$.

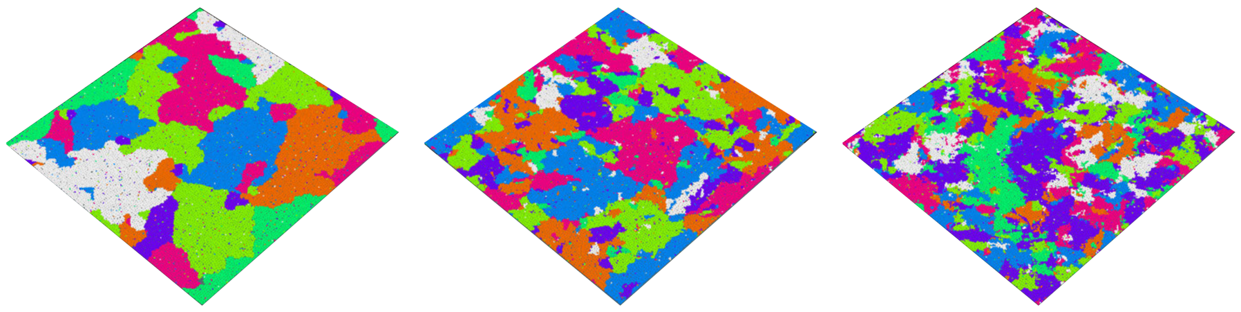


Figure 41: Forest spread rule run to completion on a 256^2 plane, $\mathit{treeVariations} = 8, \mathit{windSeedProbability} = 0.001, \mathit{limit} = 7$**. Each species is assigned a different colour.** $\mathit{blendFactor}$ **from left to right:** $0.0, 0.5, 1$**.**

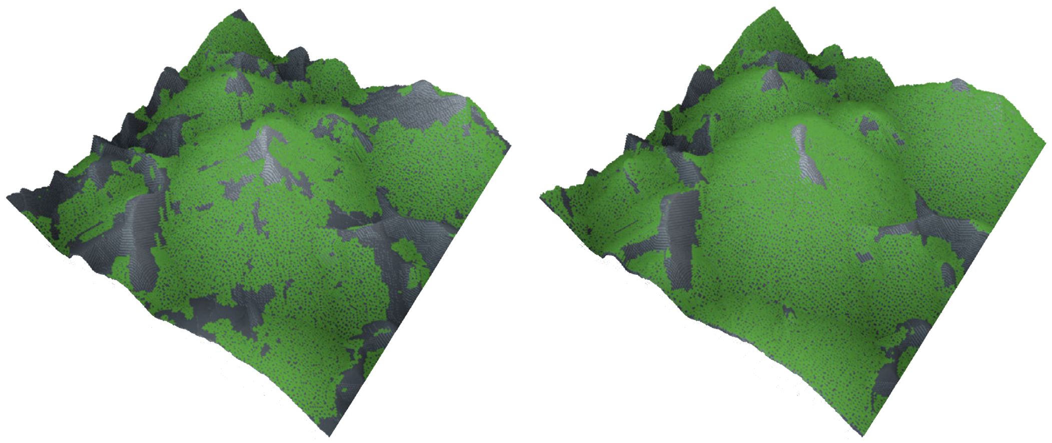


Figure 42: Forest spread rule applied on a 256^3 terrain shaped by the height displacement and sediment rules, $\mathit{treeVariations} = 1, \mathit{windSeedProb} = 0.001, \mathit{limit} = 7$**. Left:** $\mathit{soilThreshold} = 7$**, right:** $\mathit{soilThreshold} = 4$**.**

3.2.3 Stalagmites and stalactites

This rule populates a cave interior (like the ones discussed in Sections 2.1, 2.3.) with stalactite and stalagmite meshes. It is a simplified model of the natural process: Stalactites grow on the cave ceiling, when there's enough rock cells to support them and empty space underneath. Stalagmites obey the same condition with the added requirement that a stalactite should exist in *searchDistance* over the cell. Expanding the *searchDistance* parameter is computationally inexpensive, as the search is constrained to a single-cell column above the potential stalagmite. As seen in Figure 43, by using the right parameters the cave is thinly populated with no randomness in the rule.

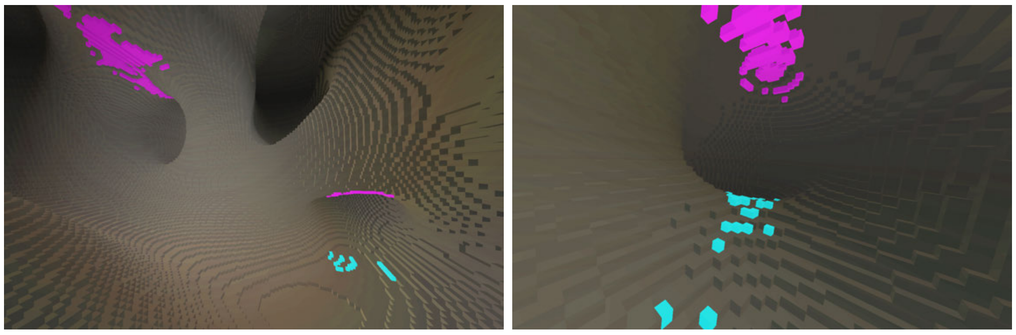


Figure 43: Cave interior population, stalactites coloured purple, stalagmites cyan. $\mathit{rockThreshold} = 9$, $\mathit{spaceLimit} = 0$, $\mathit{radius} = 25$

```
Rule: Stalgmite/stalactite spread
Parameters: rockThreshold, spaceLimit, searchDistance
Output: cell
state ← current state;
bottomCell ← cell state directly under the cell;
topCell ← cell state directly over the cell;
if cell = Air then
    bottomRockCount ← Rock cells under the cell;
    topRockCount ← Rock cells over the cell;
    // create stalactite
    if topCell = Rock and bottomCell = Air then
        aroundRockCount ← Rock cells around the cell;
        if topRockCount >= rockThreshold and aroundRockCount <= spaceLimit and
          bottomRockCount = 0 then
            return Stalactite;
    // create stalagmite
    if topCell = Air and bottomCell = Rock then
        overStalactiteCount ← Stalactite cell count in searchDistance over the cell;
        if overStalactiteCount >= 1 then
            return Stalagmite;
return cell;
```

3.3 Usage in graphics engines

The game level is populated by spawning the respective objects on the cells designated by the resulting CA grid. A problem arises when the grid resolution is small and the spawn points do not lie on the surface. In most cases we can cast a ray from the center of the spawned cell, parallel to the -y axis and snap our mesh to the first surface encountered. Depending on the desired mesh orientation and available physics tools, we could raycast upwards or using sphere overlap detection, to identify the closest surface. Afterwards, the mesh is rotated so the model's up axis aligns with the surface normal, if appropriate.

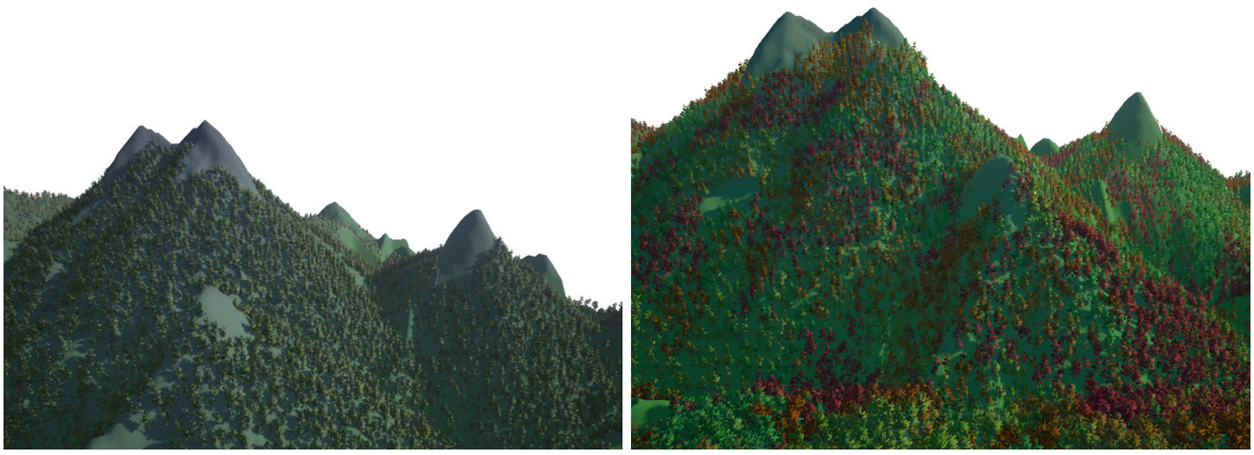


Figure 44: Simulation shown in Figure 42 imported in Unreal Engine 4. Tree species were differentiated with a hue shift. The CA generated terrain was exported as a heightmap.

An efficient way to use the grid data in conjunction with heightmap represented terrains, is through a single channel texture mask called the splatmap. Each discrete *x*, *z* coordinate vector in our world corresponds to a splatmap pixel. If a cell is inhabited that pixel is white, black otherwise. We may use additional channels to represent multiple meshes. In this case, the mesh spawn point can be deduced from the corresponding heightmap value, while the orientation is a function of the relative difference in luminance with the neighbouring pixels. Finally, because the vertex height information and the splatmap share the same UV space, we can also affect the terrain texture using the splatmap information.

A technique that applies to all procedural placing methods, is assigning a small random offset to the final object position, effectively erasing the influence of the spawn grid. Depending on the mesh in question, we may introduce scale, rotation and colour variation, resulting in a more natural-looking landscape.

The mountain in Figure 44 was populated with trees using the techniques discussed above.

4. TEXTURE GENERATION WITH CELLULAR AUTOMATA

4.1 Goals and usage

In this chapter we detail the creation of a cellular automata based texture generator. This tool aims to capture organic growth commonly seen in reaction-diffusion models, straying from the artificial, nested and repeating geometric patterns produced by conventional automata. There are of course automata able to generate colourful patterns like *cyclic cellular automata* [7] and stochastic cellular automata to name but a few. For a summary of beautiful CA generated artworks see [1].

Finding the right balance between automatic procedural generation and user guided creation is no easy task. In the case of CA it's a near impossible endeavour since the patterns they generate are fairly complex and seemingly random. Some research has been done on predicting and classifying the behaviour of automata [22], but not from an art generation standpoint.

Our goal for this tool is to find a middle ground between the chaotic or seemingly random generative behaviour and the end users intuition. To achieve this, a meaningful parametrization of the simulation is supported and the resulting behaviour is impervious to initial conditions. Strategically placed initial states might guide the formation of structures but the overall pattern development and colourization is strictly depended on the automaton rule configuration. In its current form the generator is fairly versatile in the sense that it is able to produce a variety of organic patterns and geometric shapes. Although it's use would be impractical in a professional computer graphics creation pipeline, it could serve a purpose in recreational computing and games that are somewhat abstract in nature.

4.2 Generative process

Our algorithm draws inspiration from *image kernels* [14] and automata with memory. A three-dimensional multi-state totalistic automaton evolves in an orthogonal grid, with as many cells on the *x* and *z* axis as the desired texture dimensions. The height should be at least 3, with each horizontal layer representing a colour component. The cells consider the Moore extended neighbourhood horizontally with each neighbour assigned a floating point multiplier. The multiplier matrix varies per height to introduce further variation. Vertically, cells are able to communicate with every other cell with the same *x*, *z* coordinates. The vertical neighbourhood states are again multiplied by a symmetric matrix that denotes the influence cells on different heights have on one another. The new cell state becomes the weighted average of the new and previous cell states. Finally, the structure is flattened to produce a colour for the corresponding pixels on the screen.

For each horizontal layer a number of possible states is picked between 100 to 1600. We found that using discrete states instead of continuous values aids in increasing the various classes of behaviour. When rules are evaluated, floating point numbers are rounded to the largest previous integer on every occasion. This ensures that edge conditions are met by a small number of cells, further separating their paths of evolution.

The state calculation of the horizontal neighbourhood at height h with neighbourhood radius r in pixel coordinates x, y is:

$$neighbHorizontal_{hxy} = \frac{\sum\limits_{i=-r}^{r} \left(\sum\limits_{j=-r}^{r} cell_{hij} \cdot kernelHorizontal_{hij} \right)}{(2 \cdot r + 1)^2}$$

Let H be the grid height. The matrix *kernelVertical* where $kernelVertical_{ch} \in [0..1]$, $kernelVertical_{ch} = kernelVertical_{hc}$ and $\sum\limits_{h=0}^{H} kernelVert_{ch} = 1$, represents the influence of cell with height *c* on *h*. The final neighbourhood calculation:

$$neighb_{hxy} = \sum_{c=0}^{H} (neighbHorizontal_{hxy} \cdot kernelVertical_{ch})$$

Since CA should be locally constrained a small value for the grid height or the *kernelVertical* size is appropriate (3 - 20).

Let S_h be the maximum state for cell with height *h*, M the memory size with $\sum_{m=1}^{M} memWeight_m = 1$. We compute the future state $n+1$:

$$state_{hxy_{n+1}} = \sum_{m=0}^{n-1} (state_m \cdot memWeight_m) + (neighb_{hxy} \bmod S_h) \cdot memWeight_n$$

A Hue Saturation Value vector is constructed by selecting three height layers, which is then converted to RGB:

$$RGB_{xy} = HSLToRGB(state_{1xy}, state_{2xy}, state_{3xy})$$

Although more elaborate cell colour evaluations are possible, we found that it is more intuitive to cycle through height layers for the states, than combine them into a single vector with possible loss of information. Underline structures of disregarded layers, can be discerned through their influence on those taking part in the colour calculation.

4.3 Results

In the following pages we list some images produced with the above method. These were hand-picked due to their visual interest or significant complexity rarely seen in conventional automata. To produce these figures we use a wide array of horizontal and vertical neighbourhood multipliers, memory weights and initialization methods.

In most cases, multiplier values are random, the output of some trigonometric function or expressed as a function of the neighbour radius. Note, multipliers for each neighbourhood should add up to more than one in order to advance the cell states.

The memory weight array is populated with random descending values that add up to one, with the greater being the weight for the current state.

To initialize the automaton, starting from all zero cells, we set a small number of cells in the center of the simulation to a uniformly random value, or simply randomize the entirety of the grid. Traces of the first initialization method are evident in Figure 45, where the simulation was stopped before the change propagated to the texture borders.

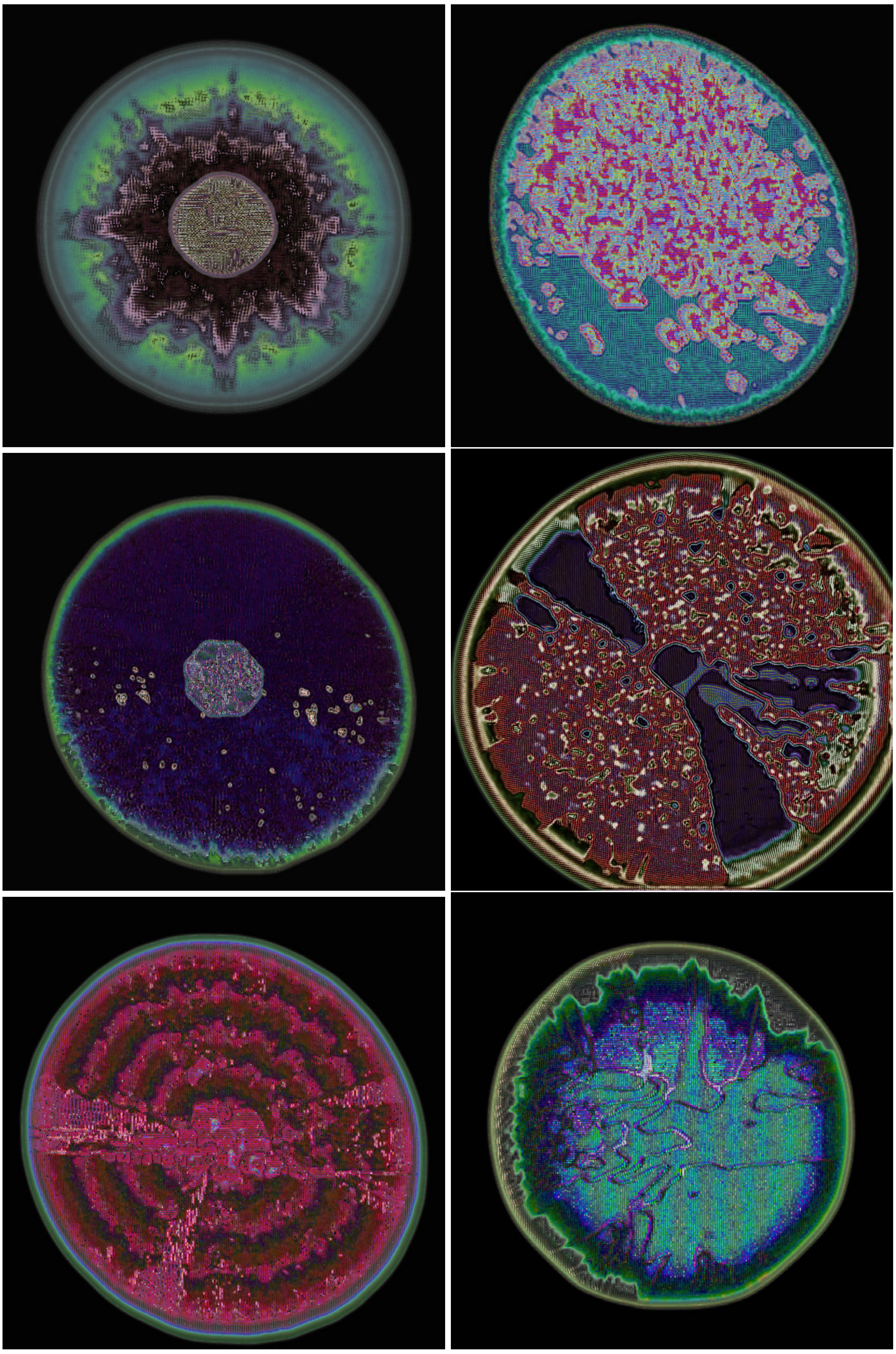


Figure 45: Texture Generator outputs, small number of iterations.

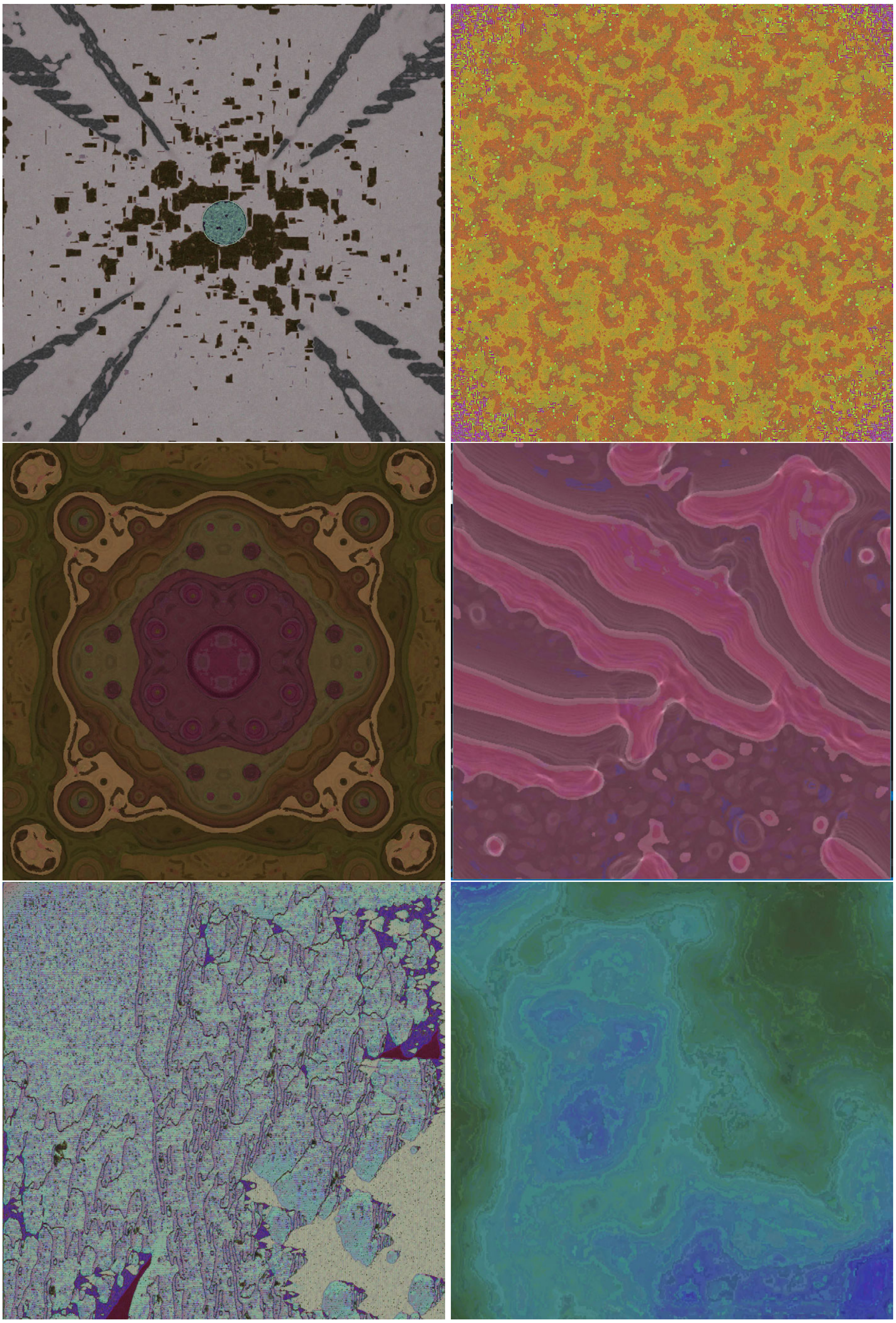


Figure 46: Texture Generator outputs.

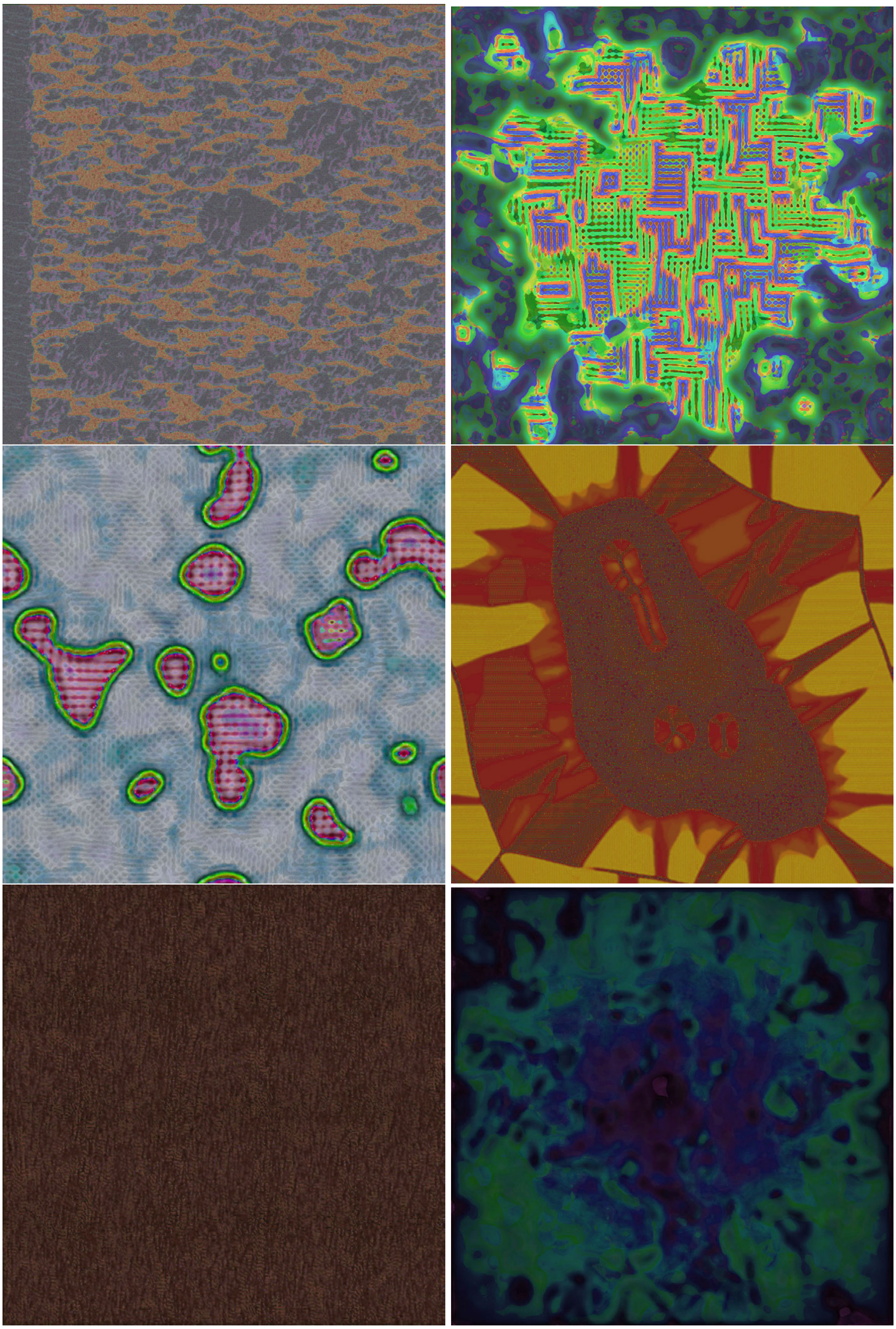


Figure 47: Texture Generator outputs

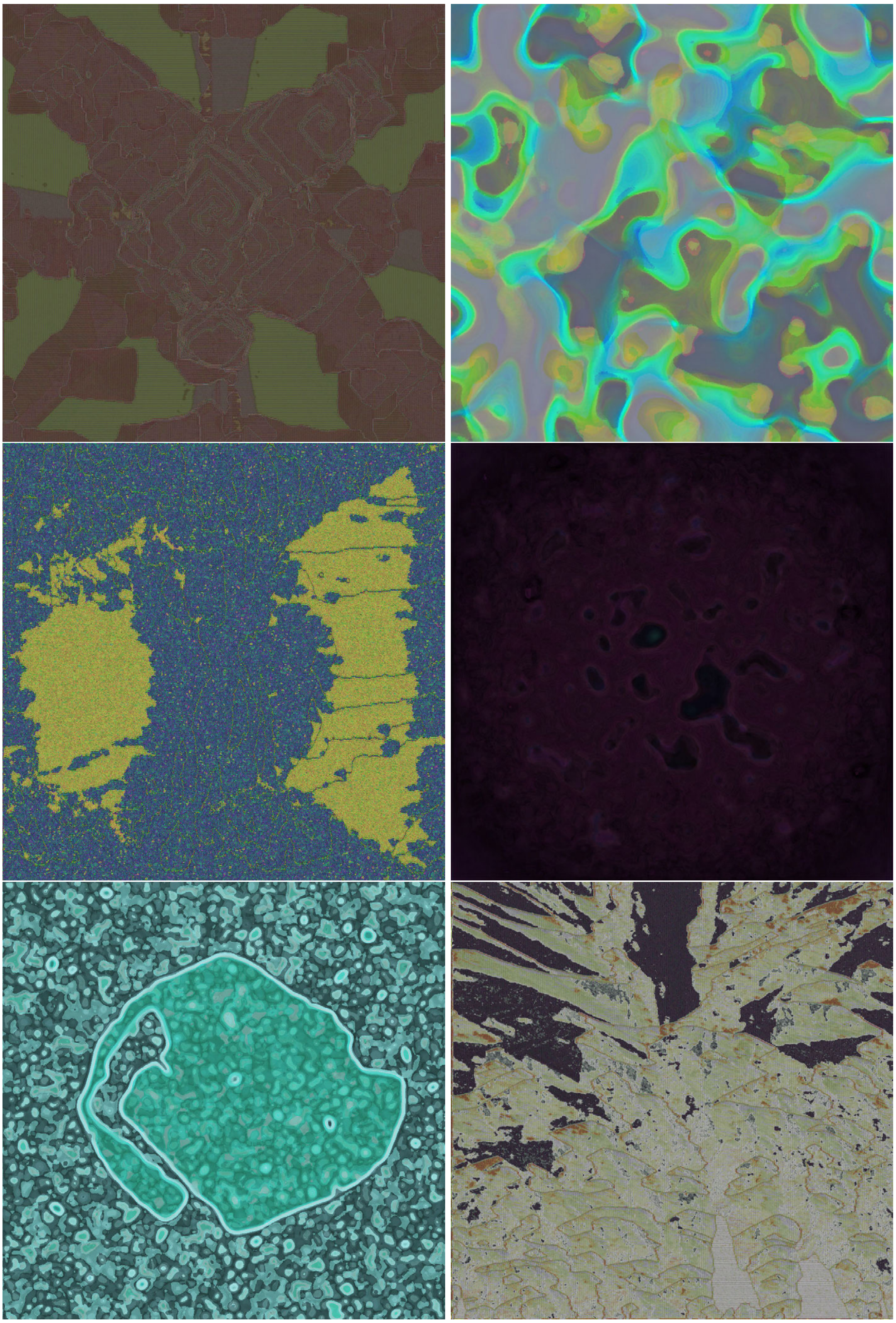


Figure 48: Texture Generator outputs

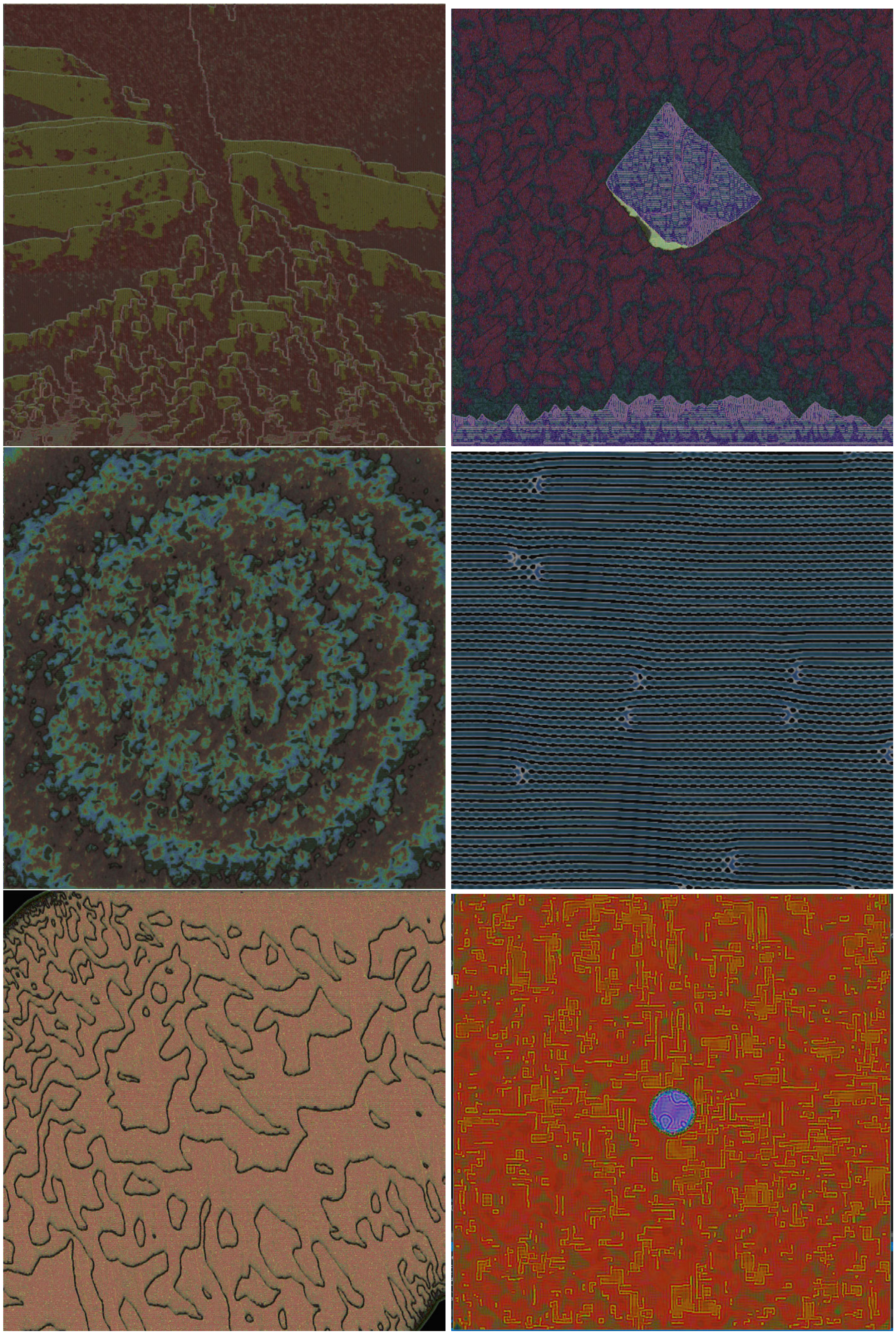


Figure 49: Texture Generator outputs

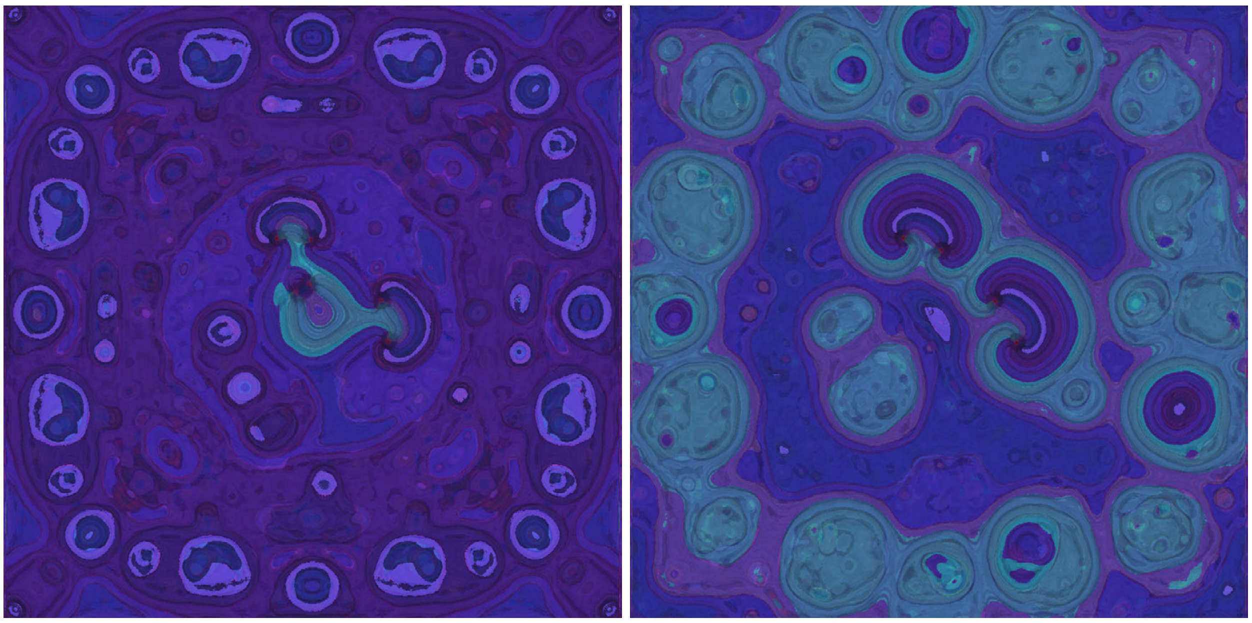


Figure 50: Texture Generator output, iteration no. 100 and 200.

4.4 Texturing landscapes

An interesting use-case of the generator is texturing procedural landscapes. By feeding the complementary data from the landscape mesh like height, ambient occlusion and curvature maps as layer initialization states in the generator, the resulting colour values conform to the landscape structure in meaningful ways. We stop the simulation after a small number of iterations to avoid over-saturation. To control the generated colours, we remap the resulting value vectors using a 3D lookup table with a desired colour palette. Results of this method are shown in Figures 51, 52, 53 and 54.

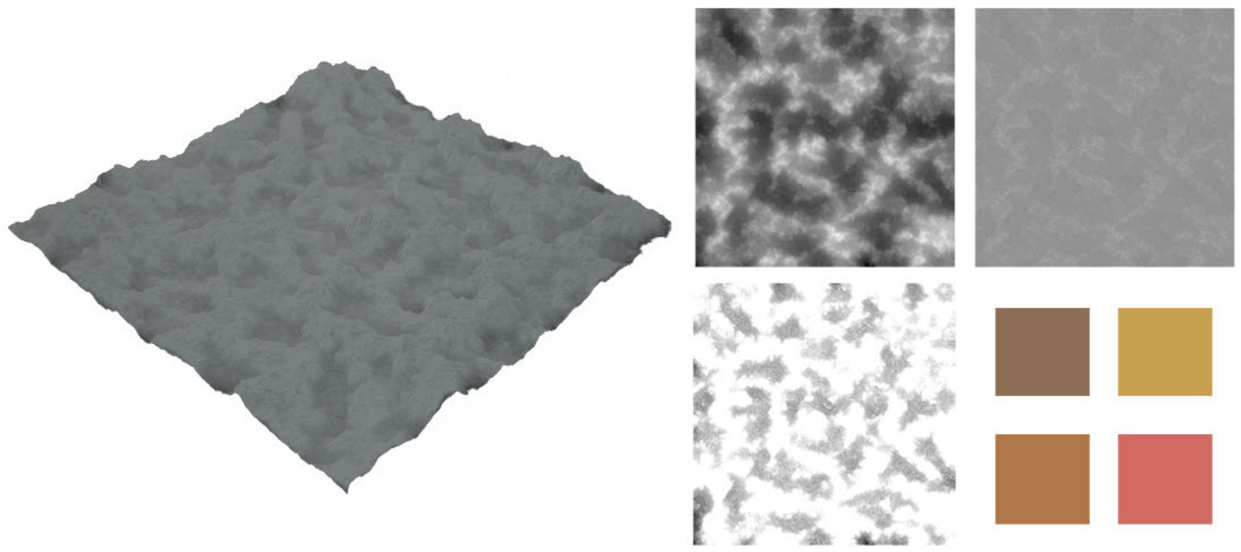


Figure 51: Original landscape, height, curvature, ambient occlusion maps and the colour palette.

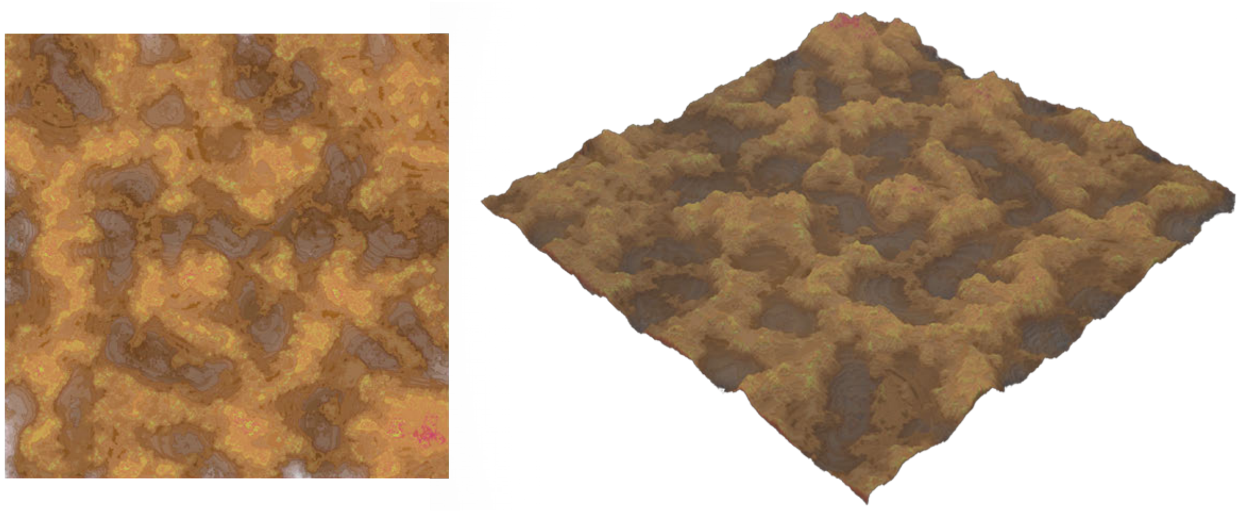


Figure 52: Left: generated texture, right: textured landscape.

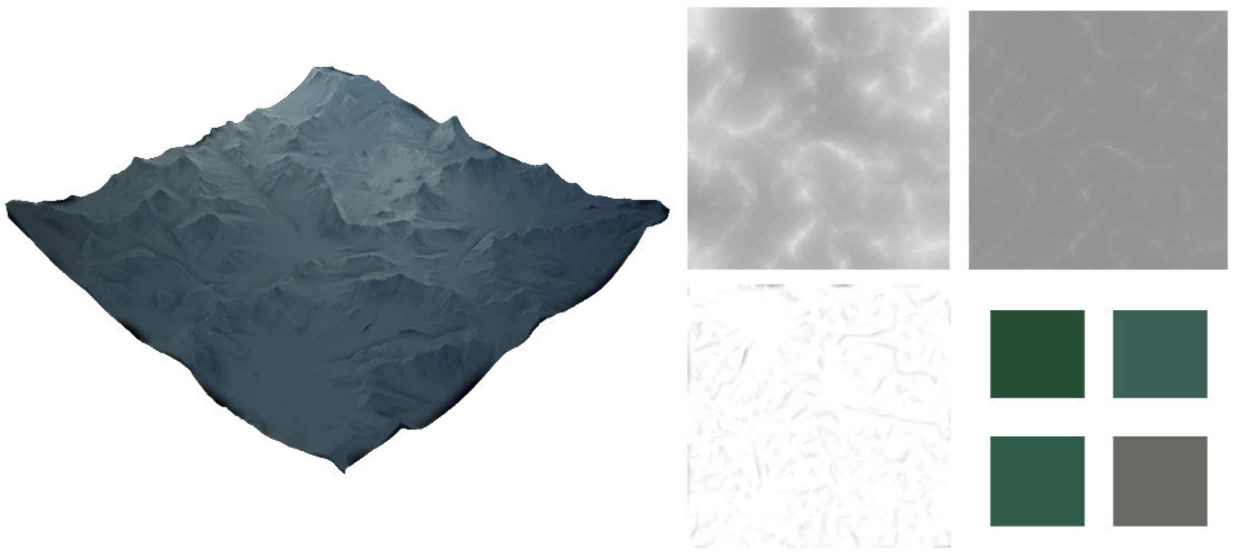


Figure 53: Original landscape, height, curvature, ambient occlusion maps and the colour palette.

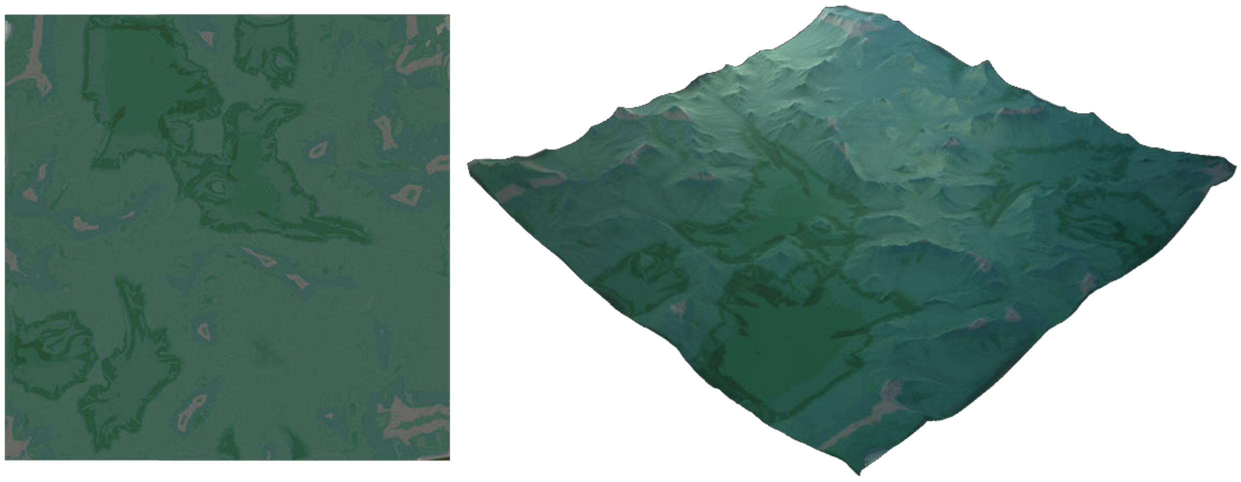


Figure 54: Left: generated texture, right: textured landscape.

5. DISCOVERING RULES THAT PRODUCE GIVEN STRUCTURES

5.1 About the procedure

Given an initial voxel structure we have to find a rule that when applied several times produces a close representation of a desired structure. The *goal structure* could be made by hand, procedurally generated or -optimally- a result of a cellular automaton. The automaton used has two states, alive or dead. The rules are limited to conventional totalistic automata rules. For our purpose, alive cells represent the solid voxels that our goal structure is comprised of.

$$\text{Initial Structure } \overset{\text{Rule genome } \times n}{\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow\rightarrow} \text{ Goal Structure}$$

If we unfold the 3D structures to a 1D array of ones and zeros, our goal is to minimize the *Hamming metric* of the End and Goal structures. Alternatively, the count of all integer 3D coordinate vectors in the simulation bounding box, whose corresponding Initial and Goal Structure cell states differ. An example calculation of the Hamming metric between two binary strings is shown in Figure 55.

```
 00110010101110
 01110000100110
  +1    +1    +1      = 3
```

Figure 55: Hamming metric calculation between two binary strings.

Possible applications of the above procedure include: compressing huge organic voxel data to a single number when a certain amount of loss is tolerated, searching for life-like rules, reverse engineering cellular automata, as well as finding equal rules for given structures and cryptography.

5.2 The genetic algorithm

A *genetic algorithm* mimics real life evolution to efficiently solve an *optimization problem*. An initial set of candidate solutions are evolved and altered towards better ones. A solution is defined by a string of properties (in our case the automaton rule) called *genome*. The evolution is an iterative process, every step of which is called a *generation*. In every generation the candidate solutions (later referred to as individuals) have their genomes altered and recombined through the process of mutation and crossover. Lastly their ability to solve the problem (fitness) is measured with an objective function. The fittest individuals are then selected through various means and are carried to the next generation. The algorithm terminates after a maximum number of iterations or -ideally- when a desired fitness value is reached for an individual in the population.

5.2.1 Genome

The *genome* is a binary representation of our rule-solution. All two-state totalistic automata rules with 26 neighbours can be represented with a binary string of 2×26 length:

01000010010001000111011101 00101101011000000011000110

The space separates the survival and birth requirements. A “1” bit in position i where $i < 26$, denotes that an alive cell may survive with $i + 1$ alive neighbours, while a bit in position *j* where $j > 26$, denotes birth when neighbour count equals $j + 1$. Using this representation we only need 52 bits per genome, that fit in a 64 bit integer. Note that the possible combinations are 2^{52}, making it impossible to exhaust the search space with a greedy algorithm.

If we wished to take into account all possible neighbour state combinations and not just their count, we would need to store a genome of 2^{27} bits. In the evaluation process the state lookup would return a 27 bit binary, and the corresponding integer would serve as an index to the genome binary array that denotes whether the cell is alive or dead in the next step. Not only it is computationally implausible, but the rule of thumb in population size is at least twice the genome length, so we would quickly run out of space.

5.2.2 Initialization

We used a population size of 2×52 with random genomes. The genome randomization was biased towards "0", with a probability of 0.7, as conventional automata rules are rarely described with more than 6 alive neighbour counts.

5.2.3 Fitness evaluation

The Hamming distance method described above, misleads the population towards locally optimal rules that produce all solid/dead cells. e.g. if the desired structure is a singe solid cell a quick solution will come up that kills every cell of the initial structure resulting in a Hamming distance of 1, but 100 % error when solid cells are concerned. Instead, we will make use of a formula that takes into account the proportion of solid and air states in the goal structure:

$$fitness = hits_{solid} \times \frac{1}{Psolid} + hits_{air} \times \frac{1}{Pair}$$

For each individual in the population, a cellular automaton is evolved several times from its genome. For every step of the automaton, the state of the grid is compared against the goal structure. A secondary fitness value is checked when two genomes have equal fitness:

$$\frac{1.0}{AutomataStepsToSolution + 1} + \frac{1.0}{CountOfOnesInGenome + 1}$$

to ensure that the shortest solution is found. The automaton stops after *maxIterations* (about 10 in our tests) iterations and the maximum fitness is returned.

The fitness function could be modified to support multi state life-based automata. In this automata a cell doesn't immediately die but loses one life point and isn't considered alive any more. A naive implementation of finding an initial life amount that explains the goal structure would take *MaxLifeSearched* more time. Although starting from a single life point going upwards, if the relative fitness of two consecutive iterations is negative we could safely stop.

5.2.4 Selection

For the selection process we used a combination of *elitism* and *fitness proportionate selection*. This ensured that the fittest genomes were carried through each generation and that low fitness genomes with possibly useful segments persevered. Ensuring genetic diversity is paramount to reaching an optimal solution. Due to the nature of the problem, rule genomes a single bit away from an optimal solution may have very low fitness. Genomes with high fitness, although admissible on their own, may become stagnant. In fitness proportionate selection the probability of each individual being selected is

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

where N denotes the population size and f_i the fitness of individual i in the population.

The probability function we used for individual selection had a denser distribution around 1.0. Although unconventional, it proved extremely efficient. Lastly, the elite 0.5% remain intact after the crossover process, in fear of losing them to an unfortunate pairing and boosting their chances of survival in the mutation process.

5.2.5 Crossover

The selected individuals are then bred using the crossover genetic operator. The *crossover operator* combines the genomes of two or three parents, generally by swapping random bit segments. A meaningful way to produce an offspring in the context of our problem, is to have it inherit the survival rule bits from the first parent and the birth bits from the second parent. We tested several crossover methods and this proved more effective, although slightly. The probability to crossover was set to 0.8. The remaining individuals are copied as they are.

5.2.6 Mutation

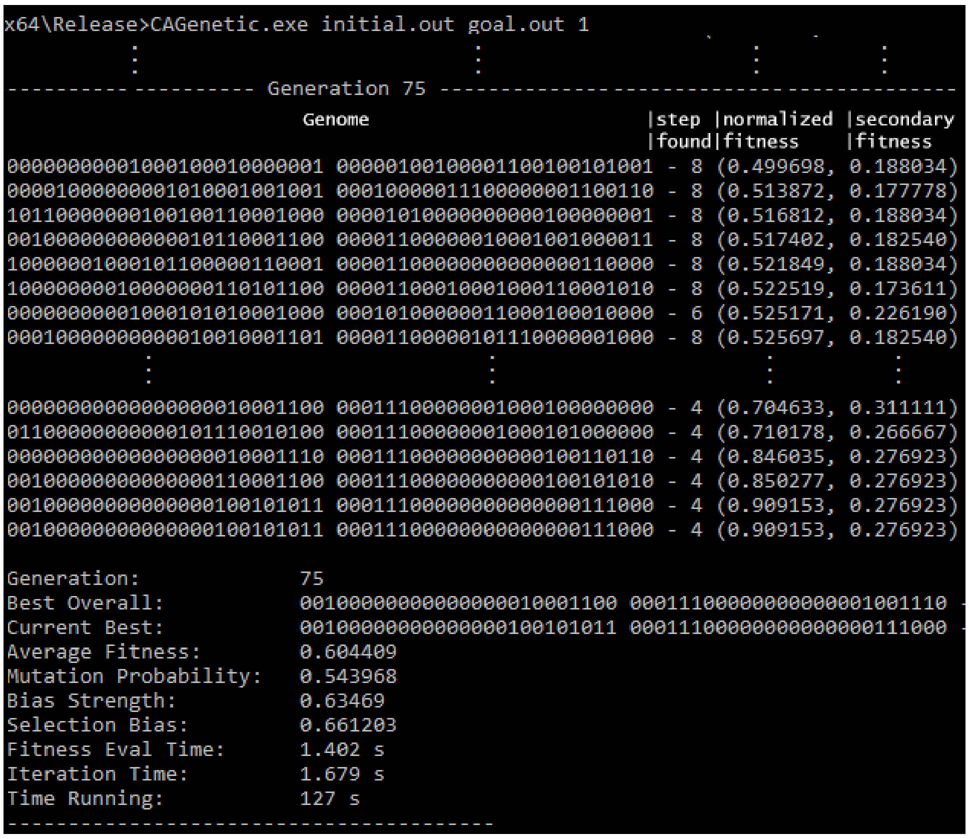
The *mutation operator* used, when applied to a genome, may transform each bit from 0 to 1, or 1 to 0 with a 10% and 20% probability respectively. The greater probability of 1 to 0 bit mutation, again serves as a guide to shorter rules. A probability of $0.9 \times averageFitness$ was used to select an individual for mutation. The mutation probability is proportionate to the average fitness of the population, to counteract elites with stagnant genomes in highly evolved generations. The remaining individuals are copied as they are to the next generation, along with the ones mutated.

5.3 Results

The above process typically converges to an optimal solution in less than 100 generations, with fitness values reaching 1.0 (100% likeness) for automata produced structures that are not life-like. The solution almost always is the shortest rule used for the production of the goal structure. For life-like automata with long periods (> 5) the process doesn't work so well, missing an exact solution half of the time. When the desired structure was made by hand or generated by other means, the algorithm aims to minimize the distance between the goal structure. In intricate shapes, the divergence goes up to 10%, which has a large visual impact, although distances as low as 0.5% were achieved in large organic masses. This method was used to find conventional notation for abstract rules. For example the simple rule $\frac{\sum_{i=1}^{26} state_i}{26} > c$, $\frac{1}{2} < c < \frac{2}{3}$ used for clustering alive cells in 3D space, resulted in the following genome:

$$00000000000001111111111111\ 00000000000001111111111111$$

Information provided for each generation by the command line application we developed is shown in Figure 56.



```
x64\Release>CAGenetic.exe initial.out goal.out 1
           :                            :                      :          :
--------------------- Generation 75 ------------------------------------------
                         Genome                         |step |normalized |secondary
                                                        |found|fitness    |fitness
0000000000100010001000001 0000010010000110010010101001 - 8 (0.499698, 0.188034)
0000100000000101000100100 1000100000111000000001100110 - 8 (0.513872, 0.177778)
1011000000001001001100010 0000001010000000000100000001 - 8 (0.516812, 0.188034)
0010000000000001011000110 0000011000000100010010000011 - 8 (0.517402, 0.182540)
1000000100010110000011000 1000001100000000000000110000 - 8 (0.521849, 0.188034)
1000000001000000011010110 0000001100010001000110001010 - 8 (0.522519, 0.173611)
0000000000100010101000100 0000101000000110001000010000 - 6 (0.525171, 0.226190)
0001000000000001001000110 1000011000001011100000001000 - 8 (0.525697, 0.182540)
            :                            :                      :          :

0000000000000000000100011 0000111000000010001000000000 - 4 (0.704633, 0.311111)
0110000000000010111001010 0000111000000010001010000000 - 4 (0.710178, 0.266667)
0000000000000000000100011 1000111000000000000100110110 - 4 (0.846035, 0.276923)
0010000000000000011000110 0000111000000000000100101010 - 4 (0.850277, 0.276923)
0010000000000000010010101 1000111000000000000000111000 - 4 (0.909153, 0.276923)
0010000000000000010010101 1000111000000000000000111000 - 4 (0.909153, 0.276923)

Generation:               75
Best Overall:             0010000000000000000100011 00 0001110000000000000001001110 -
Current Best:             0010000000000000010010101 1 0001110000000000000000111000 -
Average Fitness:          0.604409
Mutation Probability:     0.543968
Bias Strength:            0.63469
Selection Bias:           0.661203
Fitness Eval Time:        1.402 s
Iteration Time:           1.679 s
Time Running:             127 s
----------------------------------------
```

Figure 56: Generation 75 statistics. Genomes are sorted by ascending fitness.

5.4 Time, Complexity, Benchmarks

The time complexity is:

$$Generations \times Population \times CAStepsSearched \times \frac{CASpaceSize \times FitnessEval}{Cores}$$

The fitness evaluation runs exclusively on the graphics card and takes about 1 second to complete per generation for the settings described above and a 100^3 goal structure. The C++ implementation although viable for small values, doesn't scale well with simulation size.

The space complexity scales with population size only.

6. FRACTAL CELLULAR AUTOMATA

In this section we investigate the relationship between CA and fractals and discuss some CA-based methods that are able to generate them. A *fractal* is a set that exhibits a repeating pattern when observed at every scale. The capacity of CA to generate fractals was examined by Wolfram [22], Culik and Dube [5], Willson [21] and Martin [16]. A more formal definition from Mandelbrot as presented by Martin is the following.

A set X is called a *fractal* provided its *Hausdorff dimension* $h(X)$ is not an integer. Intuitively, $h(X)$ measures the growth of the number of sets of diameter ε needed to cover X when $\varepsilon \to O$. More precisely, if $X \subset \mathbb{R}^m$, let $N(\varepsilon)$ be the minimum number of m-dimensional balls of diameter ε needed to cover X. Then, if $N(\varepsilon)$ increases like $N(\varepsilon) \to \varepsilon^{-d}$ as $\varepsilon \to 0$, one says that X has *Hausdorff dimension d*.

Most of the investigation for fractal generation by CA was centered around *linear cellular automata* (LCA). LCA as defined by Wilson: We denote by P^n the set of all configurations of a Q-state's n-dimensional cellular automaton. Thus, $P^n = Q^{\mathbb{Z}^n}$. We define a *global dynamics G* on P^n as follows.

A global transition function G on the set of all the configurations P^n is a map $G\colon P^n \to P^n$ such that

- there exists a quiescent state
- there exist m neighbours $(V_i)_{i=1,...,m} \in \mathbb{Z}^n$ and a map $g\colon Q^m \to Q$ such that $\forall v \in \mathbb{Z}^n \forall \omega \in P^n$,

$$G(\omega(V)) = g(\omega(v+v_1),...,\omega(v+v_m)).$$

Then, the transition rule G on P^n is *linear* provided its *generating function g* is linear or, equivalently, provided

$$G(\omega + \tau) = G(\omega) + G(\tau).$$

Culik et al. [5] have shown that the regular evolution of linear cellular automata on simple initial configurations generates a pattern that might be fractal or self-similar. The patterns they obtain are often similar to Pascal's triangle, like the ones shown in the next section.

6.1 Self similarity in simple rules

There are examples of simple one-dimensional CA rules that present self similarity in a given interval, when visualized with time as an added dimension. One-dimensional automaton rule 90, shown in Figure 58, has this property. This rule generates the famous *Sierpiński triangle* when the grid is initialized with a single alive (state 1) cell. Its rule table, seen in Figure 57, corresponds to the binary number $01011010_2 = 90$ in *Wolfram rule notation* [22]. We observe that rule 90 is easily computable using the formula $(Neighbour_{left} + Neihgbour_{right})$ mod 2 which is also the *XOR operation*.

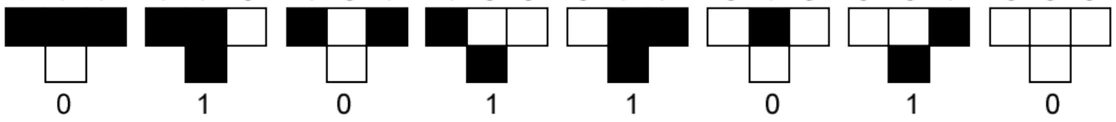


Figure 57: The rule table of one-dimensional CA rule 90.

With the initialisation of the grid treated as the first iteration, we obtain $n+1$ levels of detail for the triangle pattern for every 2^n iterations of the rule. This behaviour is shown in detail in Figure 59. 1 (2^0), 2 (2^1), 4(2^2), 8 (2^3), 16 (2^4), 32 (2^5) iterations generate 1, 2, 3, 4, 5

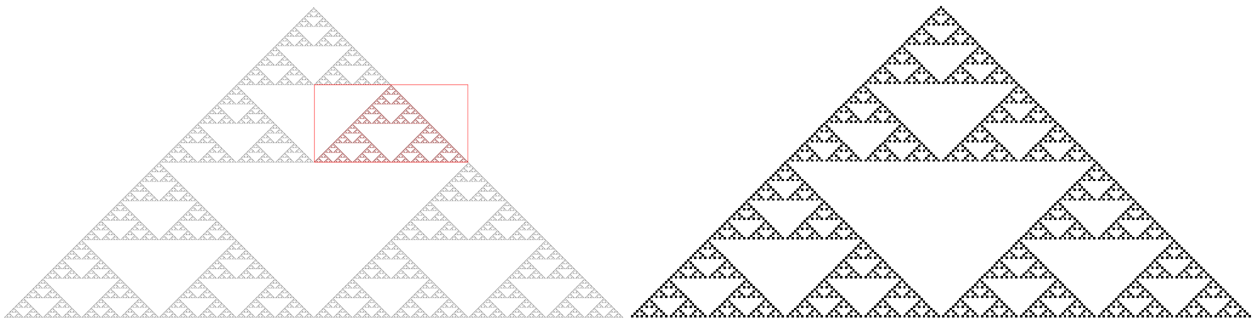


Figure 58: On the left, 511 iterations of one-dimensional CA rule 90 starting from a single alive cell. The pattern enclosed in the red rectangle is shown, enlarged, on the right.

and 6 levels of detail respectively. The cells reside in a triangular grid and are scaled to fill the same amount of space.

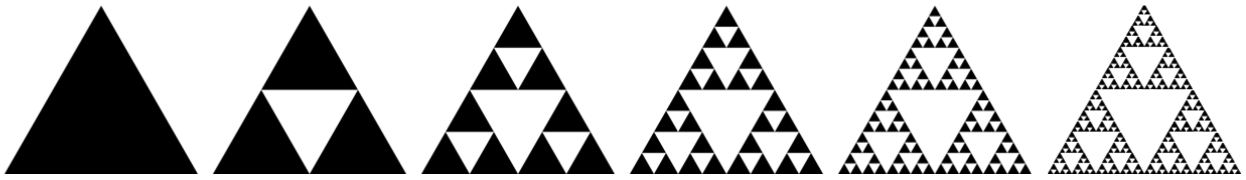


Figure 59: Sierpiński triangle construction with rule 90 in a triangular grid. From left to right, 1, 2, 4, 8, 16, 32 **iterations.**

Sierpiński-like structures are very common in one-dimensional automata. As seen in Figure 60 rules 102 and 182, among many others, are capable of forming such structures.

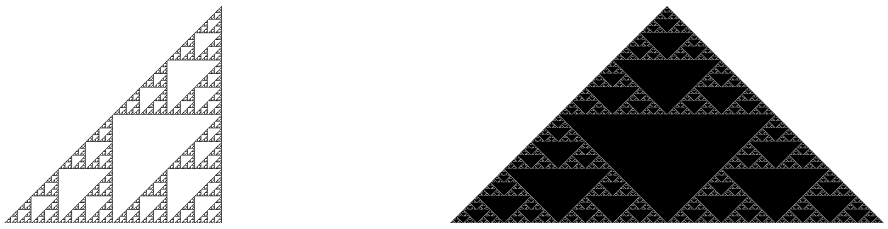


Figure 60: One-dimensional CA rules that generate a Sierpiński triangle. Left: rule 102, right: rule 182.

Culik et al. [5] provided proof that for any arbitrary radius *r*, the CA with XOR rule always generates a regular pattern, namely when started on a string ω whose length *n* is a power of 2 (any arbitrary initial finite string can be made to have a length which is a power of 2 by appending appropriate number of 0s), after n time steps the configuration is ω^{2r+1}. The XOR code for radius r is $2(4^{r+1} - 1)/3$. The following lemma by Culik [5], which can be proved by induction, explains why *XOR* CA generate perfect fractal patterns on an initial seed containing a single *l*.

Let *f* be the *XOR* CA rule for radius *r*. Then, when started on the configuration consisting of a single 1, for any *n* where *n* is a power of 2, we obtain after *n* time steps the configuration which is $2r + 1$ repetitions of the string formed by a 1 followed by $n - 1$ 0s, i.e., $G_f^n(1) = (10^{n-1})^{2r+1}$.

Culik [5] also came to the important conclusion that this regular evolution allows one to compute the value of a cell after arbitrary number of time steps in a more efficient way than to actually run the CA.

6.2 Generation of a Cantor set

Martin [16], defines the behaviour of a one-dimensional cellular automaton that embeds a Cantor set in the closed interval $H = [0, 1]$ of $\mathbb{R}$. The definition proceeds as follows.

Let $A = (Q, \delta)$ be a one-dimensional cellular automaton with states $Q = \{a, 1, q\}$ (where q denotes the quiescent state) and some other auxiliary states. Let δ be the local transition function (or generating function) for the cellular automaton. The process consists of steps starting from the initial configuration 010, or equivalently from the "white-black-white" configuration. The steps follow and are depicted in Figure 61.

1. Copy the configuration once to the right.
2. Copy the configuration a second time at the right end and paint the first copy in black.
3. Update the configuration and return to step 1.

Martin concluded that such a cellular automaton may be constructed. The duplication of a finite configuration can be done using a shift transition function as described in detail in [16].

Hence, by iterating that process and embedding the restriction to $\{0, 1\}$ of a subsequence of the configurations, we get the geometrical construction of a Cantor set. The total time between two "embeddable" configurations is thus $(7 \times$ *length*(finite configuration)$) - 2$ plus the time to come back, which is *length*(new finite configuration). These configurations occur at times given by the exponential sequence 1, 28, 278, ..., or $t_0 = 1$, $t_{n+1} = 10t_n - 2$.

The "drawing" of the Cantor set is then obtained by replacing in the subsequence all the occurrences of "0" by a segment and all the occurrences of "1" by a hole, then resizing the configurations in $[0, 1]$. The Cantor set is then the set of sites with symbol "0". This behaviour is illustrated in Figure 61.

Martin [16] showed that the process defined above can be generalized to higher-dimensional cellular spaces. In order to do that we first notice that the unidimensional cellular automaton described in the previous section can be modified as follows: instead of twice copying the initial configuration at the right end, it is also possible to copy it once at the left end and once at the right end. In other words, use the neighbourhood vector of the cellular automaton to identify the part that is copied and has to be painted in black. With that modification, the cellular automaton expands to the left and to the right, without changing the continuous representation of the configuration. The point of interest of this definition is that the copies are made by using the neighbourhood vector and might be generalized by means of this strategy.

6.3 Sierpiński carpet

Martin [16] explains that in the case of the Moore neighbourhood, it suffices to synchronize a line before copying (see Figure 62). As en example he synchronizes the left edge of the square. The synchronizing process then allows the synchronized line of cells to behave as if it were a unique cell in a unidimensional cellular automaton. Then, he applies the process depicted in Figure 62, which twice copies the initial finite configuration; in this case, however, we do not paint black (that is, rename the remaining 0s of the middle third in 1s) the middle third sub-configuration. In the case of a cellular automaton that, for instance, emits the values of a line, it does not matter whether the crossing lines are synchronized. In the case of Martin's process, we just have to obtain the first quiescent line and stop as soon as it is encountered, then copy one line into one of the directions of the neighbourhood. The return signal can be sent only (to the general) for the synchronization process. We then synchronize again in the other direction- that is, on the largest square-

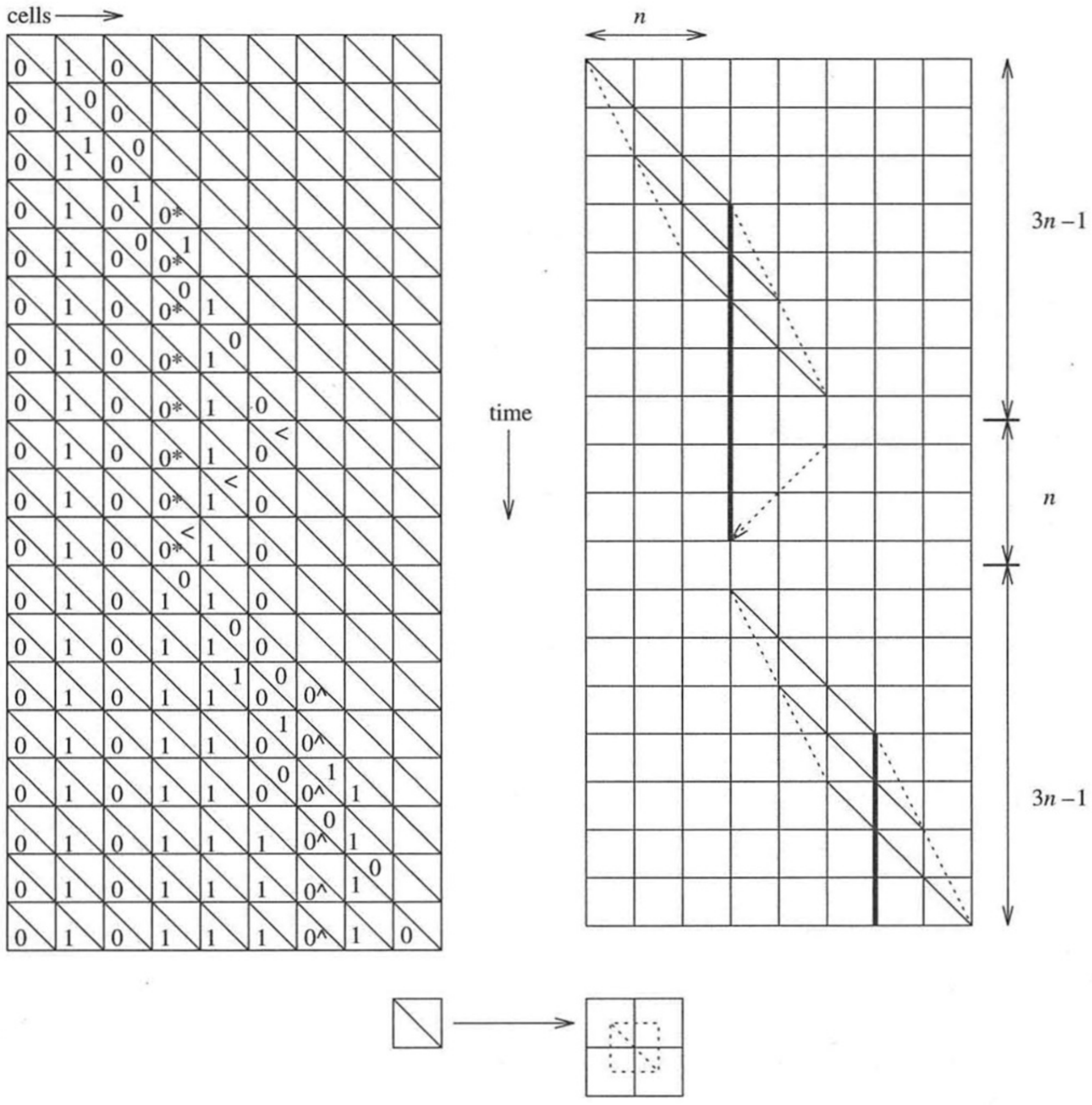


Figure 61: The two steps of behaviour: time-space states diagram and time-space diagram, from Martin [16].

and likewise copy twice the configuration with the layer containing the square to be painted black. Finally, when the copying process is over, the corners send two signals, one of slope 1 at half speed and one of slope $1/2$ at full speed, in order to choose which square has to be painted black. The black colour of the layer then enters the main state and paints black the middle third sub-configuration. Figure 63 depicts this action.

To summarize Martin's process:

1. synchronize the configuration over the *x* axis.
2. apply the copying process twice along the y axis.
3. synchronize along the y axis.
4. apply the copying process twice along the *x* axis.
5. identify the four corners of the new configuration.
6. identify the middle third sub-square and "paint it black".

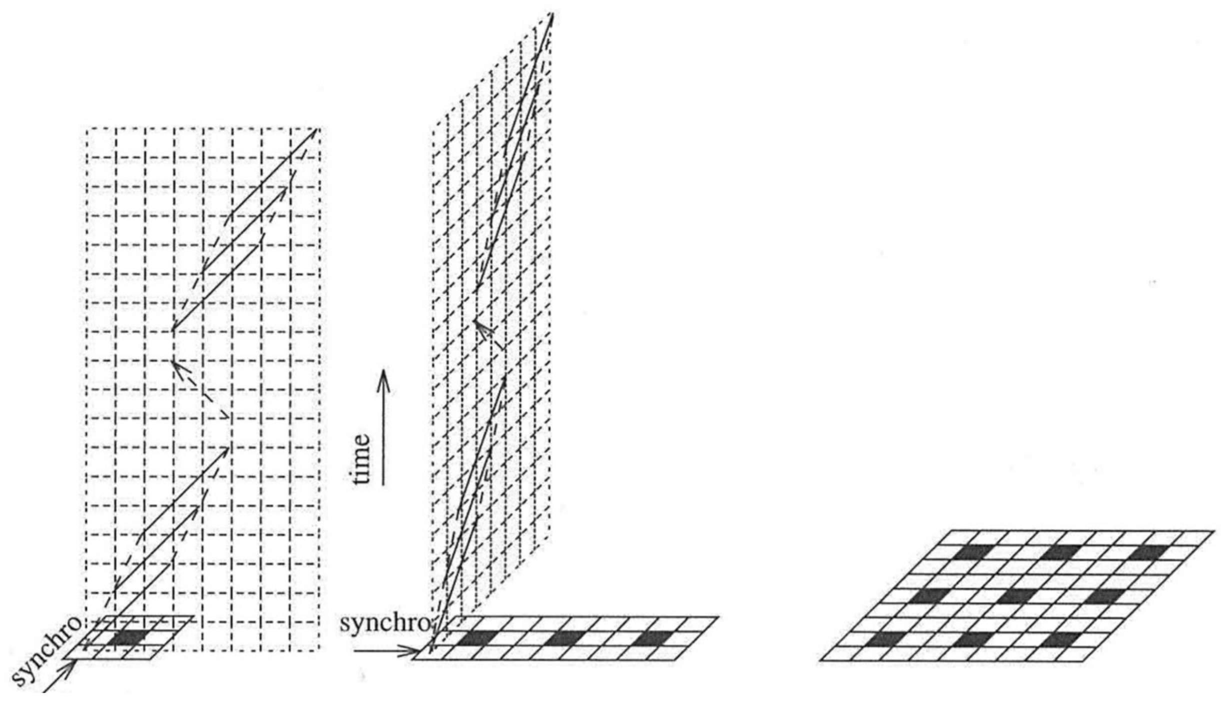


Figure 62: Copying the squares nine times: time-space diagram, from Martin [16].

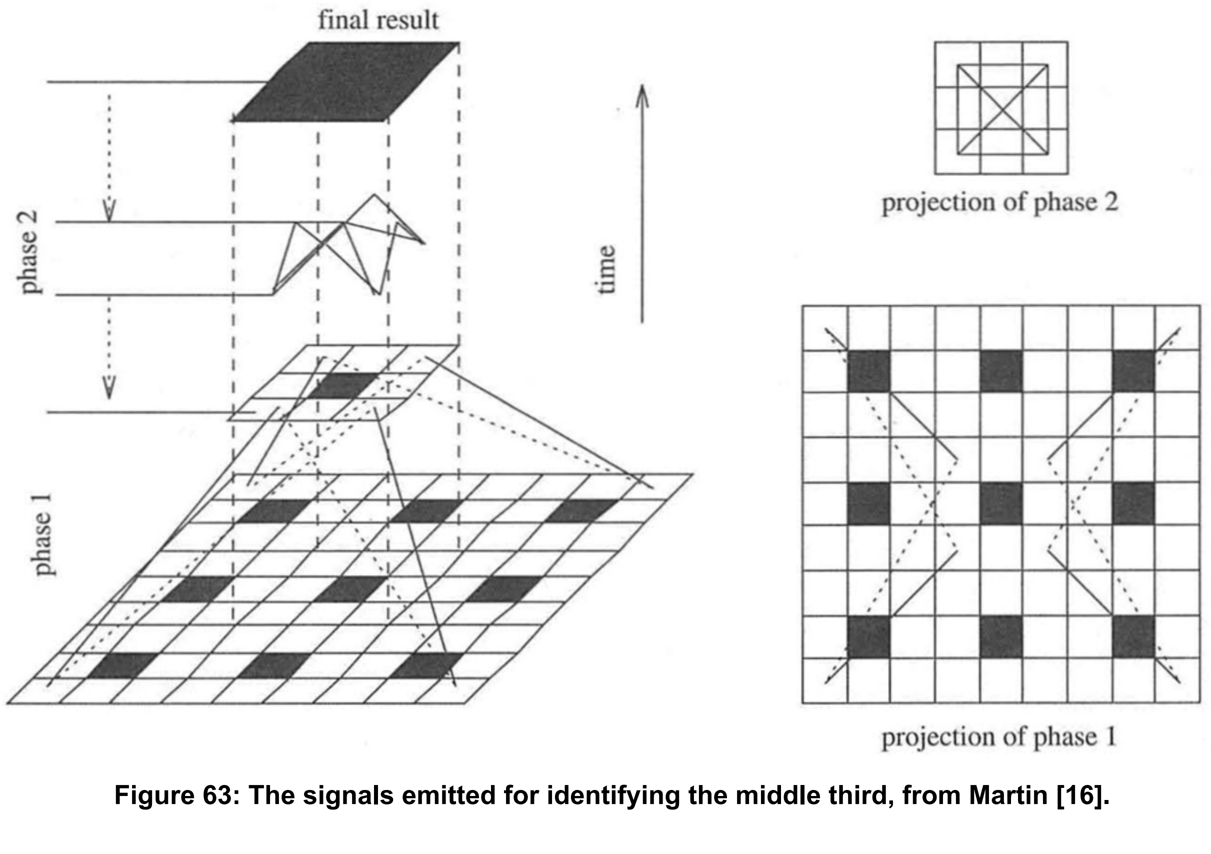


Figure 63: The signals emitted for identifying the middle third, from Martin [16].

In the case of the Moore neighbourhood, we get the promised Sierpiński carpet when embedding the configurations at exponential units of time. Martin gave justification for the computation of the Hausdorff dimension in discrete space for the two-dimensional cellular automaton that computes the Sierpiński carpet depicted. The process for drawing the Sierpiński carpet on the mesh is the one suggested in Section 5.2. He then presents the following theorem and it's proof.

Theorem 1. The number of white points of the configuration of the two-dimensional cellular automaton with the Moore neighbourhood corresponds exactly to the minimal covering of the Cantor set for balls of a diameter that corresponds to the homothetical factor. Furthermore, the Hausdorff dimension of its embedding is that of a Sierpiński carpet, namely $log8/log3$.

Proof 1. The covering of the initial configuration of the cellular automaton gives in $H^2 = [0,1] \times [0,1]$ a number of balls $N(\varepsilon) = 8$ with diameter $\varepsilon = 3^{-1}$, and in the configuration 8 white points with an homothetical ratio of $1/3$, which corresponds to the situation in H^2 and starts the induction. For the induction step, assume we have covered the nth configuration with 8^n balls having the same number of white points in the configuration. The area of the $(n+1)$th configuration is clearly $(3n+1)^2$, and contains (because of the duplications) 8×8^n white points with the same number of balls in its embedding, that is, 8^{n+1} balls. The rest of the nonquiescent part of the configuration is painted black.

Because $N_n(\varepsilon) = 8^n$ and $\varepsilon_n = 3^{-n}$, we get the equality $8^n = (3^{-n})^d$, which gives the dimension d of the theorem.

6.4 Sierpiński-Menger sponge

Martin [16] also showed that this process is analogous for a three-dimensional cellular automaton with the generalization of the Moore neighbourhood, where we obtain the well-known Sierpiński-Menger sponge depicted in Figure 64. Also the usual theorem holds for three-dimensional cellular automata.

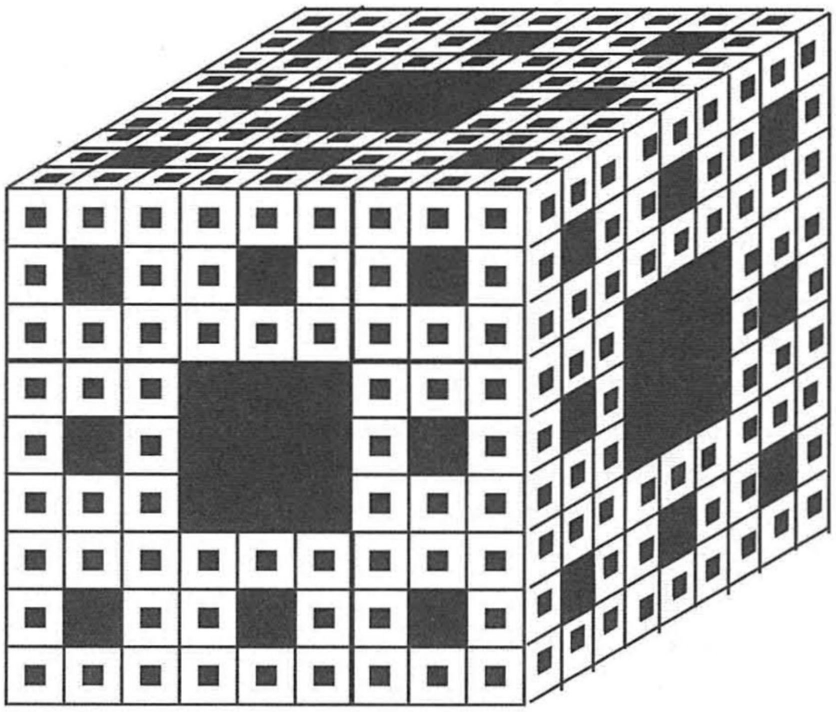


Figure 64: The Sierpiński-Menger sponge, from Martin [16].

Theorem 2. The number of white points of the configuration of the cellular automaton corresponds exactly to the minimal covering of the Sierpiński-Menger sponge for balls that correspond to the homothetical factor. Furthermore, the Hausdorff dimension of its embedding is that of the Sierpiński-Menger sponge, $\log 26/\log 3$.

One can also determine the dynamical system that counts the proportion of black points of the non quiescent part of the configuration, which is $\prod_n^{\bullet} = \frac{26}{27}\prod_{n-1}^{\bullet} + \frac{1}{27}$. The proportion of white points is denoted by $\prod_n = 1 - \prod_n^{\bullet}$.

6.5 CA-generated fractal noise

For more practical use-cases, a common procedure for producing fractal noise for use in landscape formation or texturing, is scaling and rotating a noise texture and blending it together. We used the height displacement rule discussed in Section 2.1 as a source for the noise pattern. Afterwards, we blend the pixels of each variation using the formula $a \cdot (1 - b) + b \cdot (1 - a)$, where a and b is the luminance of the two pixels. The successive blends can be seen in Figure 65 and the final result in Figure 66.

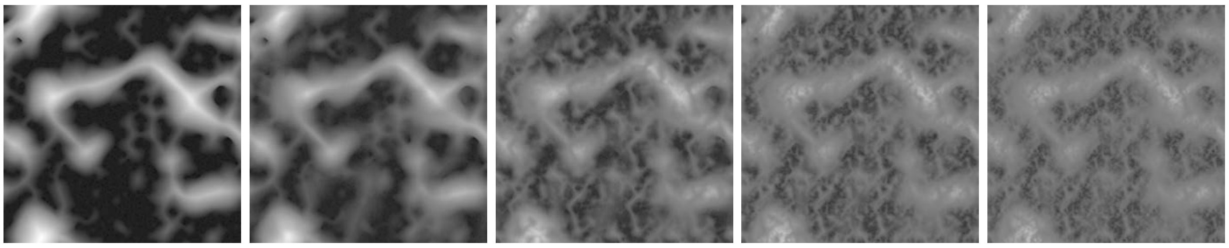


Figure 65: Successive blending of grey-scale patterns to produce fractal noise.

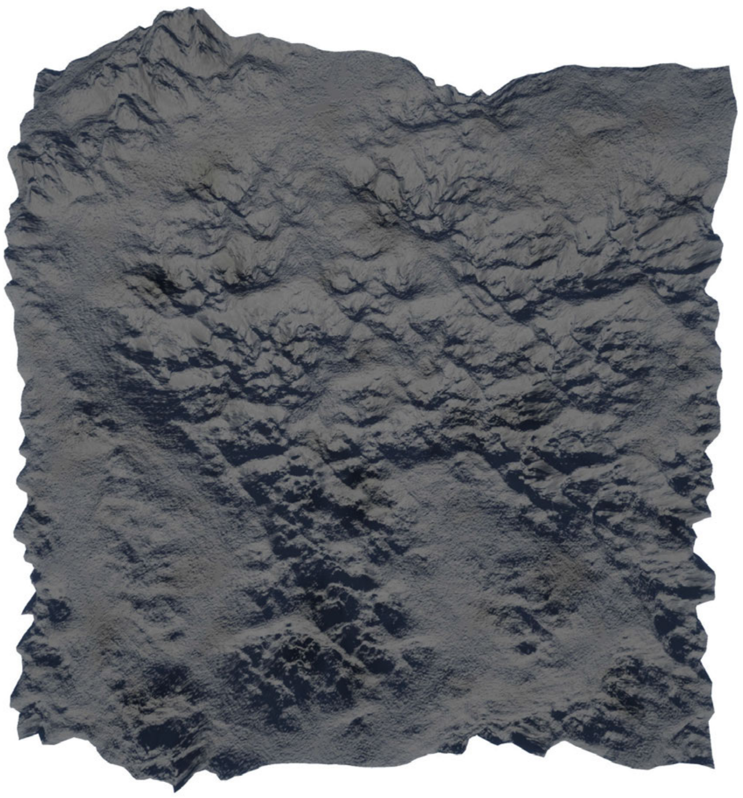


Figure 66: Top-down render of the final heightmap in Figure 65 inside Unreal Engine 4.

7. CONCLUSIONS

The purpose of this thesis is to serve as a comprehensive introduction to 3D cellular automata techniques aiding the creation of real-time computer graphics and game levels. The ones presented in Chapter 2, to the best of our knowledge, are new approaches to procedural generation, which is not surprising given the inherent limitations and unpredictability of CA. By focusing our investigation on simple rules we discovered quite a few useful configurations with the capacity to generate believable mountainous formations, cave systems, stalagmites and other intricate natural or physical structures, while being easily parameterised and combined. We also discussed techniques for effectively using the generative product of automata in a graphics engine.

Although the processing overhead and fidelity attained is satisfactory in a real-time graphics application context, given the alternatives, mainly noise functions and lightweight physics simulations, that allow greater flexibility and resolution, there is not much incentive for adopting these techniques yet. Our work in this chapter was not to no effect, as we showed that simple 3D automata rules are capable of producing a variety of structures some of which require a large stack of noise functions to reproduce, if at all. Additionally, we presented examples where 3D automata generated fairly complicated noise textures for use in texturing and particle effects. By extending our investigation to more complicated rules or hybrid CA and noise techniques for greater control over the generated voxels we could possibly cover structures that fractal noise cannot generate.

The CA based procedural placement algorithms presented in Section 3.2 can be implemented in a straightforward way inside a game engine. The foliage spread model along with artist authored guides, should be a useful tool for organically populating procedural landscapes. Furthermore, the cave structure generation method discussed in Section 2.1 provides a believable base mesh for the authoring of realistic cave levels.

The CA texture generator is an experimental tool that showcases the diverse images CA are capable of producing and aids in the procedural texturing of landscapes. Using high quality maps as seeds and artist authored colour pallets, it yields interesting results for stylized and realistic terrains.

Lastly, we introduced a genetic algorithm that searches for an automaton rule that connects two given structures. When evaluating the discovered rule on the initial structure, the relative Hamming distance ranges from 0% to 10%, with no loss of information achieved with totalistic automata produced structures that are not life-like.

INDEX

ABBREVIATIONS

CA	Cellular Automaton/a
GUI	Graphical User Interface
API	Application Programming Interface
CPU	Central Processing Unit
GPU	Graphics Processing Unit
CGI	Computer-Generated Imagery
LUT	Lookup Table
HSV	Hue Saturation Value

APPENDIX I: THE CA TOOLKIT

Several applications were developed for the purpose of this thesis. The most comprehensive being the “CA Toolkit”, which generated and rendered most of what depicted in previous figures. The applications capabilities include:

- Multiple primitive initialization functions with variable state count and distribution.
- Real time rendering of millions of voxels with multiple visualization modes.
- Real time surface extraction of CA data.
- Real time 3D automaton rule evaluation through scripting or graphical interface.
- Multiple volume boolean operations.
- Export simulation state in OBJ, native or heightmap format.
- Generate and export population data in a native format or splatmaps.

The CA Toolkit was developed on C++, OpenGL 4.3 and runs almost entirely on the GPU. As a result, all processing logic is written in *compute shaders*. The compute shader is dispatched through each cell, evaluating the new states, culling of adjacent voxels, material indices, and prepares an optimal buffer for the new voxels. The CPU handles input and initializations and makes a single instancing drawcall for the voxels or binds the vertex array for the polygon data.

In Figure 67, we see the rendering performance per voxels in the simulation. Engine overhead is included in the measurement. In Figure 68, we show the compute time per frame using the Cave rule from Subsection 2.1. With our compute method, processing time increases linearly in relation to the voxel count. Evaluating the future state of a billion voxels with a search radius of 1, requires half a second of processing time.

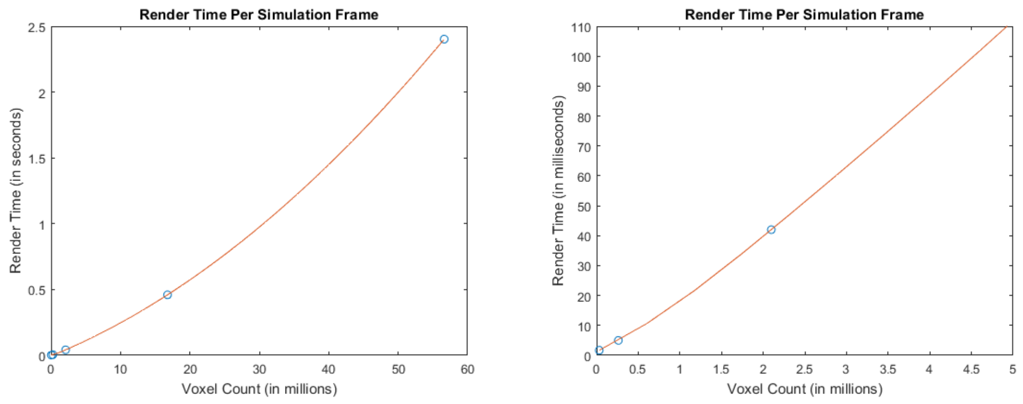


Figure 67: Time consumed on rendering each frame.

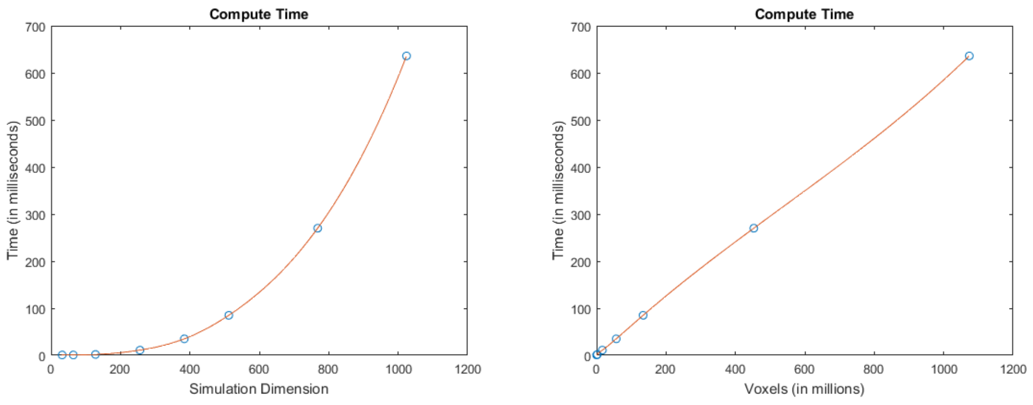


Figure 68: Time consumed on computing each frame.

REFERENCES

[1] Andrew Adamatzky and Genaro J Martínez. *Designing Beauty: The Art of Cellular Automata*. Vol. 20. Springer, 2016.

[2] Alonso-Sanz and Martín. “Elementary cellular automata with memory”. *Complex Systems* 14.2 (2003), pp. 99–126.

[3] Bays. “A note about the discovery of many new rules for the game of three-dimensional life”. *Complex Systems* 16.4 (2006), p. 381.

[4] Conway. “The game of life”. *Scientific American* 223.4 (1970), p. 4.

[5] Culik and Dube. “Fractal and recurrent behavior of cellular automata”. *Complex Systems* 3.3 (1989), pp. 253–267.

[6] RL Dobrushin, VI K͡riukov and AL Toom. *Stochastic cellular systems: ergodicity, memory, morphogenesis*. Manchester University Press, 1990.

[7] Fisch. “Cyclic cellular automata and related processes”. *Physica D: Nonlinear Phenomena* 45.1-3 (1990), pp. 19–25.

[8] Gobron and Chiba. “3D surface cellular automata and their applications”. *The Journal of Visualization and Computer Animation* 10.3 (1999), pp. 143–158.

[9] Johnson, Yannakakis and Togelius. “Cellular automata for real-time generation of infinite cave levels”. *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM. 2010, p. 10.

[10] Ju, Losasso, Schaefer and Warren. “Dual contouring of hermite data”. *ACM Transactions on Graphics (TOG)*. Vol. 21. 3. ACM. 2002, pp. 339–346.

[11] Laine and Karras. “Efficient sparse voxel octrees”. *IEEE Transactions on Visualization and Computer Graphics* 17.8 (2011), pp. 1048–1059.

[12] Lempitsky. “Surface extraction from binary volumes with higher-order smoothness”. *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*. IEEE. 2010, pp. 1197–1204.

[13] Lorensen and Cline. “Marching cubes: A high resolution 3D surface construction algorithm”. *ACM siggraph computer graphics*. Vol. 21. 4. ACM. 1987, pp. 163–169.

[14] Ludwig. “Image Convolution”. *Satellite Digital Image Analysis. Portland State University. http://web. pdx. edu/˜ jduh/courses/Archive/geog481w07/Students/Ludwig_ImageConvolution. pdf* (2015).

[15] Mark, Berechet, Mahlmann and Togelius. “Procedural Generation of 3D Caves for Games on the GPU.” *FDG*. 2015.

[16] Martin. “Inherent generation of fractals by cellular automata”. *Complex Systems* 8.5 (1994), pp. 347–366.

[17] Mitchell, Crutchfield, Das et al. “Evolving cellular automata with genetic algorithms: A review of recent work”. *Proceedings of the First International Conference on Evolutionary Computation and Its Applications (EvCA’96)*. Moscow. 1996.

[18] Hubert Nguyen. *Gpu gems 3*. Addison-Wesley Professional, 2007.

[19] Perlin. “An image synthesizer”. *ACM Siggraph Computer Graphics* 19.3 (1985), pp. 287–296.

[20] Resnick and Silverman. *Exploring emergence: The brain rules*. 1996.

[21] Willson. “Cellular automata can generate fractals”. *Discrete Applied Mathematics* 8.1 (1984), pp. 91–99.

[22] Stephen Wolfram. *A new kind of science*. Vol. 5. Wolfram media Champaign, 2002.

[23] Worley. “A cellular texture basis function”. *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM. 1996, pp. 291–294.