FH Aachen University of Applied Sciences,
Campus Jülich

Faculty 09 - Medical Engineering and Technomathematics
B.Sc. Applied Mathematics and Computer Science

Seminar Paper

Implementation of GLSL Compute Shaders in an Application for the Visualisation of Selected 2- and 3D Cellular Automata and the Exploration of their Rulespaces

Author:
Ali BERGER, 3522387

Supervisors:
Prof. Dr. rer. nat. Martin Reißel
Mr Vincent Paul Brünjes, M.Sc. RWTH

Aachen, December 19, 2023

Abstract

Cellular automata are models whose behaviour and characteristics are of both mathematical and scientific interest.

A CPU-based desktop application for the visualisation of selected 2D cellular automata and the exploration of their rulespaces (working title: CA Viz) was developed as part of a MATSE-apprenticeship at the IGMR RWTH Aachen.

In order to increase the number of possible simulation elements whilst maintaining or improving update frequency, it was decided to switch to a GPU-based algorithm. This paper covers the development and implementation of GPU-based compute shaders for 2D and 3D cellular automata in the CA Viz application.

Contents

1 Cellular Automata and the CA Viz Application

Cellular automata (CA) are mathematical simulations which can be used to model a wide variety of physical, natural, and man-made phenomena, including fluid flows [Lim90], the spread of influenza [BST05], and electric circuits [Fei08].

CA often consist of a 2D grid of simulation elements, called cells, which can be in and transition between one of several discrete states. For example, Peterson (2002) modeled forest fires spreading across a 2D landscape as cells transitioning between the following three states: contains empty space, contains trees, or contains burning trees. A cell containing trees can catch fire if its neighbouring cells contain burning trees, burning trees are replaced with empty space after a period of time, and empty cells have a probability of sprouting trees [Pet02].

1.1 What is the CA Viz Application?

The CA Viz application (Cellular Automata Visualiser) (1.1) started as a hobby project and was developed further as part of the Mathematical-Technical Software Developer (MATSE) apprenticeship at the Institute for Mechanism Theory, Machine Dynamics, and Robotics of the RWTH Aachen. The main goal of the software is to explore selected CA rulespaces and simulate the contained CA with high fidelity (in terms of update frequency and number of simulation elements) on desktop and laptop computers.

The application is programmed in C++ and uses the Open Graphics Library (OpenGL) to visualise the simulation.

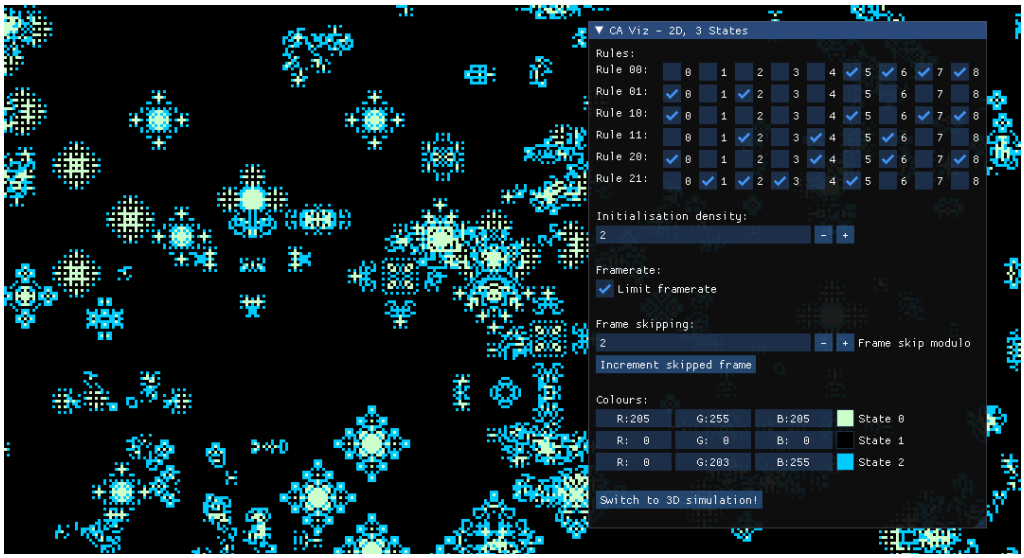


Figure 1.1: A closeup shot of the CA Viz application showing the graphical user interface and a running simulation.

1.2 Cellular Automata supported by the Application

The CA Viz application currently supports 2D and 3D orthogonal CA. Capabilities for simulating 3D CA were developed as part of this paper.

The 2D CA supported consist of an orthogonal grid of discrete cells, each of which can be in one of three states: state 0, 1, or 2 (1.2). State 0 is defined as the base state, or "dead", whilst all other states are defined as "living". In this arrangement a cell can have between 0 and 8 living neighbouring cells. The cells are updated synchronously according to a given rule 1.3.

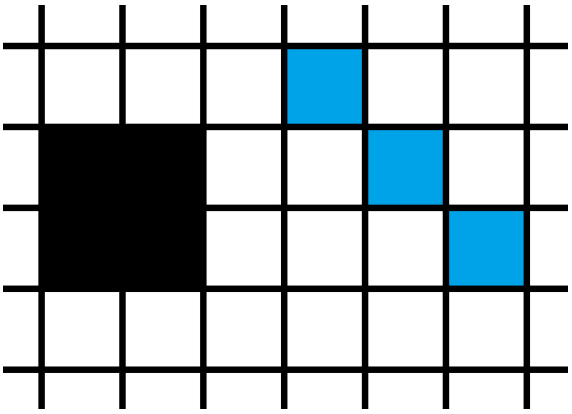


Figure 1.2: A section of grid containing cells in state 0 (white), state 1 (black), and state 2 (blue)

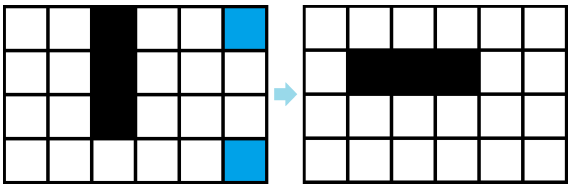


Figure 1.3: Cells of state 0 (white), 1 (black), and 2 (blue) in time step n (left) and then in time step n + 1 (right), updated according to rule R.

The simulation is outer totalistic, meaning that the state of a cell in step n+1 is fully dependent on the state of that cell and the number of its direct living neighbours in step n. How cells transition between states is determined via an aforementioned rule which contains the following sets: 00, 01, 02, 10, 11, 12, 20, 21, and 22. Each set XY contains elements representing the number of living neighbours required for a cell to transition from state X to state Y. Consider the following rule R:

$$00 = \{0, 1, 2, 4, 5, 6, 7, 8\},\ 01 = \{3\},\ 02 = \emptyset,$$
$$10 = \{0, 1, 4, 5, 6, 7, 8\},\ 11 = \{2, 3\},\ 12 = \emptyset,$$
$$20 = \{0, ..., 8\},\ 21 = 22 = \emptyset$$

Set 01 determines when a cell should transition from state 0 to 1 - if in step n a cell is in state 0 and the number of its living neighbours is contained in set 01, it then transitions to state 1 in step n + 1. The rulespace of these CA is defined as the set of all possible rules.

In order for a rule to be well defined, the sets it contains must fulfill the following criterion: exactly one set XY must encode the transition of state X to states 0, 1, or 2. As such, states X0, X1, and

X2 must be disjunct, and the union of these three sets must equal the set {0, 1, 2, 3, 4, 5, 6, 7, 8} - it must be uniquely clear what state a cell transitions to given its number of living neighbours.

The 3D CA supported by the application are defined analogously with currently only two states: 0 and 1, and thus only 4 transition sets: 00, 01, 10, and 11. A 3D orthogonal grid permits a maximum of 26 neighbours, and as such each transition set contains a subset of {0, ..., 26}.

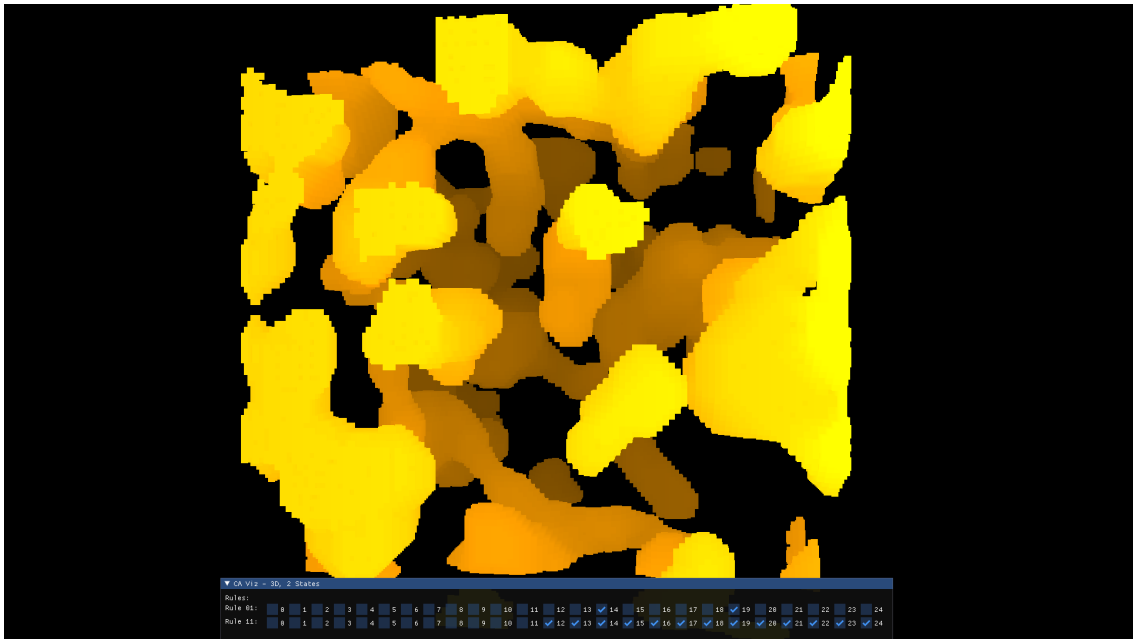


Figure 1.4: A 3D CA demonstrating clumping behaviour.

The supported CA are abstract; instead of modeling any specific system, the CA Viz application allows the user to adjust the transition rules between states, thus changing their behaviour. Future versions of the application may allow adjustment of which states count as “living” and “dead”, and may also allow an adjustable number of cell states.

1.3 Problem Statement: Improving Performance

The purpose of the application is to explore CA rulespaces and find CA with interesting behaviour. A user can define the transition sets of a rule and randomly initialise the simulation grid, then observe the resultant behaviour of the CA. Given the random initialisation of the grid, some structures and behaviours of a CA may only appear with a certain likelihood for every re-initialisation and for every step simulated. A larger grid, simulated for more steps therefore increases the chances of these behaviours and structures appearing. This leads to the following development goals for the application: to simulate CA with high update frequencies and with large grid sizes.

The displays of modern computers commonly have a maximum resolution of 1920 by 1080 pixels and an update frequency of 60 Hz, thus providing performance targets for the application: a minimum of 1920x1080 simulation elements and a steady 60 frames per second (FPS) or greater.

Due to the parallel nature of the simulation, the application of GPU resources could greatly increase both the achievable amount of simulation elements as well as the update frequency. It was thus decided to switch to a GPU-based implementation of the 2D and 3D CA, the development of which is the focus of this paper.

1.4 Methodology

This paper first evaluates common GPU frameworks for suitability (2.1) and then implements both the 2D and 3D simulations in the chosen framework, specifically OpenGL (2.2). The focus of the implementation is how simulation elements are encoded as OpenGL textures (2.2.1) and how the framework updates each element in parallel through the usage of compute shaders (2.2.2). Results are presented comparing the performance of the 2D CPU-, 2D GPU-, and 3D GPU-based simulations (3.1). These results are discussed (4.1), an example CA demonstrating the usefulness of the obtained performance increases is showcased (4.2), and the potential for improving on the implemented simulation algorithms is considered (4.3). Addendum A includes the full source code of both the 2D and 3D compute shaders, addendum B the system specifications used for the benchmarking, and addendum C contains a sampling of interesting and visually appealing CA observed whilst using the application.

2 Implementation of the GPU-based Simulation

This chapter deals with the selection of a suitable framework for GPU programming, how the information which describes simulation elements (the cells of a CA) are therein encoded, and also the implementation of the simulation logic.

2.1 Current State of the Art: choosing a GPU API

There are numerous frameworks and application programming interfaces (APIs) available for writing GPU-accelerated code, including Nvidia's proprietary Compute Unified Device Architecture (CUDA) [Corb], the Open Computing Language (OpenCL) and Open Graphics Language (OpenGL) standards from the Khronos Group [Grob] [Groc], and the Open Multi-Processing (OpenMP) API [Kle].

CUDA is only compatible with Nvidia GPUs [Cora], whilst the other APIs listed above are multi-platform. The CA Viz application targets desktops and laptops that may not have dedicated graphics processors, or may not have Nvidia GPUs, making CUDA incompatible with development goals.

OpenGL was chosen above OpenMP and OpenCL due to the rendering engine of CA Viz having been implemented in OpenGL and the OpenGL Shading Language (GLSL), allowing for any simulation code written with OpenGL and GLSL to be relatively easily integrated into the application.

2.2 Writing Simulation Code with OpenGL and GLSL

OpenGL is primarily used for writing rendering software, i.e., programs which take in information about what is to be visualised, process this information to achieve a desired effect, and then output the results to a screen, file, or data structure. OpenGL stores information in data structures called textures (arrays of pixels) [Groe] and processes information via GPU programs called shaders [Grod]. OpenGL shaders are written in the C-like language GLSL. Textures can hold information such as the colours of each pixel in an image, and shaders can apply effects to textures such as brightening each pixel.

Instead of image data, it is possible to encode simulation elements as pixels in a texture. A special type of shader for arbitrary data manipulation (a compute shader) can then be used to update each element in parallel according to the simulation's rules.

2.2.1 Encoding 2D Simulation Elements as Pixels in a Texture

The grid of cells in the simulation is implemented as a 2D texture, with each cell represented as one pixel.

The information that a pixel contains can be configured according to formats provided by OpenGL [Groa], which describe how many channels a pixel contains, how many bits large each channel is, and if these bits should be interpreted as signed or unsigned. The channels of a pixel, when used to encode image information, typically represent colours. We can use these channels as variables to hold the state information of a cell.

Consider a pixel of format GL_RGB32UI. It contains channels for red, green, and blue (corresponding to the RGB section of the name), each of which holds a 32-bit long unsigned integer.

Each cell only needs to store a single integer between 0 and 2, as such one of the most compact pixel formats was chosen, namely GL_R8I (a single red channel containing an 8 bit signed integer). The other smallest pixel format is RL_R8UI, which analogously encodes unsigned integers. The unsigned variant can represent more positive states (0 to 255), whilst the signed variant can also represent negative states (-128 to 127). For the purposes of the current simulation the difference plays no role.

2.2.2 2D Compute Shader Implementation

The compute shader needs to perform the following tasks:

1. Load the textures representing time steps n (for input) and n+1 (for output)
2. Load the transition sets
3. Determine the x and y-coordinates of the current pixel
4. Count the number of living neighbours surrounding the current pixel
5. Read the state that the of the current element from the correct pixel channel
6. Determine the state of the pixel in time step n+1 according to the transition sets
7. Write the new state of the cell to the corresponding pixel in the output texture

The GLSL source code of the shader as well as a pseudocode description can be found in addendum A.1. Code segments of interest are described below.

Lines 4 and 5 bind the input and output textures:

```
4  layout (binding = 0, r8i) uniform iimage2D img_input;
5  layout (binding = 1, r8i) uniform iimage2D img_output;
```

The keyword “uniform” declares global variables that can be sent from the main program to the graphics card. The “layout” keyword specifies the pixel format, here given as “r8i”, corresponding to the OpenGL GL_R8I format, whilst the “iimage2D” datatype declares the variables img_input and img_output as 2D images which contain signed integers.

Lines 7 through 12 declare the transition sets:

```
7  uniform int zero_zero[9] = {0, 0, 0, 0, 0, 1, 1, 0, 1};
8  uniform int zero_one[9] = {1, 0, 1, 1, 0, 0, 0, 0, 0};
9  uniform int one_zero[9] = {0, 0, 0, 1, 0, 1, 0, 1, 1};
10 uniform int one_one[9] = {0, 1, 0, 0, 0, 0, 0, 0, 0};
11 uniform int two_zero[9] = {1, 0, 1, 0, 1, 1, 0, 1, 1};
12 uniform int two_one[9] = {0, 0, 0, 1, 0, 0, 1, 0, 0};
```

GLSL does not have built-in functions for checking if an element is contained in an array. As opposed to looping through each set, the following faster method was devised: each transition set is represented as an array of 9 integers. If the integer n is in the set, position n in the array is set to 1, or 0 otherwise. The transition set 5, 6, 8 is represented as 0, 0, 0, 0, 0, 1, 1, 0, 1. To check if an element is in the array, then indexing the array with that element and casting as a boolean returns the desired result. For example, the following would return true:

```
1  bool(zero_zero[5])
```

Also of interest is how pixels and their channels are read, for example in line 33:

```
33  int current = imageLoad(img_input, ivec2(x, y)).r;
```

The “imageLoad” function takes in the desired image and the target coordinates of a pixel as parameters and returns the pixel’s channels. In the above code, the red channel is immediately read out with “.r” and stored in the variable current.

2.3 Implementation of the 3D Cellular Automata

Analogously to 2D textures, which typically encode 2D images, OpenGL provides 3D textures for the storage of volumetric data.

The compute shader (source code and pseudocode available under addendum A.2) and encoding of simulation elements in textures for the 3D simulation are implemented similarly to the 2D simulation.

The main differences include a reduction in the number of transition sets (due to the 3D simulation only currently supporting two cell states), the transition set arrays containing 27 elements (0 through 26), the pixels being indexed using 3D coordinates, and the counting of neighbouring cells now requiring 3 for-loops.

A volumetric rendering engine for the visualisation of the supported 3D CA was developed parallel to this paper.

3 Results

The 2D simulation with 1920x1080 simulation elements achieved a frame rate of 383.3 FPS and can maintain a frame rate of at least 60 FPS with 5760x3240 elements. The 3D simulation with a grid size of 128^3 achieved a frame rate of 169.3 FPS.

Several interesting CA rules where identified, illustrations of which are available in addendum C and in the discussion (4.2).

3.1 Benchmarking of the 2D CPU, 2D GPU, and 3D GPU Simulations

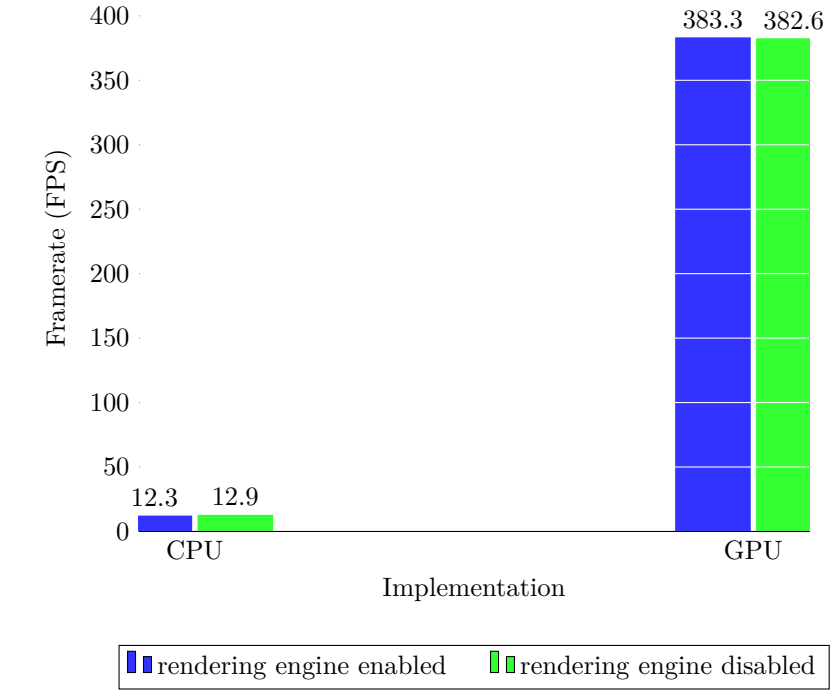
All benchmarks were run on a high-end consumer laptop, a Lenovo ThinkPad T14 with an NVIDIA GeForce MX330 graphics card, an Intel Core i7 CPU, and a display of resolution of 1920 by 1080 with and update frequency of 60 Hz. Relevant specifications of the GPU and CPU are listed in addendum B.

CA Viz uses a third-party library called Dear ImGui to implement its graphical user interface. This library also contains functions for measuring frame rates, which were used to measure the performance of the simulation loop. All frame rates listed are the highest running average of the latest 120 frames achieved within 20 seconds after starting the simulation.

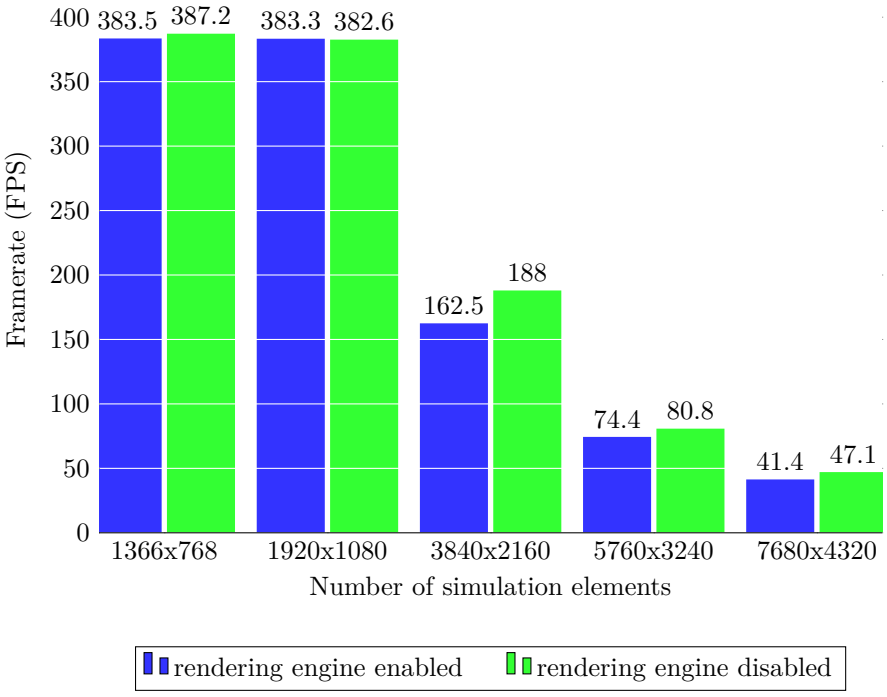
The numbers of simulation elements used in the 2D simulation benchmarks correspond to existing computer monitor display resolutions, thus ranging from 1,049,088 (1366x768) to 33,177,600 (7680x4320) elements. The sizes of the 3D grids are derived from the resolutions used for the 2D benchmarks. For example, the resolution 5760x3240 results in 18,662,400 elements. The 3D simulation uses the cube root of this amount rounded up, 266, as the dimension of each side in its 3D grid of cells. Having similar numbers of elements in both the 2D and 3D benchmarks allows for a more direct comparison of performance between the 2D and 3D simulations.

Frame rates were measured with the rendering engines both enabled and then disabled, to better determine whether performance bottlenecks were occurring in the compute shaders or when drawing the simulation.

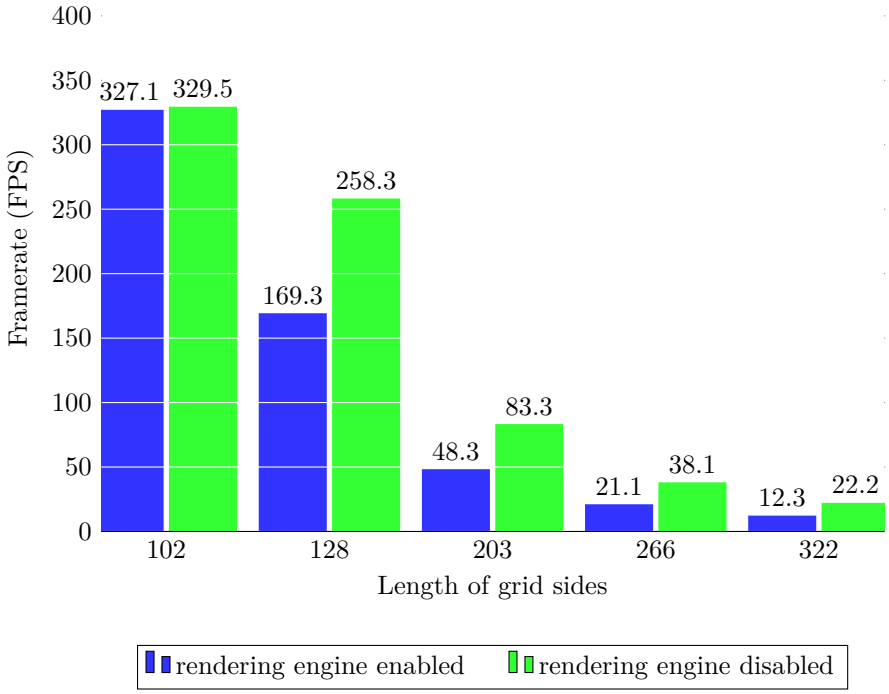
Performance of the CPU- and GPU-based Simulations with 1920x1080 Elements



Performance of the GPU-based 2D Simulation



Performance of the GPU-based 3D Simulation



4 Discussion

The main point of discussion is the comparison of the benchmark results both within and between the 2D and 3D simulations, which allows for insights into which factors influence performance of the computer shaders specifically and the application as a whole. A CA demonstrating the usefulness of increases in frame rate and amount of simulation elements is shown, and an improvement for the process of updating cell states is presented and evaluated. Furthermore, the usage of compute shaders in the automated exploration of CA rulespaces is considered.

4.1 Analysis of Benchmarks

Comparison of the frame rates achieved for different amounts of simulation elements within the 2D simulation allow for the following observations:

1. All resolutions achieved or exceeded the target frame rate of 60 FPS, except for 7680x4320 with an FPS of 41.4.
2. Resolutions 1366x768 and 1920x1080 achieved approximately the same maximum performance of 383.5 FPS and 383.3 FPS respectively, suggesting that the number of processing cores within the graphics card was not yet saturated. Given the low memory usage of the simulations, the limiting factors may be the clock speed of the graphics card and the running time of the compute shader.
3. A minor increase in frame rate resulted from disabling rendering in most cases, showing the 2D rendering engine to be of relatively low performance impact.
4. The performance increase by disabling rendering decreases between resolutions 3840x2160, 5760x3240, and 7680x4320.

Observations on the 3D simulation benchmarks are as follows:

1. The target frame rate of 60 FPS was achieved and exceeded by two grid sizes: 102^3 and 128^3.
2. Performance improvements occurred in all grid sizes when disabling rendering, with the greatest percentage and absolute increase (89.0 FPS, 52.6%) being at a grid size of 128^3, showing that the 3D rendering engine has a much higher performance cost than the 2D variant.
3. Of note is the performance increase when disabling rendering by 203^3 elements, increasing from 48.3 FPS to 83.3 FPS. This suggests that improvements to the rendering engine may allow the target frame rate of 60 FPS to be achieved.

When comparing 2D and 3D simulations with similar numbers of elements, 3D performed significantly worse than 2D in all cases, even with the rendering engine disabled. A possible explanation may be the differing O-classes of the two compute shaders: the 2 nested for-loops for counting neighbours in the 2D simulation being of class $O(n^2)$, and the 3D simulation's 3 nested for-loops being of class $O(n^3)$.

4.2 A CA found due to Improvements in Performance

As mentioned in the problem statement (1.3), certain CA behaviours and structures emerge less frequently than others. Improving the update frequency and number of supported simulation elements would allow rarer behaviours to thus be found more readily.

The below-illustrated CA was observed by the author using the GPU-based version of the CA Viz application (4.1). It takes several thousand simulation steps after initialisation before any interesting properties become apparent, and even at a resolution of 1920 by 1080 this behaviour does not occur after every initialisation.

The feature of interest is that this CA generates a type of glider[1] which takes several thousand time steps to set itself up, is dozens of cells long in width and height, and has a period of hundreds of steps. For comparison, the first glider discovered in the Conway's Game of Life CA consists of 5 cells and has a period of 4 generations [gli].

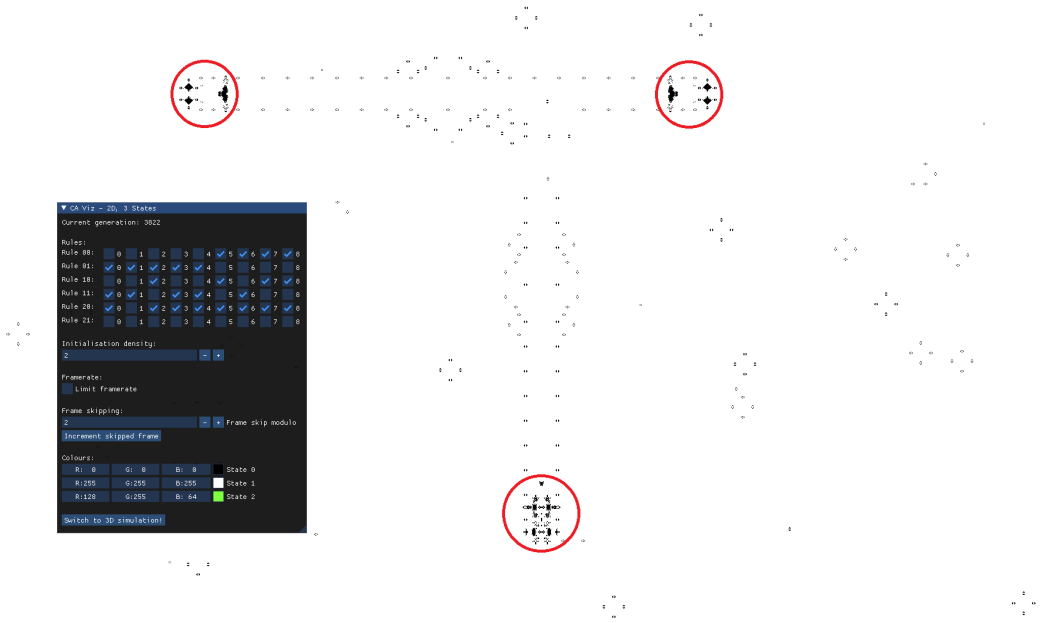


Figure 4.1: Three large gliders (circled in red) emerge after a lengthy setup process, leaving trails of oscillating patterns behind.

[1] A pattern of cells that repeats itself and moves across the grid

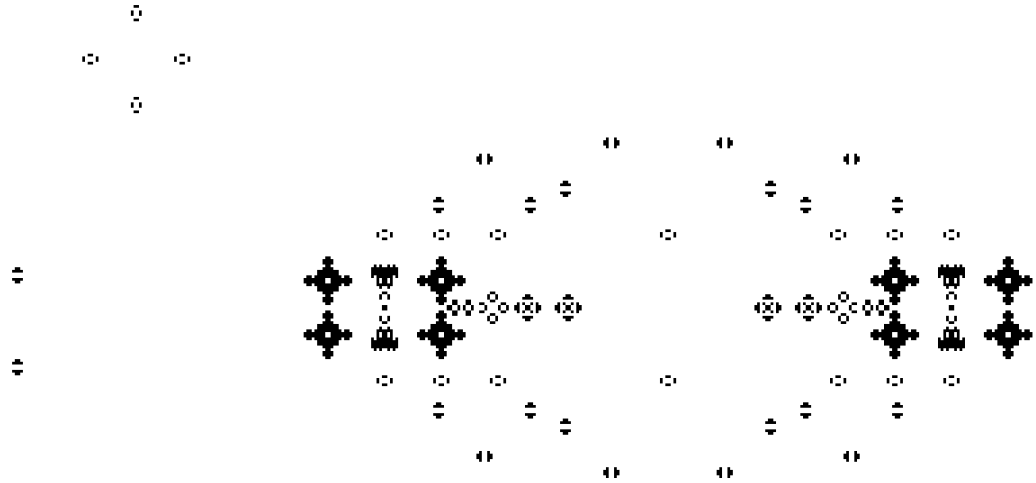


Figure 4.2: Completion of the start-up phase. Before the gliders start moving, over 2800 generations have passed.

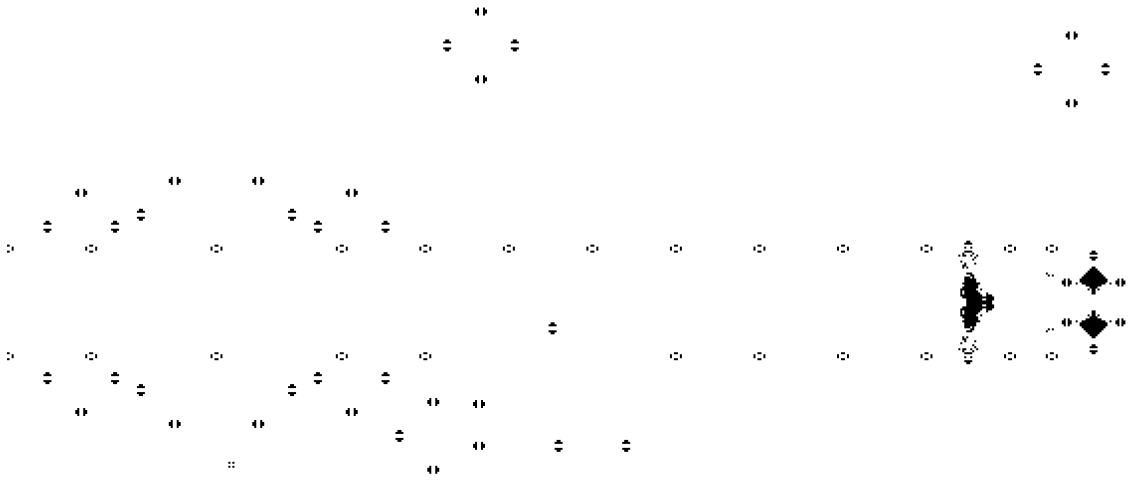


Figure 4.3: A glider several hundred generations after emerging, moving horizontally to the right.

4.3 Improvements on the Naive Algorithm

The current algorithm for updating the grids in both the 2D and 3D simulations attempts to update every cell in the grid during every simulation step. A known optimisation for efficiently updating the grids of certain CA is to only attempt to update cells whose neighbours have changed state from the previous simulation step [OWB14].

Such change information could be stored as a flag in an additional pixel channel for each cell. If a cell updates, all surrounding cells could update this flag accordingly. If this flag is set, the compute

shader can then copy over the cell state instead of recounting its neighbours and re-evaluating the transition sets.

Two potential problems with this approach present themselves. Updating the change flag of all neighbouring cells increases the runtime of the compute shader. For rules in which most cells change states during every step, this method may provide little to no benefit or perhaps reduce performance.

4.4 Uses of Compute Shaders in an automated statistical Investigation of CA Rulespaces

Whilst the goal of the CA Viz application is to facilitate the exploration of CA rulespaces, the exploration and classification of entire rulespaces in this manner is impractical. Changing the CA rule, observing the resultant CA, and then noting interesting behaviour can take upwards of 20 seconds. A rulespace of 2D outer-totalistic CA with two states has $2^{18} = 266,144$ different rules, which would take 60 days to classify by hand. The classification of such a rulespace with 3 states has $3^{27} \approx 7 \times 10^{12}$ rules and would require approximately 4,8 million years.

Measures exist for determining the complexity of certain CA [Mar00]. If such a measure could be implemented in a compute shader and still maintain comparable performance to that achieved in this paper, a complexity map of the above mentioned 2D rulespace would take under 73 hours to compute on a consumer laptop[2]. Such a map could be visualised as a 3D point cloud heat map, differentiating regions of high and low complexity, potentially assisting with locating interesting rules.

[2] Assuming 380 simulations steps are sufficient for such a measurement, and that the simulation and calculation run at 380 Hz, one second of computation time would be required per rule.

5 Conclusion

The goals of this paper were achieved and surpassed. The compute shaders developed met and exceeded targets for both the number of specified elements (1920x1080) and update frequency (60 Hz) of the simulations, allowing for the high fidelity simulation and display of CA.

Measuring the performance of the 2D and 3D compute shaders highlighted areas for improvement in both the compute shaders and the application as a whole, indicating that substantial performance increases stand to be gained from improvement of the 3D rendering engine.

A number of interesting CA were observed and documented, one of which demonstrating the usefulness of increased simulation resolution and update frequency for finding rarer CA behaviours.

An improvement to the algorithms of both compute shaders was discussed, as well as the usage of compute shaders in the automated investigation of CA rulespaces, providing opportunities for further development.

A Source Code

A.1 Source Code and Description of the Compute Shader for the 2D Simulation

The following is the GLSL source code of the 2D CA compute shader.

```
#version 430 core

layout (local_size_x = 32, local_size_y = 32) in;
layout (binding = 0, r8i) uniform iimage2D img_input;
layout (binding = 1, r8i) uniform iimage2D img_output;

uniform int zero_zero[9] = {0, 0, 0, 0, 0, 1, 1, 0, 1};
uniform int zero_one[9] = {1, 0, 1, 1, 0, 0, 0, 0, 0};
uniform int one_zero[9] = {0, 0, 0, 1, 0, 1, 0, 1, 1};
uniform int one_one[9] = {0, 1, 0, 0, 0, 0, 0, 0, 0};
uniform int two_zero[9] = {1, 0, 1, 0, 1, 1, 0, 1, 1};
uniform int two_one[9] = {0, 0, 0, 1, 0, 0, 1, 0, 0};

void main() {
 ivec2 coords = ivec2(gl_GlobalInvocationID.xy);
 int x = coords.x;
 int y = coords.y;

 ivec2 image_res = imageSize(img_input);
 int res_x = image_res.x;
 int res_y = image_res.y;

 uint counter = 0;

 for (int i = -1; i < 2; i++) {
  for (int j = -1; j < 2; j++) {
   counter +=
   int(imageLoad( img_input,
       ivec2( mod(x + i, res_x), mod(y + j, res_y))).r != 0);
  }
 }

 int current = imageLoad(img_input, ivec2(x, y)).r;

 counter -= int(current != 0);

 if ( (current == 0) && bool(zero_zero[counter]) ) {
  imageStore(img_output, coords, ivec4(0));
 } else if ( (current == 0) && bool(zero_one[counter])) {
  imageStore(img_output, coords, ivec4(1));
 } else if ( (current == 1) && bool(one_zero[counter])) {
  imageStore(img_output, coords, ivec4(0))
 } else if ( (current == 1) && bool(one_one[counter])) {
  imageStore(img_output, coords, ivec4(1));
 } else if ( (current == 2) && bool(two_zero[counter])) {
  imageStore(img_output, coords, ivec4(0));
 } else if ( (current == 2) && bool(two_one[counter])) {
  imageStore(img_output, coords, ivec4(1));
 } else {
  imageStore(img_output, coords, ivec4(2));
 }
}
```

The following is a pseudocode description of the above compute shader.

```
bind texture of step n as img_input
bind texture of step n+1 as img_output

load rule transition sets 00, 01, 10, 11, 20, 21

load height and width of textures into res_y and res_x

load (x, y)-coordinates of current pixel into variable coords

initialise variable counter = 0

for (i = -1; i < 2; i++)
for (j = -1; j < 2; j++)
if (pixel in img_input at (coords.x + i % res_x, coords.y + j % res_y) != 0)
increment counter by 1

load value of current pixel into variable current

if (current != 0)
decrement counter by 1

if a transition set XY exists that contains counter and with X == current
write Y to the pixel in img_output at (coords.x, coords.y)

if no transition sets match the above criteria
write 2 to the pixel in img_output at (coords.x, coords.y)
```

A.2 Source Code of Compute Shader for the 3D Simulation

The following is the GLSL source code of the 3D CA compute shader.

```
#version 430 core

layout (local_size_x = 8, local_size_y = 8, local_size_z = 8) in;
layout (binding = 0, r32ui) uniform uimage3D img_input;
layout (binding = 1, r32ui) uniform uimage3D img_output;

uniform int zero_one[27] = {
 0, 0, 0, 0, 0,
 0, 1, 0, 0, 0,
 0, 0, 0, 0, 0,
 0, 0, 0, 1, 0,
 0, 0, 0, 0, 0, 0, 0
};
uniform int one_one[27] = {
 0, 0, 0, 0, 0,
 1, 0, 0, 0, 1,
 1, 1, 1, 1, 1,
 1, 1, 1, 0, 1,
 1, 0, 1, 0, 0, 0, 0
};

void main() {
 ivec3 coords = ivec3(gl_GlobalInvocationID.xyz);
 int x = coords.x;
 int y = coords.y;
 int z = coords.z;

 ivec3 image_res =imageSize(img_input);

 uint counter = 0;

 for (int i = -1; i < 2; i++) {
  for (int j = -1; j < 2; j++) {
   for (int k = -1; k < 2; k++) {
    counter +=
    int(imageLoad(img_input,
     ivec3(
       mod(x + i, image_res.x),
       mod(y + j, image_res.y),
       mod(z + k, image_res.z) )).r != 0);
   }
  }
 }

 uint current = imageLoad(img_input, ivec3(x, y, z)).r;

 counter -= int(current != 0);

 if ( (current == 0) && bool(zero_one[counter]) ) {
  imageStore(img_output, coords, ivec4(1));
 } else if ( (current == 1) && bool(one_one[counter])) {
  imageStore(img_output, coords, ivec4(1));
 } else {
  imageStore(img_output, coords, ivec4(0));
 }
}
```

The following is a pseudocode description of the above compute shader.

```
bind texture of step n as img_input
bind texture of step n+1 as img_output

load rule transition sets 01, 11

load height, width, and breadth of textures into res_y, res_x, and res_z

load (x, y, z)-coordinates of current pixel into variable coords

initialise variable counter = 0

for (i = -1; i < 2; i++)
for (j = -1; j < 2; j++)
for (j = -1; k < 2; k++)
if (pixel in img_input at ( coords.x + i % res_x,
coords.y + j % res_y,
coords.z + k % res_k) != 0)

increment counter by 1

load value of current pixel into variable current

if (current != 0)
decrement counter by 1

if a transition set XY exists that contains counter and with X == current
write Y to the pixel in img_output at (coords.x, coords.y, coord.z)

if no transition sets match the above criteria
write 0 to the pixel in img_output at (coords.x, coords.y, coords.z)
```

B System Specifications

All benchmarking was performed on a Lenovo ThinkPad T14 with the following GPU and CPU:

GPU: NVIDIA GeForce MX330

- 384 cores
- Maximum clock speed of 1594 MHz

CPU: Intel Core i7-10510U

- 4 cores
- Maximum clock speed of 4.9 GHz

C Examples of Cellular Automata found with the CA Viz Application

This addendum showcases interesting or visually appealing CA observed by the author whilst developing and using the application.

The following rule (C.1) exhibits several interesting behaviours of CA. Almost every cell switches states between simulation steps, yet instead random noise, complex structures form and interact. Such formations include gliders (patterns of cells which move across the grid), glider guns (C.2), and replicators, which periodically produces several copies of themselves (C.3). Large phase-shifted regions[1] arise, all containing the above-mentioned structures (C.4).

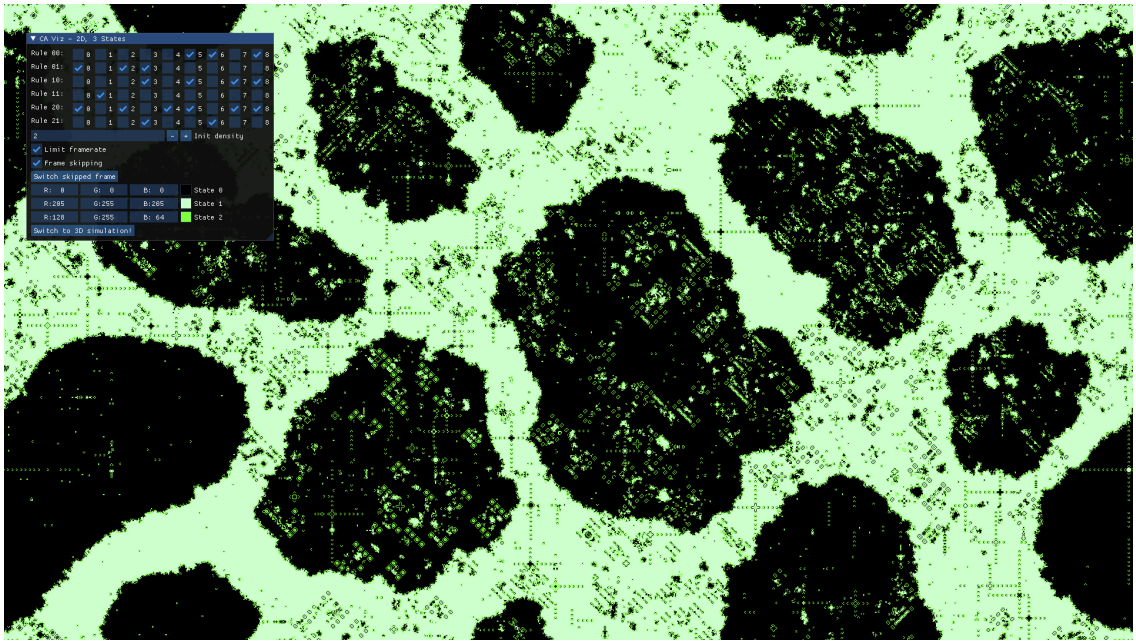


Figure C.1: Multiple behaviours can emerge from a single rule, building a mathematical ecosystem of sorts.

[1] Figure C.1 shows large black regions (cells in state 0) embedded in a light-green region (cells in state 1). In the next simulation step, these regions of cells will switch states, black turning to light-green and vice versa. They are thus one time step out of phase with each other.

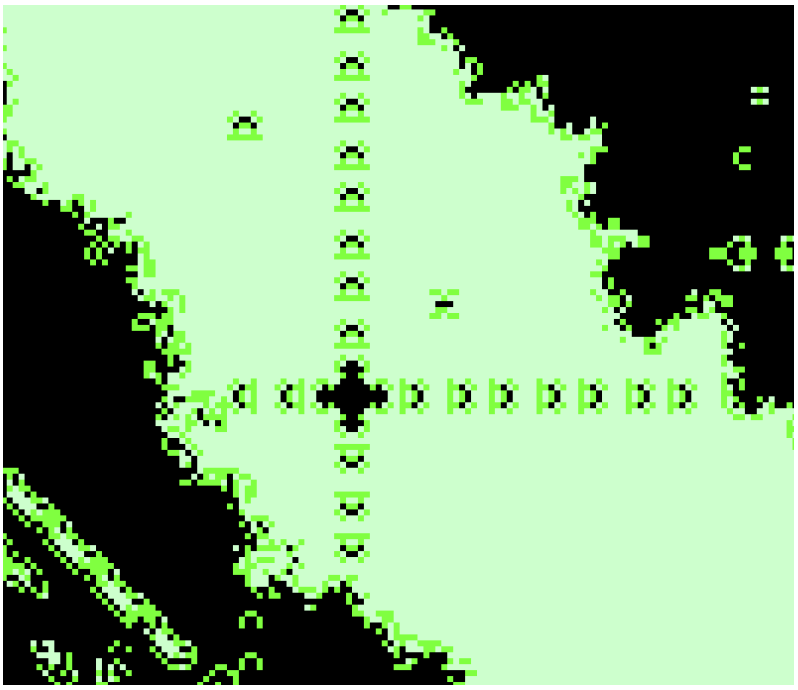


Figure C.2: A glider gun shoots gliders in all cardinal directions. A lone glider is visible in the top left, perhaps having been generated through some other means.

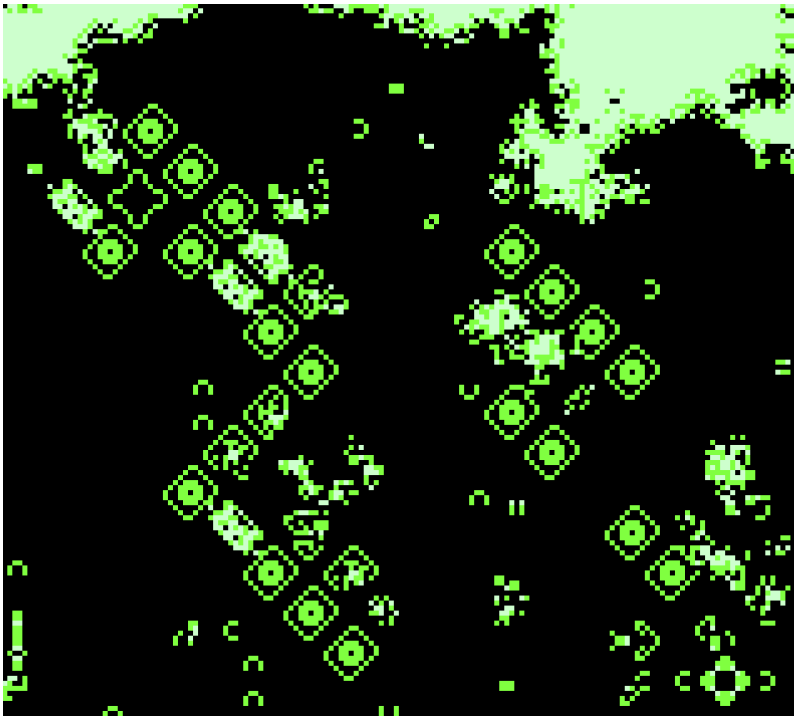


Figure C.3: Several replicators expand within a region. Interaction with other structures has interrupted the life cycle of some.

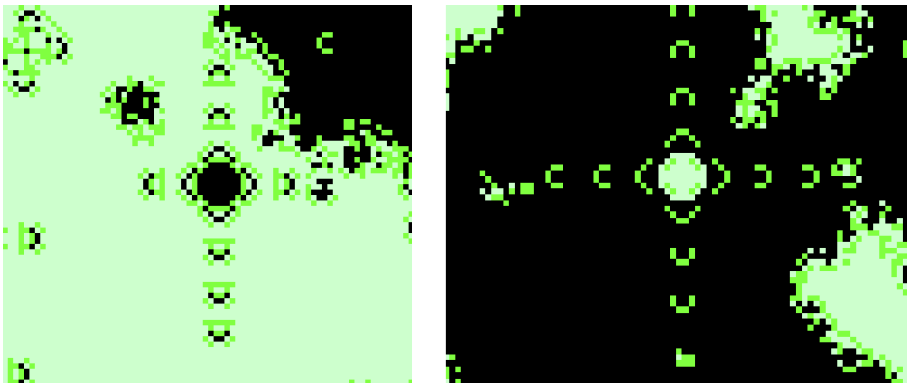


Figure C.4: A glider gun (left) and a phase-shifted variant (right).

A 3D rule demonstrating many complex behaviours was also found (C.5). This rule generates gliders, replicators, stable stationary structures, and long columns which extend themselves.

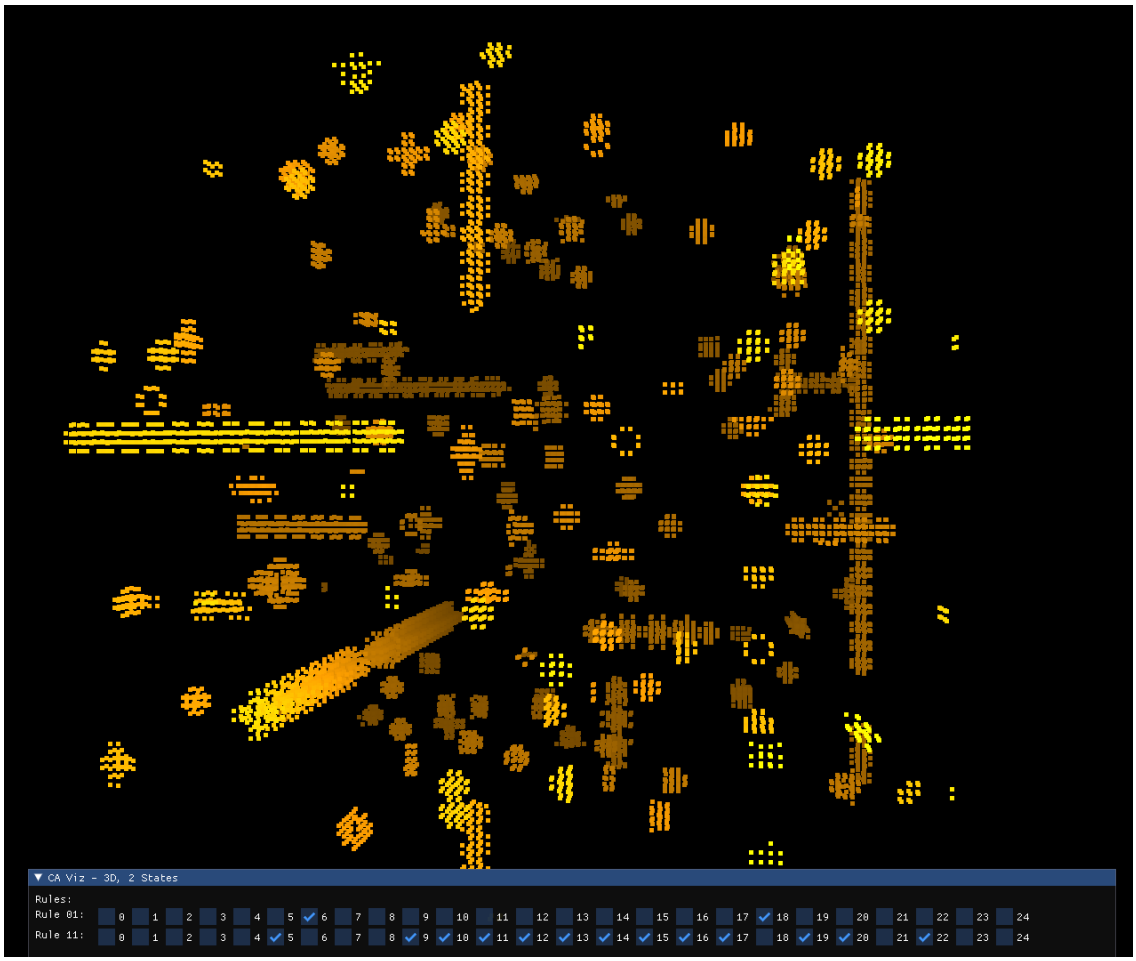


Figure C.5: A 3D CA drawn using the volumetric rendering engine developed parallel to this paper.

An interesting observation on the 2D CA rulespace with 2 states is the existence of state-inverse rules, i.e., pairs of different rules which exhibit the same behaviour, but with states 0 and 1 swapped.

For example, figure C.6 shows Conway's Game of Life (with transition sets 00 = {0, 1, 2, 4, 5, 6, 7, 8}, 01 = {3}, 10 = {0, 1, 4, 5, 6, 7, 8}, 11 = {2, 3}) has a state-inverted analogue with transition sets 00 = {5, 6}, 01 = {0, 1, 2, 3, 4, 7, 8}, 10 = {5}, 11 = {0, 1, 2, 3, 4, 6, 7, 8}. This type of symmetry might apply to all rules in this rulespace.

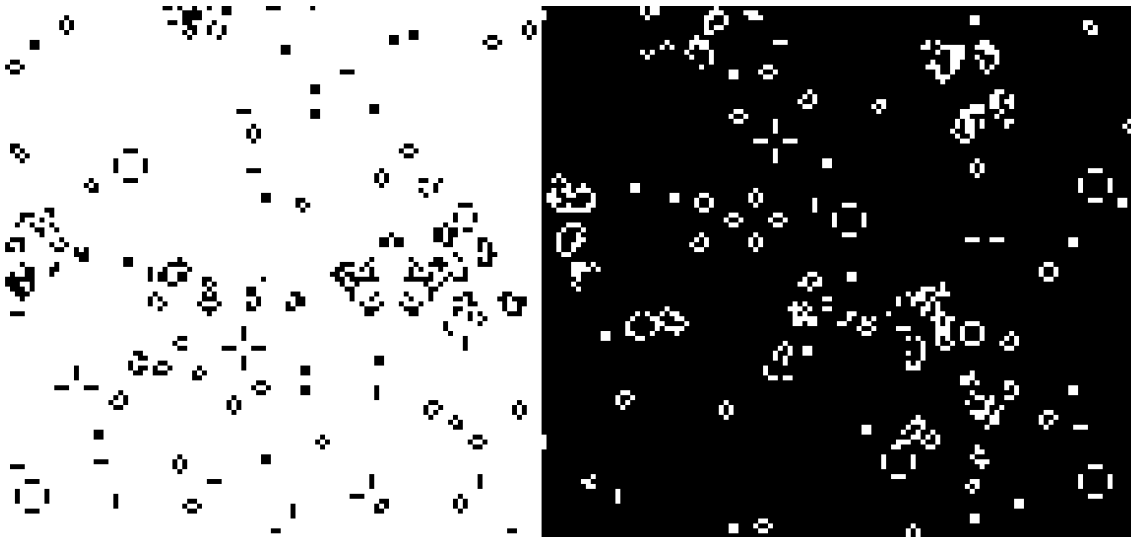


Figure C.6: Conway's Game of Life (left) and its inverted partner (right). State 0 is represented as white and state 1 as black for both rules.

The following rule has large regions which periodically transition through all three states (C.7). Smaller replicators and stable stationary patterns exist between regions.

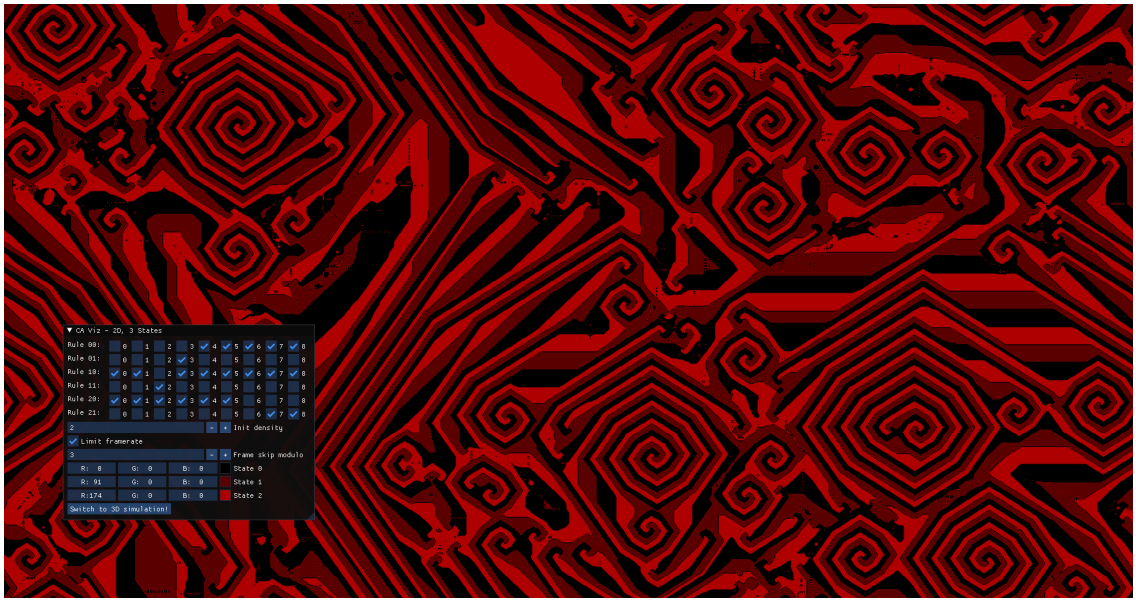


Figure C.7: Of note in the above rule are the whirlpool-like structures which consist of regions of all three states.

Perhaps the most surprising specimen found was a rule which naturally generates animated Sierpiński triangles after being randomly initialised (C.8 and C.9).

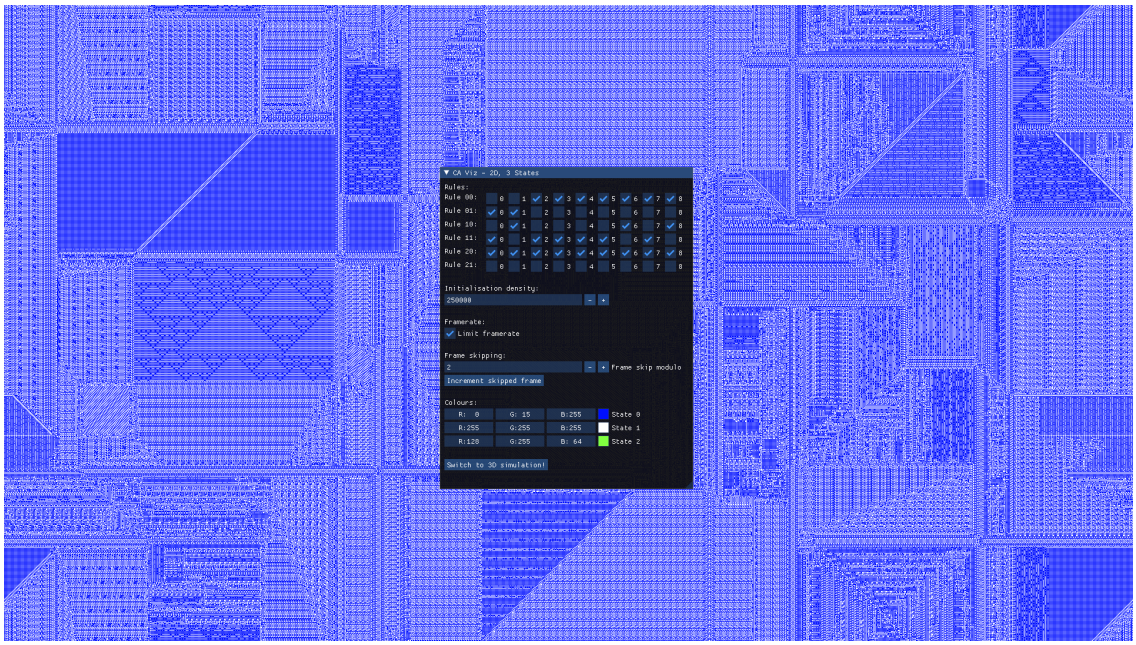


Figure C.8: Several grids of animated Sierpiński-esque triangles appear around the GUI which appear to scroll vertically and horizontally. Most prominent are those in the top-right and mid-left of the figure.

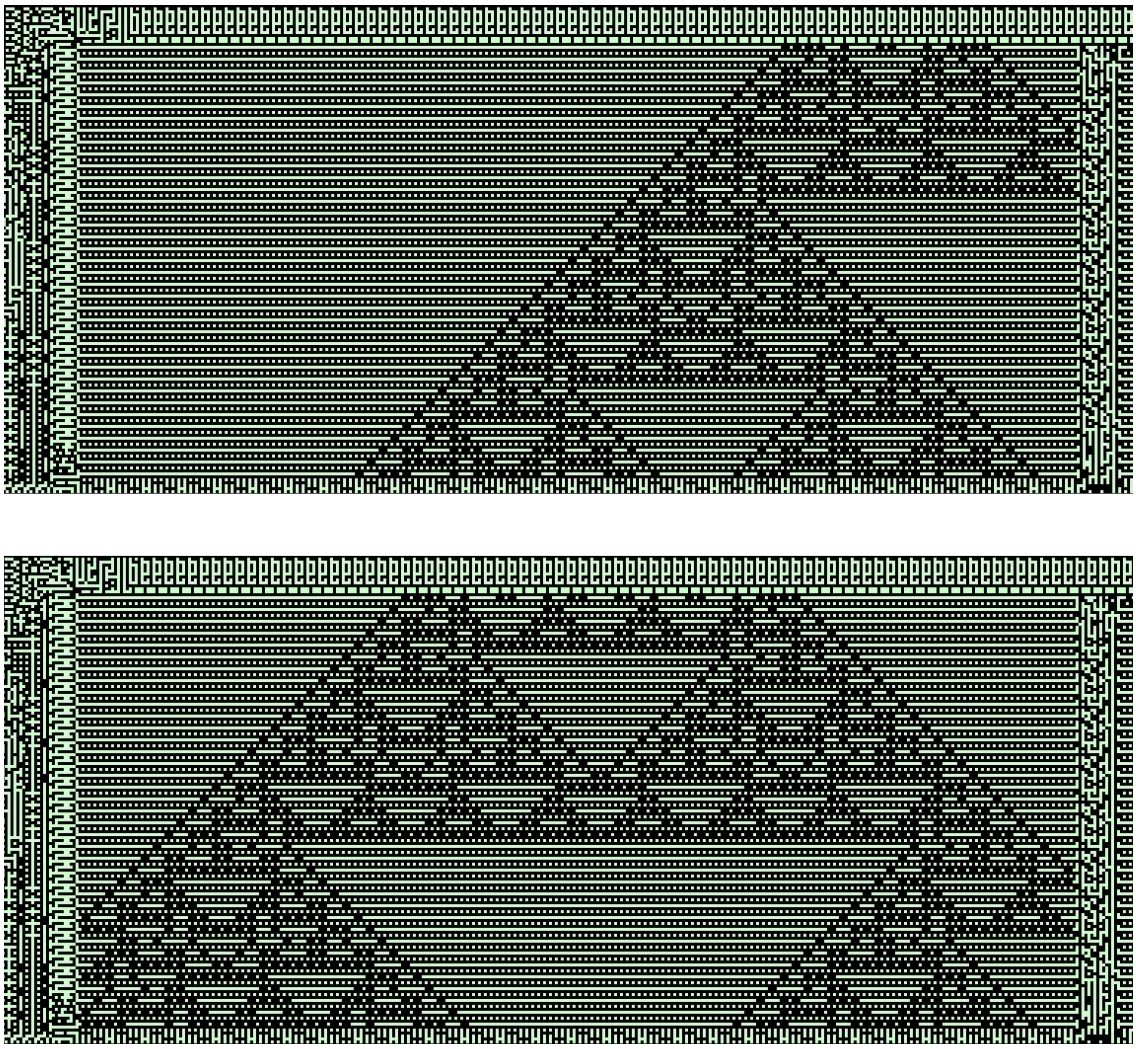


Figure C.9: A closeup section showing a Sierpiński-esque pattern (top) which appears to scroll upwards. The same section of grid (bottom) is shown several frames later.

After being initialised with an extremely low cell density (one cell in state 1 for each 250,000 cells in state 0 for example), points of growth emerge which extend outwards and then form intricate patterns when interacting with each other. In some instances, sequences of lines which appear to function as one-dimensional CA are created one after another, resulting in moving patterns.

D Bibliography

[BST05] Catherine Beauchemin, John Samuel, and Jack Tuszynski. A simple cellular automaton model for influenza a viral infections. *Journal of Theoretical Biology*, 232(2):223–234, 2005.

[Cora] Nvidia Corporation. Cuda compatibility. https://docs.nvidia.com/deploy/cuda-compatibility/. Accessed: 2023-11-24.

[Corb] Nvidia Corporation. Cuda toolkit - develop, optimize and deploy gpu-accelerated apps. https://developer.nvidia.com/cuda-toolkit. Accessed: 2023-11-24.

[Fei08] L.M.G. Feijs. Reinventing electronics with cellular automata. *Complex Systems*, 18(1):53–73, 2008.

[gli] Glider - lifewiki. https://conwaylife.com/wiki/Glider. Accessed: 2023-12-18.

[Groa] The Khronos Group. gleximage2d - opengl 4 reference pages. https://registry.khronos.org/OpenGL-Refpages/gl4/html/glTexImage2D.xhtml. Accessed: 2023-11-30.

[Grob] The Khronos Group. Opencl overview - the khronos group inc. https://www.khronos.org/opencl/. Accessed: 2023-11-24.

[Groc] The Khronos Group. Opengl overview - the khronos group inc. https://www.khronos.org/opengl/. Accessed: 2023-11-24.

[Grod] The Khronos Group. Shader - opengl wiki. https://www.khronos.org/opengl/wiki/Shader. Accessed: 2023-11-30.

[Groe] The Khronos Group. Texture - opengl wiki. https://www.khronos.org/opengl/wiki/Texture. Accessed: 2023-11-30.

[Kle] Dr.-Ing. Michael Klemm. Intro to gpu programming with the openmp api. https://www.openmp.org/wp-content/uploads/2021-10-20-Webinar-OpenMP-Offload-Programming-Introduction.pdf. Accessed: 2023-11-24.

[Lim90] H.A. Lim. Lattice-gas automaton simulations of simple fluid dynamical problems. *Mathematical and Computer Modelling*, 14:720–727, 1990.

[Mar00] Bruno Martin. Apparent entropy of cellular automata. *Complex Syst.*, 12, 2000.

[OWB14] Gadi Oxman, Shlomo Weiss, and Yair Be'ery. Computational methods for conway's game of life cellular automaton. *Journal of Computational Science*, 5(1):24–31, 2014.

[Pet02] Garry D. Peterson. Contagious disturbance, ecological memory, and the emergence of landscape pattern. *Ecosystems*, 5(4):329–338, 2002.