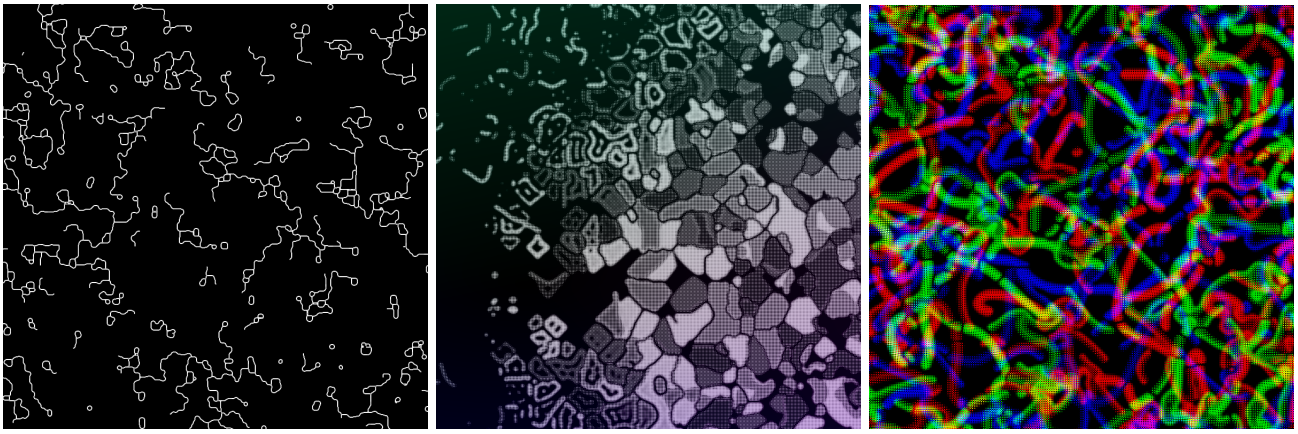
Neural Cellular Automata in 2D and 3D using Volumetric Rendering
Computer Graphics Final Project Report

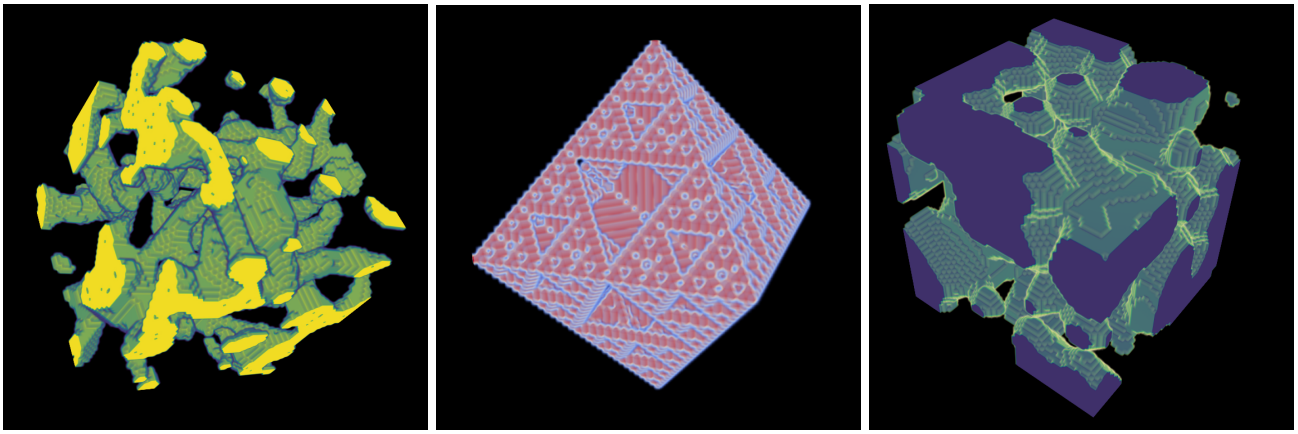
by Marco Ravelo

CS 354H: Computer Graphics
University of Texas at Austin - Spring 2023



(a) *paths* automata with *bnw* shader (b) *slime* automata with *acid* shader (c) *worms* automata with *rgb* shader

Figure 1: Examples of 2D neural cellular automatas.



(a) *caves* automata with *ybg* colormap (b) *crystal* automata with *c-w* colormap (c) *3D perlin noise* with *plasma* colormap

Figure 2: Examples of 3D cellular automatas.

Introduction

I was first introduced to neural cellular automata last year after watching this video [1] titled *"What are neural cellular automata?"*. I was familiar with cellular automata, which are discrete, abstract computational systems [2]. They are discrete, meaning they are composed of a finite set of homogeneous, simple units (referred to as cells), and they are abstract, which means they can be specified in purely mathematical terms and can be implemented using physical structures. However, the notion of *neural* cellular automata builds upon this, with a cell's state now able to be any number between 0 and 1 and its update calculated using convolution and an activation function. Below is the general formula applied to each cell using a $3 * 3$ kernel:

$$g(x, y) = w * f(x, y) = \sum_{a=-1}^{1} \sum_{b=-1}^{1} w(a, b) * f(x + a, y + b)$$

Where $g(x, y)$ is the next state, $f(x, y)$ is the current state, and w is the $3 * 3$ kernel (centered about 0,0). Once this convolution has been calculated, the result is sent through an activation function to get

the final next state of a cell. A cell can only be within the inclusive range of $(0, 1)$, so any result after applying the activation function is clamped between this range.

This slight difference leads to incredibly complex and almost life-like structures. This website titled *neuralpatterns.io* was a great inspiration for this project and showcases many different types of neural cellular automata [3]. These resources feature neural cellular automata from an academic perspective and additionally use machine learning to train their automata to grow into specific pictures [4][5].

I also wanted to explore applying this technique to 3D cellular automata. In order to do so, I implemented volumetric rendering using ray-marching to be able to visualize voxel cells inside a cube. I used various resources [6][8] and tutorials [7] to help implement my project, most notably this article titled *"Volume Rendering with WebGL"* by Will Usher [9]. Below is the front-to-back alpha compositing equation:

$$C(r) = \sum_{i=0}^{N} C(i\Delta s) * \alpha(i\Delta s) \prod_{j=0}^{i-1} (1 - \alpha(j\Delta s))$$

Where $C(r)$ is the final calculated color, $C(i\Delta s)$ is the emitted color at point $i\Delta s$, $\alpha(i\Delta s)$ is the absorption at point $i\Delta s$, and Δs is the step size through the volume. The product series inside the summation series is the attenuation term calculated at each sample point. In practice, this is generalized to a front-to-back iterative computation:

$$\hat{C}_i = \hat{C}_{i-1} + (1 - \alpha_{i-1}) * \hat{C}(i\Delta s)$$

$$\alpha_i = \alpha_{i-1} + (1 - \alpha_{i-1}) * \alpha(i\Delta s)$$

These equations use a pre-multiplied opacity for correct blending [9]:

$$\hat{C}_i(i\Delta s) = C(i\Delta s) * \alpha(i\Delta s)$$

To render this for each pixel, a ray is traced from the eye through the pixel and through the volume, performing the above calculations at each iteration inside the volume to get a final color and opacity.

Implementation

I implemented my project using the same WebGL base as was done for projects 2, 3, and 4 in this course. However, I decided not to use a majority of the provided code in *webglutils* (RenderPass.ts, CanvasAnimation.ts, Objects.ts, etc.), opting to build my application mostly from scratch. (I did end up using Camera.ts) The project is split into two sections, one for 2D rendering and the other for 3D rendering (dubbed app2D and app3D). Users can toggle between the two sections using the space button.

The 2D section is very reminiscent of *neuralpatterns.io*, utilizing the entire window as a canvas and every pixel as a cell. All calculations (including convolution and activation) are performed in the fragment shader to allow for efficient parallelism. The update loop uses the actual pixel RGBA values of the canvas as the previous state values $f(x, y)$ using *gl.readPixels(...)*. Additionally, as is done in *neuralpatterns.io*, every other update is skipped in the rendering process which reduces flickering and artifacts significantly. In essence, the draw loop for the 2D section can be summarized as such:

```
draw_loop() {
    this.draw()
    this.read()
    this.draw()
    this.read()
}
```

In my project, I included 9 different 2D automata to explore: *worms, drops, waves, paths, stars, cells, slime, lands,* and *game-of-life*, each using a unique $3 * 3$ kernel and activation function. Additionally, I implemented 4 different shader types to visualize the automata: *rgb, alpha, bnw.* and *acid*. The shader types *alpha* and *bnw* perform a single convolution per pixel, while *rgb* performs 3 separate convolutions per color value, creating a 3-layered visualization which I find very pleasing. The *acid* shader was created for fun and uses time-based calculations to create some interesting patterns. Users are able to swap between automata and shader types in real time and use the mouse to draw/erase cells as well. Some examples of 2D automata/shader combinations can be seen in Figure 1.

The 3D section uses a cube as the space where 3D voxel cells interact with one another. For my submission, I have the resolution of the cube to be fixed at $64 * 64 * 64$. Calculations are done on the CPU but on a separate worker thread from the main rendering thread. This allows for state iterations to occur asynchronously from the rendering process, meaning that state updates usually take a few frames. This works out well, as visualizing each iteration per frame would be too fast and difficult to perceive. As is mentioned in *"Volume Rendering with WebGL"*, only the back faces of the cube are rendered. Reducing the number of calculations in the fragment shader by 50% and allowing the user to zoom into the volume without clipping past the front-facing cube faces.

I included 6 different 3D automata to explore: *grow, amoeba, clouds, arch, caves,* and *crystal*; 3 static volumes: *sphere, organized,* and *random*; and a 3D perlin noise visualizer. Additionally, there are 6 different colormaps to choose from: *cool-warm, plasma, virdis, rainbow, green,* and *yellow-green-blue.* Users are able to swap between automata and colormaps in real time and use the mouse to rotate and zoom in/out of the cube. Some examples of 3D automata/colormap combinations can be seen in Figure 2.

There are a few other controls that the user has access to, such as pressing *(p)* will pause/resume the automata, pressing *(r)* will reset the current automata, and pressing *(m)* will randomize between all possible states (2D/3D, automata type, shader/colormap). This project is currently hosted on my website *here* for anyone to try out.

Limitations

There are a few key limitations I would like to address, the main one being that the 3D cellular automata are not neural-based like the 2D automata are. They instead use the classical Moore or Von Neumann cell neighbor counts to calculate their next state. This was primarily due to the fact that my 3D neural cellular automata implementations were extremely unstable. They would only exist for a few iterations and either completely dissipate into all empty cells or fill the volume with all filled cells. Following the submission of this project, I hope to explore this problem further to determine if stable 3D neural cellular automata are possible.

Additionally, even though I spent a considerable amount of time optimizing my calculations to run smoothly, there is still the issue of performance. My 2D implementations use the literal screen pixels to store cell state information, and because I am skipping every other frame, my update program is run twice per frame. This severely hinders the performance of my 2D automata. (They still run at a smooth 80 fps on my computer, but it could run at an even smoother 160 frames if implemented better.) My 3D implementation could also be re-implemented using compute shaders for each voxel cell to perform the state update function. (At the moment, the worker thread works tirelessly using multiple for loops to inefficiently do this.) My ray-marching algorithm calculated in the fragment shader could also use some improvements. Zooming in closer toward the cube causes fps drops, though it does not fall below 60 fps for any 3D automata.

References

1. Emergent Garden. "What are neural cellular automata?", YouTube (2021), *[link]*
2. Berto, Francesco and Jacopo Tagliabue. "Cellular Automata", The Stanford Encyclopedia of Philosophy (2012), *[link]*
3. Max Robinson. "neuralpatterns.io". (2021), *[link]*
4. Mordvintsev, et al. "Growing Neural Cellular Automata", Distill, (2020), *[link]*
5. Whitaker, Jonathan. "Fun With Neural Cellular Automata", (2023), *[link]*
6. Demo Volumetric Rendering of 3D perlin noise *[link]*
7. Milan Ikits et, al. "Volume Rendering Techniques", GPU Gems, Chapter 39. Volume Rendering Techniques (2007), *[link]*
8. softologyblog, "3D Cellular Automata", (2019), *[link]*
9. Will Usher, "Volume Rendering with WebGL", (2019), *[link]*