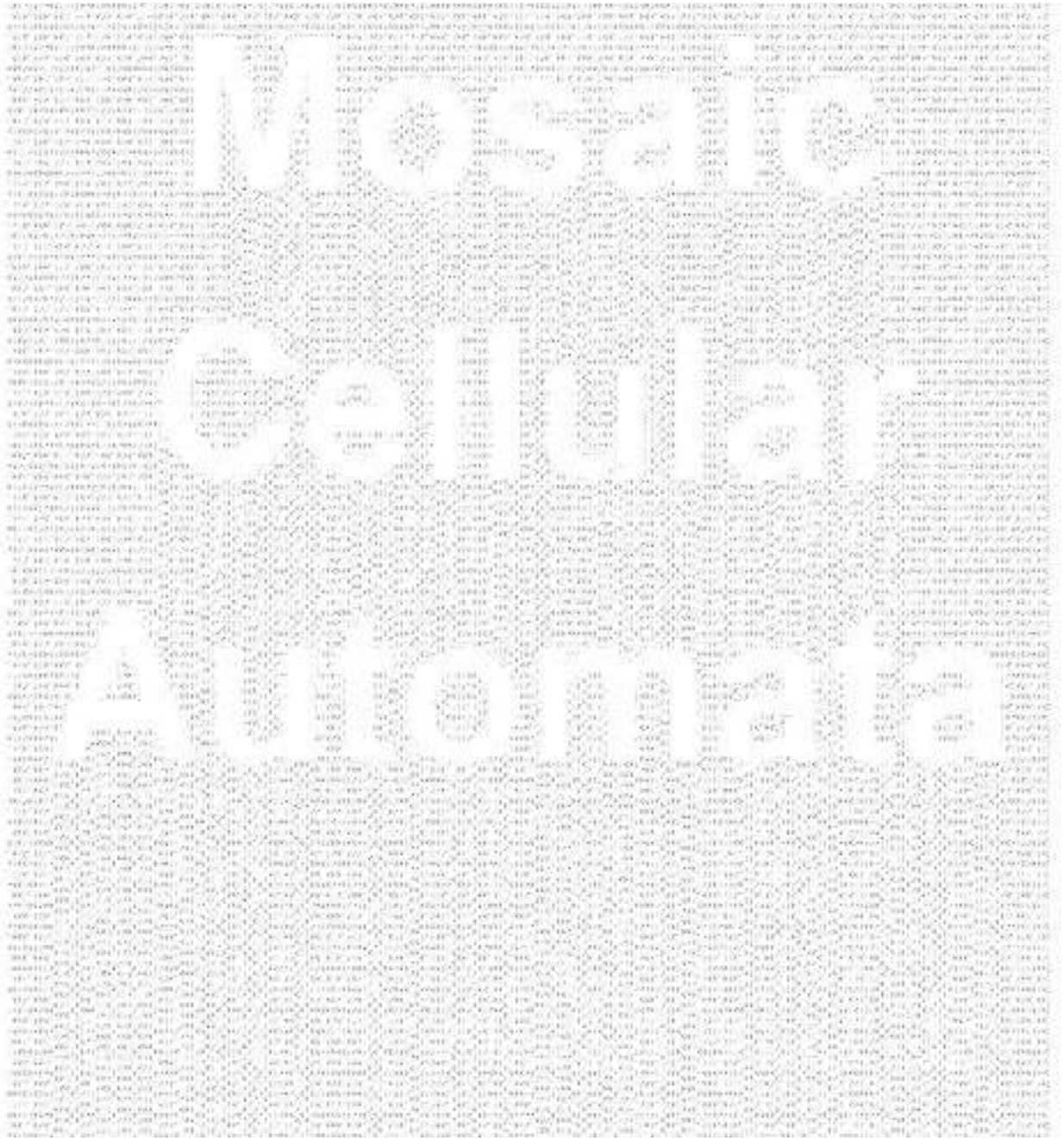

Max Proske (301235293)
Ian Springer (301209105)

IAT 455: Computational Media
Simon Fraser University
August 4th, 2016

Abstract

Mosaic Cellular Automata aims to recreate an image as a living, breathing, infinitely sustainable simulation of Conway's "Game of Life," a simulation in which a grid of pixels can be alive, dead, killed, and born depending on the state of their neighbours. Given an image as input, our method creates an initial grid of alive and dead pixels, and runs the result as a "Game of Life" simulation, requiring no further input.

Research

LifeWiki was an invaluable resource that contains the most comprehensive catalogue of cellular automata patterns available on the internet. The documentation found on this wiki assisted in compiling a detailed list of the 250 most common oscillators. <http://conwaylife.com/wiki>

Oxman's documentation of cellular automata rules allowed us to program and optimize Conway's Game of Life in Java code. However, our program expanded beyond the scope of Oxman's documentation, and required further programming to track the pixel value of cell groups and reanimated cells. <Oxman, G., Weiss, S., Be'ery, Y. 2013. *Computational Methods For Conway's Game of Life Cellular Automaton*>

Hua and Pelikan's documentation helped us compile a list of complex life forms that fit within a 6x6 grid. This allowed us to randomly generate oscillators, without worrying about neighbouring oscillators setting off a chain reaction tearing apart our image. <Hua, D., Pelikan, M. 2012. *Variations on Conway's Game of Life and Other Cellular Automata*>

Development

Max Proske: Image preparation including oscillator propagation

Oscillators

An oscillator is a pattern of pixels that die and are born in a perfect equilibrium, such that it sustains itself forever. The smallest oscillators that have an interesting lifespan are 6x6 pixels in size. Therefore, we enlarge our image by 6x, and replace each "pixel" with a 6x6 oscillator.

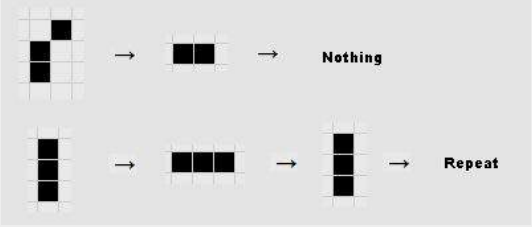

Figure 1. Oscillators example

Ordered Dither Threshold Filter

Ordered dithering uses a threshold map to fit an image to a limited colour palette, by placing dots of two colors close together to simulate the shade of a third color.

However, we are not interested in recoloring our image. Instead, we are interested in the byproduct, a distinct dithered crosshatch pattern (Figure 2). We suppress the RGB values below the calculated dither threshold value, allowing us to calculate the population density of a neighbourhood of pixels.

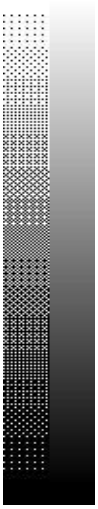

Figure 2. Dither pattern

Figure 3. Original Image

Figure 4. Ordered dither threshold filter

Population Density Matte

From here, we iterate through the image once again to assign a pixel value, based on the number of adjacent neighbouring pixels (Figure 6). White is 4 neighbours, and black is no neighbours. This allows us to group pixels into similar neighbours, which will be useful in the next step.

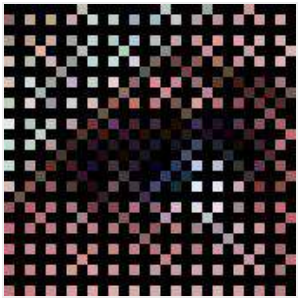

Figure 5. Original pattern

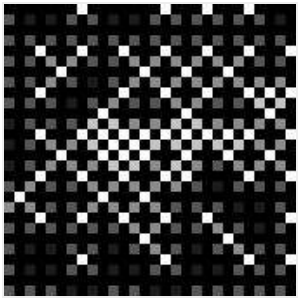

Figure 6. Matte pattern

Similar Neighbour & Bounding Box Mattes

The next matte merges pixels with similar neighbour density into cells (Figure 8). This allows us to combine the bounding boxes of atomic 6x6 oscillators into larger oscillators. However, there is still some overlapping between oscillators. This is a problem, because if the program ran now, the image would rip itself apart. To resolve this, the Bounding box matte (Figure 9) calibrates order of importance between overlapping cells. Finally, all leftover unmerged cells become atomic 6x6 oscillators, and each pixel becomes an instruction of where to place a specific width and height oscillator (Figure 10).

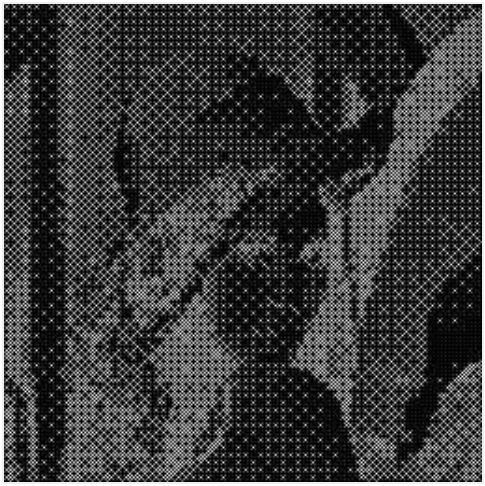

Figure 7. Population density matte

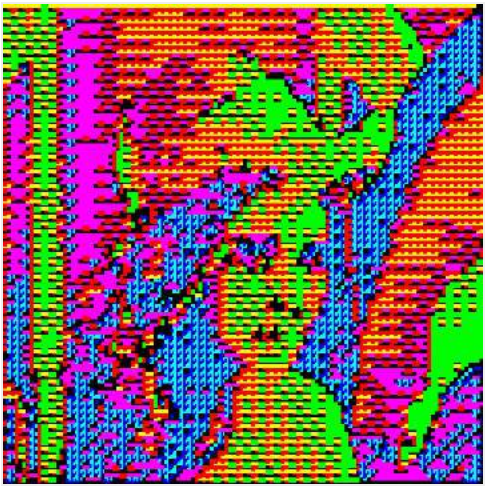

Figure 8. Similar neighbour matte

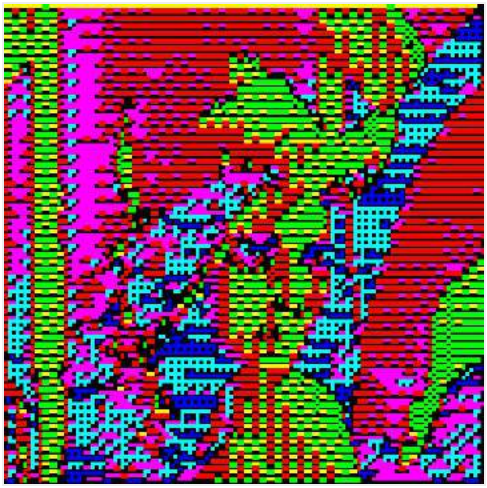

Figure 9. Improved Similar neighbour matte

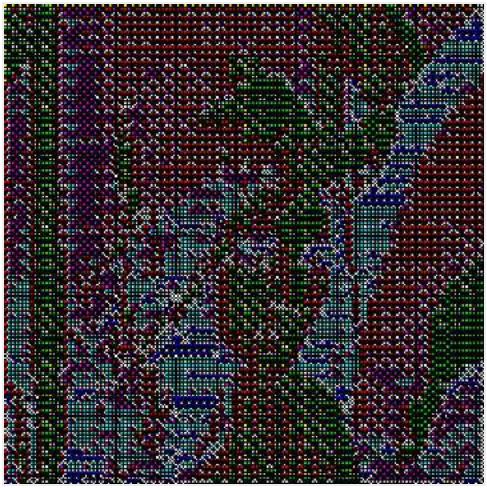

Figure 10. Bounding box matte

Alpha Matte

The Bounding box matte is then applied an alpha matte, to give the alive cells an alpha value of 255 and the dead pixels an alpha value of 0. The final image is then run through a loop that compresses the image into an arraylist of colors pixel by pixel, and passes the data to the "Game of Life" object for simulation.

Ian Springer: Structure and efficiency for the Game of Life, displaying data

The arraylist is fed into the game of life, where they're sorted into alive or dead cells based on their alpha value. Once the game has been loaded, it is run. Differing from our last implementation, the tick is now one iteration of the array. This is possible due to knowledge how how a cell calls its neighboring cells and the ability to store a previous state upon change. So, when a cell update it has cells that updated before it and cells that are yet to update after it. For all the cells before, the cell will call upon their previous state, and for the cells after, it will call

upon their current state. This is to ensure that the cells don't incorrectly check a neighbor's state. Then, once a cell tabulates its neighbors, it's told to update its own state. When doing so, it saves the current state as a previous state and then updates its current state based upon how many valid neighbors it counted. This process continues for each cell in the array until it's been iterated over.

After this, the game has been ticked fully and the program can continue. The next step is drawing it, and this process was separated out from the tick for the ability to draw multiple versions of the same game without having to tick multiple times. Essentially, the method just runs over a specified section of the array and tells each cell to draw itself at a specified location, allowing us to draw close up views without needing to iterate over the entire array.

Final Result

After weeks of planning, prototyping, and iteration, our program works as intended. It can take a single image as input, and apply our own original filters and mattes to populate a 2D grid with lifeforms that are self-contained. The final result is a living, breathing image that accurately resembles original image.

The hardest part was working out both how the neighboring pixels were to be turned into complex oscillators and how to make the game run more efficiently, but through dedication to tackling each problem individually we were able to overcome them to create a better end product.

Figure 11. Mosaic Cellular Automata

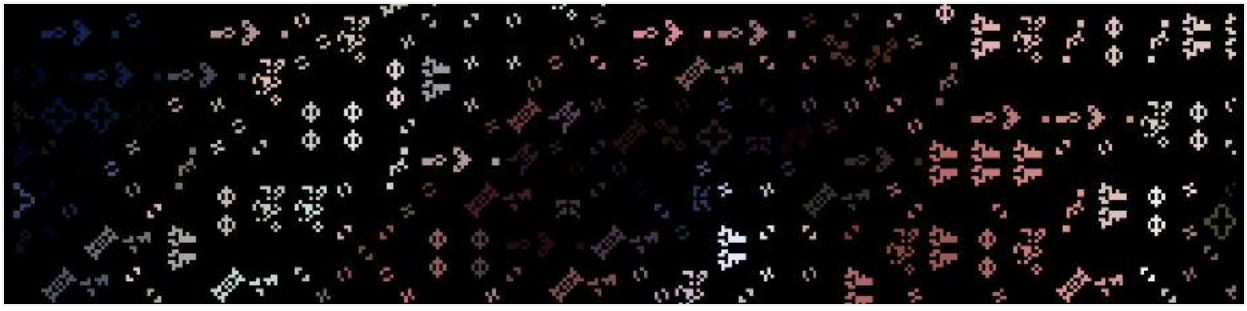

Figure 12. Mosaic Cellular Automata close-up

Conclusion

While our project idea initially felt like an unobtainable goal, through constant iteration and dozens of hours of research on the structure of the “Game of Life,” we were able to build upon previous works to produce Mosaic Cellular Automata. Our final method is an automated way of preparing a sustainable state from a single image. Not only this, but the simulation state features complicated mechanisms, and interaction between cells within it, to help introduce variety and an ethereal quality of life made up of nothing but tiny little cells that can either be alive or dead.

Overall, we managed to break down an overwhelmingly large application into manageable chunks, to successfully implement our vision while successfully maintaining image integrity.