Shape Machine: Conway's Game Of Life

William Braga

I. Abstract

Conway's Game of Life (Life) is a popular deterministic cellular automaton invented in 1970. The game features a grid made up of square cells. Cells may be born or die depending on the state of their eight neighbors; these rules were created to simulate the effects of underpopulation, normal growth, and overpopulation. Each rule can be written as a function determining whether a given cell will be alive or dead in the following iteration with respect to the number of its neighbors that are currently alive. The player of the game only acts in two ways: setting an initial configuration of the grid and stepping through iterations. Life is renowned for the seemingly-random yet reproducible designs that can be formed.

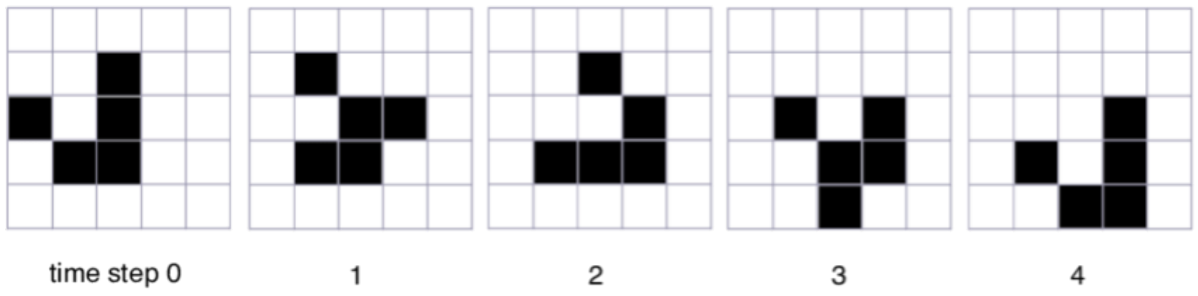
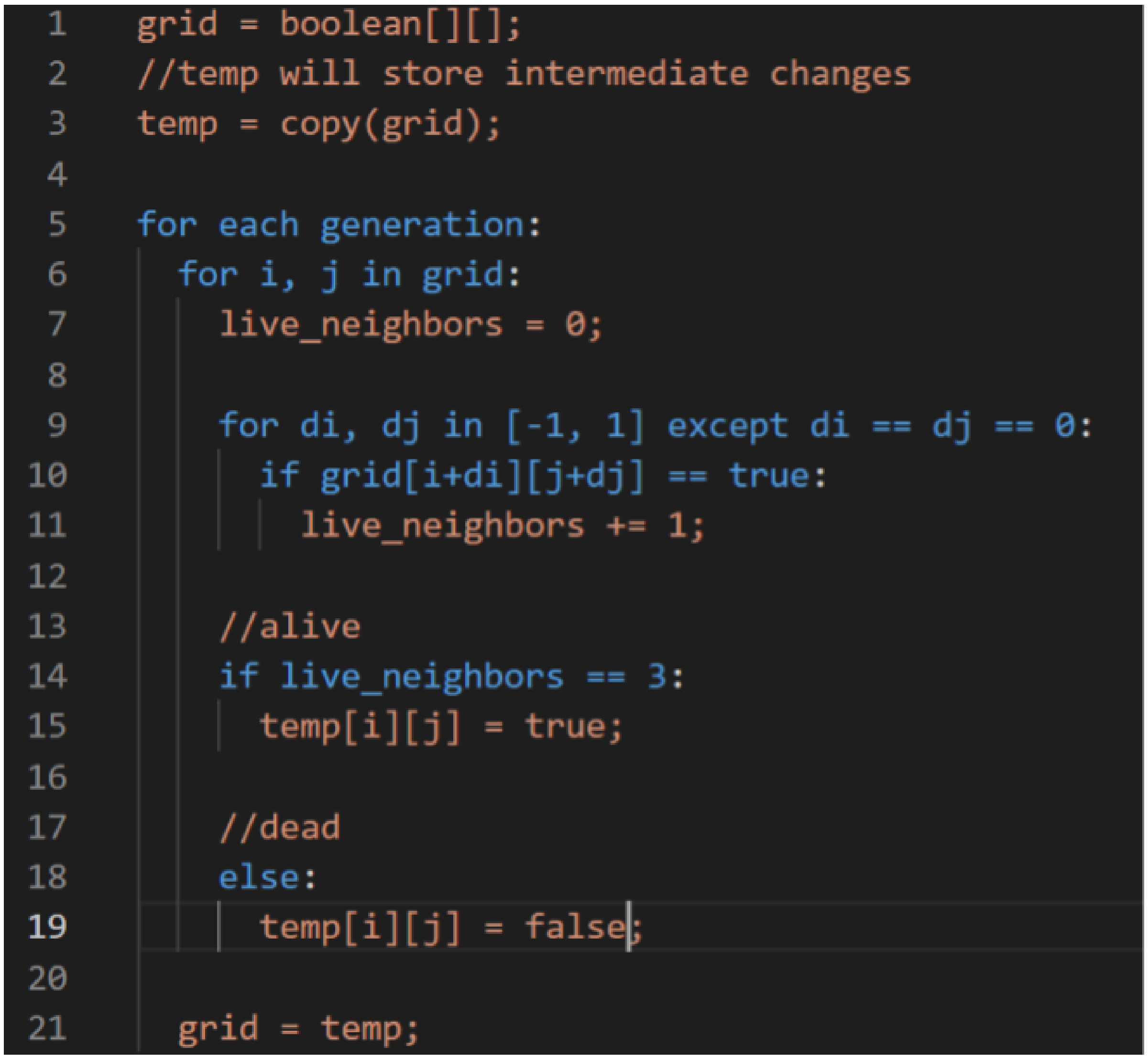


Figure 1: Glider Pattern

We conjecture Life can be implemented in the Shape Machine using a non-traditional approach. Usually, Life is implemented programmatically by iterating through each cell, counting the number of alive, adjacent cells it has, and updating its state according to the rules. In Shape Machine, we will instead use a shape grammar approach, where iterative updates are dependent solely on geometric conversions with no literal counting. These shape grammar rules will generally be defined with 3x3 cell grid shapes and markers on each cell to indicate state.

II. Challenges

There are two main considerations for the Shape Machine's Game of Life. First is technical: how should discrete iterations be handled to allow simultaneous updates? From a programmatic approach, at each iteration, a copy of the grid is made at the beginning and then merged at the end; all updates are determined using the original grid and applied to the copy. Next is efficiency: what can be done to reduce the number of explicitly defined shape rules and what can be done to improve processing time? Exploiting shape symmetry and counting logic can drastically reduce the set of rules that must be passed to the Shape Machine while retaining full functionality. For processing time, grid and shape designs must be considered in order to reduce computational load.



```
grid = boolean[][];
//temp will store intermediate changes
temp = copy(grid);

for each generation:
  for i, j in grid:
    live_neighbors = 0;

    for di, dj in [-1, 1] except di == dj == 0:
      if grid[i+di][j+dj] == true:
        live_neighbors += 1;

    //alive
    if live_neighbors == 3:
      temp[i][j] = true;

    //dead
    else:
      temp[i][j] = false;

  grid = temp;
```

Figure 2: Sample pseudocode Game of Life implementation

III. Planning

a. Structure

Implementing Life in Shape Machine requires planning on three layers of abstraction: finding all geometric cell-update rules, choosing an efficient means of encoding these rules, and designing the user display.

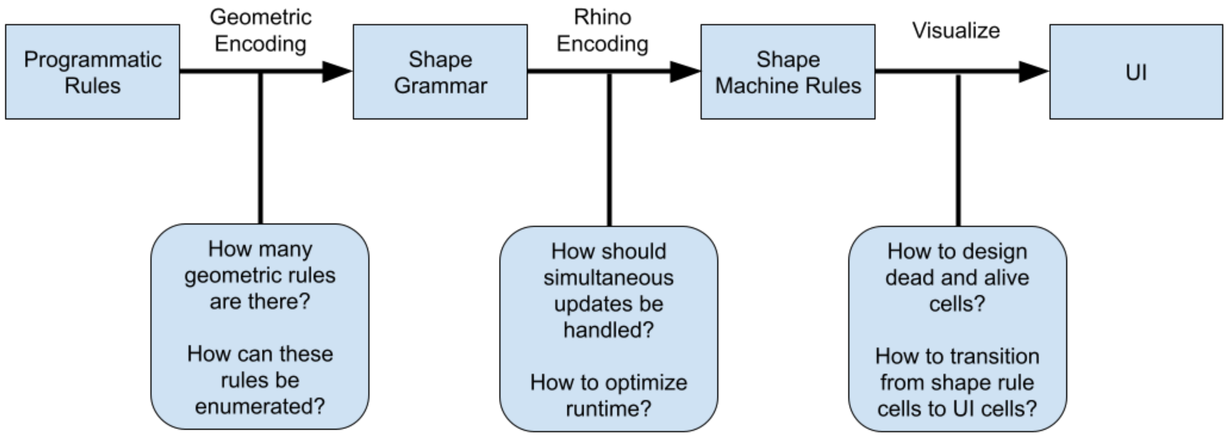

Programmatic Rules
Geometric Encoding
Shape Grammar
Rhino Encoding
Shape Machine Rules
Visualize
UI
How many geometric rules are there?
How can these rules be enumerated?
How should simultaneous updates be handled?
How to optimize runtime?
How to design dead and alive cells?
How to transition from shape rule cells to UI cells?

Figure 3: Design Process

b. Shape Grammar

The geometric design of the Game of Life rules starts with a 3x3 subgrid, where the center cell changes state depending on its eight neighboring cells. My implementation of Life in Rhino assumes the grid is sufficiently large to not need update rules around edges; all cell updates are made assuming there are always eight adjacent neighbors. In theory, the specifications for Life include an infinite grid, so no boundary conditions were officially created. Most implementations of Life choose one of two grid boundary designs: constant, where update rules do not change the values of the edges (i.e. those cells are always dead), and toroidal/modular, where patterns that leave the screen will wrap around to the other edge. If Life is able to run sufficiently fast on Shape Machine, these boundary conditions should be considered (see also: scaffold attempts). Constant boundaries can be created by having update rules reset any cells with five neighbors (edges) or three neighbors (corners) to dead every iteration. A toroidal boundary would be a significant challenge to encode in Shape Grammars. A mechanism would need to be created to broadcast a signal to the opposite edge to set those boundary cells to a certain value. The signal could be a directional label on a cell that is transferred to one of its neighbors. Setting a diagonal cell from a corner would be especially difficult. While the label is being passed, the game must pause from updating, adding more overhead to the Shape Machine.

Ignoring boundaries, there are four main geometric rule types defined by a center cell’s state and the number of alive neighbors that need to be considered: alive center and zero live neighbors, alive center and one live neighbor, dead center and three live neighbors, and alive center and four live neighbors. All other conditions either do not change the center cell’s state or would create a redundant rule (as shown in the rule enumeration). To find the number of geometric configurations a rule type has, a combination can be solved with respect to the number of live neighbors. Alive center and zero live neighbors: C(8, 0) = 1. Alive center and one live neighbor: C(8, 1) = 8. Dead center and three live neighbors: C(8, 3) = 56. Alive center and four live neighbors: C(8, 4) = 70. One way to group possible subgrids (for graphs with 3 or 4 neighbors) is to differentiate based on how many neighbors can be adjacent to each other (not including diagonals).

c. Shape Machine Rules

For the Shape Machine to recognize and apply the shape grammar, a design for subgrids must be created in Rhino. Life inherently requires much processing from the Shape Machine to iterate through all cell subgrids for each possible rule configuration. To save on time, the Rhino representation must be simple for the Shape Machine to detect and replace. In addition, rules should be condensed as much as possible using symmetry to avoid additional iterations.

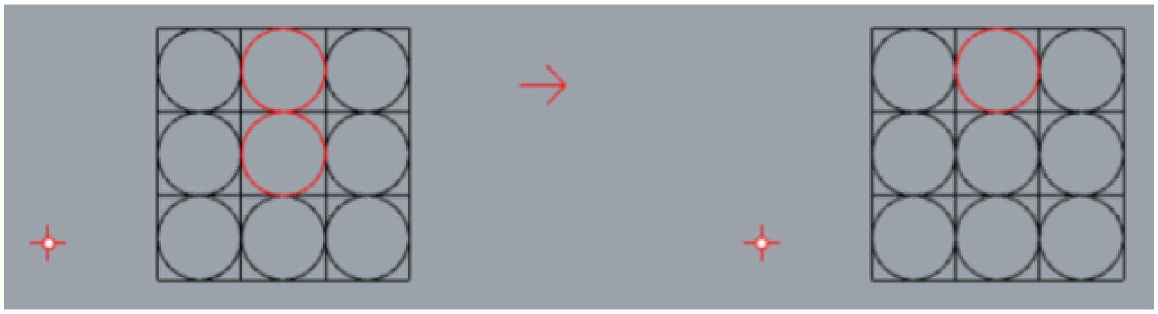


Figure 4: Shape Machine Rule

Figure 4 shows an example rule for a cell with one neighbor. The space for a cell is marked with a square. The cell state is a circle: red means alive, black means dead. With few shapes to process, the rules should be simple for the Machine to apply. The design, however, does not clearly indicate the cell state to be 1:1 with the user's front-end. Therefore, there must be a transition to a visualization stage the user will see and interact with.

The next consideration for the Shape Machine rules is handling simultaneous updates. Updates made to the grid must not affect rules applied to other cells. When programming Life, a copy of the grid is made to store updates. The original grid is queued to figure out how each cell should change. The actual changes are done to the cells of the copy. At the end, the copy is kept and becomes the original in the next iteration. The Shape Machine cannot create a copy of the grid without knowing what cells are alive or dead. Updates will have to be made on the same grid, but they cannot influence results between one rule to the next (i.e. the order of the rules should not matter). Directly using rules designed like Figure 4 would be incorrect, so modifications must be made.

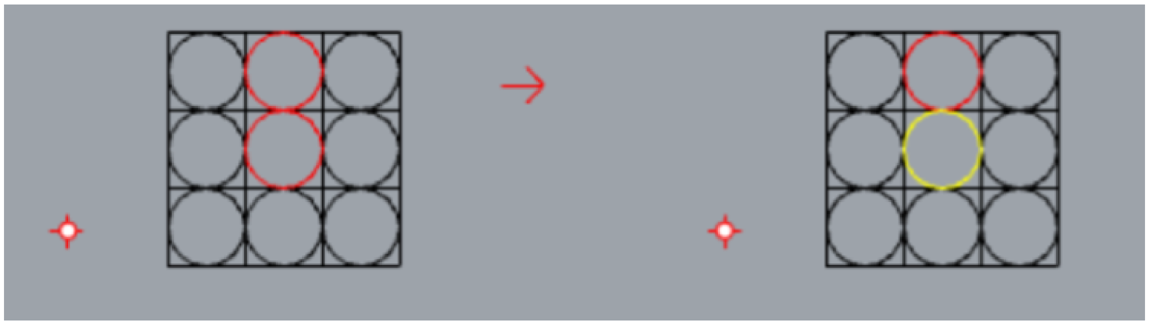


Figure 5: Shape Machine Rule - Attempt 2

Figure 5 shows an attempt at emulating a simultaneous update. The yellow circle marks a cell to-die. Cells to-live were marked by a purple circle. Only at the end of all the rules, the yellow circles will be changed to black - marking death. This attempt solves one issue but not another. Marking a cell yellow ensures that the cell will not alternate states within the same iteration. For example, the Figure 4 implementation may mark a cell as dead and then alive as neighbors are changed. Being yellow though, the cell can only change states once. The issue not solved though, is that the original state of the cell needs to be known to update its neighbors. If the Shape Machine could identify neighbors that are red/alive or yellow (having already been processed), this attempt could work. That modification can only be done in Shape Machine by copying the left hand side with all combinations of alive or yellow circles - extremely inefficient. Another design must be created that can solve both issues.

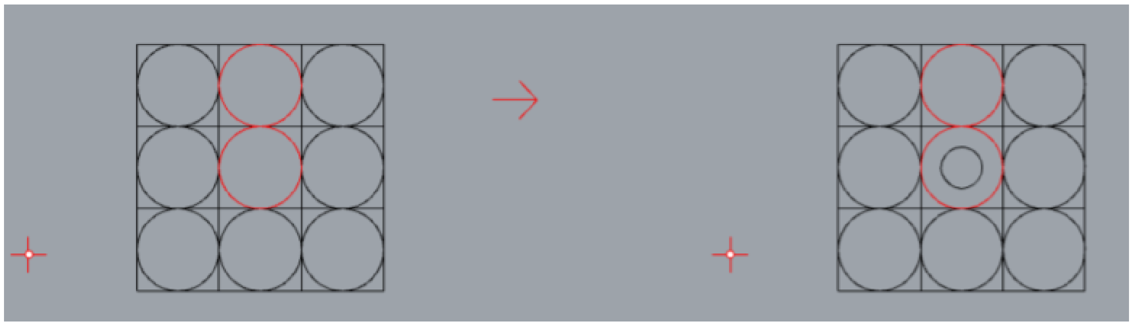


Figure 6: Shape Machine Rule - Final Attempt

Figure 6 shows my final attempt at the Shape Machine rules. It solves both simultaneous update issues. Cells cannot be updated multiple times because no rules act on the smaller inner circle that marks the state should be flipped at the end (i.e. the inner circle can only be added in one way). In addition, this design allows neighbor cells to be updated correctly because the original state still appears. The inner circle is additive, not a complete replacement. The left hand side of a rule only specifies the elements that must be included and not any that cannot appear. Therefore, the inner circle does not affect the left hand side of any rule.

d. Visualization

Ideally, the cells in Rhino could appear fully filled in to show alive cells. Also, dead cells would appear fully empty. Rhino allows a hatch to be added to a shape to fill it in, but the Shape Machine cannot act on those objects. Alive cells are modeled to be filled in using crosshatches. For the dead cells, if they are kept fully empty, later rules acting on the empty squares would apply to both dead and alive cells. This can be resolved by adding a label to all the cells and then removing labels from any cells that are alive and have a label. For simplicity, empty cells are marked with a tiny centered circle.

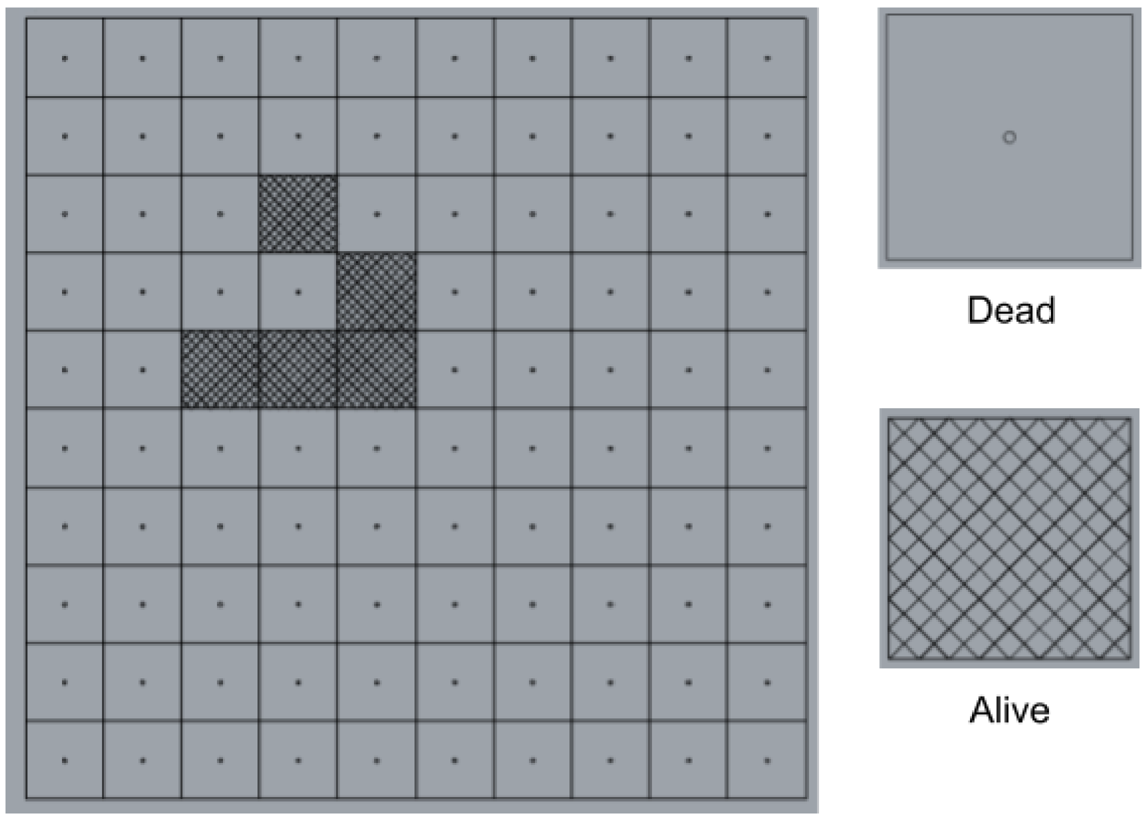


Figure 7: Grid Visualization

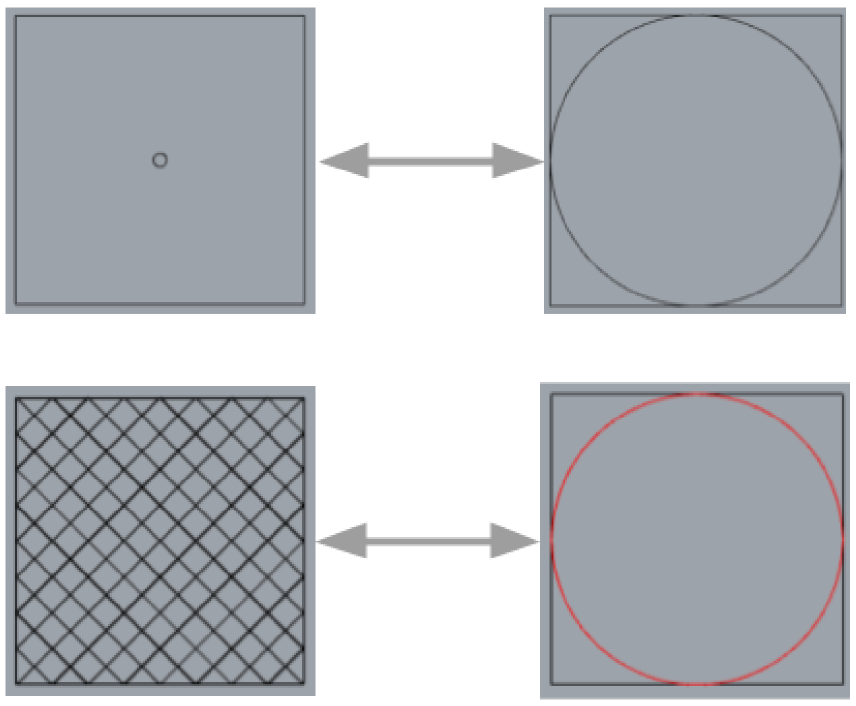
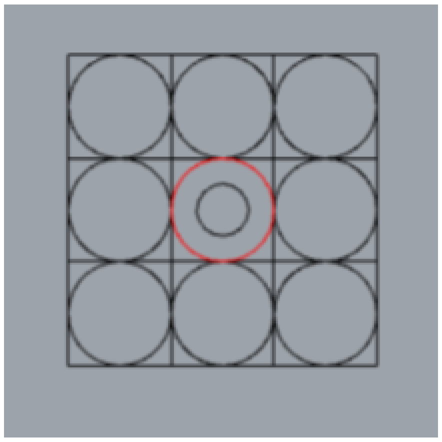


Figure 8: Visualization - Rule Conversion

IV. Rule Enumeration

The following rules depict every possible configuration of neighbors. The center cell shows how it should be updated (e.g. red circle with inner circle means it will be dead next iteration). The axes of symmetry will be stated and the number of configurations that correlates to; for example, if a graph has one axis of symmetry, it can be rotated in four ways to create four different configurations. The Shape Machine will detect those rotations, so the rule does not need to be explicit.

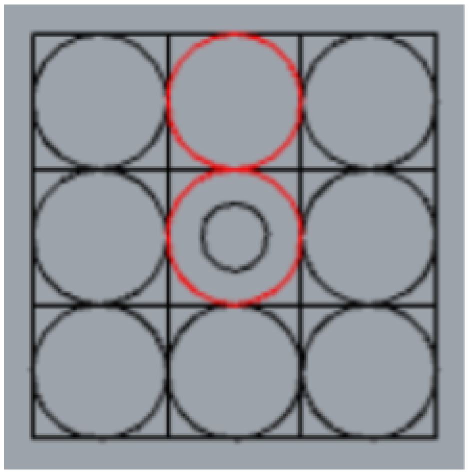
a. Zero Neighbors (1 graph)



4 axes of symmetry
1 graph

Figure 9: Zero Neighbors

b. One Neighbor (8 graphs)


1 axis
4 graphs

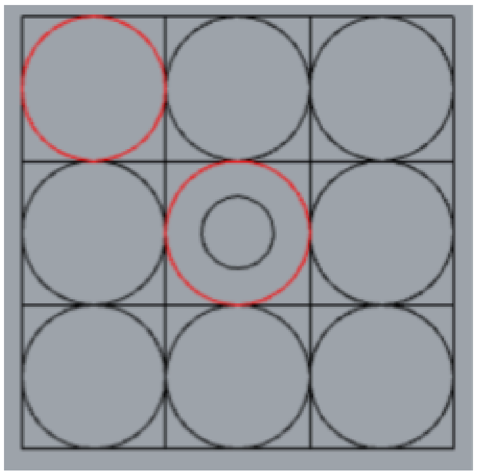
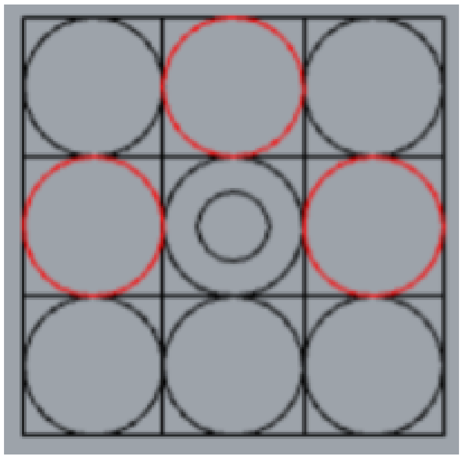

1 axis
4 graphs

Figure 10: Two Neighbors

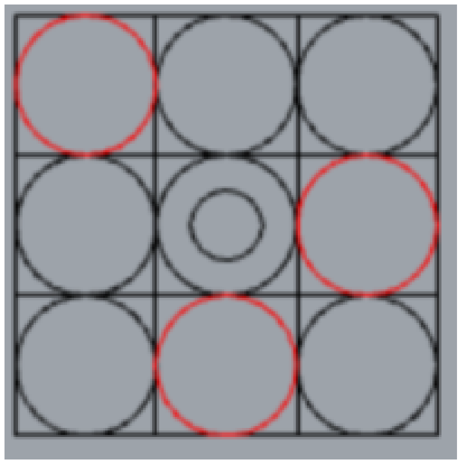
c. Two Neighbors (0 graphs)

Cells with two neighbors do not change states, so they do not need rules.

d. Three Neighbors (56 graphs)

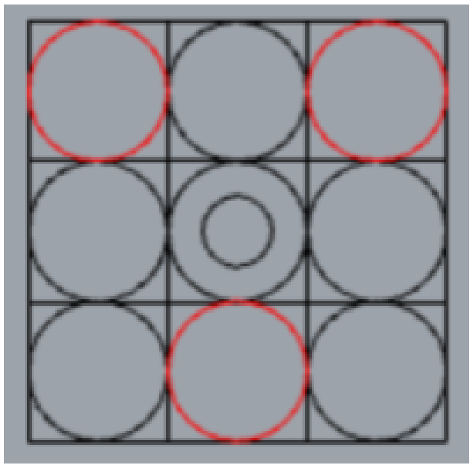


1 axis
4 graphs

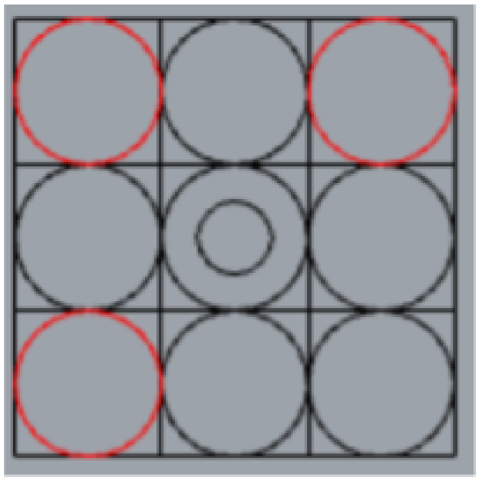


1 axis
4 graphs

Fig 11: Three Neighbors, No Adjacent, One in Top Row

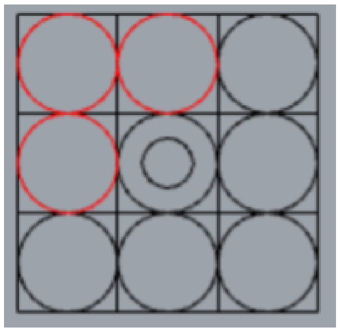


1 axis
4 graphs

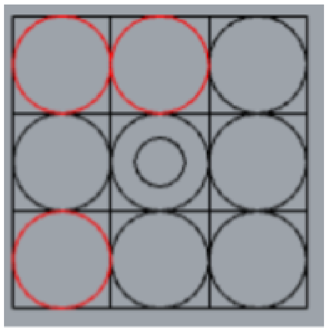


1 axis
4 graphs

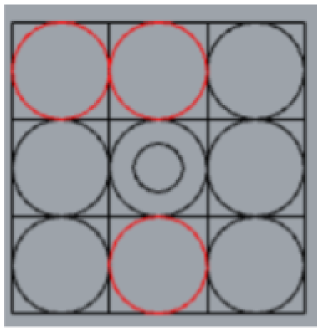
Figure 12: Three Neighbors, No Adjacent, Two in Top Row



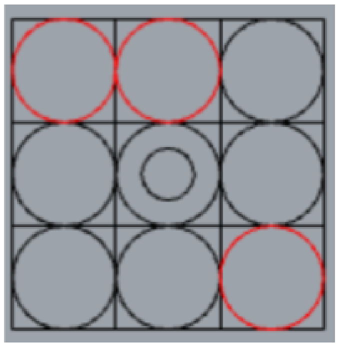
1 axis
4 graphs



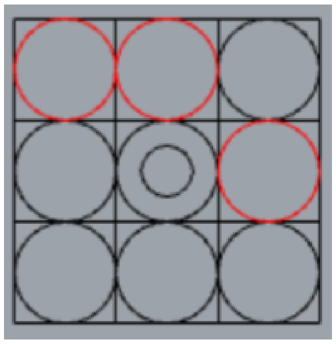
0 axes
8 graphs



0 axes
8 graphs

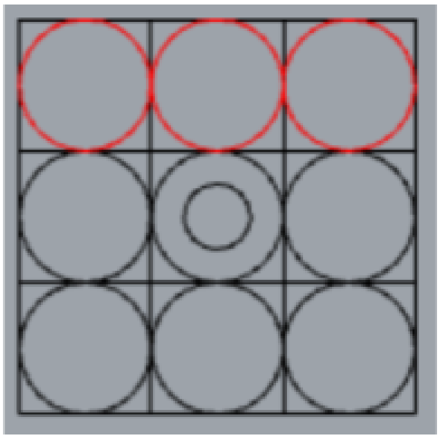


0 axes
8 graphs



0 axes
8 graphs

Fig 13: Three Neighbors, One Adjacent, Two in Top Row



1 axis
4 graphs

Fig 14: Three neighbors, Two adjacent, Three in Top Row

d. **Four Neighbors (70 graphs)**

Graphs with four neighbors do not show any dead cells. This is because these graphs can work with any configuration with *at least* four neighbors. The positions where dead cells would be filled for an exactly four-neighbor graph are left ambiguous.

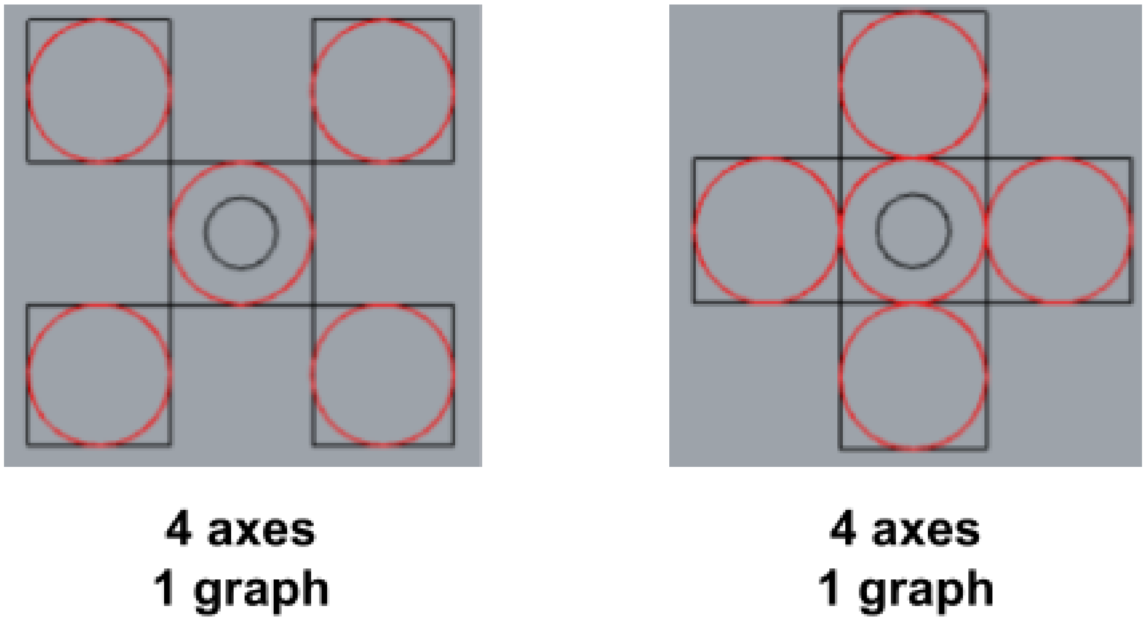


Figure 15: Four neighbors, No adjacent

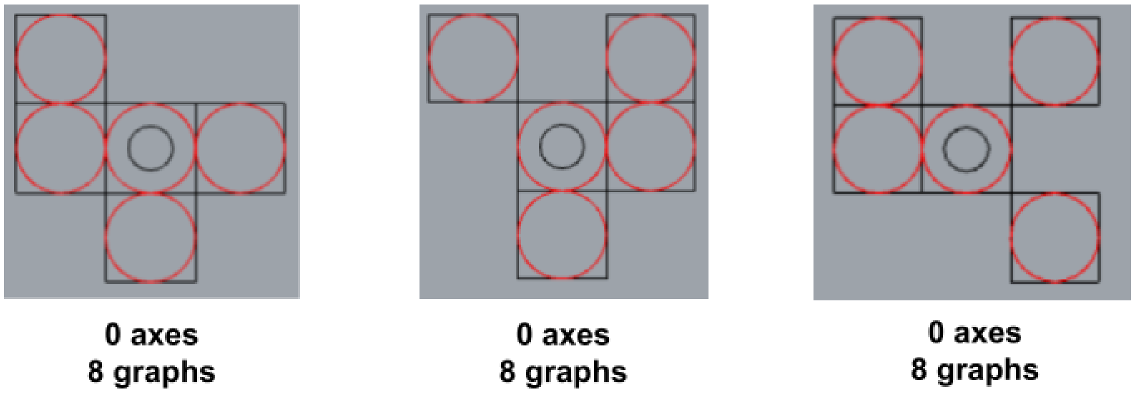
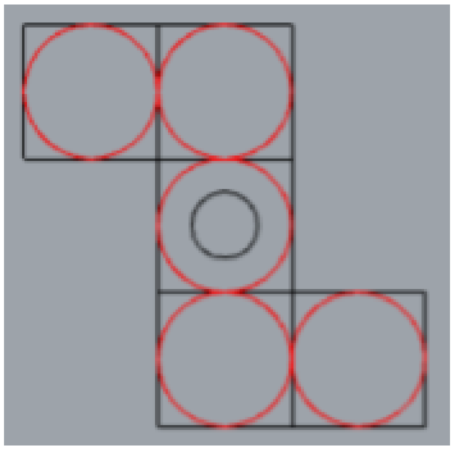
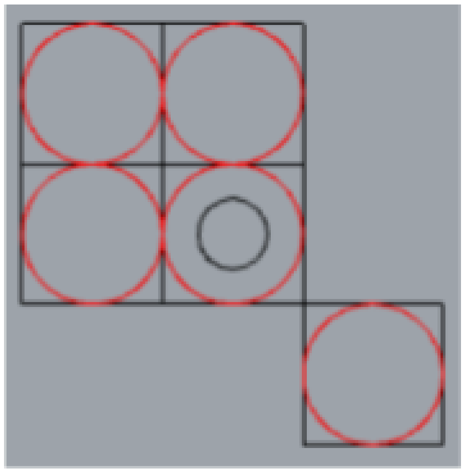


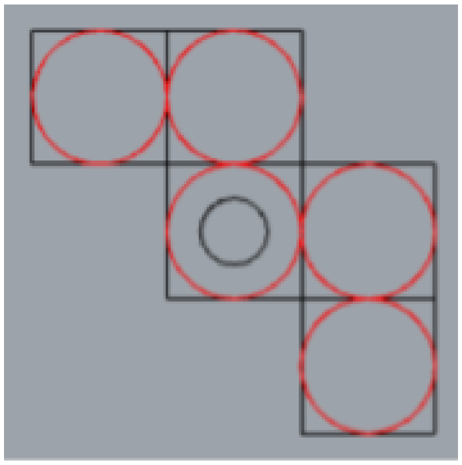
Fig 16: Four neighbors, one pair of two adjacent



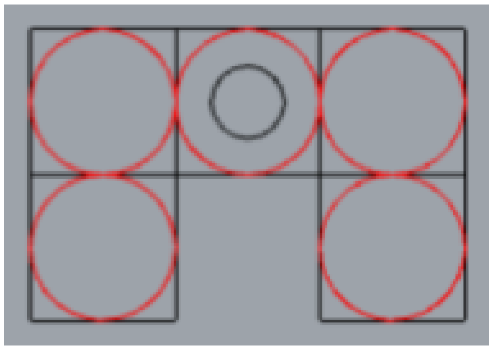
1 axis
4 graphs



1 axis
4 graphs



1 axis
4 graphs



1 axis
4 graphs

Figure 17: Four neighbors, two pair of two adjacent

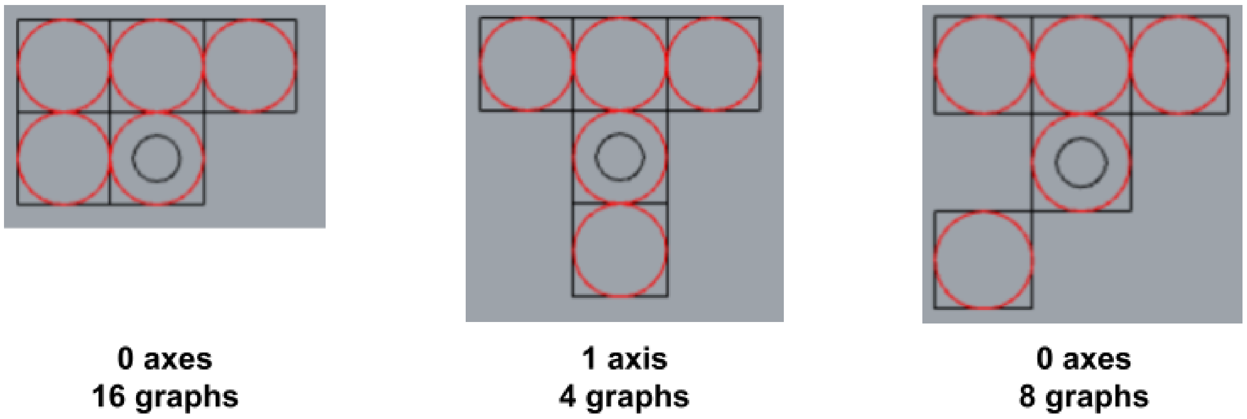


Fig 18: Four neighbors, three adjacent

e. **5-8 Neighbors (0 graphs)**

These graphs are rendered redundant by the 4 neighbor rules.

V. Results

As currently implemented, Life works on the Shape Machine. A user may create a grid of squares, set squares to alive or dead, and run the shape machine for each generation. All the rules have been counted out and implemented (for non-boundary cells) and simultaneous updates work properly.

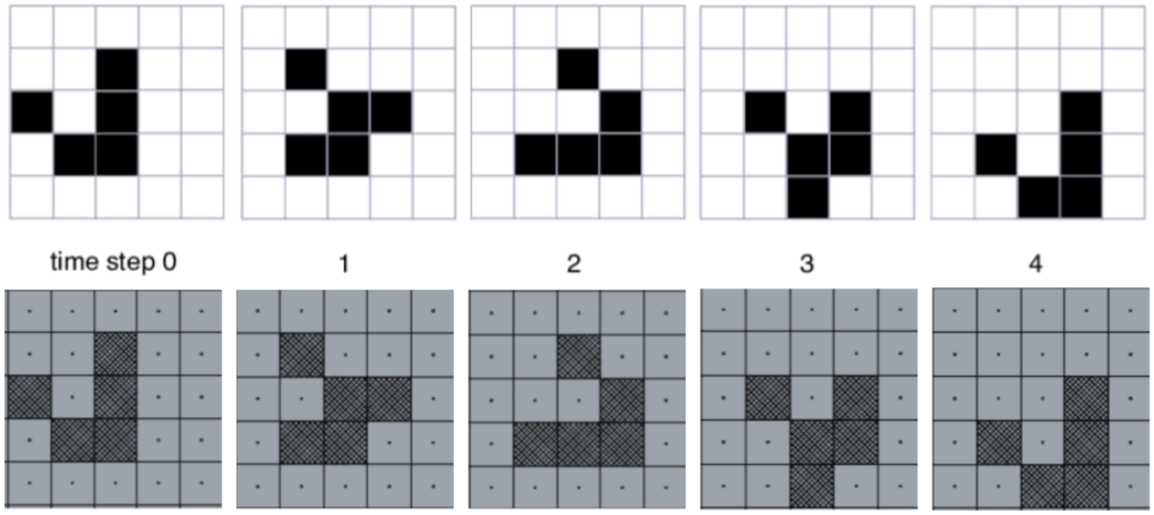
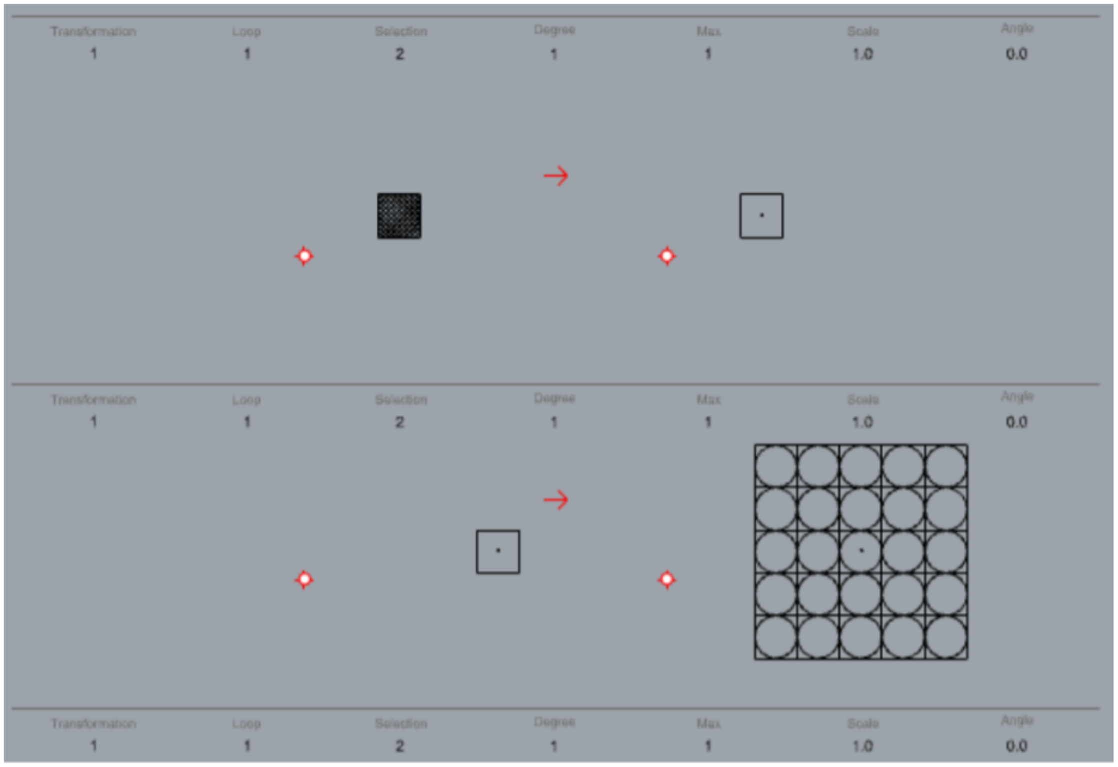


Fig 19: Gliders (Above: Life Wiki, Below: Shape Machine)

The main caveat, unfortunately, is the runtime. The images in figure 19 each took almost 10 minutes to create; that is with all the optimizations discussed. Due to the constraint, this implementation of Life would not suit as a game or an able exploration of more complicated patterns.

VI. Future Work

After these results, I attempted to create a *scaffold* optimization. Instead of running the machine on a static, fixed-size grid, have the grid only contain alive cells and no spaces for dead cells. When the machine runs, it will pad each alive cell with two layers of dead cells so that all updates can be made. After the simultaneous updates, all dead cells are removed. The scaffold machine did run faster, but there were two issues. First, removing dead cells was not simple. Since the squares need to be completely taken out, they often remove edges from cells that are alive (or remove them from dead cells making them no longer recognizable as a square). I attempted to color code the edges of alive vs. dead cells and remove all dead edges, but the Shape Machine could not properly remove all straight edges from the grid. Second, after testing the speed of the scaffold changes, I found that the speed improvement was not remarkable. On further inspection, I realized that the main bottleneck is converting the crosshatched alive cells into the back-end red circle representation. I believe it has to do with the 4 axes of symmetry and number of construction lines. The symmetry is essential though to match the symmetry of the back end rules. Having a mismatch results in the final pattern being produced four times - accounting for the ambiguity. In terms of visibility, I could not find a pattern that was as visible as the crosshatch and allowed a more efficient runtime. Any future work should first target the visualization stage for improvement.



Transformation	Loop	Selection	Degree	Max	Scale	Angle
1	1	2	1	1	1.0	0.0

Transformation	Loop	Selection	Degree	Max	Scale	Angle
1	1	2	1	1	1.0	0.0

Transformation	Loop	Selection	Degree	Max	Scale	Angle
1	1	2	1	1	1.0	0.0

Fig 20: Scaffold Padding

VII. References

Conway’s Game of Life borders rules. (2016, March 15). Mathematics Stack Exchange. https://math.stackexchange.com/questions/1699282/conways-game-of-life-borders-rules/1699329

GitHub - maxlancaster/game-of-life-with-grammar-seed: An implementation of Conway’s Game of Life seeded by basic 2D Shape Grammars. (2018). GitHub. https://github.com/maxlancaster/game-of-life-with-grammar-seed

Knight, T. K. (2002, January). Computing with emergence. MIT. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.466.3568&rep=rep1&type=pdf

LifeWiki. (n.d.). LifeWiki. https://conwaylife.com/wiki/Main_Page