The Meaning of (Conway's) Life

Joshua Turner, CSC 207, Spring 2020

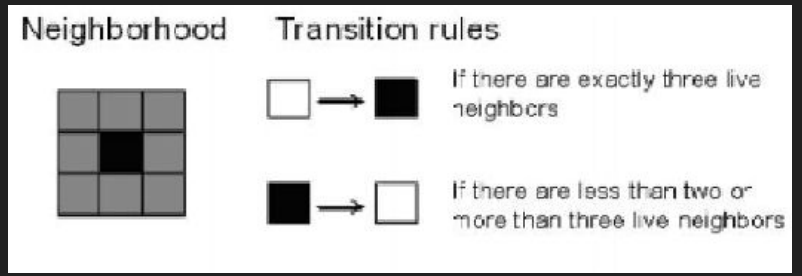
Key Terms

- Conway’s Game of Life
- Cellular automaton
- Hashlife
  - Quadtree
  - Canonicalization
  - Memoization

Imagine...

that we have a grid of cells, which can be either “alive” or “dead”, equipped with the following rules for calculating the next state of the grid:

- Live cells with exactly two or three live neighbors stay alive
- Dead cells with exactly three live neighbors come to life
- All other cells go to the dead state

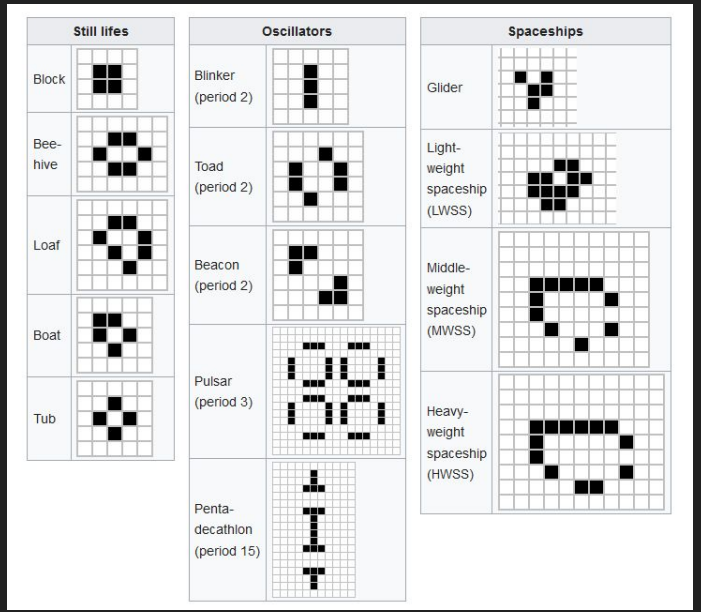


Think for a minute: what would happen? Try some small examples for yourself.

Conway's Game of Life

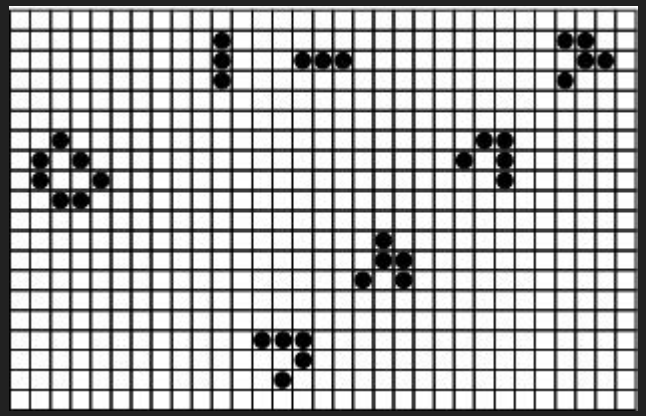
- Fascinating and complex behavior emerges!
- Recognizable, recurring structures that have diverse behavior mixed in with chaos
  - Still lifes are stable structures that do not change from generation to generation
  - Oscillators cycle through a set of states
  - Spaceships fly across the grid
- This is called *Conway's Game of Life*

Look at the Block and Blinker. In light of the rules from the previous slide, can you see why they exhibit the behavior they do?



Cellular Automata

- *Life* is a specific case of a more general class of structure called a *cellular automaton.*
- A cellular automaton can have *any* set of rules for evolving the grid, as well as any number of states.
    - More complex automata can also encode specific transitions for *specific* neighbor states, rather than any configuration with the same number yield the same result

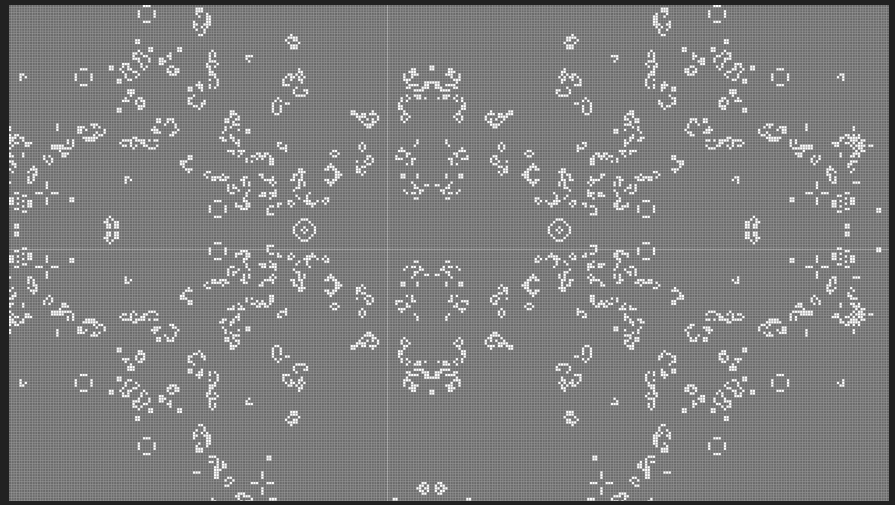


A still image just doesn't capture it. Conway's Life is hypnotizing to watch.

One of my favorite cellular automata solves mazes!

So, what is this project?

- My goal was to create an environment in which users could experiment and play with simple cellular automata.
- This includes interacting with the grid by drawing, creating rules for grid evolution, and watching the grid evolve through automatic playback.



Big Questions

1. How do we compute evolutions for an apparently infinite grid?
2. How do we do it in such a way that is not absolutely disgusting, efficiency-wise?

What would you *do?*

Hashlife: A Truly Awesome Algorithm

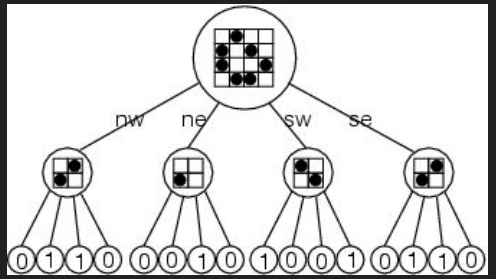
- *Hashlife* is here to save the day.
- By taking advantage of the fact that CA's are deterministic and have frequent regular patterns, Hashlife speeds up the process.
- Here are the key insights of Hashlife:
  - Quadtrees
  - Canonicalization
  - Memoization

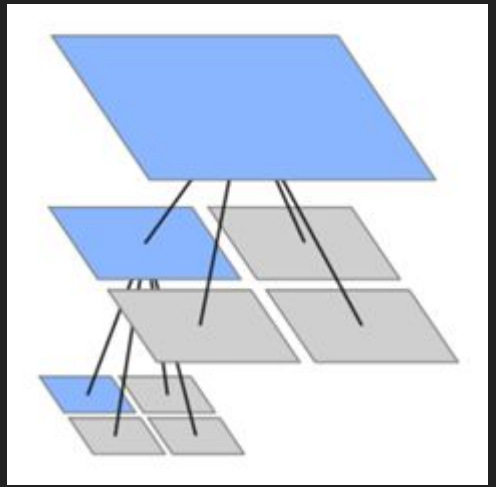
Bill Gosper, the genius behind Hashlife

Quadtree

- *Quadtree* - A tree in which each node has four children
- To deal with computing on an infinite grid, we represent the region of the grid that has been explored by live cells with a quadtree!

But wait, doesn't this mean that, in addition to all the leaf nodes, we also have to store a bunch of unnecessary parent nodes?



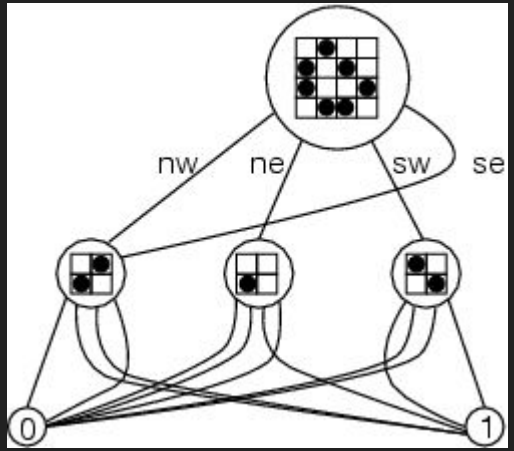


In our Quadtree representation, the grid is sliced into four, and each of those slices into four...

Not so fast!

Canonicalization

- We use a magical process called *canonicalization*.
- With canonicalization, we have duplicate nodes point to the same place in memory.
- Thus, much space is saved and regular structures can be identified by the program.



A canonicalized Loaf structure. Notice that the NW and SE nodes are the same, so they point to they same location.

Memoization

Two ways to compute the Fibonacci numbers:

(a)

```
int fib(int n) {
   if (n < 3)
      return 1 ;
   return fib(n-1) + fib(n-2) ;
}
```

O(2^n) :(

(b)

```
int cache[] ;
int fib(int n) {
   if (n < 3)
      return 1 ;
   if (cache[n])
      return cache[n] ;
   return cache[n] = fib(n-1) + fib(n-2) ;
}
```

O(1) :)

Memoization

- Similarly, instead of computing the next generation of a node *every time*, we can compute it *once* and then *store it!*

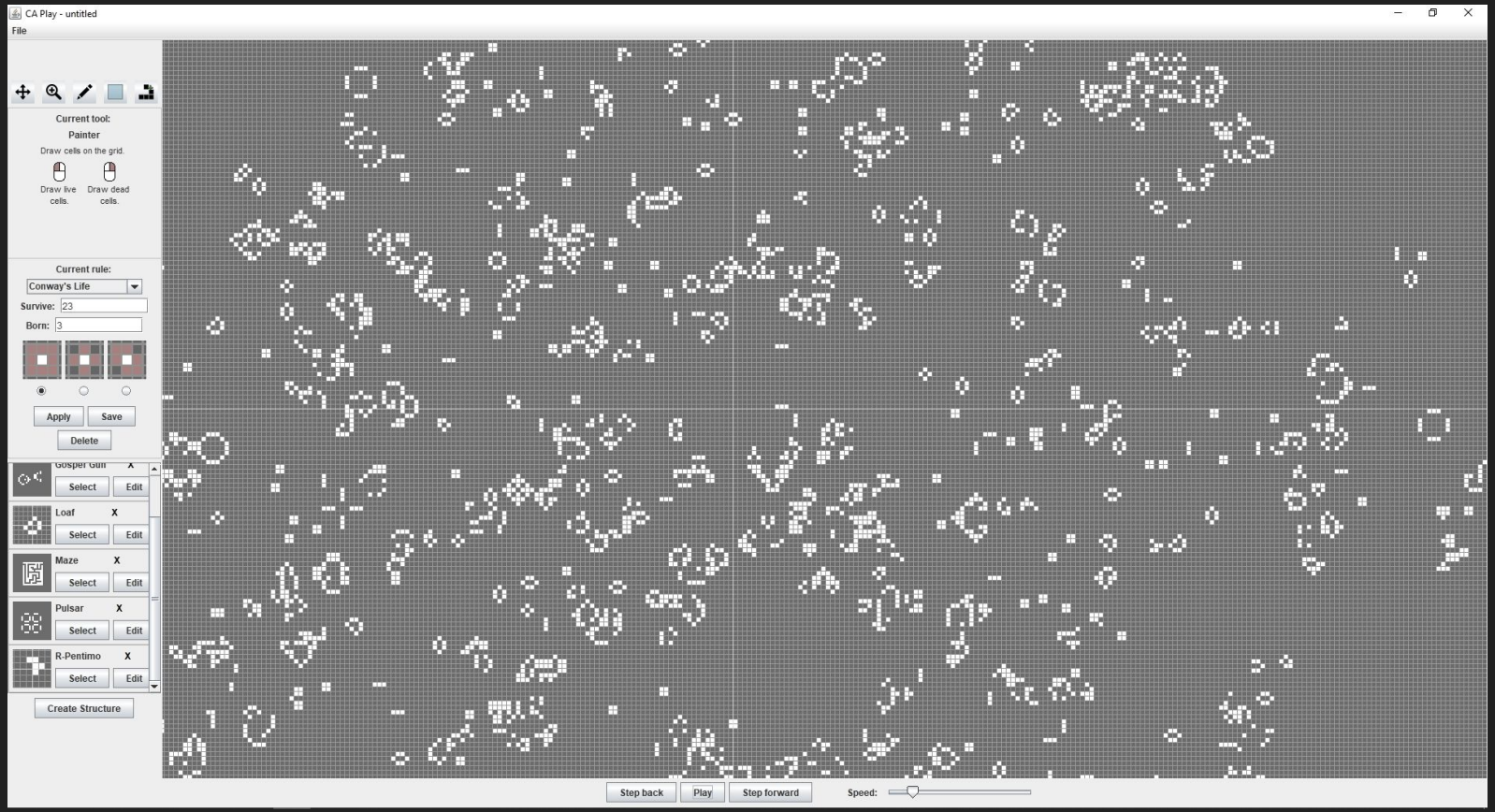
It doesn't matter what the other cells in the grid are. We already know what the interior of this square will look like in the future.

Hashlife: What We've Learned

- The Quadtree allows us to represent the grid with repeated subdivisions, capturing chunks of space.
- Canonicalization allows us to have identical nodes be represented by the same immutable node.
- Memoization leverages the past two ideas by storing the results of transitions to eliminate redundant computation.
- The result: pure magic!

All this Hashlife stuff is going on under the hood, but here's what the user actually sees...

Welcome to CA Play



Features

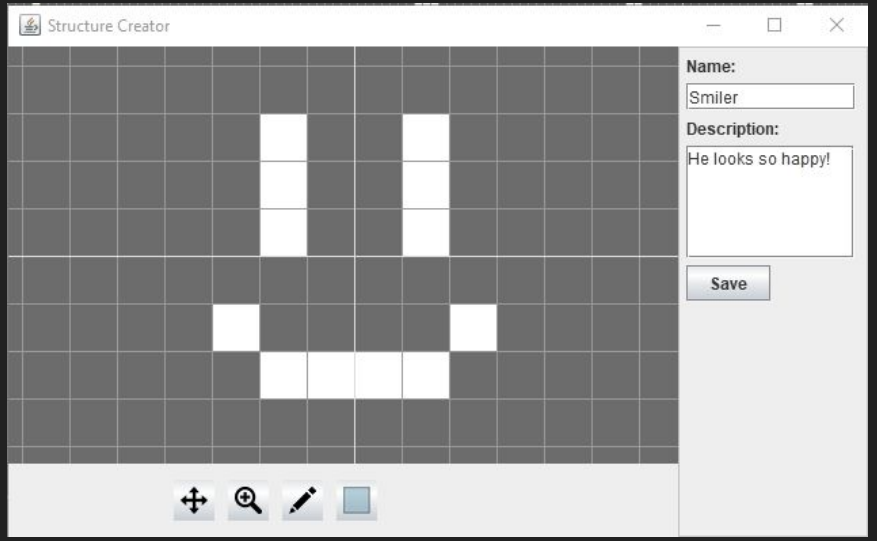
- Structures
  - Structure Creator
  - Structure Menu
- Tools
- Rule Editor
- Grid playing
- Grid saving

Structures

- I previously mentioned that many cellular automata have patterns with interesting properties.
- So that users don’t have to draw these patterns manually every time, the program allows users to draw it once, then after that put it on the grid with a single click.

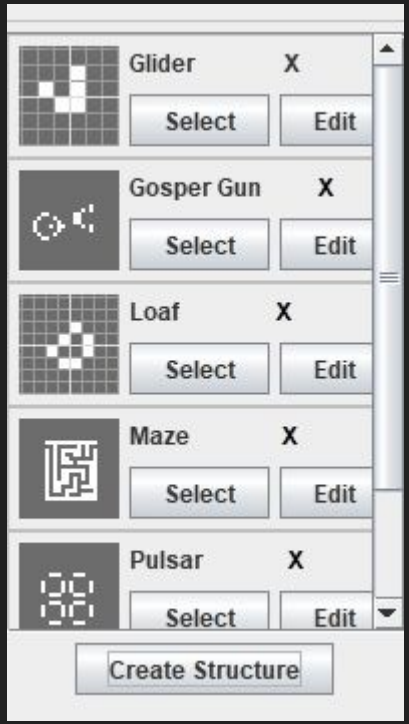
Structure Creator

- Users create structures in the (gasp!) Structure Creator
- In this window, users draw cells and user other tools to create the structure
- They then name, describe, and save



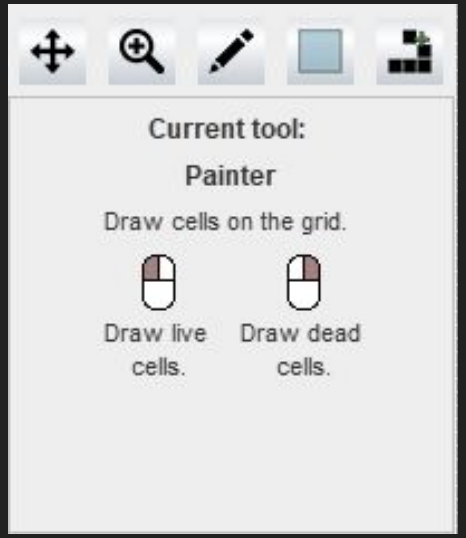
Structure Selector

- Once a user has created a structure, it ends up in the Structure Selector menu, which shows all saved Structures.
- Here, users can select a structure to place, edit a structure, or delete it.



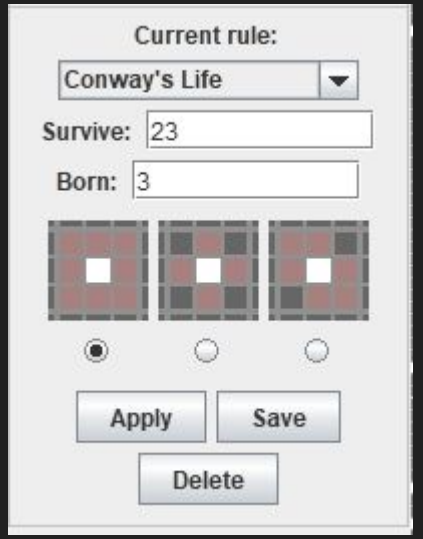
Tools

- The Tools menu allows users to select which tool to use on the grid, as well as provides information about the selected tool.
- From left to right, we have:
  - Mover - Move around the grid
  - Zoomer - Zoom in and out
  - Painter - Draw and erase cells
  - Selector - Select regions to fill, delete, randomize, or invert
  - Structure Adder - Add structures to the grid and select regions to turn into structures



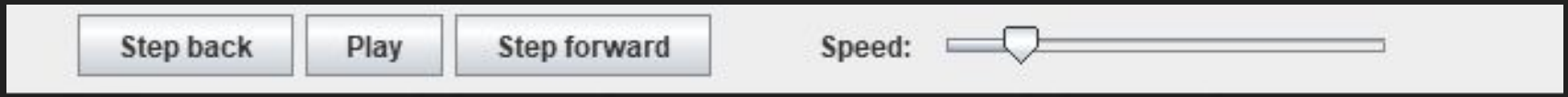
Rule Editor

- The rule editor lets users customize, save, and load rules to create new cellular automata.
- In addition to changing the neighbor counts, the user can also change the neighborhood shape.



Grid playing/saving

- Just watching the automaton evolve is almost hypnotizing.
- The user can do this with the bottom bar. They can play/pause, change evolution speed, and step forward/backward.
- Also, users who make an especially lovely grid can save it for experiment and play later.



Future directions

I will most likely continue work on this project. Future things I do may include:

- Requiring the user to associate structures with rulesets
- Expanding to more states (though not too many, because Hashlife won't like it!) and transition types
- Tweak bugs
- And most importantly… let the user select pretty colors for states!

Lesson Learned

I spent two weeks thinking about how I was going to implement my original project of creating an environment for exploring *any* cellular automaton, including options for larger neighborhoods, more exotic transitions, and cells with continuous states. And it fell flat on its face. Why? I didn't realize that I designed my algorithm for computing grid evolutions under the implicit assumption that I would have **unlimited computing power**. The algorithm simply went through the grid *cell by cell* and computed the transition *every single time*. It was *very slow*. So I had to scrap that version and produce this simpler one. So, here is the lesson I learned: **don't write bad algorithms**. Always think about how your idea for an algorithm will actually work in practice.

Further reading

Hashlife:

- https://www.drdobbs.com/jvm/an-algorithm-for-compressing-space-and-t/184406478?pgno=1
- http://www.thelowlyprogrammer.com/2011/05/game-of-life-part-2-hashlife.html

CA:

- https://towardsdatascience.com/algorithmic-beauty-an-introduction-to-cellular-automata-f53179b3cf8f
- https://plato.stanford.edu/entries/cellular-automata/
-