2-D Cellular Automata, Life, and Computing

Michael Hughes

January 15, 2009

In this chapter, we transition from 1D cellular automata to two-dimensional CAs. We focus exclusively on the most popular two-dimensional cellular automata, known as the "Game of Life".

The Game of Life is not at all a game in the sense of a multi-player variety. Instead, we use the word game in the sense of a solitary pastime. The Game of Life is a 2D CA originally invented by British mathematician John Conway as an intriguing form of mental recreation. We'll see why Conway called his game "life" later on.

1 What is a 2D CA?

To understand Conway's game, we first must discuss what it means for a cellular automata to be two-dimensional.

As with 1D CAs, a 2D CA describes how spatially organized entities change state over time depending on their neighbors. The difference is simply that we extend the spatial arrangement of cells into two dimensions. So rather than a finite sequence or a ring, as we discussed last chapter, we place the cells in a square grid structure.

Every cell's state changes over time depends on the current state of the cell as well as the current states of its neighbors. Traditionally, we define a given cell's 2D neighborhood as all cells which share at least one corner or edge with the given cell. So in two-dimensions, each cell has 8 neighbors which influence its future.

The state of each cell is usually a binary value, either on or off. Due to the complexity of determining rules that consider the various possible states each neighbor individually, 2D CAs are often totalistic. This means the arrangement of on and off states among a cell's neighbors does not matter, only the total number of on and off neighbors.

Although CAs are not always intended to model specific physical systems, one application of totalistic 2D CAs is as simulations of how biological populations might evolve over time and space. In this application, we can think of an "on" cell as representing the location of a living organism, while an "off" cell might represent an empty, life-free location. An appropriate rule system, then, would account for how an organism's spatial neighbors might influence its birth

or death. Let's conduct a brief thought experiment to discover how this could work.

First, consider birth: in order for an organism to come into existence where no organism existed before, biology tells us it must have at least one parent. So we might require that a small number of nearby cells be alive in the current generation in order for a cell that is off in the current generation to turn "on" in the next generation.

Next, consider death: we can reason through natural selection that an organism may not survive if it must compete for resources with many neighbors. This suggests a rule that forces cells with many living neighbors in the current generation to die off in the next generation.

These ideas sparked Conway's development of a fascinating rule system for a 2D binary totalistic CA known as the "Game of Life". Familiar with the concept of cellular automata, Conway sought to create a CA defined by rules which would lead to interesting, almost life-like behavior. What did he mean by life-like? Well, in particular, Conway had these goals in mind: [1]

1. There should be no initial pattern for which there is a simple proof that the population can grow without limit.

2. There should be initial patterns that apparently do grow without limit.

3. There should be simple initial patterns that grow and change for a considerable period of time before coming to end in three possible ways: fading away completely (from overcrowding or becoming too sparse), settling into a stable configuration that remains unchanged thereafter, or entering an oscillating phase in which they repeat an endless cycle of two or more periods.

After considerable experimentation, Conway identified a system of fairly simple rules which yielded non-obvious, life-like behavior. They are: [2]

1. Births: Each empty cell adjacent to exactly three neighbors–no more, no fewer–is a birth cell. It should turn "on" at the next generation.

2. Deaths: Each cell with four or more neighbors dies (is removed) from overpopulation. Every cell with one neighbor or none dies from isolation. These should turn "off" in the next generation.

These are the rules for the Game of Life can result in very interesting behavior, as we will see later. First, we must think about how to play the game.

[1] Taken directly from Marvin Gardner's 1970 *Scientific American* article on Conway's Game of Life. Available online at `http://www.ibiblio.org/lifepatterns/october1970.html`

[2] Ibid.

2 Playing the Game

Perhaps the best way to understand the Game of Life is to play it. That is, we can experiment with some initial configurations and begin to identify interesting results and patterns.

2.1 By-Hand

Conway and others originally played this game by hand. According to Martin Gardner[3], "To play life you must have a fairly large checkerboard and a plentiful supply of flat counters of two colors. (Small checkers or poker chips do nicely.) An Oriental "go" board can be used if you can find flat counters that are small enough to fit within its cells. (Go stones are unusable because they are not flat.) It is possible to work with pencil and graph paper but it is much easier, particularly for beginners, to use counters and a board."

After setting up an initial configuration, playing by hand involves updating the state of each counter by flipping it to its appropriate state based on its neighbors. The biggest caveat here is that the entire array of cells transitions from one time step to the next *simultaneously*. We cannot flip a cell counter immediately after we compute its next state. Instead, to avoid errors in chronology we must wait until all of its neighbors have been determined. Thus, the by-hand method requires a great deal of patience, and furthermore is not very feasible for a very large game board.

2.2 Simulation

Thankfully, modern computing allows us to simulate many generations of a very large 2D CA rather efficiently. Perhaps the most straightforward implementation involves encoding a 2D matrix or array to represent the CA, in which the ith row and jth column holds either a 1 or a 0 to represent whether the cell at location (i, j) is alive or dead. While this implementation is simple enough that almost anyone with a little programming experience can create a working simulation, it is not the most efficient process.

Bill Gosper, a mathematician who has worked extensively on the Game of Life, outlines a considerably more efficient data structure, commonly known as HashLife[4] [5], for representing this CA. The essential idea here is to utilize a tree-like structure to represent the square grid at various levels of detail. The root note holds the entire grid, while each of its four children represent one quarter of the entire space. As we continue down the tree, each node has four children, one for each of the four quadrants that comprise its square fraction of the grid. The leaf nodes of the tree, of course, are single bits.

[3] Ibid

[4] The original reference is: R. William Gosper. "Exploiting Regularities in Large Cellular Spaces," Physica 10D, 1984.

[5] See Tomas G. Rokicki's tutorial on HashLife for a more intuitive description: `http://www.ddj.com/hpc-high-performance-computing/184406478?pgno=1`

This structure is itself not very difficult to understand. The complicated ideas that make it efficient, though, come next. Instead of storing an array-like structure of ones and zeros at each level of the tree, which would cost a lot of memory and be very redundant, we can instead take advantage of a computer science concept known as *canonicalization*. This means if we expect to be storing redundant elements in a tree or other collection, we only need to keep track of one particular instance of the element. For example, the two-by-two grid consisting of all four dead cells will probably be used a lot in a given game. So instead of storing a different 2 by 2 matrix each time, we only need to keep one instance of this 2 by 2 matrix in memory at all times and use the idea of referencing to allow multiple nodes to refer to this object.

Canonicalization allows us to considerably reduce our memory cost for the simulation, at least compared to the basic tree structure idea. But where is the run-time speed up? The significant time efficiency HashLife provides comes from memorization. That is, for a given canonical block, we only need to compute its next-generation result once. After that, we can store the result in memory and associate it with our canonical block, so that the next time we need to determine what happens next to a 2-by-2 block with all four dead cells, we can quickly jump to the answer with very little (constant time) overhead.

Of course, implementing the HashLife optimization scheme is not quite as easy as we have described it. Determining a given cell's future state often depends on neighbors outside its own square block, so we must track and refer to other blocks in the tree appropriately. But when correctly implemented, the speed-up given by HashLife is "astronomical", allowing a given initial configuration to run for "trillions of generations" and "billions of cells" [6].

3 Patterns and Complexity in the Game of Life

Enough about programming. Let us now examine the interesting patterns and features which evolve in the Game of Life.

Conway's original interest in this game was to determine whether or not an initial pattern existed such that the number of living cells would grow without bound as time went on. We therefore classify patterns in the Game of Life based on their behavior over time and space. We describe and illustrate a few well-known types below.

3.1 Still-Lifes

The most basic pattern type in the Game of Life is one that never changes over time. Examples include the block and the beehive, as shown in Figure 1.

[6] Ibid.

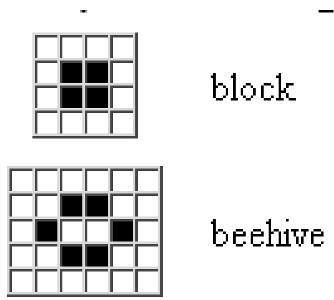


Figure 1: Common Still-Life Patterns

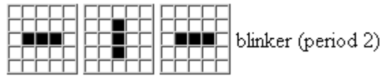


Figure 2: Blinker: A Period 2 Oscillator

3.2 Oscillators

Another fundamental pattern type is an oscillator. This type of pattern will regularly repeat a series of configurations over time. The three cell blinker has a period two, transitioning from its horizontal state to its vertical state in one step and from this vertical state back to horizontal state in the next, as shown in Figure 2

3.3 Gliders

A glider is a pattern which moves spatially across the grid over time. Its exact configuration may oscillate with some period, but there is a definite regular structure that translates through the grid. A simple glider is shown in figure 3. We see this particular glider transitions its entire structure one square down and one square to the left after 5 generations. Given enough time, it will continue to move diagonally across the board without limit (unless of course it collides with another pattern).

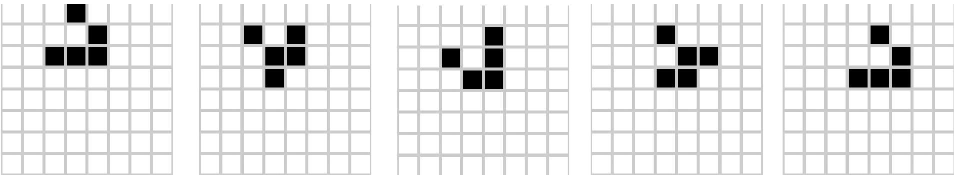
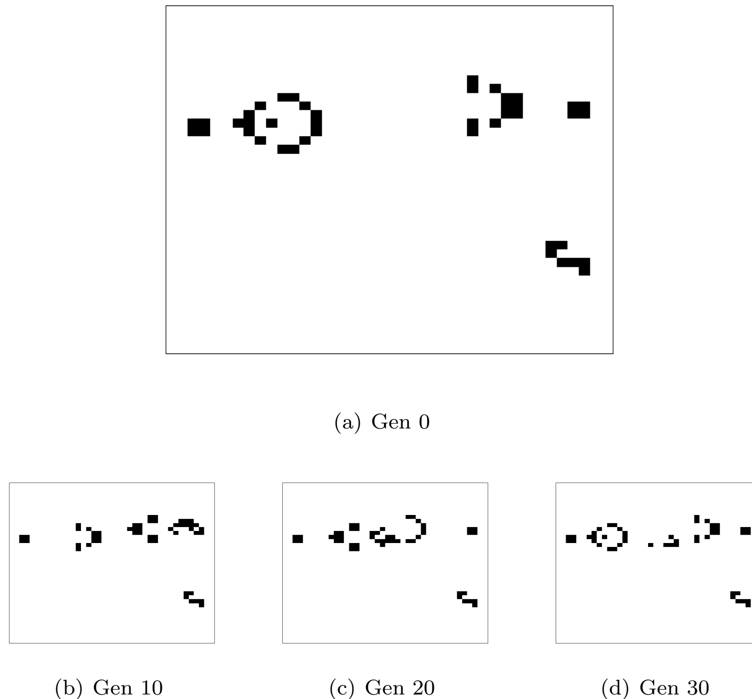


Figure 3: Glider



(a) Gen 0

(b) Gen 10 (c) Gen 20 (d) Gen 30

Figure 4: Time evolution of the Gosper Gun with an eater.

Conway realized how important gliders could be in his pursuit of an initial configuration which grows forever. He reasoned that if a configuration could be found that produced gliders with some regular period, the resulting behavior over time would yield an ever-increasing number of gliders. He called this hypothetical glider producing configuration a "gun", though Conway did not succeed in discovering a gun in his early work. It was left to future researchers to resolve this question.

3.4 Glider Gun

A glider-producing gun was eventually found by Bill Gosper, and named the Gosper Gun in his honor. It produces a new glider every 30 generations. We can see this gun in Figure 4. In recent years, thanks to advanced software search techniques and efficient simulations like HashLife, many more guns with many different periods have been found. However, Gosper's Gun remains a very important and interesting configuration, as we will see later.

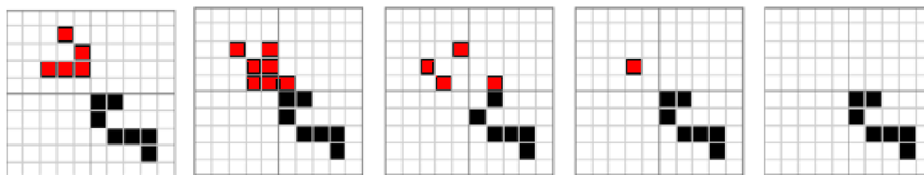


Figure 5: Eater absorbing a glider over 5 generations (view left to right)

3.5 Eater

Is there a still-life configuration that can obliterate a glider without disappearing? It turns out that a rather simple configuration, commonly known as an "eater" among GOL enthusiasts, performs this function well. We see the result of a glider colliding with this eater in Figure 5.

Although interesting in its own right, the eater will prove useful as we transition our view of the Game of Life from an interesting CA to a potential thinking machine.

4 Thinking with Life

Based on the intricate patterns and complexity we describe above, we can safely label the Game of Life a Class IV automata under Wolfram's classification scheme. The complexity which can evolve in the Game of Life is staggering. Perhaps the most interesting aspect of the Game of Life is that it has been shown to be Turing-complete. Practically, this means given enough time, a large enough grid space, and the right initial configuration, we can use the simple rules of this 2D CA to perform any computation that a computer can do.

This fact may be difficult to swallow, and understandably so. Although the proof of GOL's completeness is far too complicated for this work, we can instead offer a practical discussion, using the patterns outlined in the previous discussion, of how we might use the Game of Life to perform basic mathematical functions like NOT and AND. We follow the treatment first proposed by Jean-Phillipe Renard [7]using his LogiCell application [8].

4.1 Signals

Like silicon-based computers, the basic computations performed in our Game of Life computer operate on binary signals. The inputs and outputs of all processes, as well as any stored values, can all be represented as a collection of "on" and

[7]Rennard. Implementation of logical functions in the game of life. Available online: `http://arxiv.org/ftp/cs/papers/0406/0406009.pdf`

[8]`http://www.rennard.org/alife/`

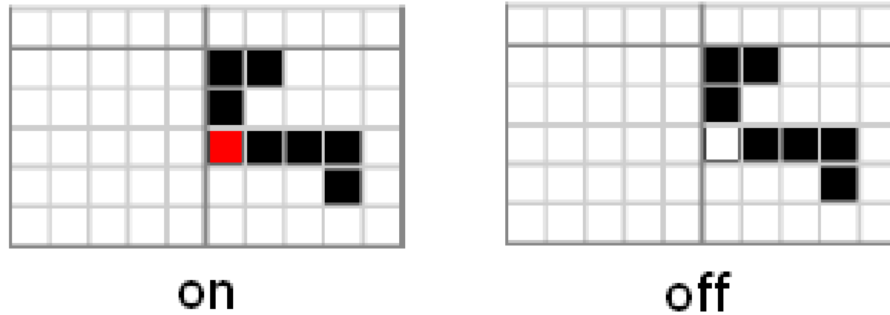


Figure 6: Eater control bit (shown in red)

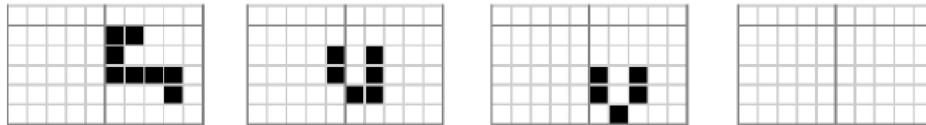


Figure 7: Time evolution of a disabled eater

"off" signals. But instead of using high or low voltage across a wire to represent binary data, as in a real computer, within the Game of Life we represent data using a glider stream. A signal is "on" if a glider stream is present and flowing, and the signal is "off" of the stream is blocked and non-existent.

4.2 Input

In order to send signals at all, we need some way of setting a glider stream to be on or off to begin with. To accomplish this, we may use a Gosper Gun to fire a stream of gliders. However, we'd like some way to *control* whether or not a stream turns on or off.

To solve this problem, we use the eater pattern discussed earlier, with one minor variant. We add an disable cell to the eater, as shown in Figure 6. When the disable cell is dead, the eater behaves as a still-life that blocks gliders. When the disable cell is alive, however, the eater will die and disappear after a few generations, as shown in Figure 7. Thus, if we place a controlled eater in the path of a glider stream, we can turn the stream on or off by setting the control bit of the eater on or off.

The full control of the glider stream that the disable cell of the eater gives us allows us to represent a binary input to our GOL CPU using just a Gosper Gun and an appropriately aligned and controlled eater, as shown in Figure 8. We see that the off or zero input yields no glider stream after 60 generations,

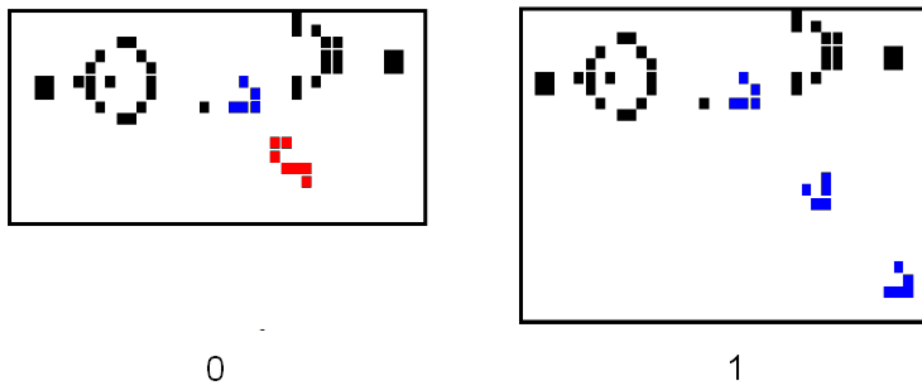


Figure 8: A disabled and enabled binary input after 60 generations.

while the on input shoots a glider stream.

This gives a straightforward initial configuration to create input gates within our GOL CPU. We simply place a Gosper Gun with an appropriately controlled eater onto our grid, and voila! We have a source of gliders that will represent either a 0 or a 1 in our binary computations.

4.3 Output

How do we detect output? A binary computation will of course yield a 0 or a 1 output value. How do we determine this value?

Just like we may set the on or off value of a specific cell on the board initially to produce input, we can read the state of a specific cell on the grid after some period of time has passed to whether a binary computation results in an on or off, true or false, 1 or 0 result.

We may use another eater pattern, as shown in Figure 9, to detect when a glider stream is produced. We see that a few generations into the collision, a specific cell next to the eater turns on briefly, before eventually dying off. Using careful calculation, we can determine the exact generation which we expect this bit to turn on if the output of our operation is true. Knowing this, we need only to read the state of this cell at this generation to determine the output of our computation.

4.4 Signal Interaction: Collision Computing

But how do we get signals to interact? We cannot simply pass on inputs as outputs. Where does the actual computing happen?

The fundamental property of glider streams that makes computing possible is their collision dynamics. We observe that two colliding streams of gliders will annihilate each other, leaving nothing alive behind, as shown in Figure 10.

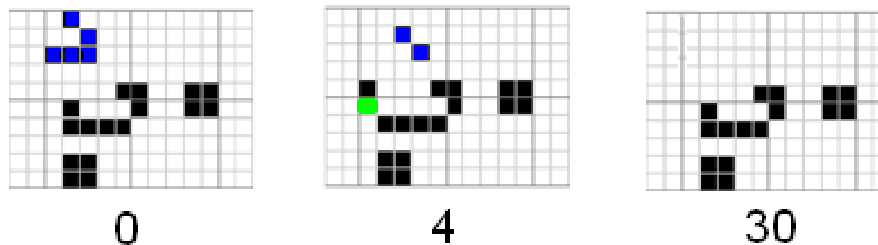


Figure 9: The state of the output gate after collision with an incoming glider.

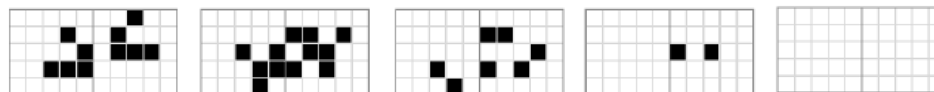


Figure 10: Glider collision over 5 generations (view left to right)

As we will show in the next few sections, glider collision allows us to simulate some fundamental bit-wise operations of the typical binary computer. Specifically, we'll show constructions for the NOT gate and the AND gate. The OR gate is also possible, though a little more complex. We refer the reader to Rennard's work for more information.

4.5 NOT

One of the most basic operations we may want to perform on a signal is to reverse its state. That is, given an on input, we produce an off output, and vice versa. Computer scientists refer to a piece of hardware that completes this function as a NOT gate. It turns out that we can create an analogous pattern within the GOL that will produce a glider stream given an off input but no stream when its input is on.

To build the NOT gate pattern, we simply place a single binary input pattern (Gosper Gun plus controlled eater) onto our grid and add another Gosper Gun nearby whose output stream will collide perpendicularly with our input. The initial components of a NOT gate are shown in Figure 11. We can toggle the input by turning the red cell on and off, while we observe the result of the NOT operation by reading the green cell of the output detector after a fixed amount of time. Figure 12 shows the result of the NOT computation for both possible input values: 0 and 1. We see that when the input A is on, its signal collides with the Gun signal (collision center marked with a red "X") which eliminates both from the grid space. However, when A is off, the Gun signal is free to travel all the way into the output detector and force the output bit to on. This is the correct functionality of a NOT gate.

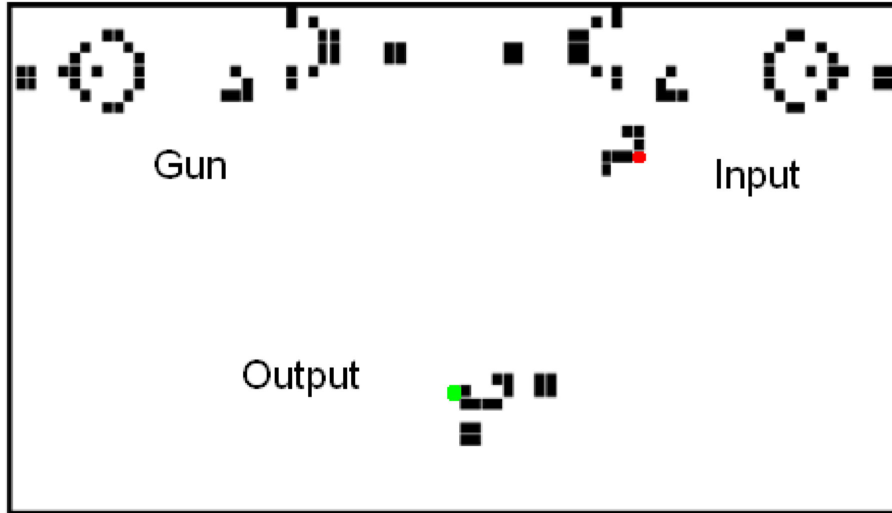


Figure 11: Components of the NOT gate.

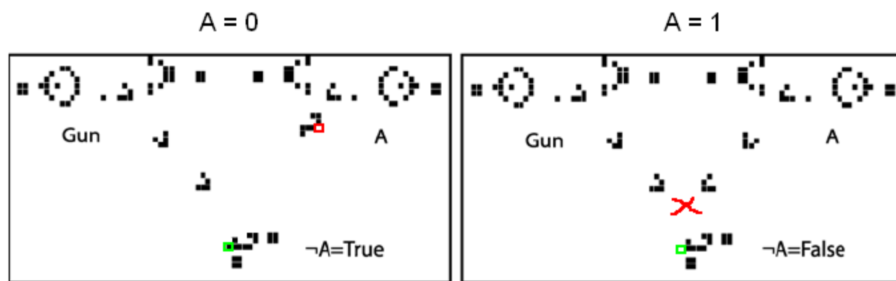


Figure 12: The result of NOT operation on all possible inputs.

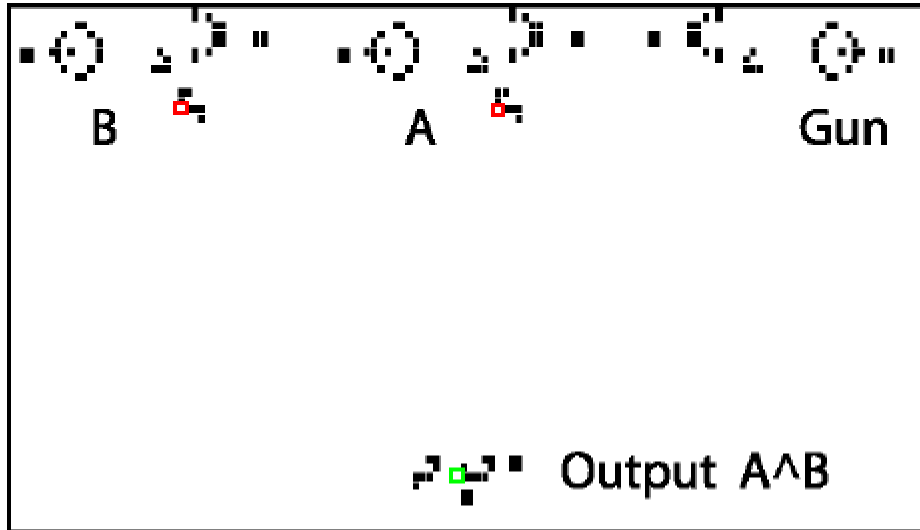


Figure 13: Components of the AND gate.

4.6 AND

But what about computing using multiple input signals? Let's build an initial GOL pattern that will take two binary input signals, A and B, and determine whether they are both on. Computer scientists know this function in hardware as the AND gate.

The components of our initial pattern are all pattens we've discussed before. We use two binary input patterns (Gosper Gun plus controlled eater) to represent A and B. We will arrange these so their glider streams will fly in parallel. Like the NOT gate, we also add a standard Gosper Gun nearby which will shoot a perpendicular glider stream. The interaction between this Gun signal and both A and B will produce our desired result. We place an output detector where the eventual result of the AND will be observed. These components and their arrangement are shown in Figure 13. Remember that we can toggle the inputs by turning the red cells on and off, while we observe the result of the AND operation by reading the green cell in the output detector after a fixed amount of time.

Note that in addition to our typical output collector we also have added an extra eater to block the Gun signal from propagating outside of the gate. This isn't necessary when we only care about the AND computation in isolation, but if we use multiple gates together we clearly don't want an extra glider signal flying across our grid space and screwing up our computations.

We can see what happens for all possible input combinations in Figure **??**. When both A and B are off, the Gun simply fires into the extra eater, and the output bit never changes from off. When A alone is on, the glider signal

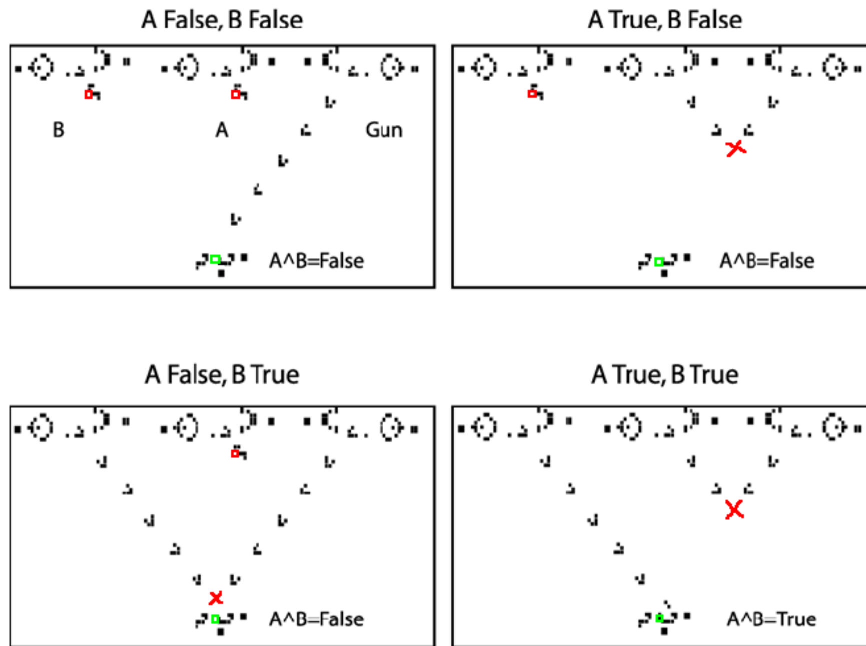


Figure 14: The result of AND operation for all possible inputs.

from A collides with the Gun signal, which annihilates both signals (collision is denoted with an "X" in the figure) and again never changes the output bit from off. When B alone is on, its signal collides and eliminates the Gun signal closer to the output detector, but still far enough away to leave the output at 0. However, when A and B are both on, A's signal anihilates the Gun stream, leaving B's signal free to fly into the detector and set the output bit. This proves the effectiveness of our AND gate.

5 Final Remarks

Hopefully this section has related some idea of the interesting patterns the Game of Life has to offer, as well as some idea of what it means to compute using cellular automata. For more information, please see the original work of Rennard in this area [9]

---

[9]Ibid.