Modular Design of Cellular Automaton in Rust

Shuyang Sun
Harrisburg University of Science and Technology
ssun14@my.harrisburgu.edu

***Abstract*—This paper is the literature review of a computer simulation project for CISC 614. This project is about cellular automaton, and we will go through the history and background of cellular automaton, the primary objective of this project, our design choices of the simulation and how that affects our code structure. The primary purpose of this project is to provide a flexible and easy-to-use software framework to study different types of cellular automatons. We found that with our design it is very easy to implement unconventional cellular automatons with minimal effort, by reusing the vast majority of the code that comes with our framework.**

***Index Terms*—Cellular Automaton, Game of Life, Computer Simulation, Rust**

I. Introduction

Created by the founding father of the famous Monte Carlo simulation - Stanislaw Ulam, and his colleague John von Neumann, cellular automaton (CA) was first used to study the growth of crystals and self-replicating robots in the 1940s in Los Alamos National Laboratory.

Cellular automatons are systems formed by a collection of individual agents interacting with each other based on certain rules. Traditional CA systems consist grids of cells, each with a state selected from a set of discrete states. The system evolves by discrete steps, with each step referred to as a "generation", each cell's state in the next generation is determined by its neighbors' states and a fixed set of rules. Before the simulation starts, each cell is assigned with an initial state. Figure 1 and Figure 2 are snapshots of a one-dimensional and two-dimensional cellular automaton, respectively.

As more and more people started researching cellular automatons, their form has evolved into other types of system with extremely flexible rules. For example, the grid system does not have to be one or two dimensional, it can be any finite dimension or not a grid system at all; another way to extend traditional system is by combining learning automaton and cellular automaton, so each cell's rule is adaptive instead of fixed. As researchers started exploring more complex configurations, more interesting behaviors and challenges arise.

Now with some basic knowledge about cellular automatons, let us explore a bit more about their history before diving deeper.

Cellular automatons are not purely the interest of academia; they have practical use cases as well. Researchers have used cellular automaton to model freeway traffic and verified the benefit of start-stop waves in real-life traffic [1]. Another example is simulating forest fire spread using a combination of CA and machine learning [6].

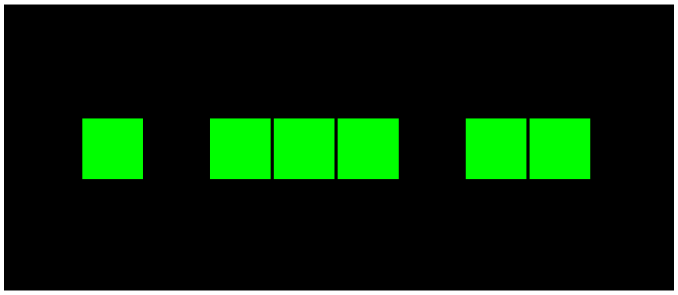
Fig. 1. 1-D Cellular Automaton

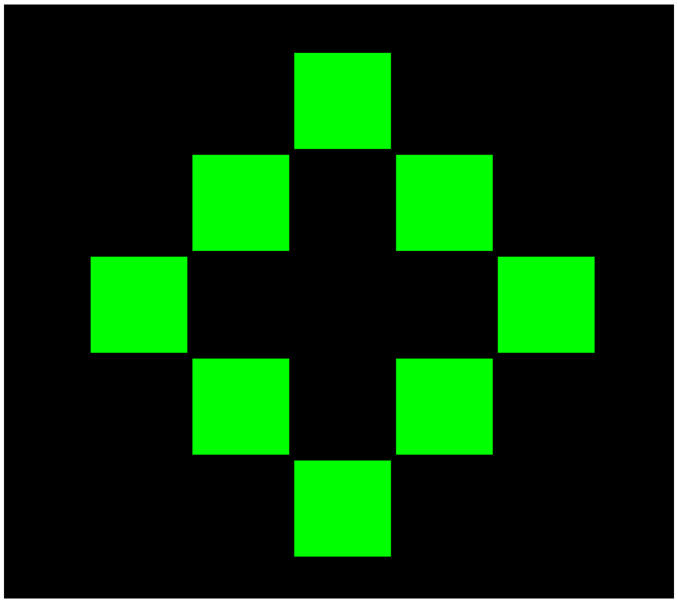
Fig. 2. 2-D Cellular Automaton

Throughout the mid-90s, researchers came up with many different CA systems, one of the most famous ones is John Conway's Game of Life. Next, let us take a deeper dive into that and other CA systems.

II. Previous Work

A. Game of Life

John Horton Conway (1937-2020) was a British mathematician. During his years in Cambridge he invented a particular CA system called The Game of Life, this system was introduced to the public in 1970.

The Game of Life is defined on a two-dimensional grid system, with each cell having one of two binary states: alive or dead. Each cell's neighbors are defined as the surrounding 8 cells, this neighborhood rule is usually referred to as the Moore neighborhood. The evolution rule is as following:

1) An alive cell with 2 or 3 alive neighbors survives.
2) A dead cell with exactly 3 alive neighbors becomes alive.
3) Otherwise, an alive cell dies and a dead cell stays dead.

Although the rule might seem complex at first glance, it is much simpler compared to other CA systems, especially considering its capability. This logic can be expressed by a one-line logical condition in most programming languages:

```
isAlive = (aliveCount == 3) || isAlive && aliveCount == 2
```

Although the rules are very simple, it is special because of this seemingly random rule can generate many repeating patterns. Researchers have been able to build complex systems by implementing logical gates with the simple rules of The Game of Life [5]. Figure 3 is a snapshot of the logical gate AND built in Game of Life.

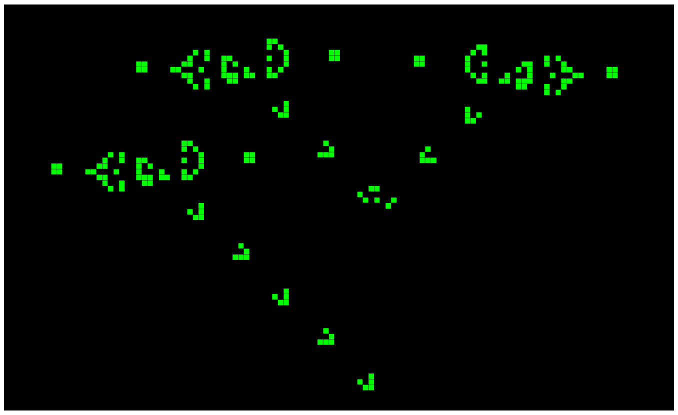

Fig. 3. AND Gate with TRUE, TRUE Input

As we have seen from the study of the growth of crystals and Game of Life, CA systems can be used in the context of both real-world experimental simulation and theoretical research. Some may argue that Game of Life is a simplified version of death and reproduction, but its connection to real-world evolution may not seem obvious to many people. However, the next few examples we will be looking at are closely tied to real-world systems.

B. Traffic System Design

Traffic system design is a very challenging problem that is closely related to our daily lives. It is unrealistic to test out different design choices in real life, not only it is costly to do so, but less optimal design can also lead to consequences anywhere from reduced traffic flow to increased fatal traffic accidents. This is exactly the reason traffic simulation is a necessity when it comes to traffic design. There are many systems designed specifically for traffic simulation, but researchers have also tried using other well-studied generic systems to model traffic simulation, including cellular automatons.

In a freeway traffic simulation created in 1992, the authors modeled multiple aspects of a freeway traffic using both CA and fluid-dynamical systems, including traffic stops, car velocity, acceleration and deceleration, etc. Randomization was also used to simulate the uncertainty of real-world traffic situations. The authors found that CA systems have advantages not only in computing resource requirement, but also how closely it mimics individual driver decisions due to its discrete nature [1]. The computational advantage of CA systems is shown in another paper that simulated pedestrian behavior using a two-dimensional CA, the computational complexity only grew linearly with number of pedestrians simulated, instead of the square number of it in some other methods [7]. For inner-city traffic system design, CA systems can also be used to model traffic lights [9].

C. Urban Growth

Another example of CA system being used to simulate modern life is the study of urbanization. Like our previous traffic simulation example, urban growth is very similar in the sense that the system is composed of many different individual agents, which behaves in different ways based on the state of their local and non-local neighbors. A research conducted in 1997 utilized CA systems to predict the urban growth of San Francisco Bay area.

The CA system used in this study is relatively more complex than traditional CA, since not only it modeled many data points, but also uses dynamic self-modifying rule system. Self-modifying models provide many benefits such as: more closely model the flexibility of real-world system, easier extension to existing model by linking external data, being able to manipulate and simulate more complex mathematical functions [4].

Like urban growth, biological cell growth also consists individual cells interacting with their neighbors, with complex parameters at play such as blood flow and surrounding tissue. Indeed, researchers have studied tumor growth using CA models [8].

D. Natural Disaster

CA systems have also been used to study natural disaster. A good model of natural disaster events is important for urban design and designing emergency evacuation policies to prevent or reduce damage. Most natural disasters occur in the environment which there are many individual moving parts, and it is difficult to study the behavior of given system analytically. Forest fire is one of the natural disasters that is challenging to simulate, not only the terrain has to be part of the consideration, but many other uncontrollable factors such as humidity and wind direction/speed can also drastically affect the outcome.

A group of researchers were able to effectively model forest fire by combining traditional CA and Extreme Learning Machine (ELM). In their paper published in 2017, by modeling a forest fire by assigning each cell an ignite probability and using an ELM to learn the ignite probability from historical data, researchers were able to produce a well performing model of forest fire spread [6].

CA models of natural disaster can be combined with other CA models such as CA model of evacuation process with obstacles [3] for emergency response planning.

III. Objective

As we have seen from previous sections, CA systems has a wide range of utility ranging from pure interest of research to practical applications that could have an impact on real-world policy design. However, most researchers and studies must implement the system from scratch, even though the core fundamental of most cellular automaton systems are very similar.

The goal of this project is to provide a generic software framework to construct cellular automaton systems, while making sure it is high-performance, easy to use and makes as little assumption as possible about the specific use case.

In the next section we will discuss the design and implementation of this software framework, as well as a few examples of how we can easily extend this framework by adding/modifying components.

IV. Software Design

In this section we will be discussing the design of our software framework. We will review some example CA systems we have looked at to understand what assumptions we can or cannot make, the software architecture design, and finally our choice of programming language. First, let us start with some assumptions and requirements.

A. Cell State

The first observation we can make is about cells' states since it is a relatively simpler aspects of design. All examples we have seen so far uses discrete cell states, which can be easily represented by enumerations or integers, however one can certainly construct a CA system with continuous states and a continuous function as evolution rule. In fact, we do not want to put much limit on the state of cell at all, they can be any primitive or user-defined type in the program if it plays well with other parts of the system.

B. Cellular Space

Moving on to more complicated aspects of our system, we quickly realize that we cannot make assumptions about the dimensionality of our cell space. We have seen examples of one-dimensional and two-dimensional CA systems, three-dimensional or higher are not rare either, although anything above three-dimensional would certainly require rendering methods that can reduce the dimensionality to make sense of what is going on. Upon closer inspection, maybe we do not have to stop at grid-based system at all, nothing is limiting us from constructing a cellular automaton from any random data structure.

Before diving deeper into our cell space, we have to discuss the relationship between logical data structure and visualization. It is well-known that in any software design it is a good idea to decouple model and user interface (UI), because one data can be represented in many ways. The same is true in our context, one data structure can be visualized in many ways and we do not want to make one dependent on the other.

For example, we can construct a CA system within a graph or a tree structure, then define each cell's neighbors to be its neighbors in the graph or children in the tree. If we know how to iterate through the data structure and the concept of "neighbors" is well defined, this space should be valid, and no further constraint is necessary. The number of cells in the space does not even have to be finite, it can grow or shrink dynamically while the cells evolve.

C. Neighboring Rule

We have briefly mentioned the relationship between our cell space and neighboring rule, let us explore that further. A neighboring rule does not have to be coupled with cell space either, since one cell space can use multiple neighboring rules and vice versa. For example, on a two-dimensional grid CA system, two commonly used neighboring rule are Moore and Von Neumann. The Moore neighboring rule defines the neighbors of a cell its 8 surrounding cells, while the Von Neumann neighboring rule defines them to be the 4 adjacent cells excluding the other 4 in the corners. Similarly, the same neighboring rule can be applied on different cell spaces if they share the same cell iteration rule (i.e., using the same index system).

Neighboring rule does not need to be coupled with cell states either. As we have discussed before, we do not really put any constraint on the cell state, and we should not need to know the state of the cell to calculate its neighbors.

Neighboring rule, cell space and cell state should be three separate components that do not need to know about the inner workings of each other.

D. Evolution Strategy

Evolution strategy is the first connecting piece of all three aspects we have discussed above. A strategy component is responsible of transitioning the current state of the system into its next state. To achieve this, the strategy component needs to understand where all the cells are, the state of all cells, who are their neighbors, and then calculate the next state for each cell.

It seems like evolution strategy should be considered as a higher-level concept that knows how states, space and neighboring work with each other. However, let us explore if that is a good idea based on examples we have seen.

Although a strategy component is responsible for updating all cells, for each individual cell it only needs to know its current state and its neighbors' states. Given this information, the strategy should not need to know how to traverse the space or what is the neighboring rule. Another thing to consider is that all cells may not want to have the same evolution strategy. Researchers may want to study how multiple evolution strategy interact with each other, or in an extreme case, each cell can have its own dynamic strategy that is governed by a learning algorithm and changes throughout the duration of simulation.

It should be clear now that although the strategy component can be considered a high-level component that needs to know the interaction between other parts of the system, it is far more than necessary and not a modular design. Evolution strategy should be local to a specific cell, and it is the choice of our framework user to decide if all cells share the same strategy or not.

E. Data and Persistence

The data we have in this simulation is simple: states of all cells at each iteration. However, we need to answer the question of at what time do we need to access this data? Is it live during the simulation? Or is it after the simulation in which case we need to consider data persistence?

The answer to the first question should be an easy "yes". In most cases, we want to visualize the simulation as it is playing out. We need to design our system such that at every iteration it shares the current state of the system to other extensible components, while not knowing what they may do with the data.

We can get the answer to our second question by looking at the application of machine learning in examples we looked at above. Machine learning methods were mostly applied locally for each cell [2], [6], however, we should not limit ourselves to only apply machine learning on a local scale. One use case for applying machine learning models on a global scale would be using it to analyze simulation data to find patterns given a set of rules, like the glider pattern and logical gates found in Game of Life [5]. Besides machine learning, other analysis can also be performed after the simulation has finished, so the answer to our second question is also "yes".

F. Visualization

Visualization is one of the core components of a CA system. Although CA systems can be studied from a purely analytical perspective, pretty much all studies we have looked at visualize the system in one way or the other. Watching the system evolve and seeing patterns emerging from chaos is one of the most satisfying activities we can do with cellular automatons.

Lucky for us, the visualization system only need to communicate with the data component of our system (persisted data if we want to implement replay), it does not need to know anything about the neighboring rule or evolution strategy. This allows us to place our visualization component within our data processing pipeline, just like any other algorithm we might want to apply to live-generated data. This design decouples the visualization component from logical cell space structure, the exact same visualization code can be used on many different systems if we know how to visualize each individual state.

G. Performance

Performance may or may not be important depending on the use case. For an analytical research project that analyzes how the system evolved after the simulation, performance may not be a big concern, since it only relies on persisted data. For a project with live visualization as priority, performance is key, just like video games - higher frame rate directly translates to more pleasing visual effects.

Due to the nature of our system, there can be many cells to simulate, so we have to maximize the performance to satisfy all use cases.

On a grid-based system, the most optimal implementation given today's technology would involve much GPU programming. Technologies like CUDA makes general-purpose GPU (GPGPU) programming more accessible than ever before. Although CA systems like Game of Life can run extremely fast with a GPU implementation, other systems may not be able to be implemented on GPU at all. As we have discussed before, each cell may have its own complicated evolution strategy, which is very difficult to run parallel on GPU.

Our system needs to be high-performance by being able to massively parallelize most of the computation without relying on GPU processing. This directly affects our choice of programming language, which we will discuss next.

H. Programming Language

Considering all the design aspects we have talked about, an object-oriented programming language is likely a better choice, instead of a functional one. Due to the performance requirements, an interpreted language or any language without parallelism may not be a good choice.

The most efficient language suitable for this task is probably C/C++, however, it is well known C/C++ programs are prone to memory-related bugs and difficult to implement multi-threaded algorithms [11], [12]. Rust is an alternative to C/C++, which is memory safe and makes it much easier to implement parallel algorithms [13], [14].

V. Triangular Cell Space

We have mentioned that the purpose of our modular design is so that it is easy for researchers to implement unconventional cellular automatons without starting from scratch. In this section we will explore what is required to implement a two-dimensional triangular cellular space instead of a grid one. Our objective is to reuse the vast majority of our existing code while only swapping out minimal amount of components. First, let us take a look at the triangular cell space we are trying to implement.

As shown in Figure 4, our goal is to implement the triangular cellular space on the right side.

It seems like to implement a triangular cellular automaton system, we need to implement everything from scratch, since the geometry of triangular system is so different from grid-based system. But since our goal of developing this framework is to minimize effort and maximize code reusability, let us explore each component of our framework and see how to implement this new cellular automaton system with minimal effort.

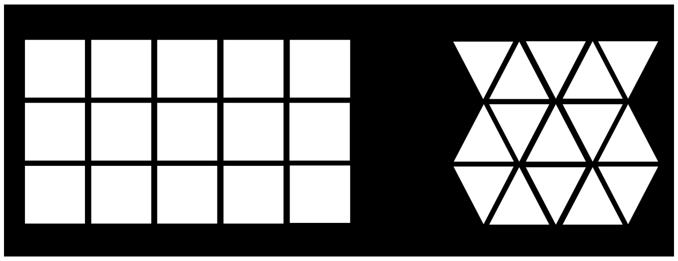

Fig. 4. Geometry Comparison

A. Cell State

The geometry of our cellular space should not concern the state of cells, any state can be used on a grid-based CA should be able to be applied on the triangular CA. We do not need to implement any additional restriction for the state component.

B. Cellular Space

Cell space is the component that is responsible of iterating through cells. A grid-based system can be easily iterated through integer index along each axis, but it is not so obvious for triangular cells. Since there is a big difference in geometry, our first instinct may be creating a different cell iterator.

For example, we can start from the cell in the center, and identify each cell by the path it can take starting from the center cell. There are three directions we can travel from each triangular cell: left, right, and up or down (depending on whether the cell is pointing upwards or downwards). Instead of being represented by a tuple of integers, the index of each cell would be an array of directions.

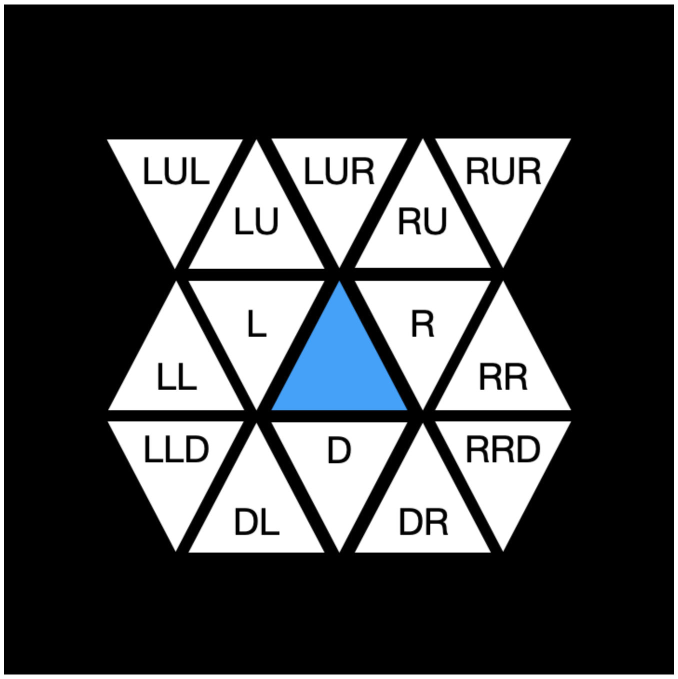

Fig. 5. Indexing by Path

If we label left, right, up and down as “L”, “R”, “U” and “D” respectively, we would then represent each triangular cell’s index as shown in Figure 5.

However, implementing a new cell index like this means we cannot easily reuse other components we already implemented for grid-based cellular space. We also run into a problem which there are multiple (infinite to be exact) representations for the same index, since there are infinite paths we can reach from one cell to another. Let us explore an easier solution that maximizes code reusability.

One observation we can make is that any diamond shape can be composed from two triangles. If we convert every single square in our grid-based system into a triangle, then combine every two pairs of it into a diamond shape, we would have a diamond shaped grid cellular space. This cellular space is as easy to iterate as the our original square one, since diamonds are simply squares with a shear transformation. Figure 6 shows how we can represent each triangular cell’s index with the exact same two-dimensional coordinate system used on square cells.

In fact, we do not need to make any change to cellular space component, we can reuse all of the code we wrote for grid-based system and iterate through the new space just fine. From the cellular space component point of view, iterating through a square board is logically exactly the same as iterating through a triangular board. This perfectly demonstrates the benefit of decoupling components during our design, cell iteration does not need to know the inner workings of other components and can stay unchanged. The extra complexity introduced by the change in geometry will be handled elegantly by other components, as we will see soon.

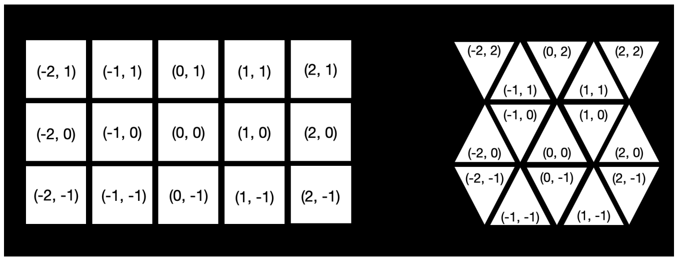

Fig. 6. Indices Comparison

C. Neighboring Rule

It is up to us to define any arbitrary neighboring rule as we wish for our new triangular cell space, but it probably makes the most sense if we start from existing rules for gird-based CA system. The most common neighboring rule used in grid-based cellular space is Moore [15], it defines the neighbors of a cell to be its eight surrounding cells. Another less common rule is von Neumann [15], which defines a cell’s neighbors to be adjacent four cells, excluding ones connected by the four corners.

In this example, we will mimic the Moore neighboring rule for our triangular cells. We define the neighbors for a triangular cell to be its six surrounding cells: three connected by sides and three connected by corners. Figure 7 shows neighbors of the center cell (cell 5) in grid and triangular cellular spaces respectively.

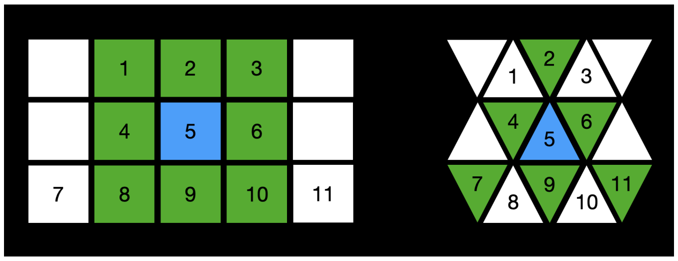

Fig. 7. Moore Neighboring Rule

Upon closer inspection, we quickly discover that there is a bidirectional one-to-one mapping between neighbors in the grid cellular space and neighbors in the triangular cellular space. As shown in Figure 8, the neighbors for the center cell in the triangular version are the same as von Neumann neighbors in the square version, but with the addition of two more cells extended out on the x-axis (cell 7 and 11). The exact position of the two extended cells depend on whether the triangle is pointing up or down.

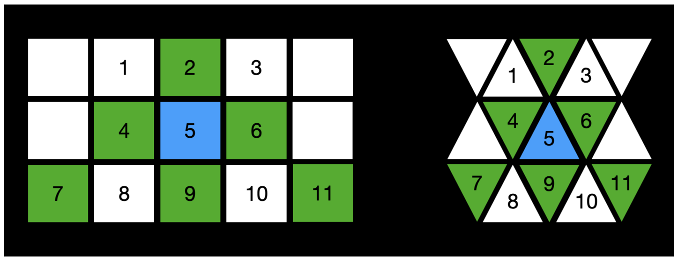

Fig. 8. Moore Neighbors Mapping

With this information, we can easily implement our triangular Moore neighboring rule with our existing implementation of grid-based components like following:

```
function mooreTriangular(idx)
      isPointingUp = (idx.x + idx.y) % 2 == 0
      yIdx = isPointingUp ? idx.y - 1 : idx.y + 1
      additional = [(idx.x - 2, yIdx), (idx.x + 2, yIdx)]
      return vonNeumann(idx).extend(additional)
```

The actual implementation in the code base is not far off from one presented above. Turns out, we can implement the geometry and neighboring rule for our new triangular cellular space with only a few lines of code, the only component needed to change was the neighboring calculation.

The rest of the components can stay exactly the same, because at this point it is clear that a triangular cellular space is logically equivalent to a grid one with a specific neighboring rule. However, although it is possible to visualize the new neighboring rule with our existing grid-based implementation, it is not as intuitive as a direct visualization of triangular cells. Now we will talk about changes we need to make to the visualization component to make it visually more pleasing.

D. *Visualization*

The visualization component is a part of data processing pipeline. It has two responsibilities: how to render each state and the geometry of each cell at any given index. As stated before, we did not make any changes to the state, so state-representation code can stay exactly the same. We do need to change the shape and positioning calculation to take the new geometry into account, and luckily for us we do not need to make much change to that either.

Translation and scaling transformations have already been implemented for grid-based system, because we need to render cells at different indices and scale them based on the window size. All we need to do is to implement new vertex and fragment shaders for triangle shapes and add rotation to our transformation matrix, since a triangular cell can be pointing either upward or downward.

If we denote our vector as v, translation matrix as T, rotation (clockwise) matrix as R and scale matrix as S, then we have:

$$v = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix},$$
$$T = \begin{bmatrix} 1 & 0 & 0 & H_x \\ 0 & 1 & 0 & H_y \\ 0 & 0 & 1 & H_z \\ 0 & 0 & 0 & 1 \end{bmatrix},$$
$$R = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$
$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 1 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Applying transformation in the order of scaling, rotating then translating gives us:

$$Av = TRSv,$$
$$A = \begin{bmatrix} S_x\cos\theta & S_y\sin\theta & 0 & H_x \\ -S_x\sin\theta & S_y\cos\theta & 0 & H_y \\ 0 & 0 & S_z & H_y \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We can use the above matrix to calculate transformation for both square and triangular cells. In reality, the change we had to make to our implementation for triangular cellular space was only about 200 lines of Rust code. The new neighboring rule was implemented with one small function, and the vast majority of the new code added were for rendering calculation purpose.

With very minimal effort, we now have a triangular cellular space that shares all the existing functionality of grid-based CA system such as saving and loading replay.

E. Triangular CA Examples

The common standard to describe the rule in a cellular automaton is by using string with syntax "survival/born/states/neighbor". For example, the rule of Game of Life can be represented as "2,3/3/2/M", with each part stands for the following:

1) 2,3: any alive cell survives to the next generation if there are 2 or 3 alive neighbors.
2) 3: any dead cell with 3 alive neighbors will be reborn.
3) 2: there are 2 states.
4) M: use Moore neighboring rule.

When the number of states n is greater than 2, that means a cell can be at any state from 0 to $n-1$. A cell is only considered alive when it is at state $n-1$, when it dies it enters a count down stage before it can be reborn again. A cell at any stage between $n-2$ to 0 is considered a dead cell, but only cells at state 0 can be reborn.

Figure 9 to 11 are snapshots of a CA system generated from rule "3,4,5/2/4/M", with half the cell initialized as alive at the beginning of the simulation. This rule is nicknamed "Star Wars". Different colors represent different states.

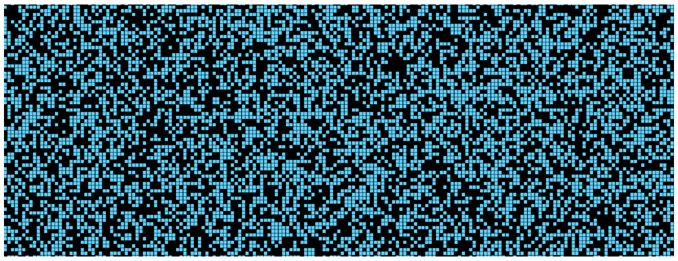
Fig. 9. Star Wars 1

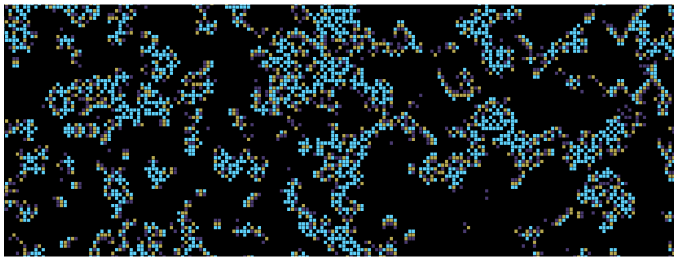
Fig. 10. Star Wars 2

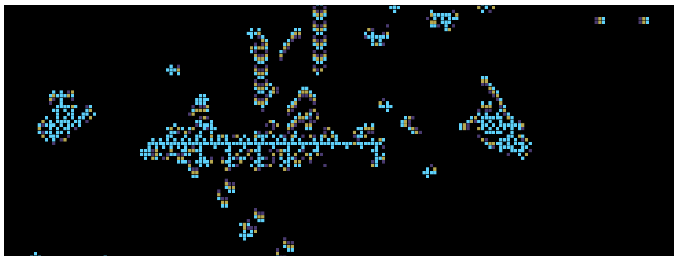
Fig. 11. Star Wars 3

As shown in the figures, the "Star Wars" rule generates many moving patterns that are seemingly shooting out projectiles and battling with each other. Now we can use exactly the same rule but apply it to a triangular cellular space and observe what happens. Figure 12 to 14 shows snapshots of the triangular version of "Star Wars".

We realized this triangular version of the system reached a stable state much more quickly, and the system is left with many infinite repeating patterns such as the one shown in Figure 15.

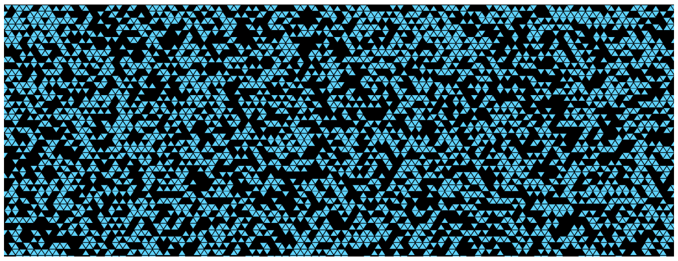
Fig. 12. Triangular Star Wars 1

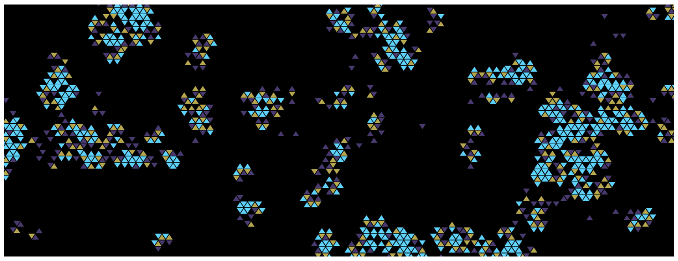
Fig. 13. Triangular Star Wars 2

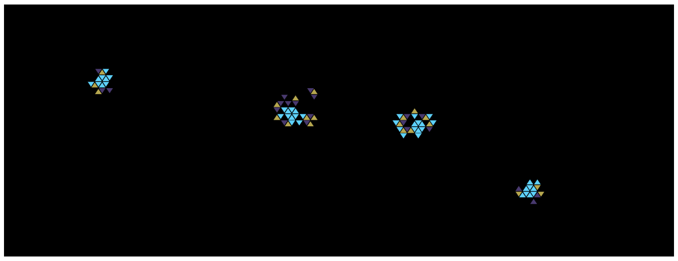
Fig. 14. Triangular Star Wars 3

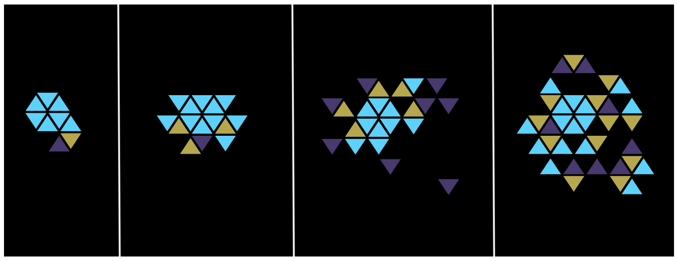
Fig. 15. Triangular Star Wars Repeating Pattern

Another rule we can study is rule "3,4,5/2,4/25/M", nicknamed "Bombers". Figure 16 to 19 are snapshots of the square version. As we can see this configuration reaches a stable state in the end with many infinite repeating patterns.

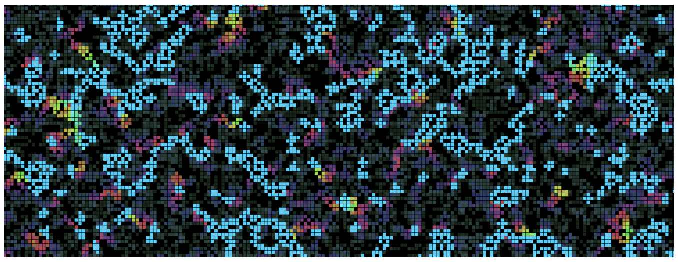
Fig. 16. Bombers 1

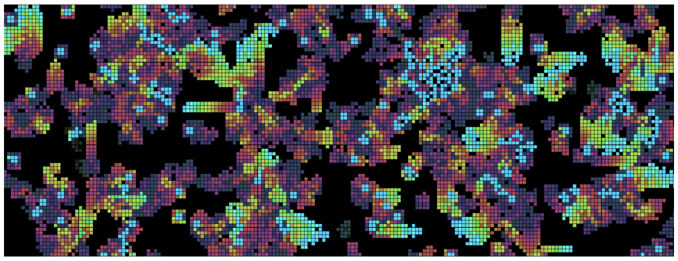
Fig. 17. Bombers 2

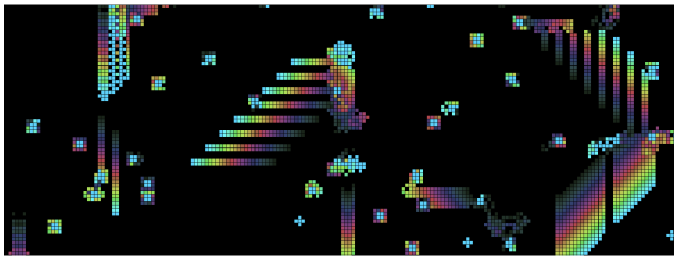
Fig. 18. Bombers 3

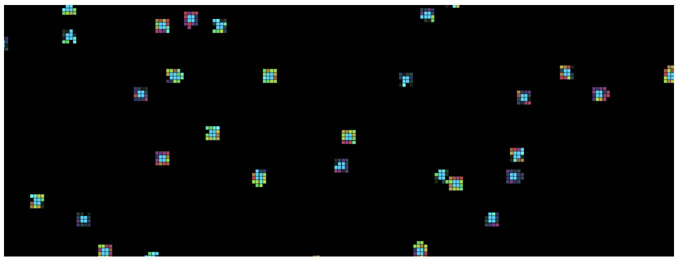
Fig. 19. Bombers 4

When we apply the same rule in the triangular cellular space, we get some interesting behaviors. Figures 20 to 23 are snapshots of the triangular version of "Bombers".

The triangular version of this rule is quite interesting because once the system reaches a stable state, it contains not only stable infinite repeating patterns, but also patterns that generates and destroys signals similar to the "Glider Gun" and "Glider Eater" in Game of Life. Examples of such can be seen in Figure 22 and 23. With patterns like this, one can certainly build programs that are Turing complete [16] within this triangular cellular space.

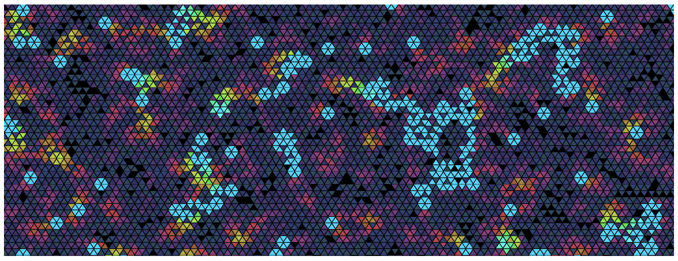
Fig. 20. Triangular Bombers 1

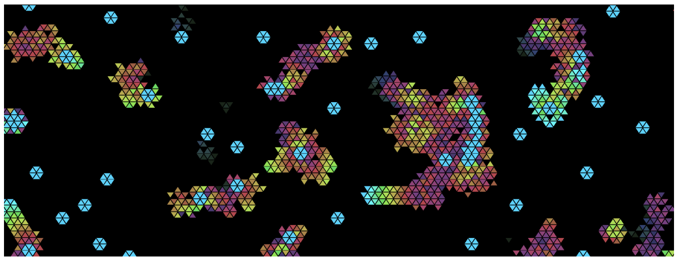
Fig. 21. Triangular Bombers 2

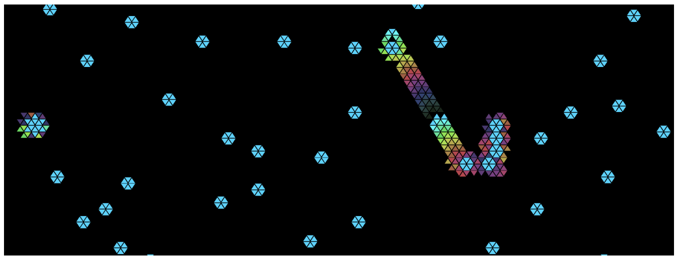
Fig. 22. Triangular Bombers 3

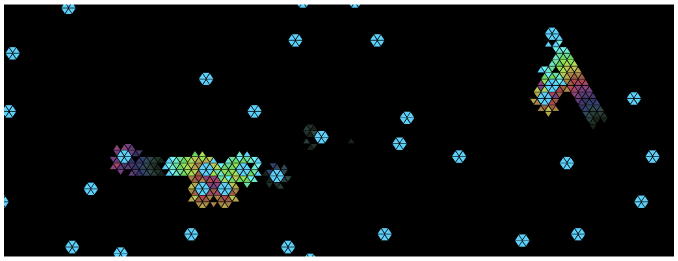
Fig. 23. Triangular Bombers 4

VI. Future Work

Currently most abstraction and modular design happened on the cellular automaton aspect, very little abstraction were done on the data processing pipeline and graphical rendering part.

By abstracting those two components better, it will be even easier for researches to analyze simulation data and implement new visualizations.

Adding more standard component implementation can also help showcase the flexibility of the framework. For example, three-dimensional grid-based cellular automaton back-end is already implemented, but currently there is not a visualization option available yet.

VII. Conclusion

We have demonstrated the very minimal effort used to implement triangular cellular space. While this specific cellular space may have very limited application in practical application, it is not difficult to imagine using this framework to research other cellular automaton systems which do have applications in real-life. For example, implementing a hexagonal cellular space following similar process may help studying behaviors of beehive.

This project is the result of learning from all previous cellular automaton systems. It can be a framework to help researchers implement various high-performance CA systems without designing and implementing everything from scratch. The project not only implements multiple standard cellular automaton components, but also contains a few unconventional components to demonstrate how flexible and extensible this framework is. In short, this framework is “battery included but removable”.

References

[1] Nagel, K., & Schreckenberg, M. (1992). A cellular automaton model for freeway traffic. Journal de physique I, 2(12), 2221-2229.

[2] Richards, F. C., Meyer, T. P., & Packard, N. H. (1990). Extracting cellular automaton rules directly from experimental data. Physica D: Nonlinear Phenomena, 45(1-3), 189-202.

[3] Varas, A., Cornejo, M. D., Mainemer, D., Toledo, B., Rogan, J., Munoz, V., & Valdivia, J. A. (2007). Cellular automaton model for evacuation process with obstacles. Physica A: Statistical Mechanics and its Applications, 382(2), 631-642.

[4] Clarke, K. C., Hoppen, S., & Gaydos, L. (1997). A self-modifying cellular automaton model of historical urbanization in the San Francisco Bay area. Environment and planning B: Planning and design, 24(2

[5] Rennard, J. P. (2002). Implementation of logical functions in the game of life. In Collision-based computing (pp. 491-512). Springer, London.

[6] Zheng, Z., Huang, W., Li, S., & Zeng, Y. (2017). Forest fire spread simulating model using cellular automaton with extreme learning machine. Ecological Modelling, 348, 33-43.

[7] Burstedde, C., Klauck, K., Schadschneider, A., & Zittartz, J. (2001). Simulation of pedestrian dynamics using a two-dimensional cellular automaton. Physica A: Statistical Mechanics and its Applications, 295(3-4), 507-525.

[8] Alarcón, T., Byrne, H. M., & Maini, P. K. (2003). A cellular automaton model for tumour growth in inhomogeneous environment. Journal of theoretical biology, 225(2), 257-274.

[9] Brockfeld, E., Barlovic, R., Schadschneider, A., & Schreckenberg, M. (2001). Optimizing traffic lights in a cellular automaton model for city traffic. Physical review E, 64(5), 056132.

[10] Gerlee, P., & Anderson, A. R. (2007). Stability analysis of a hybrid cellular automaton model of cell colony growth. Physical Review E, 75(5), 051911.

[11] Hovemeyer, D., Spacco, J., & Pugh, W. (2005, September). Evaluating and tuning a static analysis to find null pointer bugs. In Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (pp. 13-19).

[12] Xu, W., DuVarney, D. C., & Sekar, R. (2004, October). An efficient and backwards-compatible transformation to ensure memory safety of C programs. In Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering (pp. 117-126).

[13] Jung, R., Jourdan, J. H., Krebbers, R., & Dreyer, D. (2017). RustBelt: Securing the foundations of the Rust programming language. Proceedings of the ACM on Programming Languages, 2(POPL), 1-34.

[14] FISHER, G., & YEH, C. Comparing Producer-Consumer Implementations in Go, Rust, and C.

[15] Davies, C. H. J. (1995). The effect of neighbourhood on the kinetics of a cellular automaton recrystallisation model. Scripta metallurgica et materialia, 33(7), 1139-1143.

[16] Sutner, K. (1995). On the computational complexity of finite cellular automata. Journal of Computer and System Sciences, 50(1), 87-97.