A Gamified and Multiplayer-Networked Discrete Simulation for Cellular Automata

13 April 2021

Aaron Fox

For Those Curious about Research into Cellular Automata's Applications

Table of Contents

I. Executive Summary

The goal of this project is to design and implement a multiplayer, interactive, and playable cellular automaton simulation. The inspiration of this multiplayer networked game is the late John Conway's "Game of Life," which follows simple discrete simulation rules in its base version to facilitate a Turing complete "zero-player" simulation (so called because the player sets the initial configuration and can only observe the next discrete steps in the base version.) Rather than following the original zero-player nature of this simulation, this project's simulation is gamified such that it can be played over a network by multiple players to compete and create the cellular automata with the largest number of alive cells, turning a zero-player simulation into an *n*-player game. The game can be played at http://www.aaronfox.me/game_of_life as of the writing of this text.

While the original rules of the Game of Life are used in the current iteration of the game, they can easily be adapted in code and changed to make the rules unique to that game. The game was created using the Phaser JavaScript library for the game and UI logic, NodeJS for the server logic, and HTML5 and CSS for the presentation logic. The network follows a client-server architecture for communicating back and forth between the server and its clients. This networked simulation has many potential applications such as in chemistry, physics, biology, communication, and beyond, and therefore this discrete cellular automata simulation can serve fare more purposes than mere entertainment at the patterns that can be formed by the cellular automata.

II. System Description

a) Needs Assessment and System Requirements

While several of the zero-player Game of Life simulations can readily be found on the web, a truly *n*-player networked version of simulating the Game of Life remains unique and elusive on the web. This simulation is therefore more interesting and can yield further research and applications than the original version of the game can. The many applications of studying cellular automata, such as in fields of biology, chemistry, physics, and communication that are already well-researched, can be extended further by allowing the interaction of multiple players in this simulation. There is a larger potential of exploring more details of the natural sciences with more players, and this simulation, while also being a fun and gamified for those not caring about the research aspect of this project, can produce interesting revelations building on previous studies of the zero-game cellular automata performed in the past.

System requirements of this project include an ability for multiple players to access a grid of cells to place to simulate the rules of the Game of Life. The players can click on empty cells to outline the positions the user would like to place their cells. They can then place the cells by clicking a button. Meanwhile, a five-second step timer provides a visual cue to the user of any next steps to give players a sense of each step and give them time to plan their cell placements. The players can also all see other player's living cells as well which are defined in random color selections to differentiate each player. Text indicating the number of currently alive cells and the cells that the player can place will be presented to each player as well. Then, for every step of the

game, the rules of the classic Game of Life are applied. Simplified, the rules to be followed are:

- that any living cell with two or three neighbors moves onto the next level (at the right population)
- that any dead cell with three neighbors at the prior discrete step will become living in the next step (also at the right population), and
- that all the other cells not fitting with the first two rules become dead if not already by either underpopulation or overpopulation.

These rules are applied at every step and each alive and dead cell for every player is displayed to the player accordingly. The UI indicating the time left for the current step, the number of alive cells the player has, and the number of cells the user has left to place are also all updated accordingly.

This simulation is run as a web application meant for laptop and desktop computers to allow for proper width of screen, and therefore has the minimum requirements of requiring a user to have a computer that can run a web browser. Any operating system that can utilize a web browser such as Google Chrome, Mozilla Firefox, or Apple Safari, and has access to a network via Wi-Fi or ethernet should be able to meet the minimum requirements of running this simulation. The minimum system requirements of running a browser such as Google Chrome, including approximately 100MB of hard drive space and 128MB of RAM, are also required to run this simulation [1].

b) System Specifications and Standards Requirements

The language implementation of choice for this project is JavaScript, which was chosen for its ease of communicating real-time information via WebSockets using the socket.io node module. The Phaser game framework is used to design this simulation and is additionally written in JavaScript, making it easy to cross-collaborate communication between the client and server for this networked simulation. For communicating across the networks, NodeJS and Express.js are used for their ease of back end networking development. For representing the information in a web application, HTML5's Canvas element is used alongside JavaScript for communicating the relevant simulation information visually. This is all then hosted using Heroku so that it can be accessed by anyone with a network connection.

Regarding standards requirements, the system must be able be stable enough over the network to not have noticeable latency in displaying the cells of each player. At least eight individual players should be supported over the network without a notice of performance degradation. More support for further players can be added as required, but this would require a more expensive Heroku Dyno to support the load on the server and is currently not supported by the free Dyno being used. So long as the graphical performance is not hindered by this many players, the core standards of this simulation are met. Additionally, the user must be able to interact at all times with the grid of cells to allow for real-time placement of the simulation without noticeable latency.

c) Realistic Design Constraints

As this project was done within the confines of limited economic resources, only Heroku's free Dyno was used, which are virtualized Linux containers for executing the supplied server-side and client-side JavaScript code of this project. The free dyno is limited in its ability to scale to resource demands, as it is really intended for sandbox and personal hobby usage. This limits the number of users that can access the network and once and leads to a degradation in server communication performance over time. Optimizations were made to the program to help limit the size of data being transferred between users, but the limits of the free virtualized Linux container for hosting this service is real constraint in the design of this multiplayer simulation.

Additionally, the lack of proper time synching using JavaScript limits how real-time the simulation can be across users. This inevitably leads to some players being out-of-sync and can cause some players to see their graphics at a slower pace than others. General gameplay strategy and abilities are unaffected by this, but the visualization struggles are a very real constraint when it comes to attempts at time synchronization with NodeJS. Lastly, the memory and speed of the Phaser game engine is inherently limited in its capabilities by its use of JavaScript. As the language execution is not comparable to other languages such as C and C++, the performance of the simulation is constrained by the speed and memory limits of the JavaScript language. Possible optimizations of using other gaming libraries could be used to speed up the game's communication and ease these restraints.

d) Security and Privacy Considerations

As this simulation is hosted over an unsecure HTTP network rather than an SSL-certified HTTPS network, there is inherent insecurity for this simulation. If a player were very determined, they could theoretically crack down to the source code and possibly change some of the player data, but Phaser obfuscates so much of this information and blocks easy access to the network, making this a difficult feat. As this simulation does not concern any private data or any purchases and essentially functions as a game and simulation, strong privacy considerations were not deemed a critical aspect of this system. If this simulation were to continue to grow, however, this could be hosted via HTTPS and have more secure user authentication and fight against any potential DDoS or malicious attacks in the future. However, a lack of any personal input or user data in this system does not make this consideration a priority, for how a user thinks about simulating their Game of Life strategies is not necessary information a potential user would likely care much about anonymizing.

e) System Design Diagrams

The basic client-server architecture design of the main NodeJS simulation server and its bidirectional communications with *n* simulation player clients can be seen in Figure 1.

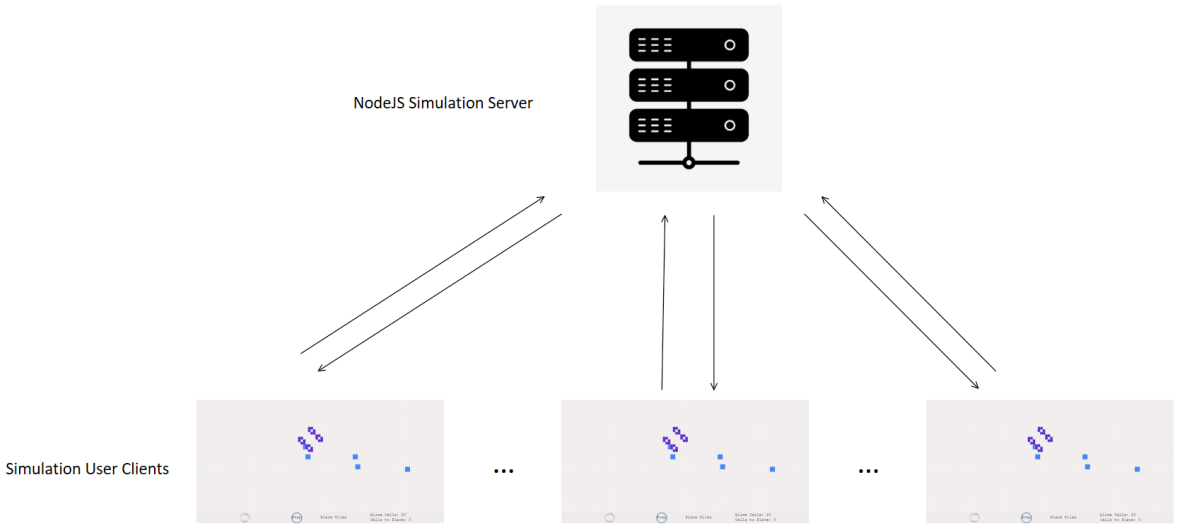


Figure 1. Overarching client-server architecture of the simulation.

This client-server architecture allows for *n* clients to join and send and receive information from the main simulation server. The simulation server sends time synchronization messages every five seconds to indicate a step should be taken. It then takes in each socket's player information and distributes it to all the other players. The sending of the server's time synchronization message works unidirectionally as seen in Figure 2.

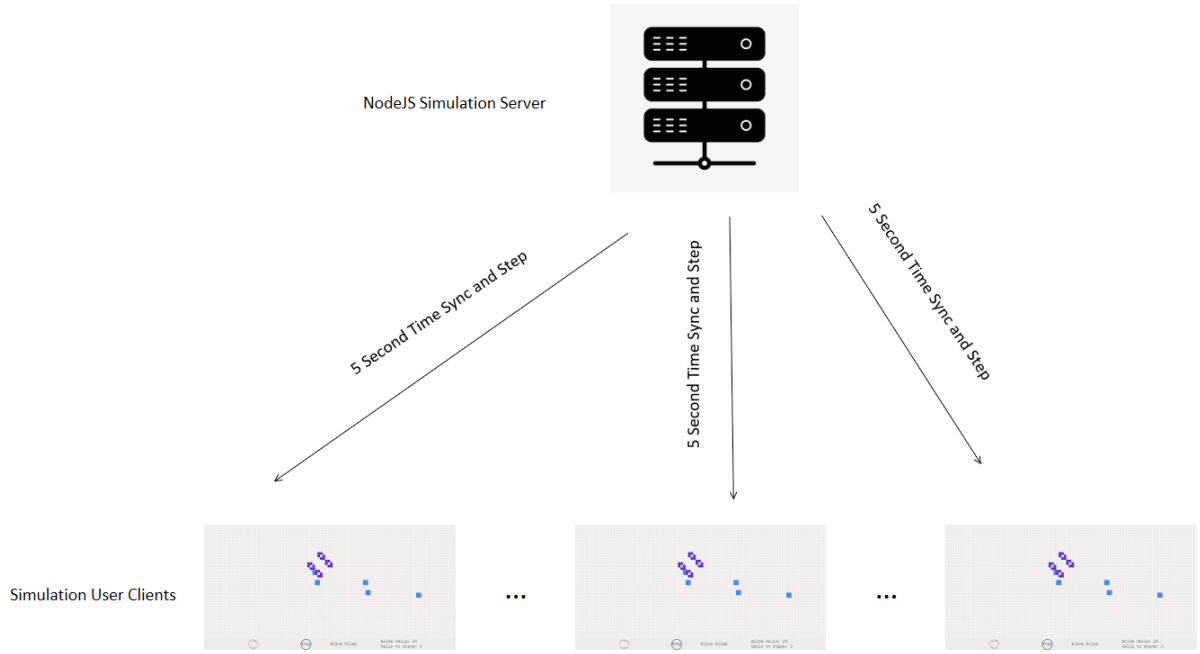


Figure 2. Unidirectional time synchronization messages sent for every 5-second step.

In addition to the messages sent by the server to all the user clients, sometimes a player has to communicate to the other players that it placed a new cell, for example. A client therefore has to first emit a message to the server. The server then processes the client's message and, as appropriate, then broadcasts the changes appropriately to all the other clients that the server is currently connected with. This interaction can be seen in Figure 3.

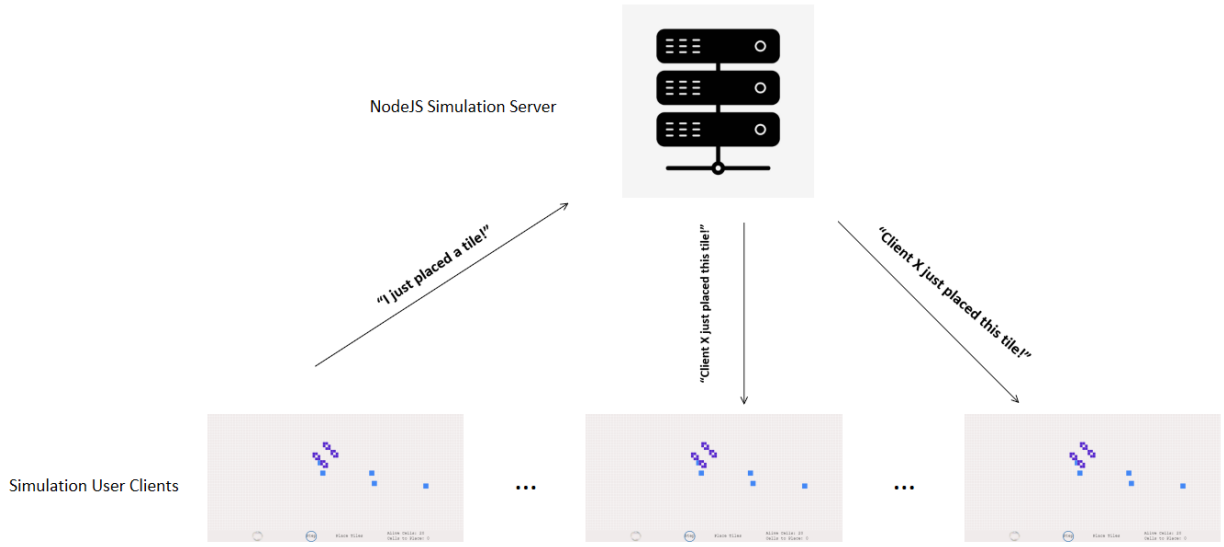


Figure 3. Example of a player communicating their cell/tile placements to other clients.

It should be noted that Figures 1, 2, and 3 all omit the network layer of the Internet which must be used to connect the clients and the server, but this is implied in the diagram as being between the arrows.

f) Hardware/Software Overview

With respect to the hardware required to run this program, a client must have access to a computer or tablet that supports a wide enough (1000x500) screen for displaying the entire user grid. Additionally, the client must be able to run a web browser such as Google Chrome and have a connection to the internet via Wi-Fi or ethernet. A bare minimum of around 100MB of hard drive space and 128MB of RAM should be used for running this web application. The user also needs some form of an input device for simulating clicks, whether that is mouse, trackpad, or touch screen. The development of this system was performed on a Lenovo ThinkPad Yoga P40 and the

hosting of the web server was done through Heroku's virtualized Linux containers on their host servers.

The software required for a user to run the program is a simple web browser such as Google Chrome, Apple Safari, or Mozilla Firefox. The software used to design this program included Visual Studio Code, the Phaser game framework, NodeJS, Express.js, Socket.io, Heroku, and the basics of HTML5, JavaScript, and CSS for display.

g) Economical, Technical, and Time Constraints

A central economical constraint surrounding the performance of this project concerned the dyno type used by Heroku the cloud platform. As there was no budget for this project, Heroku's free dyno was used, and, as this container is meant for sandboxes and experiments, allows for fewer process types, has less available RAM, and lacks autoscaling capabilities. This severely limits the ability for the server-client architecture to scale as the clients grow and limits the number of available clients to about 8 to still run appropriately. Given a proper budget, Heroku's more expensive dynos could be used instead to make up for this deficit.

The main technical constraint of this project concerned the limited processing power contained within the free dyno and the ability for NodeJS to rapidly receive all the player information from the clients, process that data, and then communicate that information back to other relevant clients. Additionally, the use of JavaScript does not compare with a more streamlined programming language and/or gaming framework such as C++ and Godot, for example. The choice of technology used for this system could therefore be changed to optimize performance, speed, and communication.

Lastly, the relevant JavaScript frameworks were selected for their ease of rapid prototyping and development. Given a time budget larger than a semester, the performance and speed of the system could be greatly improved with a framework and language that is designed for creating optimal game performance such as Unity and C#, Godot and C++, or Unreal Engine and C++. This could be a future addition and could help improve on the previously mentioned constraints, but would require more time to design, develop, and test than the use of JavaScript as in this project took.

III. Detailed Implementation

a) Hardware Details

As this was a more software-oriented project, the relevant hardware details of this project are minimal in comparison to the software details. With respect to the development details, this system was developed on a Windows 10 machine with an Intel Core i7-6600U CPU @ 2.60 GHz, 2808 MHz, 2 Cores, and 4 Logical Processors. The computer's localhost was used in the development phase as a base server for testing out the multiplayer network functionality. To host in Heroku over a network, Heroku's virtual Linux containers running which in turn run on Amazon's EC2 cloud-computing platform. Amazon's physical computers are therefore serving as the main network communicator using their Intel Xeon Scalable processors on their specialized hardware designed for hosting web services in the cloud.

b) Software Details

There are three central files relevant to the design, deployment, and appearance of the networked simulation. The relevant files are `server.js`, `game.js`, and `index.html`. `Server.js` controls the backend networking aspect and time synchronization of the project. `Game.js` contains the logic and actions involved in the game. Index.html provides the visual appearance of the game itself and displays the UI to the user.

Server Logic (server.js) Implementation

All of `server.js` is based on NodeJS, a back-end JavaScript environment that can easily run on cloud services such as Heroku. NodeJS allows for using many of it's packages and modules using npm, NodeJs' package manager. Many different npm packages were used, most notably Express.js and socket.io, which facilitate the back end development, routing, and the use of WebSockets for real-time client-server communication.

The core logic of `server.js` is based on the Express.js, which is used for back end web development on top of NodeJS. It serves as the base web framework and focuses on setting the port environment of the server and allows for creating the server which uses WebSockets to facilitate communication between the main server and its clients. Express.js is used to send files and the route to the game found in index.html.

After connecting the server created by Express.js, WebSockets are used and created using the socket.io Node module. WebSockets are a protocol used for full-duplex (two-way) communication over just one TCP connection. This allows for nearly instantly and consistently communicating between each server and client without

having to consistently poll the server. Instead, event-driven responses are triggered and received to trigger necessary events in the simulation in real time. Once the server's WebSocket is created and connected to `server.js's` main server, the server was then set to listen to connections and disconnections from clients to this server.

Upon detecting a new connection, the server's `socket.io` listener will then add a new player with the client's unique socket ID to an array of players that the server keeps track of. Each player is assigned several variables, including the number of cells the player has left to place, the locations of the currently placed cells, the color of the client's cells, and the number of cells on the grid. The server then emits all currently existing players to the new socket so that the player can include the all the other already-existing players to their respective grid. Next, the server broadcasts to all the other existing players the current player's new information such as the new player's cell color and player ID so that the other players in the simulation are aware of the new client.

Inside the connection listener of the server, a disconnection listener is also implemented so that a player can be deleted from the current players array, and all players are alerted of the client's disconnection so that they can remove the leaving client from their list of players. There are also two more listeners inside the connection listener. The `tilePlaced` listener, which is sent whenever a particular client places a cell in a tile, updates the current player's placed cell tile locations with the new cell tile location and then broadcasts the updated player's tiles to all the other connected players. The last listener inside the connection listener, `clearCells`, simply clears out all

the player's tiles and likewise broadcasts to all the other connected clients that the relevant player's cell placements is now empty. The last section of `server.js` includes a `setInterval` call that is used for time synchronization and triggering a step every 5 seconds. This step function is emitted to all players and forces time synchronization upon each player and alerts the players to update their simulation to the next discrete step.

Game Logic (game.js) Implementation

The game logic is contained with `game.js`. This file takes advantage of the Phaser game framework API to allow for abstracting away much of the complications of displaying the UI and coordinating the game logic. The initial lines of the file serve to initialize the canvas configuration of the game implemented in an HTML5 Canvas element using Phaser Game API. The `create` and `preload` functions are initialized in this configuration, as well as several global variables that are referenced throughout the script, such as the graphics, references to buttons and text objects, references to the player object, the size of the grid, and other important variables to keep track of such as the number of steps left to increment the player's cells that they can place.

Next, the `preload` function is used to load in a PNG of a bubble icon which is used to visually represent to each client how much time is left until the next step by floating to a circle labeled step. This image serves as a playful visual cue to the client of how much time is remaining until the next discrete step occurs in the simulation. The create function, which is the main portion of the game logic script, contains the `socket.io`

WebSocket information, all the socket listeners, button and text implementations, and input logic for the device the client is using (mouse, touchscreen, trackpad, etc.)

After adding the respective interactive hovering, clicking, and removal of the mouse pointers on the buttons and setting the respective texts and shapes, the `socket.io` listeners are set up such that they can be received from the server whenever the server is ready to deliver new information to each client. The client `currentPlayers` listener, for example, sets up the UI to draw all the other currently connected clients' cell placements as well as the current players' cell placements. Another listener, `newPlayer`, listens for the server to alert the client that another client has been added to the server and then adds that player to its Phaser group object for managing all other clients. The next listener, `step`, is used for time synchronization between each client as the server calls it every 5 seconds, effectively restarting the discrete time steps when the clients inevitably get out of step with each other. As there is no global clock in this system, this is the next best alternative to syncing the steps of each client. In the `step` listener, the number of tiles a certain client increases every `STEPS_REQUIRED_TO_INCREMENT_CELLS_TO_PLACE` steps, which is normally set to 5, to make the simulation more interesting and dynamic. Next the Game of Life rules are applied by calling the `applyGoLRules` function, which is discussed later on in this section.

The `create` function then implements a disconnected listener that the server calls whenever another client disconnects so that the still-connected client can remove that user's current cells and information from their current `players` array. An `otherTileWasPlaced` listener is also implemented here, and this simply draws all the

other player's current cells whenever the server alerts the client that there are more cells to draw in the tiles. The `graphics` object is also implemented here using Phaser's graphics API for easily drawing and creating shapes, images, and text inside. An initial grid drawing of empty squares is also drawn for the UI, and, for the last aspect of the `create` function, the actual placing of tiles logic is implemented so that users can place an outline of cells they would like to place in available cells blocks. The client can later place those outlined tiles by clicking the "Place Tiles" button text at the bottom of the Canvas UI.

Several helper functions follow the main `create` function, and all the functions are called inside the `create` function to support its many listeners that allow for the networked discrete simulation to take place. The `redrawGrid` function, for example, simply clears the current graphics object to prevent an overload of memory drawing new shapes and graphics objects over old objects, redraws the empty grid, and then calls on the `placeFilledTiles` and `drawTilesToPlace` functions to redraw the UI of the other player's current cell placements and to redraw the current client's cells to place, respectively.

While there are many auxiliary functions such as `getNumberOfNeighboringBlocks`, which returns the number of alive cells bordering a current cell out of the 8 possible neighboring adjacent and diagonal-adjacent blocks, and `getLocationIndex`, which returns the index of a grid if it exists or -1 otherwise, the central function to this simulation lies in the `applyGoLRules` function.

The Game of Life Simulation (`applyGoLRules`)

The `applyGoLRules` function begins to apply the new simulation rules by first creating a copy from the current player's `placeTileLocations` object reference. A copy is needed to prevent corrupting the current data of the placed cell locations array. For example, if a cell was determined to be dead and removed from the original place cell locations array instead of the copy of the new placed cell locations array, then that would interfere with the logic of other neighboring cells as those cells would not be able to understand that, in the previous discrete step, it had a neighboring live cell, which would corrupt the logic of the simulation. Since each step is discrete, a copy of the referenced cell locations array must be made for every call to this function. The function then checks for the length of the place tile locations array. If it is less than 3, then all current alive cells must die as that means that no cells will have 2 or 3 neighbors, which is a requirement for a cell to become living.

Otherwise, two nested for loops iterate over every grid and column of the grid. In the nested for loop, the current element is assigned the current `i` and `j` coordinates that represents the coordinates of each cell tile location in the grid, and a `numNeighbors` variable stores the number of neighbors of that tile space that could potentially house the cell based on its Moore neighborhood, which involves the 8 nearest neighboring cells: the adjacent cell blocks and the diagonally adjacent cell blocks. The `getNumberOfNeighboringBlocks` function works by iterating through all 8 possible neighbors and incrementally counting if an alive cell is in those neighboring blocks. This function is able to determine if a cell exists in the neighboring cells using the

`getLocationIndex` function which returns the index of an existing cell in the current alive aells array and -1 otherwise.

If the current cell whose number of alive neighboring cells was just calculated in `numNeighbors` in the main `applyGoLRules` function is alive itself, then we check if the alive cell has less than two alive neighboring cells or greater than three neighboring alive cells. If so, then the cell is deemed dead and is removed from the new copy of the tile placement array. Otherwise, the alive cell is left alone as it remains alive per the rules laid out in the Game of Life. If the cell is dead, an if statement occurs that checks if the dead cell has three neighbors. If so, then the dead cell becomes alive and is added to the new copy of the tile placements.

After the two nested for loops complete, the old place tile locations is overwritten by the copy and the UI text and visuals are redrawn and updated accordingly.

Software Development Implementation

With respect to the software development implementation, Visual Studio Code was used to develop the code for its excellent syntax highlighting and error checking for JavaScript files. GitHub and git were used for version control and keeping track of changes to this code. The testing of the output was done using Google Chrome and Mozilla Firefox browsers. The windows command line was used for starting the servers and the `nodemon` package was used for rapid development in restarting the server whenever file changes were made.

IV. Evaluation Experimental Procedure and Result Analysis

As this was an interactive and gamified discrete simulation requiring different inputs of networks, this system was most easily tested, when ready, among my different friends and family that could all access the server from different computers. I would play along the simulation with them and receive their feedback on ease of understanding the UI, quality and latency of the server, and visual appeal of the system, among other metrics. This was effectively an iterative and qualitative process as the core aspects of the game and the rules were put into place and followed but the design and structure of the game could always be improved. Based on the feedback and results from friends and family, I was able to make changes to the discrete gamified simulation, such as adding a bubble Phaser Tween that helped provide visual cues to each client of the time left in the current step (before, it was assumed the user knew it was every five seconds.)

After setting up some basic metrics such as acceptable lag, latency, and drawing of the grid for many different objects, I made sure to continually optimize the code to meet those metrics. Different approaches of data structures, including hash maps, arrays, and 2D arrays all tested and analyzed for their speed performance and memory usage. As soon as the optimal way of structuring and passing the data between clients and server was found, the optimal structures and algorithms were kept in the code to minimize lag and improve UI speeds.

The resulting system can be found and played at TODO URL HERE. Some screenshots of the game and some of the different Game of Life structures can be seen in Figure 4.

V. Societal Impact

As this is a gamified networked simulation of the Game of Life, the main societal impact should be improving society's happiness, curiosity, and cleverness when individuals are able to learn and realize the many cool patterns that arise in the simulation based on the configuration of the cells that they placed. As with all games and simulations however, this game could have the possibility of worsening human productivity via game addiction, and that must be a serious ethical consideration. Placements on how long an IP address could have access to the server in one day, for example, could help to prevent any possible addictions occurring.

As this is hosted over an HTTP network rather than an HTTPS network, meaning that all user data sent over the network is unencrypted, there are possible legal issues with storing user input information per the EU's GDPR rules, for example. Although none of the current information being stored involves personal data and only collects input data based on the user's simulation inputs, a lawyer may have to investigate the storing and collecting of any user input data in countries with more strict rules and regulations. Any lawsuit is unlikely, however, as the data is not being stored permanently in a database, for that is not in the system requirements of this project. If this project were to be extended to include storing user information via usernames and passwords along with their simulation data from where they left off, however, then more concern would have to be made on the security and integrity of this application to

avoid any legal issues in the future concerning inappropriately storing personal data such as passwords.

VI. Contributions of Project to Society

The study of cellular automata has many different applications for research into real-life systems in chemistry, physics, biology, computer science, and beyond. Many patterns in nature, for example, can be almost exactly simulated using cellular automata. The Cone Snail, for example, follows the rule of cellular automata in the design of its shell as its shell is able to excrete different piments that either turn a cell "dead" or "alive," each represented by a different color [2]. Other applications of study of cellular automata include cephalopods, botanical stomas, and fibroblasts, for example [3][4][5].

Besides this already striking research applications into simulating nature, many other aspects of the natural sciences can also be explored, including chemistry, physics, computer science, and communication [6][7][8]. As it stands, much of the research into the applications of using cellular automata simulations have not involved multiplayer networked simulations. Doing so may allow for researchers and otherwise curious minds to explore even more applications and possibilities of simulating the natural sciences and computer science with multiple different input configurations from many different simulation clients with the ability to consistently update and change the simulation strategies as the system learns and adapts to the cell placements.

Although the system would need server hardware improvements to handle larger server loads, there are many areas where this simulation could be extended to

simulate even larger applications such as generative music and more complex mathematical and physics models. The possibility of research applications to this are therefore far larger possible contributions to society than the mere fun and entertainment that can be had from playing this gamified simulation.

VII. Conclusion

This project certainly aided in bringing together much of what I learned in undergrad, including how to use optimal data structures, which algorithms to use, networking using a client-server architecture, the basics of automata theory, how to bring together complex software engineering systems successfully, and, of course, creating simulations in my simulation and modeling of discrete systems courses. While many of these courses taught me a breadth of their respective subjects, this project really helped me to understand the depth of bringing all these concepts together. In addition, some of my more theoretical, non-programming classes, such as my automata theory class for example, were able to be realized and thoroughly understood in this project in this real-world application, therefore bridging the important gap between theoretical learning and practical application. I had to do my own research into many of the tools, concepts, and software used in the development of this application, but this research was made much easier thanks to the excellent foundation provided to me during my undergrad courses. Many more extensions to this project, including security analysis, more complex distributed cloud systems and servers, and the inclusion of a database could also provide further integrations and extensions to the excellent undergrad classes and experiences that UofL has to offer.

VIII. Appendix

a) References

[1] "Chrome Browser system requirements," *Google Chrome Enterprise Help*. [Online]. Available: https://support.google.com/chrome/a/answer/7100626?hl=en. [Accessed: 07-Mar-2021].

[2] Coombs, Stephen, "The Geometry and Pigmentation of Seashells", pp. 3–4, 2009.

[3] Peak D, West JD, Messinger SM, Mott KA, "Evidence for complex, collective dynamics and emergent, distributed computation in plants," *Proc Natl Acad Sci U S A*. 2004.

[4] Andrew Packard, "A 'neural' net that can be seen with the naked eye," *Neuronal Coding Of Perceptual Systems*, pp. 397–402, 2001.

[5] Y. Bouligand, "Disordered Systems and Biological Organization," 1986.

[6] A. K. Dewdney, "The hodgepodge machine makes waves," *Scientific American*, p. 104, 1988.

[7] Sethna, James P. "Statistical Mechanics: Entropy, Order Parameters, and Complexity." *Oxford University Press*, 2008.

[8] Chowdhury, D. Roy; Basu, S.; Gupta, I. Sen; Chaudhuri, P. Pal, "Design of CAECC - cellular automata based error correcting code," 1994

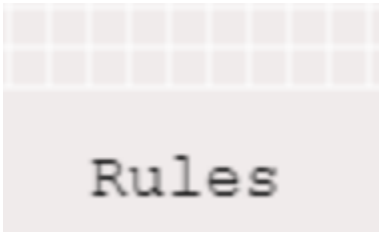
b) User Manual

The User Manual for this is referenced in the game itself located at http://www.aaronfox.me/game_of_life and can be accessed there or at https://docs.google.com/document/d/169VO83FgXEXiv1NImkiVdlav88jXB17FoYv6h9FYMQA/edit?usp=sharing.

For reference, they are placed here as well.

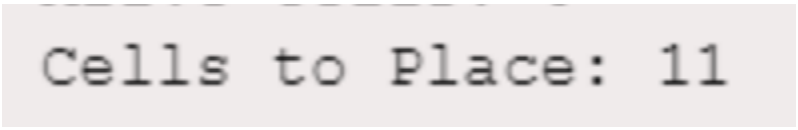
A Gamified and Multiplayer-Networked Discrete Simulation for Cellular Automata Ruleset

The rules of the game are pretty straight forward. To place a potential cell, click on any cell that you do *not already* have a cell on to place it. For example, a cell that looks like



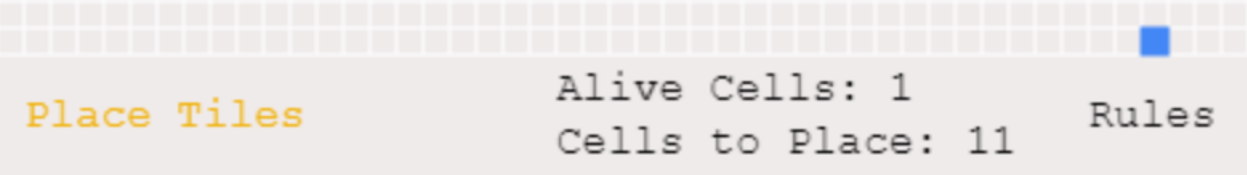
will become

If you click on the empty cell above the ‘L’ in rules. A cell outlined in blue will mean that you have a potential cell placement. Notice how this reduces the number of cells you have to place in the UI to 11, as seen in the UI below:



This indicates the number of cells that you may place at a time. You begin with 12, and you *will get one more* cell to place every 5 steps as long as the sum of the potential tiles to place plus the outlined cells does not exceed 12.

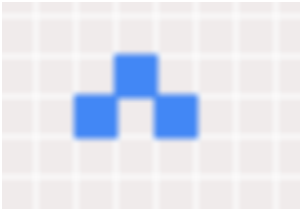
Click the ‘Place Tiles’ button to place a cell. This will turn the outlined cell completely shaded blue, like so:



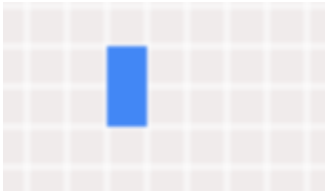
Note how the number of alive cells incremented as you added a new alive cell. Did your single cell die right away at the next step? That's expected behavior, because this simulation is following the rules of the late John Conway's Game of Life. The game works in discrete steps of 5 which are indicated by the Bubble animation. When the bubble reaches the 'Step' circle, a new discrete step occurs. To elaborate on the rules of the game, they are:

• that any living cell with two or three neighbors moves onto the next level (at the right population)
• that any dead cell with three neighbors at the prior discrete step will become living in the next step (also at the right population), and
• that all the other cells not fitting with the first two rules become dead if not already by either underpopulation or overpopulation.

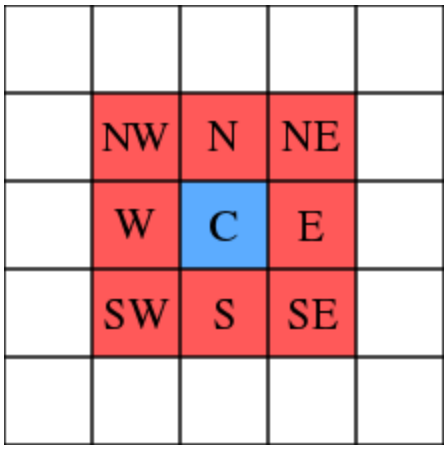
To explain each rule, for example, picture these three cells:



What is the next step? Well, it will become:



That is because the topmost alive (blue) cell of the first image has two neighbors according to a Moore neighborhood, which looks like:

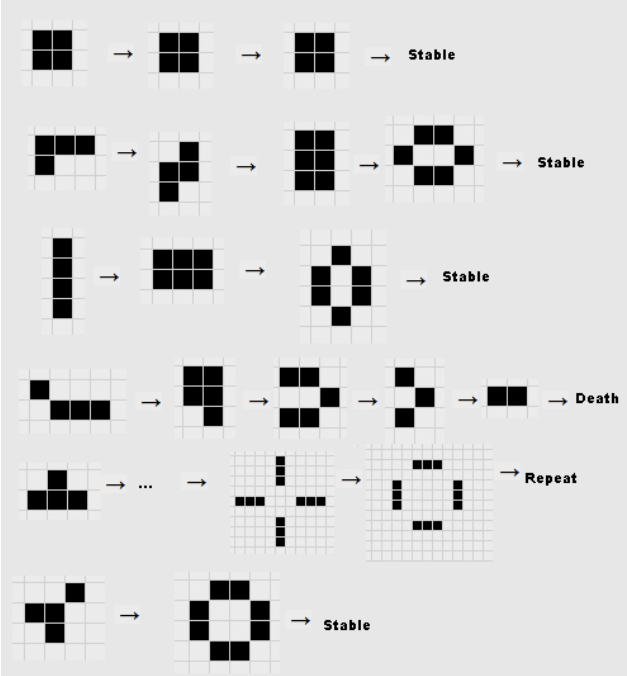


Note how the topmost blue cell has a neighbor at SW and SE according to its Moore Neighborhood. The topmost blue cell in the original image will therefore survive. The leftmost and rightmost blue cells, however, die as they only have one neighbor.

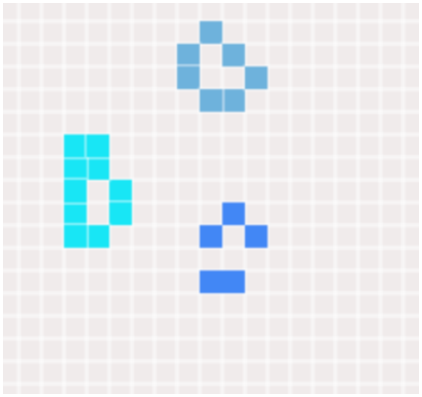
Additionally, note how a brand new cell was born in the second image! This was because it had exactly three neighbors, ones at N, W, and E at the chart above. All other cells die or remain dead that don't follow these rules.

These simple rules can lead to many amazing and complex figures and simulations that have applications in biology, chemistry, physics, and beyond! For example, cone snail shells can be simulated with this cellular automata in this way!

Some examples of cool shapes you can make can be seen below:



Other players will show up as randomly generated colors, like so:



I encourage you to explore your own shapes, patterns, and to compete with others to find the the maximum number of alive cells you can achieve!

If you have any questions, concerns, or would like to reach out to me, feel free to message me on LinkedIn or elsewhere!

c) Source Code

The relevant source code of this project is included here.

server.js:

```
var express = require('express'),
    app = express()//,
// bodyParser = require('body-parser'),
// methodOverride = require('method-override')

var express = require('express');
var app = express();

app.set('port', (process.env.PORT || 8081));

app.use(express.static(__dirname + '/public'));

// ==================================== BEGIN Multi game logic ====================================
// var server = require('http').Server(app);
// // NOTE: changed this from original
// var io = require('socket.io')(server);

var server = app.listen(app.get('port'));
var io = require('socket.io')(server);

// Keep track of all players in game with players
var players = {};

const MAX_TILES_TO_PLACE = 12;

io.on('connection', function (socket) {
    console.log('a user connected');
    // create a new player and add it to our players object
    players[socket.id] = {
        tilesToPlace: MAX_TILES_TO_PLACE,
        placedTileLocations: [],
        tilesToPlaceLocations: [],
        numberOfTilesOnBoard: 0,
        playerId: socket.id,
        color: '0x' + (Math.floor(Math.random() * 16777215).toString(16))
    };
    // Send the current players to this player socket only
    // Note: socket.emit sends objects to just this socket
    //       while socket.broadcast.emit sends to all other sockets
    socket.emit('currentPlayers', players);

    // send the current scores
    // socket.emit('scoreUpdate', scores);

    // Let all other players know of this new player
    socket.broadcast.emit('newPlayer', players[socket.id]);

    socket.on('disconnect', function () {
        console.log('user disconnected');
        // remove this player from our players object
        delete players[socket.id];
        // Check if this is last player
        if (players.length == undefined) {
            // Reset board since no players left
            // This is done automatically by logic of game.js
        }
        // emit a message to all players to remove this player
        io.emit('disconnected', socket.id);
    });

    // Sends to all other players that this player placed a new tile
    socket.on('tilePlaced', function(tileData) {
        players[socket.id].placedTileLocations.push(tileData);

        // Emit message to all players that tile was placed
        socket.broadcast.emit('otherTileWasPlaced', players[socket.id])
    })

    // Clears out all previous tiles to clear board of any previously removed
    // tiles that were once placed
    socket.on('clearCells', function () {
        players[socket.id].placedTileLocations = []

        // Emit message to all players to clear out current placedTileLocations array
        socket.broadcast.emit('otherTileWasPlaced', players[socket.id])
```

```
    })
});

app.get('/multi_game', function (req, res) {
    // res.render('pages/multi_game');
    res.sendFile(__dirname + '/views/pages/multi_game.html');
    // res.sendFile('pages/multi_game.html');

});
// slide = 0
setInterval(step, 5000); // advance slides every 5 seconds

function step() {
    io.sockets.emit('step', 2);
}
```

game.js:

```
// Author: Aaron Fox
// Description: Game logic for creating an adjustable gamified version of the Game of Life cellular automata simulation.

// Configuration variables for the game and canvas
let CANVAS_WIDTH = 1000
let CANVAS_HEIGHT= 500

var config = {
    type: Phaser.AUTO,
    parent: '#canvas',
    width: CANVAS_WIDTH,
    height: CANVAS_HEIGHT,
    backgroundColor: '#f0ebeb',
    scene: {
        preload: preload,
        create: create,
    }
};

// Refer to this for scaling game window
// game = new Phaser.Game(window.innerWidth * window.devicePixelRatio, window.innerHeight * window.devicePixelRatio, Phas
er.CANVAS, 'gameArea');
const MAX_TILES_TO_PLACE = 12;
const STEPS_REQUIRED_TO_INCREMENT_CELLS_TO_PLACE = 5;

// Global variables used
var game = new Phaser.Game(config);
var graphics;
var placeButton;
var rulesButton;
var player = 0;
var socket;
var bubbleTween;
var aliveCellsText;
var cellsLeftToPlaceText;
var steps_since_cell_to_place_incremented = 0;
const MIN_NEIGHBORS_TO_SURVIVE = 2;
const MAX_NEIGHBORS_TO_SURVIVE = 3;

// Determine the size of the cells of the game and canvas
let hspace = 10;
let size = {
    x: CANVAS_WIDTH / hspace,
    y: (CANVAS_HEIGHT - 50) / hspace
}

// Used for loading the image of a bubble
function preload() {
    this.load.image('bubble', 'assets/smaller_bubble.png');
}

// Main create function that maintains its listeners, UI adjustments, and takes in input
```

```
function create() {
    var self = this;
    this.socket = io();
    socket = this.socket;
    this.otherPlayers = this.add.group();

    // Progress bar represented as bubble
    var image = this.add.image(100, (CANVAS_HEIGHT - (50 / 2)), 'bubble');

    bubbleTween = this.tweens.add({
        targets: image,
        x: 400,
        duration: 5000,
        ease: 'Sine.easeInOut',
        loop: -1,
        loopDelay: 0
    });

    var r2 = this.add.circle(405, 475, 20);
    r2.setStrokeStyle(2, 0x1a65ac);

    // Visual texts and buttons to display
    stepText = this.add.text(CANVAS_WIDTH / 2 - 115, (CANVAS_HEIGHT - (50 / 2) - 10), 'Step', { fill: '#000000' })
    aliveCellsText = this.add.text(CANVAS_WIDTH / 2 + 200, (CANVAS_HEIGHT - (50 / 2) - 20), 'Alive Cells: 0', { fill: '#0
00000' })
    cellsLeftToPlaceText = this.add.text(CANVAS_WIDTH / 2 + 200, (CANVAS_HEIGHT - (50 / 2)), 'Cells to Place: ' + MAX_TIL
ES_TO_PLACE, { fill: '#000000' })

    placeButton = this.add.text(CANVAS_WIDTH / 2, (CANVAS_HEIGHT - (50 / 2) - 10), 'Place Tiles', {fill: '#000000'})
    .setInteractive()
    .on('pointerdown', () => placeTiles())
    .on('pointerover', () => placeButtonHoverState())
    .on('pointerout', () => placeButtonRestState());

    // Rules button
    rulesButton = this.add.text(CANVAS_WIDTH / 2 + 400, (CANVAS_HEIGHT - (50 / 2) - 10), 'Rules', { fill: '#000000' })
        .setInteractive()
        .on('pointerdown', () => displayRules())
        .on('pointerover', () => rulesButtonHoverState())
        .on('pointerout', () => rulesButtonRestState());

    // When a new player is added, add all players including current player
    this.socket.on('currentPlayers', function(players) {
        Object.keys(players).forEach(function (id) {
            if (players[id].playerId === self.socket.id) {
                addPlayer(self, players[id]);
                player = players[id];
            } else {
                addOtherPlayer(self, players[id]);
                // Draw other players tiles as well
                drawTiles(self, players[id]);
            }
        });
    });

    // When a new player is added, add player to current players of this socket
    this.socket.on('newPlayer', function(playerInfo) {
        addOtherPlayer(self, playerInfo);
    });

    // Main step function that is set by setInterval call in the server
    this.socket.on('step', function (playerInfo) {
        bubbleTween.restart();

        // Increment number of cells user can place by one if not already at max tiles to place
        // Ensure to also check current tiles to place as well to make sure user isn't trying to sneak more alive cells i
n
        // than they're allowed
        if (player.tilesToPlace + player.tilesToPlaceLocations.length < MAX_TILES_TO_PLACE) {
            if (steps_since_cell_to_place_incremented == STEPS_REQUIRED_TO_INCREMENT_CELLS_TO_PLACE) {
                steps_since_cell_to_place_incremented = 0;

                // Increment player's cells to place count
                player.tilesToPlace++;
                updateCellsToPlaceText();

            } else {
                steps_since_cell_to_place_incremented++;
            }
```

```
    } else {
        steps_since_cell_to_place_incremented = 0;
    }

    // Apply GoL rules here appropriately
    applyGoLRules();
});

// Remove game object from game
this.socket.on('disconnected', function (playerId) {
    self.otherPlayers.getChildren().forEach(function (otherPlayer) {
        if (playerId === otherPlayer.playerId) {
            otherPlayer.destroy();
        }
    });
});

// Draw all other player's tiles with this socket
this.socket.on('otherTileWasPlaced', function(playerInfo) {
    var color = 0xffffff;
    var thickness = 1;
    var alpha = 1;
    graphics.lineStyle(thickness, color, alpha);
    self.otherPlayers.getChildren().forEach(function (otherPlayer) {
        if (playerInfo.playerId === otherPlayer.playerId && player.playerId != otherPlayer.playerId) {
            // Update other player's tiles
            drawTiles(self, playerInfo);
        }
    });
});

// Main graphics object upon which all the UI is drawn
graphics = this.add.graphics({
    lineStyle: {
        width: 1,
        color: 0xffffff,
        alpha: 1
    }
});

// Draw initial grid
for (let ix = 0; ix < size.x; ix++) {
    for (let iy = 0; iy < size.y; iy++) {
        graphics.strokeRect(ix * hspace, iy * hspace, hspace, hspace);
    }
}

// Upon a user clicking in an empty cell, have it be placed in the cell tiles to be placed structure.
this.input.on('pointerdown', (pointer) => {
    if (pointer.isDown) {
        var color = 0x4287f5;
        var alpha = 1.0;
        graphics.fillStyle(color, alpha);
        var color = 0x4287f5;
        var thickness = 1;
        var alpha = 1;
        graphics.lineStyle(thickness, color, alpha);
        // Round position to next greatest hspace
        let x = Math.floor(pointer.position.x / hspace) * hspace;
        let y = Math.floor(pointer.position.y / hspace) * hspace;
        // Check for clicking on already existing square so we can remove that square
        containsLocationIndex = getLocationIndex(player.tilesToPlaceLocations, { x: x, y: y });
        if (containsLocationIndex > -1) {
            player.tilesToPlaceLocations.splice(containsLocationIndex, 1)
            player.tilesToPlace++;
            updateCellsToPlaceText();
            var color = 0xffffff;
            var thickness = 1;
            var alpha = 1;
            graphics.lineStyle(thickness, color, alpha);
            graphics.strokeRect(x, y, hspace, hspace);

            // Must check if any tiles near this tile. If so, must recolor those as well
            adjacentNeighbors = getAdjacentNeighboringBlocks({ x: x, y: y });
            if (adjacentNeighbors.length > 0) {
                var color = 0x4287f5;
                var thickness = 1;
                var alpha = 1;
                graphics.lineStyle(thickness, color, alpha);
```

```
                    neighbors.forEach(function (element) {
                        graphics.strokeRect(element.x, element.y, hspace, hspace);
                    });
                }
                // End adjacent neighbors check
            } else if (x < size.x * hspace && y < size.y * hspace && player.tilesToPlace > 0) {
                // Also check if tile is already in placedTileLocations
                // Here, simply include tile in tiles to place array
                // Subtract amount of tiles player can place
                var cellIsDead = true;
                for (var i = 0; i < player.placedTileLocations.length; i++) {
                    if (x == player.placedTileLocations[i].x && y == player.placedTileLocations[i].y) {
                        cellIsDead = false;
                        break;
                    }
                }
                if (cellIsDead) {
                    player.tilesToPlace--;
                    updateCellsToPlaceText();
                    graphics.strokeRect(x, y, hspace, hspace);
                    // Emit placed tile
                    player.tilesToPlaceLocations.push({ x: x, y: y });
                }
            }
        }
    })

}

// Clears grid and then redraws the user's tiles to place and filled tiles
function redrawGrid() {
    // Draw initial grid
    graphics.clear();
    for (let ix = 0; ix < size.x; ix++) {
        for (let iy = 0; iy < size.y; iy++) {
            graphics.strokeRect(ix * hspace, iy * hspace, hspace, hspace);
        }
    }
    placeFilledTiles();
    drawTilesToPlace();
}

// Main Game of Life function. Applies the rules of Game of Life
// as set in the constant variables in this method.
// The performance of the game can be adjusted based on changes
// to the constants such as setting MIN_NEIGHBORS_TO_SURVIVE to a number other than 2
// and MAX_NEIGHBORS_TO_SURVIVE to a number other than 3, which are the default settings
// for the traditional Game of Life game.
function applyGoLRules() {
    // Clone of grid
    var newTilePlacements = [...player.placedTileLocations]

    if (newTilePlacements.length < 3) {
        newTilePlacements = []
    } else {
        // Iterate through each row of grid
        for (var i = 0; i < CANVAS_WIDTH; i += hspace) {
            // Iterate through each column
            for (var j = 0; j < CANVAS_HEIGHT - 50; j += hspace) {
                // Get number of neighbors for this tile
                currElement = { x: i, y: j };
                numNeighbors = getNumberOfNeighboringBlocks(currElement);

                index = getLocationIndex(player.placedTileLocations, currElement);
                // If cell is alive
                if (index > -1) {
                    if (numNeighbors < MIN_NEIGHBORS_TO_SURVIVE || numNeighbors > MAX_NEIGHBORS_TO_SURVIVE) {
                        // Remove this cell from placedTileLocations since it died
                        for (var k = 0; k < newTilePlacements.length; k++) {
                            if (newTilePlacements[k].x == currElement.x && newTilePlacements[k].y == currElement.y) {
                                newTilePlacements.splice(k, 1);
                                break;
                            }
                        }
                    }
                } else {
                    // Otherwise, this cell is dead and should be alive if 3 neighbors
                    if (numNeighbors == 3) {
                        newTilePlacements.push(currElement);
```

```
                    }
                }
            }
        }
    }

    // Now update player data
    player.placedTileLocations = newTilePlacements;

    // Redraw grid and update it
    redrawGrid();

    // Update UI accordingly
    updateAliveCellsText();
}

// Updates the number of alive cells text UI to alert the user of their current 'score'
function updateAliveCellsText() {
    aliveCellsText.text = 'Alive Cells: ' + player.placedTileLocations.length;
}

// Updates the number of cells a user has left to place in the UI
function updateCellsToPlaceText() {
    cellsLeftToPlaceText.text = 'Cells to Place: ' + player.tilesToPlace;
}

// Returns neighboring blocks of a cell
function getAdjacentNeighboringBlocks(location) {
    // Check up, right, down, left
    var xLocs = [hspace, 0, -1 * hspace, 0]
    var yLocs = [0, hspace, 0, -1 * hspace]
    neighbors = []
    for (var i = 0; i < xLocs.length; i++) {
        currLocation = {x: location.x + xLocs[i], y: location.y + yLocs[i]};
        locationIndex = getLocationIndex(player.tilesToPlaceLocations, currLocation);
        if (locationIndex > -1) {
            // Then add this to neighbors
            neighbors.push(player.tilesToPlaceLocations[locationIndex]);
        }
    }
    return neighbors;
}

// Returns number of neighboring blocks of a cell
function getNumberOfAdjacentNeighboringBlocks(location) {
    // Check up, right, down, left
    var xLocs = [hspace, 0, -1 * hspace, 0]
    var yLocs = [0, hspace, 0, -1 * hspace]
    count = 0
    for (var i = 0; i < xLocs.length; i++) {
        currLocation = { x: location.x + xLocs[i], y: location.y + yLocs[i] };
        locationIndex = getLocationIndex(player.placedTileLocations, currLocation);
        if (locationIndex > -1) {
            count++;
        }
    }
    return count;
}

// Returns number of neighboring blocks of a cell out of a possible 8
function getNumberOfNeighboringBlocks(location) {
    // Check up, up-right, right, down-right, down, down-left, left, up-left
    var xLocs = [hspace, hspace, 0, -1 * hspace, -1 * hspace, -1 * hspace, 0, hspace]
    var yLocs = [0, hspace, hspace, hspace, 0, -1 * hspace, -1 * hspace, -1 * hspace]
    count = 0
    for (var i = 0; i < xLocs.length; i++) {
        currLocation = { x: location.x + xLocs[i], y: location.y + yLocs[i] };
        locationIndex = getLocationIndex(player.placedTileLocations, currLocation);
        if (locationIndex > -1) {
            count++;
        }
    }
    return count;
}

// Checks if an array contains x and y locations already
// Returns index of element if found and -1 otherwise
function getLocationIndex(array, location) {
    for (var i = 0; i < array.length; i++) {
```

```
        element = array[i];

        if (element.x == location.x && element.y == location.y) {
            return i;
        }
    }
    return -1;
}

// For when user is hovering over 'Place Tile' button
function placeButtonHoverState() {
    placeButton.setStyle({ fill: '#f0b207' });
}

// For when no mouser hovering or clicking on 'Place Tile' button
function placeButtonRestState() {
    placeButton.setStyle({ fill: '#000' });
}

// For when user is hovering over 'Rules' button
function rulesButtonHoverState() {
    rulesButton.setStyle({ fill: '#f0b207' });
}

// For when no mouser hovering or clicking on 'Rules' button
function rulesButtonRestState() {
    rulesButton.setStyle({ fill: '#000' });
}

// Displays rules to user
function displayRules() {
    window.open('https://docs.google.com/document/d/169VO83FgXEXiv1NImkiVdlav88jXB17FoYv6h9FYMQA/edit?usp=sharing', '_bla
nk');
}

// Draws out other player's cell tiles
function drawTiles(self, playerInfo) {
    graphics.fillStyle(playerInfo.color, 1.0);
    playerInfo.placedTileLocations.forEach(function(element, index) {
        graphics.strokeRect(element.x, element.y, hspace, hspace);
        graphics.fillRect(element.x, element.y, hspace, hspace);
    });
}

// Draws all tiles of the user
function drawTilesToPlace(self) {
    var color = 0x4287f5;
    var thickness = 1;
    var alpha = 1;
    graphics.lineStyle(thickness, color, alpha);
    player.tilesToPlaceLocations.forEach(function (element, index) {
        graphics.strokeRect(element.x, element.y, hspace, hspace);
    });
}

// Adds player to its own group
function addPlayer(self, playerInfo) {
    self.test = self.add.image();
}

// Adds another player to the current Phaser Group for group management
function addOtherPlayer(self, playerInfo) {
    const otherPlayer = self.add.image();
    otherPlayer.playerId = playerInfo.playerId;
    self.otherPlayers.add(otherPlayer);
}

// Places tiles currently in user's tile to place locations when the user clicks
// 'Place Tiles' button
function placeTiles(self) {
    // Convert all tiles to place and put in placedTilesLocations
    for (var i = 0; i < player.tilesToPlaceLocations.length; i++) {
        player.placedTileLocations.push(player.tilesToPlaceLocations[i]);
    }
    // Empty out tilesToPlaceLocations
    player.tilesToPlaceLocations = [];
    // Fill in all placed tiles
    // Placed currently filled tiles
    placeFilledTiles();
```

```
}

// Places user's tiles and emits new filled tiles to all other players as well
function placeFilledTiles(self) {
    var color = 0x4287f5;
    var alpha = 1.0;
    graphics.fillStyle(color, alpha);
    // First clear out all previous tiles to clear board of any previously removed
    // tiles that were once placed
    socket.emit('clearCells')
    for (var i = 0; i < player.placedTileLocations.length; i++) {
        element = player.placedTileLocations[i];
        graphics.fillRect(element.x, element.y, hspace, hspace);

        // Emit tilePlaced call here
        socket.emit('tilePlaced', { x: element.x, y: element.y })
    }
    updateAliveCellsText();
}
```

index.html:

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
</head>

<body>
    <script src="/socket.io/socket.io.js"></script>
    <!-- <script src="//cdn.jsdelivr.net/npm/phaser@3.0.0/dist/phaser.min.js"></script> -->
    <script src="https://cdnjs.cloudflare.com/ajax/libs/phaser/3.52.0/phaser.min.js"></script>
    <script src="js/game.js"></script>
</body>

</html>
```