UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Cellular Automata Wars

Propusă de

Ionuț Arhire

Sesiunea: *Iulie, 2018*

Coordonator științific:

Drd. Olariu Florin

Table of Contents

1. Introduction

1.1. Main Idea

This application is targeted towards users that want to learn more about cellular automata (CA) or users that already have an interest in this domain, offering them the opportunity to discover new types of cellular automata and new rules while also having fun and interacting with friends.

1.2. Motivation

My main inspiration for this project was my interest in cellular automata. I wanted to do something innovative with it, something that would appeal to anyone, from CA enthusiast to people without a technical knowledge background.

As such, a browser-based multiplayer game was the ideal solution. Furthermore, it was a new technological challenge for me as I never used web-sockets-based frameworks until now and never built a multiplayer game before, so I took it as my chance to try it, learn something new, and have fun with it.

1.3. Methodology

Server-Client architecture using web-sockets for 'alive' match events and REST API for request-response type of tasks (e.g.: match creation).

1.4. Summary of Solution

The matches can be created by a single user. This user can share the url to the match with anyone. After enough users use the link to enter the match, the first game of the match can start. The idea of each game is to win by having the most 'alive' cells on the board by the end of the last generation (more details on this terminology in "CA theory in CA Wars" section, page 6).

2. Contributions

Most CA simulators have an ‘old’ look to them accompanied by an even ‘older’ looking page design, and none of them can be experienced between multiple people. My app does exactly the opposite while offering a fun experience to the user.

My contributions are:

- reframing of an ‘old’ concept, that of cellular automata.
- pleasant and modern front-end design.
- modern user interface mechanics (transparent menus, background animation, overlay modals).
- an architecture that can be easily expanded upon, both on front-end and on back-end.
- designing the app with performance in mind (e.g.: the matches are stored in RAM memory for faster memory access and are cleared when all players disconnect).
- intuitive and easily extendable persistence mechanism (using relational database instead of a NoSql one for more flexibility in the future).
- intuitive and ‘to-the-point’ game design.
- supporting a community of enthusiasts.

I decided that this documentation would be best separated in 5 chapters:

- App Flow - this chapter summarises the functionalities found in the menus and general mechanic behind starting to play.
- CA theory in CA Wars - this chapter briefly explains the theory behind cellular automata, the rules used in CA Wars and most of all, the particularities of the app in the context of CA theory.
- Game Flow - this chapter thoroughly explains the flow of a single game using relevant examples. Reading this chapter will probably make the best job in painting the big picture of the app.

- User Interface - this chapter thoroughly explains the bonus functionalities the user can access during playing that make the game more customizable, easy to play, and fun.
- Architecture - this chapter summarises the front-end and back-end architectures, also explaining why certain strategies were chosen instead of others.
- Code Examples – this chapter is used to explain critical methods/functions from back-end and front-end.

3. App Flow

The app will welcome the user through a main menu after which the user can opt for:

- creating a new multiplayer match.
- learning how the game works and what each option does.
- finding about the author through an ‘About’ section.

When creating a new multiplayer match, the user will be able to customize the following settings:

- Number of players.
- CA rule set which will determine the evolution of each game of the match.
- Maximum number of generations resulted from the CA evolution.
- Number of rows of the play field.
- Number of columns of the play field.

After match creation, the user will receive an url from the server, which he will use to share the match with his/her friends. Then, the match will begin. A match is a succession of games, each game ending in a win or a draw between more players.

4. CA theory in CA Wars

In this chapter, we will look into the basic theory of CA and how it relates to the flow of a game.

4.1. Basic CA

A cellular automaton consists of a regular grid of cells, each cell being in one of a finite number of states, such as ‘on’ and ‘off’ (‘alive’ and ‘‘dead’’, respectively). The grid can be in any finite number of dimensions. For each cell, a set of cells called its neighbourhood is defined relative to the specified cell. An initial state (time t = 0) is selected by assigning a state for each cell of the grid.

A new generation is created (advancing t by 1), according to some fixed rule that determines the new state of each cell in terms of the current state of the cell and the states of the cells in its neighbourhood. Typically, the rule for updating the state of cells is the same for each cell and does not change over time, and is applied to the whole grid simultaneously.

4.2. CA in the game

For CA Wars, only 2-dimensional CA are used, so the generations will evolve on a 2-dimensional grid.

Cellular automata are often simulated on a finite grid rather than an infinite one. In two dimensions, the universe would be a rectangle instead of an infinite plane. The obvious problem with finite grids is how to handle the cells on the edges.

CA Wars uses finite grids. Finite grids were chosen to make the games more competitive by shifting the focus from ‘expanding’ to ‘attacking’ and ‘defending’. Another great reason is performance (easier to compute on a finite grid).

In CA Wars, ‘edge’ cells are handled with a ‘toroidal’ arrangement: when one goes off the top, one comes in at the corresponding position on the bottom, and when one goes off the left, one comes in on the right.

This can be visualized as taping the left and right edges of the rectangle to form a tube, then taping the top and bottom edges of the tube to form a torus (doughnut shape).

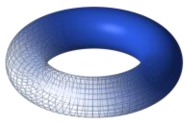

Fig. 1- A torus

The reason for using this 'toroidal' arrangement is ease of programming (using modular arithmetic functions).

The rules used in CA Wars are called 'Life-like' rules.

To understand what 'Life-like' rules refer to, we will have to analyse the most renowned CA rule:

4.3. Game of Life

The universe of the Game of Life is a 2-dimensional grid of square cells, each of which is in one of two possible states, 'alive' or ''dead''. Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. This is called the 'Moore neighbourhood', which you can observe in Fig. 2.

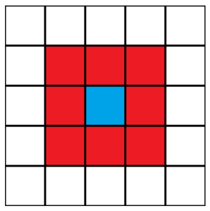

Fig. 2- Moore neighbourhood

At each step in time, the following transitions occur:

1. Any ‘alive’ cell with fewer than two ‘alive’ neighbours dies, as if caused by under population.
2. Any ‘alive’ cell with two or three ‘alive’ neighbours lives on to the next generation.
3. Any ‘alive’ cell with more than three ‘alive’ neighbours dies, as if by overpopulation.
4. Any ‘dead’ cell with exactly three ‘alive’ neighbours becomes a ‘alive’ cell, as if by reproduction.

The first generation is created by applying the above rules simultaneously to every cell in the seed; births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a tick. Each generation is a pure function of the preceding one. The rules continue to be applied repeatedly to create further generations.

4.4. Life-like CA

A CA is ‘Life-like’ (in the sense of being similar to Conway’s Game of Life) if it meets the following criteria:

- The array of cells of the automaton has two dimensions.
- Each cell of the automaton has two states, ‘alive’ and ‘‘dead’’.
- The neighbourhood of each cell is the Moore neighbourhood.
- In each time step of the automaton, the new state of a cell can be expressed as a function of the number of adjacent cells that are in the ‘alive’ state and of the cell’s own state; that is, the rule is outer ‘totalistic’ (sometimes called ‘semitotalistic’).

This class of cellular automata is named after the Game of Life, the most famous rule, which meets all of these criteria. Many different terms are used to describe this class. It is common to refer to it as the “Life family” or to simply use phrases like “similar to Life”, but we will refer to these kind of rules as “life-likes” for simplicity.

There are three standard notations for describing life-likes. In the notation we will be using, Game of Life can be expressed as follows: ‘S23/B3’. ‘S’ stands for ‘survival’ and ‘B’ stands for ‘birth’. ‘S23/B3’ would therefore be read as: “The cell will survive if it has exactly 2 or 3 ‘alive’ neighbours / the cell will become ‘alive’ if it has exactly 3 ‘alive’ neighbours”.

There are $2^{18} = 262{,}144$ possible life-likes, only a small fraction of which have been studied in detail.

4.5. Mirek's Cellebration (MCell)

Mirek's Cellebration (MCell) is a 32-bit Windows program whose main purpose is exploring existing and creating new rules and patterns of 1-D and 2-D Cellular Automata.

CA Wars uses all the MCell built-in Life-like rules:

Name	Rule(S/B)	Character
2x2	125/36	Chaotic
34 Life	34/34	Exploding
Amoeba	1358/357	Chaotic
Assimilation	4567/345	Stable
Coagulations	235678/378	Exploding
Conway's Life	23/3	Chaotic
Coral	45678/3	Exploding
Day & Night	34678/3678	Stable
Diamoeba	5678/35678	Chaotic
Flakes	012345678/3	Expanding
Gnarl	1/1	Exploding
HighLife	23/36	Chaotic
Long life	5/345	Stable
Maze	12345/3	Exploding
Mazectric	1234/3	Exploding
Move	245/368	Stable
Pseudo life	238/357	Chaotic
Replicator	1357/1357	Exploding
Seeds (2)	/2	Exploding

Serviettes	/234	Exploding
Stains	235678/3678	Stable
WalledCities	2345/45678	Stable

5. Game Flow

Step 1: Map Generation

A simple 2-dimensional orthogonal grid of minimum size 4x4 and maximum size 50x50, that is partitioned such as each player would get the same amount of 'territory'.

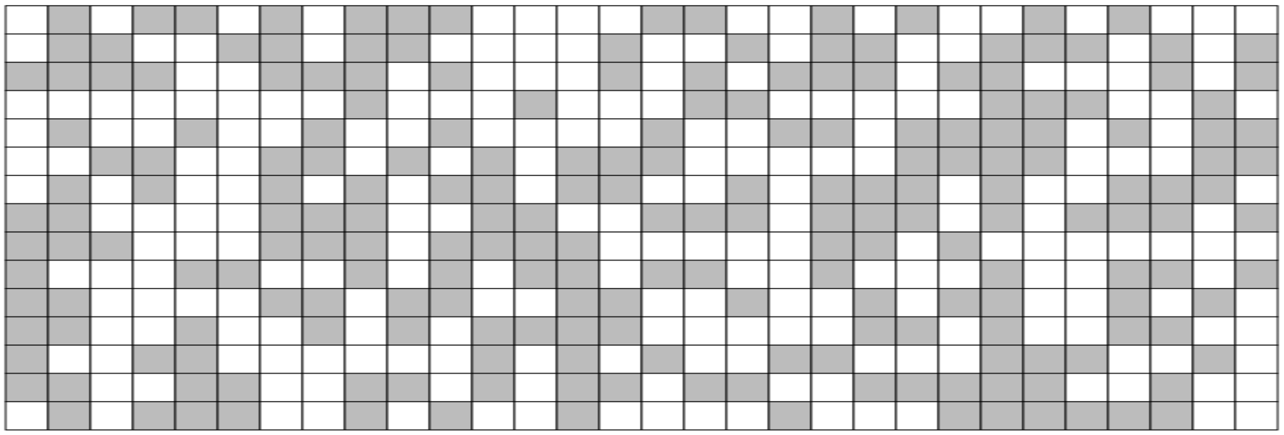

Fig. 3 - Example of how the match map could look for a player

The strategy for map generation is randomly selecting, for each cell, which player can 'own' it. Given the uniform distribution of number of 'viable' cells per player, we can be sure of the fact that each player will get approximately the same amount of 'territory'.

Another advantage of using this method is the fact that there are very low chances for a player to get a large clump of 'territory', meaning that the player is constrained to using patterns that don't require a lot of cells, so he can more easily predict a game and employ a certain strategy.

Step 2: Initial Configurations

Each player will activate cells as to create the most favourable initial configuration for himself/herself.

Step 3: Automata Evolution

We will now analyse a simplified case of automata evolution that can happen during a game:

We will start from the following initial configuration:

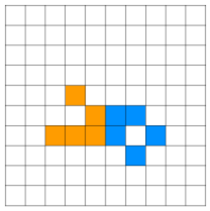

Fig. 4 - Initial configuration

Here, player I has the orange cells and player II the blue ones.

The CA we will be using is Game of Life (GOF).

GOF consists of the following transitions:

1. Any 'alive' cell with fewer than two 'alive' neighbours dies, as if caused by under population.
2. Any 'alive' cell with two or three 'alive' neighbours lives on to the next generation.
3. Any 'alive' cell with more than three 'alive' neighbours dies, as if by overpopulation.
4. Any 'dead' cell with exactly three 'alive' neighbours becomes 'alive', as if by reproduction.

We can see in the picture that player '1' has a "Glider", a pattern of GOF under the category: "spaceship" (for the way it "moves"). Player '2' has a "Boat", also called a "still life".

The initial configuration evolves as follows:

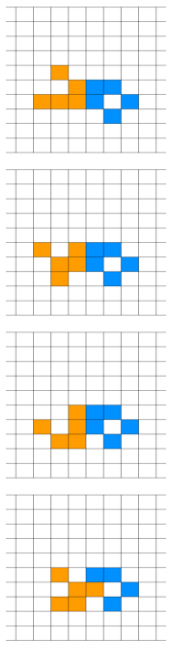

Fig. 5 - First evolution stages

The last panel of Fig. 5 shows how a cell can overwrite an opponent cell.

At each iteration of the evolution, the transitions for a player are computed as if the opponent's cells are all 'dead'. The only problem here would be the case when a cell wants to be activated with orange and blue at the same time. In this case, the cell will remain 'dead' for both players.

Getting back to the evolution, the next state is:

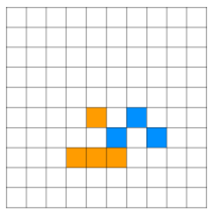

Fig. 6 - Fifth generation

To analyse how we arrived at this state we will use this:

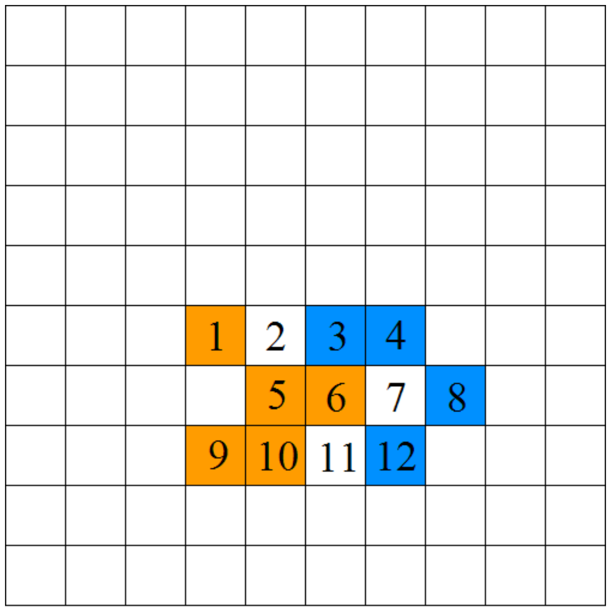

Fig. 7 - Evolution analysis

Cell 1: died because of transition 1

Cell 2: was born because of transition 4; neighbours: {1,5,6}

Cell 3: died because of transition 1

Cell 4: lived because of transition 2

Cell 5: died because of transition 3

Cell 6: was reborn as a blue cell because of transition 4; neighbours: {3,4,12}

Cell 7: stayed ‘dead’

Cell 8: lived because of transition 2

Cell 9: lived because of transition 2

Cell 10: lived because of transition 2

Cell 11: was born because of transition 4; neighbours: {5,6,10}

Cell 12: died because of transition 1

The next state is:

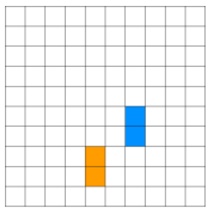

Fig. 8 - Last generation

After this state, simultaneous ‘death’ follows for both players’ cells. In this case, the game concluded with a draw, so neither of the players will get a point.

6. User Interface

In this section, important buttons and UI functionalities that are not entirely self-explanatory will be explained. All of the elements that are about to be discussed are also shown and explained in the 'Help' section of the main menu of the app.

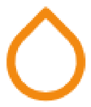

Fig. 9 - The Droplet

The Droplet: When activated, you can erase your 'alive' cells by left clicking or left click dragging over them.

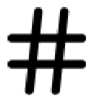

Fig. 10 - The Hash

The Hash: When activated, the black border of the grid will become white, thus giving the game view a different, cleaner look.

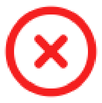

Fig. 11 - The Exit

The Exit: When clicked, it opens a confirmation dialogue before letting the user exit.

The following buttons appear only after the game has finished and right before resetting the game.

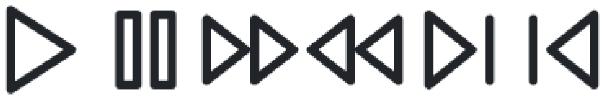

Fig. 12 - The Media Buttons

The Media Buttons: They work exactly like video player buttons but instead of video frames they work with CA evolution generations.

Fig. 13 - The Save

The Save: When clicked, it automatically downloads each CA generation as png in the default download folder of your browser.

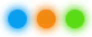

Fig. 14 - The 'Who-Sent' Dots

The 'Who-Sent' Dots: Each dot means that the player of that color already sent its initial configuration. The order of the dots, from left to right corresponds to the real order in which the players sent their respective initial configurations. So in the example above, the 'blue' player sent first, followed by the 'orange' and then the 'green' player.

The game view is also dynamic in the sense that you can move the camera around by dragging with right click over the canvas surface or zooming in and out using the mouse scroll.

7. Architecture

In this chapter, we will discuss the general architecture behind the back-end and the front-end.

7.1. Server-side Architecture

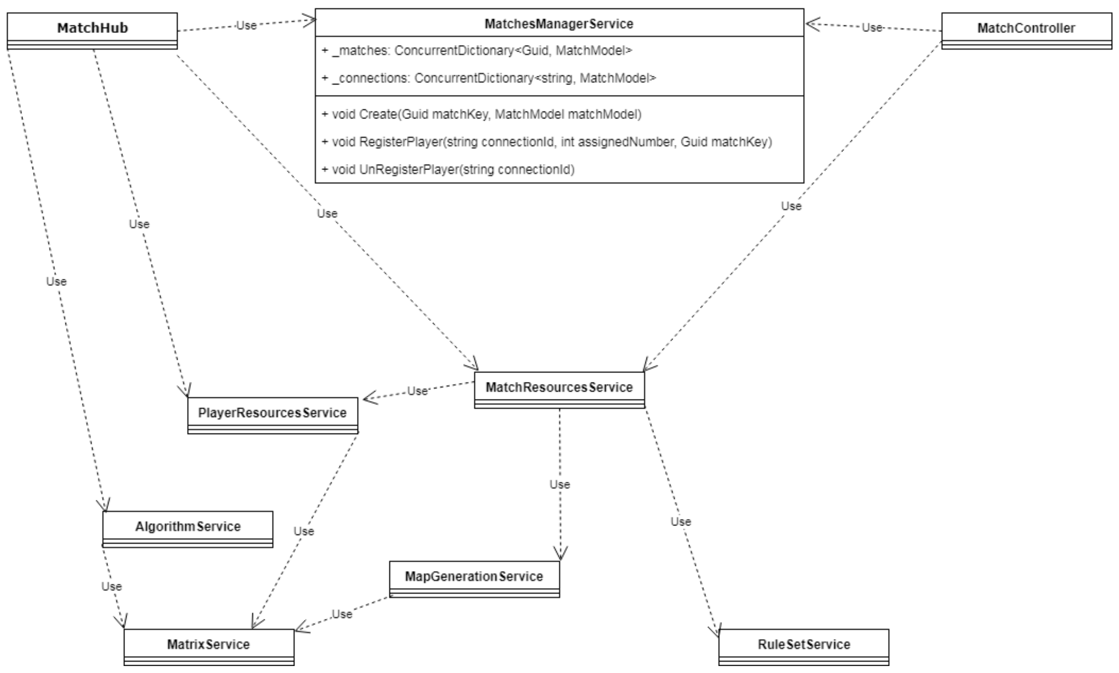
MatchHub
Use
MatchesManagerService
+ _matches: ConcurrentDictionary<Guid, MatchModel>
+ _connections: ConcurrentDictionary<string, MatchModel>
+ void Create(Guid matchKey, MatchModel matchModel)
+ void RegisterPlayer(string connectionId, int assignedNumber, Guid matchKey)
+ void UnRegisterPlayer(string connectionId)
Use
MatchController
Use
Use
Use
Use
MatchResourcesService
Use
PlayerResourcesService
AlgorithmService
Use
Use
Use
MapGenerationService
Use
Use
MatrixService
RuleSetService

Fig. 15 - Server architecture

This is the dependency graph of the server architecture with details about the MatchesManagerService class that acts like a container class for the state of the server.

Going from top to bottom, we have the two classes that control the traffic of the server: the MatchHub class that concerns itself with real-time communication and the MatchController class that concerns itself with request-response type of communication.

The MatchController class was mainly implemented so that the MatchHub wouldn't get overworked, trying to also handle request-response type tasks. Doing it this way also offers the advantage of having explicit separation of types of communication for easier extendibility and maintainability.

The MatchesManagerService class has two concurrent dictionaries that hold the state of the server, namely:

- _matches dictionary that stores the correspondence between a match's match key and the match object from RAM memory.
- _connections, that stores the correspondence between a connection id of a user and a match object from RAM memory.

The _connections dictionary's purpose is to more easily get the match a player is in case the player disconnects from the match.

The MatchesManagerService class also has three very important methods:

- Create: Stores a match in _matches dictionary.
- RegisterPlayer: Stores the connection in _connections dictionary and adds the player model to the match model.
- UnRegisterPlayer: Is called when a player disconnects from the match. The method makes sure the information about the player is erased so that the match can continue normally.

In the lower half of the diagram we can see all the other services used on the back-end.

- MatchResourcesService: Creates match models for storing them in the MatchesManagerService.
- PlayerResourcesService: Creates player models that will be stored in a match model. This service is also responsible with producing personalized maps for each player (for more details, see the "23Code Examples" section, page 23).
- AlgorithmService: Runs the CA evolution based on the rule set found in the match model and returns all a list of generations and the winner (or let's you know a draw happened if that is the case).

- MapGenerationService: Returns the match map. The match map is stored in memory as a grid with each element being the assigned number of a player. So if the first cell of the grid has the value ‘1’ that means that only player ‘1’ has control over that cell.
- MatrixService: A wrapper for basic matrix operations that should be globally available.
- RuleSetService: A wrapper for the repository that is used to parse the rules stored in the database (for more details, see the “Code Examples” section, page 23).

7.2. Client-side Architecture

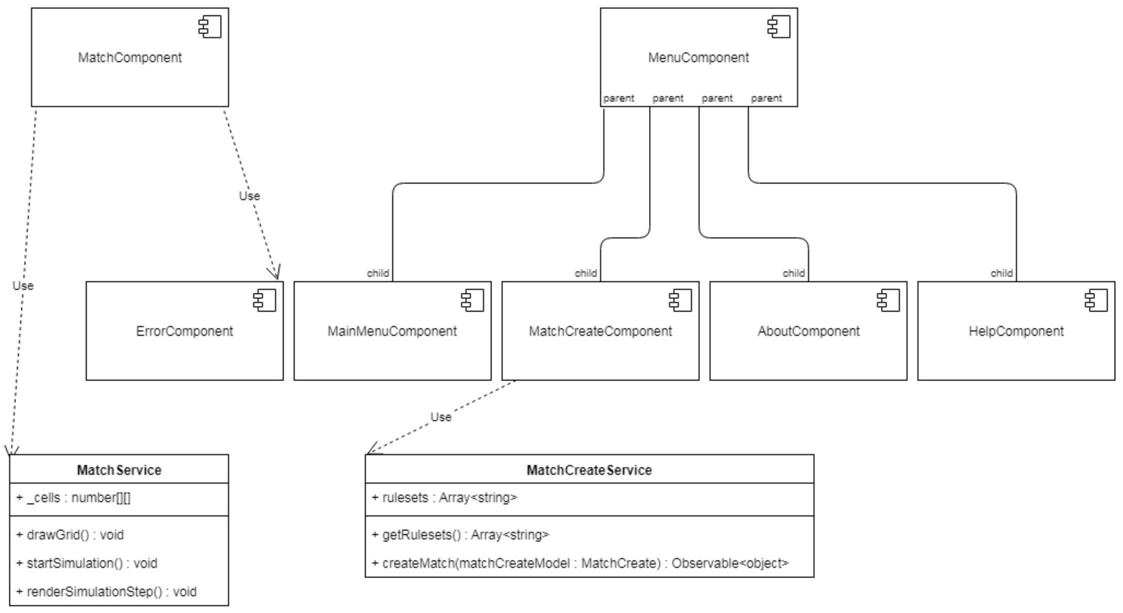

Fig. 16 - Client architecture

This is the dependency graph of the client architecture with details about the service classes.

In the upper half of the diagram we have the components and in the lower half we got the services.

7.2.1. Parent Components

- MatchComponent: this component establishes the connection with the MatchHub on the back-end and manages the communication between the hub and the component. This component delegates its computing tasks to the MatchService.

- MenuComponent: this component establishes a template for all the menus of the app and is the parent for all its child components, meaning that a child component can't be rendered without first rendering the MenuComponent. For a menu like the 'help' menu the url will be 'menu/help' and for 'about' menu the url will be 'menu/about'. In other words, menu is a parent to those components also from a routing perspective.

7.2.2. Other Components

- ErrorComponent: this component is instantiated by the MatchComponent whenever there was an error in connecting to the match.

Possible errors are:

- The match key isn't a valid Guid.
- The match is full (the match already started).
- The match is non-existent.

The remaining components are self-explanatory, in that they correspond to their respective views. For example, the MatchCreateComponent corresponds to the match creation menu.

7.2.3. Services

- MatchService: as stated above, MatchService is used by the MatchComponent to delegate its computing tasks. The service also maintains the state of the grid in the _cells 2d array.

  MatchService functions:

- drawGrid: is used to render the grid. The color of the grid border can be modified through the activation of the "Hash" button (for more details, see the "User Interface" section, page 16).

- startSimulation: is used to set a recurring task (recurs at every 60 ms) that renders a new generation of the finished game at every firing.
- renderSimulationStep: is used to set the _cells to the current generation and then call drawGrid to render everything on the game view.

- MatchCreateService: this service is where the MatchCreateComponent gets its rulesets from. MatchCreateService also has the responsability of sending the match create model to the back-end for actual creation.

MatchCreateService functions:

- getRuleSets: this function returns the _rulesets array.
- createMatch: this function makes a http post request with the match create model to the corresponding endpoint in MatchController of the server. It also returns an observable that can be used to easily track and manage the response of the server.

8. Code Examples

In this section, we will take a look and analyse some critical methods/functions both on the back-end and on front-end.

8.1. Back-End Samples

```
public async Task SendConfig(string unparsedMatchKey, float[,] playerConfig, int
assignedNumber) {
    var matchKey = Guid.Parse(unparsedMatchKey);

    var match = this._matchesManagerService.GetMatchModel(matchKey);
    var playerId = this._matchesManagerService.FindPlayerId(Context.ConnectionId,
match);

    match.InitialConfigs.Add(new InitialConfigModel(playerConfig, playerId));

    await Clients.Group(matchKey.ToString()).SendAsync("PlayerSent", assignedNumber);

    if (match.InitialConfigs.Count == match.Players.Count) {
        var result = this._algorithmService.RunGame(match);

        await Clients.Group(matchKey.ToString()).SendAsync("Game", result);
        match.InitialConfigs.Clear();
    }
}
```

Fig. 17 - MatchHub SendConfig method

This method is invoked remotely by the client-side, which also is responsible with providing all the parameters. The client-app sends the initial configuration through this method.

The Parameters:

1. unparsedMatchKey: guid converted to string that represents the match key.
2. playerConfig: a 2d uniform array of float elements that represents the initial configuration of the player.

The 2d uniform float array has the following cell values:

- the assigned number of the player, which means that the cell is ‘alive’.
- ‘-1’, which means that the cell is ‘‘dead’’.
- ‘-2’, which means that the cell is part of enemy ‘territory’.

3. assignedNumber: the number assigned to the player. If the player has the assigned number ‘2’ then we call him player ‘2’.

What it does:

The method registers the initial configuration that the player sent with the corresponding match (extracted using the match key) and notifies all the players in that match which player sent his/her initial configuration.

If the number of initial configurations is equal to the number of all the players currently in the match, so, if all players connected to the match sent their initial configurations, then the algorithm service is used to compute all the generations of the game, until the number o generations reaches the maximum iterations number (or until all cells are ‘dead’). At the end, Each player receives the results and the initial configurations are erased from memory.

```
public float[,] GetPersonalizedMap(float[,] map, int assignedNumber) {
    var pmap = this._matrixService.CopyMatrix(map);

    for (int i = 0; i < pmap.GetLength(0); i++)
    {
        for (int j = 0; j < pmap.GetLength(1); j++)
        {
            if(pmap[i,j] == assignedNumber) {
                pmap[i,j] = -1;
            }
            else {
                pmap[i,j] = -2;
            }
        }
    }

    return pmap;
}
```

Fig. 18 - PlayerResourcesService GetPersonalizedMap method

This method is invoked when preparing player resources to be part of the match model dto.

The Parameters:

1. map: a 2d uniform array of float elements that represents the map for the whole match (for more details about the match map, see the "Server-side Architecture" section, page 18, MapGenerationService subsection).
2. assignedNumber: the number assigned to the player.

What it does:

Assigns '-1' to cells the player is allowed to control and '-2' to cells that make up the enemy 'territory'. The result is going to be sent to the client-app which will correctly display enemy 'territory' as greyed out squares that the player can't control.

```
public LifeLikeModel GetByName(string name) {
    var repoResult = this._lifeLikeRepo.GetByName(name);

    var forSurvival = this.ParseNeighbours(repoResult.ForSurvival);
    var forBirth = this.ParseNeighbours(repoResult.ForBirth);

    var newLifeLikeModel = new LifeLikeModel(repoResult.Name, forSurvival,
forBirth, repoResult.Character);

    return newLifeLikeModel;
}
```

Fig. 19 - RuleSetService GetByName method

This method is used to get the life like rule model from the database during match creation.

The Parameters:

1. name: the name of the ruleset.

What it does:

It uses the repository for life like rules to get the life like entity out of the database. Next, the ‘ForSurvival’ and ‘ForBirth’ strings are parsed so they can be converted into List for convenience and clarity.

For example, let’s take the “2x2” life like rule. Its S/B is “125/36” which means the ‘ForSurvival’ string will be stored as “125” and ‘ForBirth’ will be “36”. These strings will be converted as stated above, using the following method.

```
private List<byte> ParseNeighbours(string neighbours) {
    var result = new List<byte>();

    foreach (char c in neighbours) {
        result.Add((byte)Char.GetNumericValue(c));
    }

    return result;
}
```

Fig. 20 - RuleSetService ParseNeighbours method

This method is used to parse the ‘ForSurvival’ and ‘ForBirth’ strings for any life like rule.

The Parameters:

1. neighbours: the string to be parsed.

What is does:

Creates a new list of bytes, takes each character of the string and converts it to byte type and then appends it to the list. The method returns the resulting list that is further used by the algorithm service in computing the next generation.

```
public async Task SendResources(string unparsedMatchKey) {
    var matchKey = Guid.Parse(unparsedMatchKey);
    if (this._matchesManagerService.GameModelExists(matchKey)) {
        var match = this._matchesManagerService.GetMatchModel(matchKey);

        if (match.PlayerNumbers.Count == 0) {
            throw new HubException($"The match with the key
\"{unparsedMatchKey}\" is already full!");
        }

        var matchModelDto = this._mapper.Map<MatchModelDto>(match);

        matchModelDto.AssignedNumber =
this._playerResourcesService.AssignNumber(match);
        matchModelDto.Map =
this._playerResourcesService.GetPersonalizedMap(match.Map,
matchModelDto.AssignedNumber);

        await Clients.Caller.SendAsync("Resources", matchModelDto);
        await Groups.AddToGroupAsync(Context.ConnectionId,
matchKey.ToString());
        this._matchesManagerService.RegisterPlayer(Context.ConnectionId,
matchModelDto.AssignedNumber, matchKey);
    }
    else {
        throw new HubException($"The match with the key \"{unparsedMatchKey}\"
is non-existent!");
    }
}
```
Fig. 21 - MatchHub SendResources method

This method is invoked remotely by the client-side, which also is responsible with providing all the parameters. The client-app uses this method to connect to a match and receive the required resources to start the game view.

The Parameters:

1. unparsedMatchKey: guid converted to string that represents the match key.

What it does:

Checks if the match with the corresponding match key exists. If it doesn’t exist, the method throws an error that will be further used on the client-side to display the corresponding message to the user.

If the match has all the player numbers assigned then the match is full and so the method throws an error and the same situation as above repeats itself.

If the match is existent and is not full, the match model is mapped unto a match model dto that carries only the info that is of interest to the client-app, a player number is assigned randomly from those still available, a personalized map is created and the method finally sends the resources to the client-app, adds the connection id of the user (Context.ConnectionId) to the singalR group assigned for the match and registers the player with the match.

```
public int AssignNumber(MatchModel match) {
    var playerNumbers = match.PlayerNumbers;

    var randomizer = new Random();
    var idx = randomizer.Next(playerNumbers.Count);
    var result = playerNumbers[idx];
    playerNumbers.RemoveAt(idx);

    if (this._env.IsDevelopment())
    {
        if (playerNumbers.Count == 0)
        {
            match.PlayerNumbers = InitPlayerNumbers(match.Players.Count);
        }
    }


    return result;
}
```

Fig. 22 - PlayerResourcesService AssignedNumber method

This method is used to assign a number to a player so that the player cells can be easily identified in the grid.

The Parameters:

1. MatchModel: the model for the match the player takes part in.

What it does:

Gets all the still available player numbers of the match and randomly selects one for the user and then removes the extracted number from the player numbers list. The selected number is returned.

If we run the app in development mode and all the player numbers have been assigned for the match received as parameter, then we reinitialize the player numbers so that new players can join the match even if all the previous players disconnected from it. This is because it’s easier to debug when a match can’t die off.

```
public void RegisterPlayer(string connectionId, int assignedNumber, Guid
matchKey) {
    var match = this._matches[matchKey];
    this._connections.TryAdd(connectionId, match);

    match.Players[assignedNumber].ConnectionId = connectionId;
}
```

Fig. 23 - MatchesManagerService RegisterPlayer method

This method is used to register a player when he request an existent match that didn’t already start.

The Parameters:

1. connectionId: the connection id of the user (accesible through Context.ConnectionId).
2. assignedNumber: the number assigned to the player.
3. matchKey: the key of the match (the unique identifier for the match).

What is does:

Registers the player with the match and keeps track of its connection to the match so it's easier to unregister him/her, as you will see in the following method.

```
public void UnRegisterPlayer(string connectionId) {
    var match = this._connections[connectionId];

    var playerId = this.FindPlayerId(connectionId, match);

    this.RemoveInitialConfig(match, playerId);
    match.Players.RemoveAt(playerId);
    this.RemoveConnection(connectionId);

    if (match.Players.Count == 0) {
        this.RemoveMatch(match);
    }
}
```

Fig. 24 - MatchesManagerService UnRegisterPlayer method

This method is used to unregister a player when he diconnects from the match.

This method doesn't get fired when running the server in development mode so that the match doesn't get removed when all players disconnect. This is because it's easier to debug when the match stays indefinitely available.

The Parameters:

1. connectionId: the connection id of the user (accesible through Context.ConnectionId).

What it does:

It selects the match that corresponds to the player's connection id. If the player already sent his/her initial configuration, that initial configuration gets removed. It removes the player from the match and it removes the link between the player connection id and the match (this.RemoveConnection(connectionId)). If all the players disconnected, the method removes the match from memory so that performance won't be affected by inactive matches that stay stagnant in memory.

8.2. Front-End Samples

```
public ngAfterViewInit(): void {
  this.getResourcesStatus().subscribe((_gotResources) => {
    if (_gotResources) {
      this._matchService.init(this.playGrid.nativeElement,
this.toolbar.nativeElement, this._dimensions, this._playerResources,
this._map);
      this._matchService.drawGrid();
      this._matchService.captureEvents(this._assignedNum);
    }
  });
}
```

Fig. 25 - MatchComponent ngAfterViewInit function

This function fires when the view gets rendered and is used to initialize match service and the game view.

What it does:

The initialization code mentioned above runs only if the client already received the necessary resources. getResourcesStatus() returns an observable that the function subscribes to using a lambda function that has the boolean parameter '_gotResources'. If '_gotResources' is set to true, match service gets initialized, the grid gets rendered and the event handlers for the game view (used for camera movement on right click drag and scroll) are registered.

```
public startSimulation(): void {
  this._simulationInterval = setInterval(() => {
  if (this._currGameStateIdx < this._generations.length - 1) {
    this._currGameStateIdx += 1;
      this.renderSimulationStep();
    }
  }, 60);
}
```
Fig. 26 - MatchService startSimulation function

This function is called when the user clicks on the ‘play’ button, after the generations for that respective game arrived from the server.

What it does:

This function is used to set a recurring task (recurs at every 60 ms) that renders a new generation of the finished game at every firing.

setInterval is called, provided with a lambda function that concerns itself with rendering each generation in the appropiate order and a value that represents the time interval between two firings measured in milliseconds.

The resulting task is stored in a class variable ’_simulationInterval’ so that it can be deleted once media buttons like ‘pause’ or ‘rewind’ are hit, and by doing this, stopping the recurring task. This behaviour can be better understood by analyzing the following functions:

```
public stopSimulation(): void {
  clearInterval(this._simulationInterval);
}
```
Fig. 27 - MatchService stopSimulation function

```
public skipSimulationBack(): void {
  this.stopSimulation();
  this._currGameStateIdx = 0;
  this.renderSimulationStep();
}
```
Fig. 28 - MatchService skipSimulationBack function

The first function is used to clear the recurring task, while the second function is used when hitting the ‘rewind’ button.

After hitting the ‘rewind’ button, skipSimulationBack gets fired and the recurring task gets cleared, followed by setting the ’_currGameStateIdx’ (which represents the current generation index) to 0 (the absolute beginning) and thus rendering the first generation.

```
public saveGenerationsAsImgs() {
  for (let idx = 0; idx < this._generations.length; idx++) {
    this._cells = this._generations[idx];
    this.drawGrid();

    this._canvas.toBlob(function(blob) {
      saveAs(blob, idx + ".png");
    });
  }
}
```

Fig. 29 - MatchService saveGenerationsAsImgs function

This function is called when the user hits the ‘save’ button.

What it does:

It goes through each generation and at each step, it renders the grid and saves the canvas contents in .png format in the default download folder of the user’s browser.

9. Conclusion

The main goal of this thesis was to put cellular automata in a completely new, fresh, perspective that would appeal to anyone. However, I didn't want it to only seem like a fun idea, but also be a fun idea and experience, so I knew there was a long way to go in terms of front-end work, which was quite scary for me, given that I wasn't confident of my front-end engineering and design skills.

I also wanted to learn a lot and explore, so I chose to use the latest updates for each major technology in the project, even though I knew I wouldn't find the most extensive documentation given their recent launches.

At the beginning, it was pretty hard researching and getting used to signalR. I also had to open some issues on the aspnet/signalR github repository to better understand what this and the older version of singalR have in common and what is different.

I also had to get used to the front-end canvas and its mouse input event handling and had to do a lot of brainstorming to figure out the design on the go, because I wanted it to be my own invention, with as little outside help as possible.

However, I think the hardest thing was to make hard architectural choices. I made some architectural decisions that turned out to be not as succesful as I had hoped and then had to redo the whole thing. But all those bad decision were great lessons for me and I discovered how much more important was 'planning ahead' than I originally thought.

I also experimented with when to refactor my code and found out it's better to do it early most of the times.

Visual Studio Code offered me the flexibility I needed for both back-end and front-end and its lightness probably saved my calm a few times.

Ideas for the future

In the future, the game could accommodate multiple classes of CAs, not only life-like CAs. However, the greatest addition would be a singleplayer mode in which you can test out new ideas by playing with a single color or more and see how they interact.

Personal Opinion

The result of my work until now is an application meeting the basic objectives I was set out to achieve. I also learned a lot from working on this project, both technical knowledge and organizational wisdom.

When I say ‘organizational wisdom’ I’m referring to:

- reading as much as you can about the used frameworks before deciding on the architecture.
- refactoring early (even when you are the only person working on the project).
- realizing the importance of OOP principles.
- discussing your ideas with someone else.

I am also more confident now in my ability to produce good looking and dynamic UI.

All in all, it was a thrilling experience that only made me more passionate and less scared about real-life, big projects.

10. Technologies

- C#
- .NET Core
- SignalR
- Angular 6
- Automapper
- Material Design Bootstrap
- SQL Server

11. Bibliography

1. **Official documentation for SignalR,** https://docs.microsoft.com/en-us/aspnet/core/signalr/?view=aspnetcore-2.1
2. **McCell "Life-Like" CA rules,** http://psoup.math.wisc.edu/mcell/rullex_life.html
3. **Basic CA theory,** https://en.wikipedia.org/wiki/Cellular_automaton
4. **Game of Life,** https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
5. **"Life-Like" CA,** https://en.wikipedia.org/wiki/Life-like_cellular_automaton
6. **What is Mirek's Cellebration?,** http://www.mirekw.com/ca/whatis_mcell.html
7. **Official documentation for Automapper,** http://docs.automapper.org/en/stable/index.html
8. **CSS hints and tips,** https://css-tricks.com/
9. **Official documentation for MDBootstrap,** https://mdbootstrap.com/
10. **Angular-feather angular library for icons,** https://github.com/michaelbazos/angular-feather
11. **Official documentation for Angular,** https://angular.io/docs
12. **Reasons for choosing Angular,** https://medium.com/angular-japan-user-group/why-developers-and-companies-choose-angular-4c9ba6098e1c
13. **Stack Overflow Developer Survey 2018,** https://insights.stackoverflow.com/survey/2018/