Bulletin of the *Transilvania* University of Braşov • Vol 5(54) No. 2 - 2012
Series III: Mathematics, Informatics, Physics, 137-146

SURVIVOR GAME

Andrei IVAN[1] and Daniela MARINESCU[2]

Abstract

We present in this paper a one-player game, based on the cellular automata. This game uses the rules of the *Game of Life* but this set of rules is extended, depending on each state. This extension of the set of rules is made for a more realistic model. The story of the *Survivor Game* is based on the science fiction novel of H.G.Wells *The Time Machine*, 1895, and on the two movies with the same title, produced in 1960 and in 2002.

2000 *Mathematics Subject Classification:* 37B15, 68Q80, 97R80, 97M70.
Key words: cellular automata, computer game, recreational computing, behavioural and social science.

1 Introduction

Searching for a self-reproducing machine J. Von Neumann, with the help of Stan Ulam [5] defined, in the 40s and 50s, a discrete spatio-temporal simulation system named cellular automaton (CA). The systematic studies appear in the paper of Wolfram, Langton and others in the 80s, including classification, analysis of computation universality. In 2002 Wolfram [12] introduced Cellular Automata in many fields, mathematics, physics, biology, social sciences.

Nowadays, the Cellular Automata are used as modelling tools in the field of computer science [13] for VLSI design, image processing, data compression, encryption, pattern classification, parallel processing architecture and computer game. One of the most known games that used 2D cellular automata is the Game of Life, proposed by Conway in 1970[2, 11]. There are many other games which have been modelled through CA. For example: the firing squad[4], the firing mob[1], the queen bee[8] and also the game of iterated prisoners dilemma[6, 7].

We propose in the following, a new game, named Survivor-Game, based also on CA, which used an extension of the set of rules of the Game of Life.

[1]SC Tehmin, Braşov, Romania, e-mail: subzero.ivan@gmail.com
[2]Faculty of Mathematics and Informatics, *Transilvania* University of Braşov, Romania, e-mail: mdaniela@unitbv.ro

2 Cellular automata and Game of Life

The cellular automaton [9, 10, 3] is a discrete model studied in physics, compatibility theory, mathematics, biology and micro-structure modelling.

By the definition of J. Von Neumann [5], the Cellular Automaton (CA) is a finite two-dimensional array of cells. The cells are arranged in a square-grid [3]. For every cell we can consider the five-cell neighbourhood (self and four orthogonal neighbours) or the nine-cell neighbourhood (self and all the cells that have a common border with the initial cell). The five and nine neighbourhood, are termed as *Von Neumann* and *Moore neighbourhood*, respectively. Each individual cell is in a specific state which changes over time depending on the states of its five or nine-cell neighbourhood. The change is defined by a transition rule.

One example of this cellular automaton is an automaton where the cells have only two states, *on* and *off*, and the transition rule from the moment t to the moment $t+1$ should be that a cell is *on* only if exactly two cells from the neighbourhood are *on*, otherwise the cell is *off*.

In paper [8] A. Smith has shown that a nine-cell neighbourhood CA, with two states per cell and appropriate rules is capable of universal computation. This model of CA with a specified set of rules has been used by Conway to create the *Game of Life*. Studying the Von Neumann model of CA, which has 29 states, Conway founded a more simply model with 2 states and 4 rules. This game is important from a theoretical point of view because it has the power of a Turing machine. It is also important because of the surprising kind in which life can evolve.

Cellular automata are mostly simulated on a finite square-grid. Obviously, in two dimensions, the universe is a rectangle. A delicate problem about these grids is how we handle the cells that reside on the edges of the rectangle because the way we process them may have impact on the whole grid. This problem can be treated as follows:

1. We can leave the values of edge cells constant.
2. We can define different rules for edge cells, but this is pretty complex as it requires more rules to handle those exceptions.
3. Perhaps the best approach in case of edge cells is to handle the grid as a toroidal shape: when we need the cell above an edge cell situated on the first row, we return the corresponding cell from the last row and so forth.

The game of life is a zero-player game which means that the evolution depends on the initial configuration without any intervention of the player. So the player must introduce an initial configuration and after that he only observes the evolution of the game. From

[3]Other arrangements are also possible. For example, they can be arranged in a hex-grid.

the Game of Life point of view the universe is an orthogonal array of cells, with two states: alive or dead. Every cell interacts with the other eight cells from the nine-cell neighbourhood. The following rules are applied to each cell at every step:

1. Every cell which has less than two live neighbours becomes dead.
2. Every live cell with two or three live neighbours remains alive.
3. The live cell with more than three live neighbours becomes dead by overcrowding.
4. A dead cell with exact three live neighbours becomes alive.

The initial configuration is named *seed* of the system. A generation is computed by obtaining a new state for every cell, simultaneously, but this generation is memorized in a separate array, which becomes the new generation of the system. The period of time between the calculations of two generations is named a *tick*. In the Game of Life it is possible to have an initial configuration which remains unchanged, a pattern named *still lives*, which reappears after some generation, named *oscillators*, and patterns that translate themselves across the board, named *spaceships*.

3 Survivor Game

3.1 Concept

Being fans of computer strategy games, we wanted to create a game where the player finds himself in the situation of analyzing, calculating and thinking his moves so that the game's objectives will be completed. Having touched the cellular automata field, we were sure that these can be applied in modelling such a game. Unfortunately, the cellular automata reside on static rules and there is no need for the user's intervention to determine a new generation. Because we wanted a nice game, entertaining and exciting for the player, the characteristics of cellular automata didn't help very much. We have studied the graphical elements that can emerge from the Game of Life, but these are pretty hard to control. For example, a ship from the Game of Life could be easily controlled by the user with the arrow keys, but the charm of cellular automata would suffer and the player's experience wouldn't be so great. On the other hand, we can see that *CA* are not usually applied in games. Still we didn't give up; we knew that there must be some way around. After some studies and serious brainstorming, we concluded that we could create something playable by extending the Game of Life rules.

Finally, we elaborated a concept which, of course, was modified during the study of its practicality, a model that certainly is not perfect even in the current shape. Besides the set of rules in the Game of Life, the concept was enriched with many other rules depending on the states. The number of the states was increased to absorb the player in the game's world. Briefly, the player controls two types of entities: pawns and shelters. These must be controlled in such a way that the player destroys the enemy's traps and troops on the map. If the player runs out of pawns, the battle is lost. The player can choose to place one of the two entities in one of the free cells. When the player is certain of the repercussions of his move, he lets the game engine apply the rules for each cell (there is a king of system called **turn-based** that can be seen in many strategy games). In the concept there is a variable which represents the current time of the day which is incremented after each turn. The set of rules takes into account this variable. The moments of the day are: *sunrise*, *midday*, *sunset* and *midnight*.

For simplicity, we preferred to take into account the day-time compressing them into two: day and night. The possible states of a cell are:

1. **Pawn** (Player) - the entity that helps the player accomplish the game's objectives. Basically, a new generation will be calculated by applying the Game of Life rules.
2. **Shelter** - the helper entity which offers protection to all the pawns nearby. If too many pawns are nearby, the shelter will vanish.
3. **Enemy** - comes out at night surrounding the traps. Any unprotected pawn nearby will die.
4. **Trap** - the player must destroy these cells. A trap will be destroyed when a certain number of pawns surrounds it.
5. **Walkable** - a cell where the player can move.
6. **Unpassable** - an element used mainly for decorating the map, but which can increase the difficulty of the game.

Now, the rules will be presented. For the ease of reading, 4 variables will be defined: *tc*, *sc*, *ec* and *pc*; these mean the number of enemy traps, shelters, enemies or pawns nearby the current cell. The function **survivor** will be applied to a *state* type variable.

$$gl(s, on, off) = \left\{ \begin{array}{ll} on & \text{if } s = on \text{ and } (c = 2 \text{ or } c = 3) \\ on & \text{if } s = off \text{ and } c = 3 \\ off & \text{otherwise} \end{array} \right\}$$

where $c =$ the number of the cells around which are *on*

1. **Player**

$$survivor(Player) = \left\{ \begin{array}{ll} Walkable & \text{if } sc \leq 0 \text{ and } ec > 0 \\ Player & \text{if } sc \geq 0 \\ gl(Player, Player, Walkable) & \text{otherwise} \end{array} \right\}$$

2. **Shelter**

$$survivor(Shelter) = \left\{ \begin{array}{ll} Shelter & \text{if } pc \geq 2 \text{ and } pc \leq 3 \\ Walkable & \text{otherwise} \end{array} \right\}$$

3. **Enemy**

$$survivor(Enemy) = \left\{ \begin{array}{ll} Walkable & \text{if } sc > 0 \text{ or if it is day} \\ Enemy & \text{otherwise} \end{array} \right\}$$

4. **Trap**

$$survivor(Trap) = \left\{ \begin{array}{ll} Walkable & \text{if } pc \geq 3 \text{ and } pc \leq 4 \\ Trap & \text{otherwise} \end{array} \right\}$$

5. **Walkable**

$$survivor(Walkable) = \left\{ \begin{array}{ll} Enemy & \text{if it is night, } tc > 0 \text{ and } sc \leq 0 \\ gl(Player, Player, Walkable) & \text{otherwise} \end{array} \right\}$$

6. **Unpassable**

$$survivor(Unpassable) = \left\{ \begin{array}{ll} Unpassable & \text{always} \end{array} \right\}$$

Algorithm 1 Game of Life algorithm

```
function GameOfLife($cell$, $stateOn$, $stateOff$) ▷ Calculate state according to Game of Life
    $neighboursOn \leftarrow$ StateCount($cell, stateOn$)
    $state \leftarrow cell.state$
    if $state = stateOn$ AND ($neighboursOn = 2 \quad OR \quad neighboursOn = 3$) then
        return $stateOn$
    end if
    if $state = stateOff$ AND $neighboursOn = 3$ then
        return $stateOn$
    end if
    return $stateOff$
end function
```

Algorithm 2 Complete algorithm

```
function StateCount($cell$, $state$)
    return Number of states nearby $cell$ that are equal to $state$
end function
function CalculateState($cell$)                    ▷ Calculate the new state of a cell
    $state \leftarrow cell.state$
    $trapCount \leftarrow$ StateCount($cell, Trap$)
    $shelterCount \leftarrow$ StateCount($cell, Shelter$)
    $playerCount \leftarrow$ StateCount($cell, Player$)
    $enemyCount \leftarrow$ StateCount($cell, Enemy$)
    if $state = Walkable$ then
        if $IsNight \quad AND \quad trapCount > 0 \quad AND \quad shelterCount \leq 0$ then
     return $Enemy$
        end if
     return GameOfLife($cell, Player, Walkable$)
    end if
    if $state = Player$ then
        if $shelterCount \leq 0 \quad AND \quad enemyCount > 0$ then  return $Walkable$
        end if
        if $shelterCount > 0$ then  return $Player$
        end if
     return GameOfLife($cell, Player, Walkable$)
    end if
    if $state = Shelter$ then
        if $playerCount \geq 2 \quad AND \quad playerCount \leq 3$ then  return $Shelter$
        else return $Walkable$
        end if
    end if
    if $state = Enemy$ then
        if $IsDay$ then
     return $Walkable$
        end if
        if $shelterCount > 0$ then  return $Walkable$
        end if
     return $Enemy$
    end if
    if $state = Trap$ then
        if $playerCount \geq 3 \quad AND \quad playerCount \leq 4$ then  return $Walkable$
        end if
     return $trap$
    end if
     return $state$
end function
```

3.2 Presentation

Due to the multiple possibilities offered by Microsoft Technologies like *C#* and *WPF*, we decided that the application would fit very well a *Windows Phone 7* profile. The game has a nice UI which also tells the player the story behind the game and how to play it. The game has 5 levels which must be unlocked in order to play them. The first level is always unlocked, but the others must be unlocked by finishing the previous levels. For choosing a certain level, we have developed a pie-level chooser control which is manipulated with the touch screen (Fig. 1).

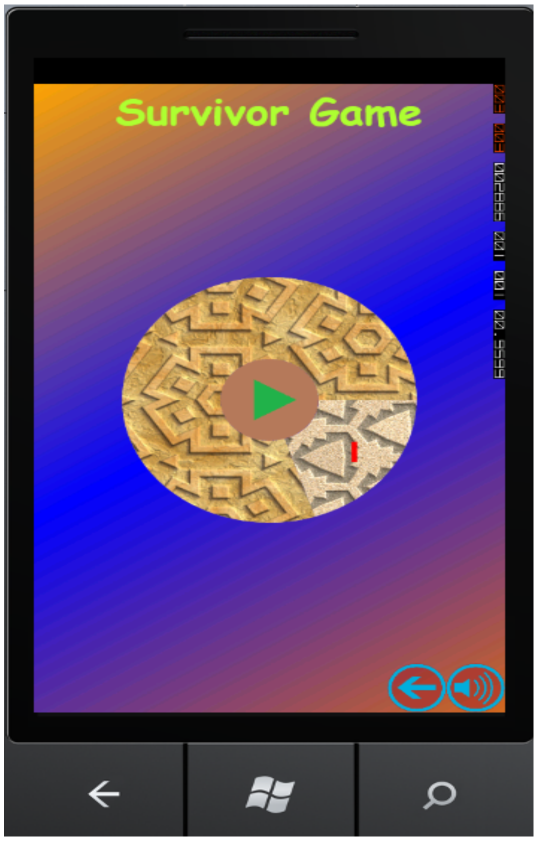

Figure 1: Pie level chooser control. The selected level is level 1.

In figure 2 we have a screenshot of the game play. The top bar shows some details about the current game: day-time, number of players, enemies and traps on the map. Finally, a progress-bar tells us the overall progress of the game. The bottom bar is reserved for the two possible moves the player can make. The rest of the user interface is occupied by the main grid. After choosing the type of move to make, the player is shown the free cells he can move in, then presses the *OK* button, if he is sure about the move. After that, the game engine will apply the rules to all the grid cells. At the same time, the engine will check if the win or lose conditions are satisfied and takes the corresponding action.

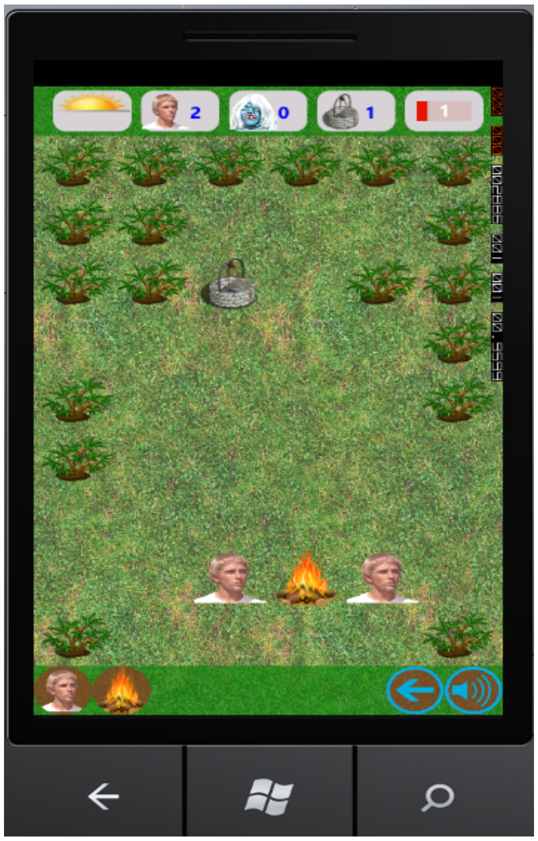
Figure 2: Gameplay screenshot. We can see two Players sitting near their shelter and an enemy outpost which must be destroyed.

3.3 Implementation

The game is developed in Silverlight for WP7 using Visual Studio 2010. The code respects the **MVVM** [4] pattern, model that consists in the separation of business logic from the user interface. Mostly based on **MVC** [5], its scope is developing modern rich-UI applications in an event-driven way (for example HTML5, WPF and Silverlight). It was also designed to make use of *data-binding* in WPF, letting the developers create bindings from the **markup** language to *view-model*. The separation of concepts lets designers and programmers focus on their tasks and easily collaborate with each other.

Very often seen in the game's implementation is the **provider pattern**. Its role is to supply objects of a certain kind to the calling layer. Usually, these objects have an associated key (usually String). The pairs ($name$: String, $object$: Object) are stored in a dictionary. This functionality is similar with the **multiton pattern** and can be

[4]Model View View-Model.
[5]Model View Controller.

completed with *lazy initialization* [6]. Because the dictionary can have only unique keys, their associated key must be therefore unique.

4 Conclusions

Starting from a well known *Game of Life*, we designed a single-player game, where the player interacts with the computer for survival in some *bad* conditions. The player controls two types of entities: pawns and shelters. These must be controlled in such way that the player destroys the enemy's traps and troops on the map. If the player runs out of pawns, the battle is lost. The player can choose to place one of the two entities in one of the free cells. When the player is certain of the repercussions of his move, he lets the game engine apply the rules for each cell.

Due to the fact that the game is based on cellular automaton, it is also useful in students' education for understanding and use of the rules of CA.

References

[1] Culik, K. and Dube, S., *An efficient solution of the firing Mob problem*, Theoretical Computer Science, **91**, 57-69, December 1991.

[2] Ganguly, N., Sikdar, B.K, Deutsch, A., Canright, G. and Chaudhuri, P.P., *A survey on cellular automata*, Tech. rep. (2003). http://www.cs.unibo.it/bison/publications/CAsurvey.pdf

[3] Garzon, M., *Models of massive parallelism.* Analyse of Cellular Automata and Neural Networks, Springer,1995.

[4] Moore, E., editor, *Sequential machines. Selected papers*, Addison-Wesley Publishing Company., Inc., Redwood City, CA., 1964.

[5] Nemann, J. von *The theory of self-reproducing automata*, A.W. Burks(ed), Univ. of Illinois Press, Urbana and London, 1966.

[6] Nowak, M.A. and May, R.M., *The spatial dilemmas of evolution*, International Journal of Bifurcation and Chaos, **3**, 35-787, 1993.

[7] Schweitzer, F., Behera, L. and Muehlenbein, H., *Evolution of cooperation in a spatial prisoner's dilemma*, Advances in complex systems, **5(2&3)** (2003), 269-301.

[8] Smith, A., *Introduction to and swurvey of polyautomata theory*, Automata, Languages, Development, North Holland, Publishing Co, 1976.

[9] `http://en.wikipedia.org/wiki/Cellular_automaton`.

[6] A pair is not added into a dictionary unless explicitly needed.

[10] http://en.wikipedia.org/wiki/Elementary_cellular_automaton.

[11] http://en.wikipedia.org/wiki/Conway's_Game_of_Life.

[12] Wolfram, S., *New Kind of Science*, A. Wolfram Media, Inc, 2002., http://www.wolframscience.com/

[13] Wolfram, S., *High speed computing: scientific application and algorithm design*, ed. Robert B. Wilhelmson, University of Illinois Press, 1988.