The Game of Life Game

EE 381V Final Project

Tarek Allam and Jackson Lightfoot

May 7, 2021

1 Motivation

John Conway's Game of Life is perhaps the most famous example of cellular automata and unconventional computing. However, this fame is typically restricted to those in fields such as math and computer science. Even some of our engineering friends say, "Like the board game?" when we mention Game of Life to them.

To increase awareness and educate the general public about the Game of Life, we decided to make a video game based off cellular automata. Traditionally, the Game of Life is a zero player game, meaning once an initial configuration is set, the game requires no further input or player to run. To those who do not understand the significance or underlying mechanics, a simple simulator may quickly bore them as they have no control once a simulation begins. In order to make the experience more engaging for a curious player, we decided to turn the Game of Life into an interactive single player game.

Some options we considered were generic 1D, 2D, and 3D cellular automata with a playable character attempting to navigate the evolving grid. The high level design we chose is a 3D platformer where a controllable character can walk forward, backward, left, and right, as well as jump. The ground underneath the player is a 2D cellular automata grid that continually evolves. The player can only stand on alive cells, so they must jump from block to block in order to avoid falling into lava. Ultimately, the goal of the player is to survive as long as possible, with an in game timer tracking progress. Developing an intuition for visually evaluating how the grid evolves is necessary for the player to survive longer, hence helping the player increase their understanding and appreciation of cellular automata.

2 Implementation Details

2.1 Unity

Unity is an open source, comprehensive game development platform with an emphasis on accessibility for students and indie developers, so as such we considered it an ideal vehicle for our project. It offers a suite of tools for building games, from rendering aids to a powerful scripting platform to audio and lighting systems. Projects are broken into scenes

which modularize various gameplay arenas including levels and menus, and each scene contains a hierarchy of objects which controls its functionality; broadly speaking, these objects fall into three main categories: controllers, prefabs, and assets. Controllers offer scripting functionality with C#, inheriting from Unity's `MonoBehaviour` class which connects it to the engine's editor, allowing developers to quickly connect scripts and objects together in logical patterns. Prefabs, short for prefabrications, are objects which have been constructed outside of the game to have certain properties visually or functionally, and are generally used to construct the scene's physical environment. Lastly, assets are simply items such as audio files, textures, and shaders which impact the user's sensory experience.

2.2 3D Character Controller

The 3D character controller is a C# script which allows the user object (a simple pill capsule) to collide, jump, and move. Jumping is bound to the space bar, and movement to the WASD keys. Collision and movement is handled by Unity's `CharacterController` component, which performs advanced checks to determine physics effects on its parent object. The controller also features a ground checking object, which determines if the player is standing on a surface and corrects their velocity if that is the case.

2.3 Evolving Grid

Once we achieved a working 3D character controller on a grid of cubes, the next step was to implement an evolving grid that simulates 2D cellular automata. Controlling the grid was done using a C# script and Unity's `Update()` function, which is called before every frame update. This function checks if a certain period of time has occurred since the last grid update, and if so calls the `UpdateGrid()` function. The `UpdateGrid()` function iterates through an array that holds the current state of each cell in the grid and counts the number of neighboring alive cells. It then uses a rule set to determine if a particular cell should be dead or alive the next iteration. If the state of the cell changed from the previous iteration, the grid cube in that location of the game will be spawned or despawned appropriately.

To allow for generic 2D cellular automata simulation, a generic rule set was implemented as a 2 by 9 boolean array, where the first index corresponds to whether or not the current cell is alive (0 for dead, 1 for alive), the second index corresponds to the number of alive neighbors (0 through 8), and the boolean value stored determines the next state of the cell (`false` for dead, `true` for alive). For example, the Game of Life rule set would be encoded with `true` at indices (1, 2), (1, 3), and (0, 3), and with `false` everywhere else. For boundary conditions, there are two different options implemented: wrap and no wrap. With wrap, a cell at a particular boundary has neighbors on the opposite side of the grid, while no wrap simply implies a lack of neighbors at the boundaries.

Since the goal of the game is to survive as long as possible, a key problem we realized was that some configurations will result in cells that never die. Thus, a player could simply stand on one of these cells forever. To account for this, we added the functionality of transparent cells to the grid. When a cell is alive for a certain number of iterations, it will become transparent and the player will fall through. The cell is still treated as an alive cell by the rule set, so it does not tamper with the simulation.

2.4 Level Design

Each level of the game is defined by an initial configuration and a rule set. To allow for the user to customize their own levels and rule sets, this was implemented via a simple text input file. The input file allows comments and encodes the following parameters:

- Grid size
- Wrap or no wrap flag
- Number of initially alive cells
- Alive cell coordinates
  - The first coordinate is where the player will spawn
- Number of rules
- Rule definitions
  - Only rules that result in an alive cell are specified

The `example_input.txt` file in the "Level Templates" directory of the repository details how to format the text input files. Additionally, the `level_template.txt` file provides a template with less comments to be copied for new level creation. Adding custom levels to the game entails adding the levels to the appropriate directory depending on your OS:

- Windows Builds/The Game of Life Game_Data/StreamingAssets/Levels
- macOS Builds.app/Contents/Resources/Data/StreamingAssets/Levels

The game itself includes 10 levels, each inspired by classic game of life configurations. These include a variety of oscillators, spaceships, and of course the Gosper glider gun. The 10 included levels are as follows:

1. Gliders
2. Blinkers (period 2)
3. Toads (period 2)
4. Pulsar (period 3)
5. Light Weight Space Ship
6. Middle Weight Space Ship
7. Heavy Weight Space Ship
8. Far Gliders
9. Pentadecathlon (period 15)
10. Gosper Glider Gun

2.5 In Game HUD

To display relevant information to the player during game play, a heads up display (HUD) was added to display the timer and score in game. The timer starts counting from 0 once the evolving grid starts, and the score is computed as a function of time in seconds multiplied by net distance accumulated (ignoring the vertical, or jumping direction). This encourages the player to move around the grid as much as possible to set a higher score. Also included in the HUD are a countdown timer before the game starts and a game over screen when the player falls off the grid. Upon falling off of the grid, the player still has control of the camera and can look upwards to watch the simulation continually progress.

2.6 Main Menu

The main menu is a simple canvas object with two UI groups for the main menu and the options menu, which each have controllers allowing them to interact with themselves and each other. The main menu controller simply switches scenes if play is pressed, changes the active UI group of options is pressed, and quits the application if quit is pressed. The options menu allows the user to select the level which play will open, as well as the difficulty, sound, and music settings. UI components are wired using Unity's `EventHandler` functionality, which enables UI elements to send signals when different actions occur; these signals then call scripts which complete the process.

2.7 Music

A quality soundtrack is a crucial component of any entertaining video game. In the spirit of Game of Life, the 4 unique formations of a glider were used to generate a musical melody. Each of the 4 formations of a glider can fit in a 3 by 4 grid. To generate a melody from this grid, each formation corresponds to a beat of a musical measure. The length 3 dimension encoded rhythmic triplets for each beat, and the length 4 dimension encoded 4 different musical notes. To make the soundtrack sound both peaceful and playful, C Major 7 and F Major 7 chords were alternated as the 4 note encoding. More specifically, these encodings were (E, G, B, C) and (E, F, A, C). A visualization of this encoding is provided in figure 1.

The main melody present in the soundtrack was generated using this glider encoding, and the rest of the parts were written and recorded by Jackson, including a section of piano improvisation to give the soundtrack more life. The live recording of the soundtrack can be viewed on **YouTube** (click for link).

3 Challenges

We faced a variety of challenges during development. Most prominently, neither of us were familiar with Unity, so we had to learn everything from the ground up, including C#. While similar to Java, C# still had some nuances which delayed us in getting started. We also had issues with the GitHub repository, as some of Unity's auto-generated files did not mesh well when committed, so we had to take special care not to include those files while resolving any issues the exclusion caused. The mouse behavior was also tedious to implement, especially

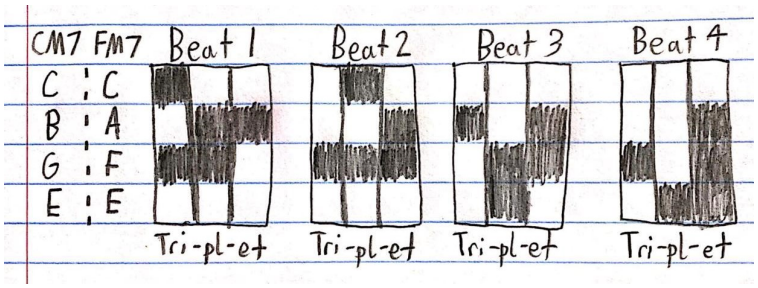

Figure 1: Glider configurations were used to generate the soundtrack

because Unity uses highly complicated quaternions in order to represent rotation, which we had to interact with directly in order to rotate the camera and player object. Lastly, asset management posed a problem; when we first built the project, none of the static resources loaded properly; however, with some effort debugging, we were able to discover the `StreamingAssets` utility, which solved our issues.

4 Conclusion

In conclusion, we have successfully created a playable game based on John Conway's Game of Life, titled "The Game of Life Game". To play the game, simply download the executable from the appropriate OS Build directory in the public **GitHub repository** (click for link) of the project. We hope to share this project with both friends and family to help increase general knowledge about the Game of Life, 2D cellular automata, and unconventional computation!