Requirements Specification
Game of Life

J. Anthony Sterrett, Jr.

1 Core Concepts

1.1 Conway's Game of Life

Conway's Game of Life is an example of a *cellular automaton*, which governs a two-dimensional grid of "cells" which are each in one of two states – "dead", or "alive". As time passes in the grid, cells may be born or die according to the states of their neighbours – specifically, an alive cell stays alive if two or three of its neighbours are alive; a dead cell comes to life if three of its neighbours are alive; under any other circumstance, a cell becomes (or stays) dead.

Conway's Game of Life is not the only cellular automaton, but it is the most famous example, and most other two-dimensional CA's result in "boring" worlds. Conway's Game of Life, on the other hand, results in dynamic worlds with interesting properties, the explication of which is beyond the scope of this document.

1.2 Multiplayer

The concept of a Cellular Automaton may be cast into the milieu of a multiplayer game thus: cells have a dead state, and one "live" state for each player. After each time step, each player selects one cell to mutate; the world then resolves their mutations and executes the next time step according to the rules of the cellular automaton.

A natural extension of Conway's Game of Life into a two-player game follows:

- If a dead cell has three live neighbours, it changes to the state which is most represented among its live neighbours.
- If a live cell has two live neighbours, and those neighbours are in the same state, it changes to the state of those neighbours.
- If a live cell has two live neighbours, and those neighbours are in different states, the cell stays in its previous state.
- If a live cell has three live neighbours, it changes to the state which is most represented among its live neighbours.
- Under any other circumstance, a cell remains or becomes dead.

2 Specification

2.1 Gameplay

The gameplay, then, is rendered thus. A board is drawn on-screen, measuring eight cells by eight cells, and twenty of these cells are coloured – ten for each player, those of the player's chosen from the lower left-hand

quadrant, and those of the opponent's chosen from the upper right-hand quadrant. The starting cells are selected in a manner such that the state of the board exhibits 180-degree rotational symmetry.

At this point, the player clicks (or touches, for a touch-screen device) one cell of their choice. At the same time, their opponent – run by a computer script – selects a cell as their move. Once both moves are submitted, the board is mutated thus:

- If both players select the same cell, no mutation occurs
- Otherwise, if a player selects one of their opponent's cells, that cell becomes dead.
- Otherwise, if a player selects a dead cell, that cell becomes one of theirs.

After this mutation occurs, the game calculates the next world state according to the cellular automaton's rule, the updated game board is displayed to each player, and the players once again have a chance to select a cell.

The game continues until either one of the players forfeits, or else one of the players has no more cells on the board. At that point, the other player is declared the winner.

2.2 Code

The majority of the project will be written in C++, using the SDL2 library for graphics, and using Lua scripts for certain dynamic elements, as expounded upon below. Here is a trace of how the code flow should proceed:

- The user is presented with a main menu, which offers options to either begin a game with the default settings, or to load a custom AI script or a World script.
- A thread is spawned which acts as a server, listening on a predefined port on the local machine. This server accesses the designated World script to control the world transitions.
- Another thread is spawned which acts as the opponent. This thread accesses the designated AI script to control the opponent's moves.
- The main menu disappears, and a window is spawned which houses the game board. The game then proceeds until one or both players have no more cells on the board.

2.2.1 Scripting

There are two opportunities for Lua scripting in this project.

The first is called an AI script. This script will consist of a function called move() which takes as parameter the current World state, and returns a choice of cell which will represent the computer player's move.

The second is called a World script. This script will consist of a function called tick() which takes as an in-out parameter the current World state, and changes it to the next World state.

The existence of these bits of logic as scripts allows users to customize their game experience. By changing the AI script, players can play against interesting opponents, and discover new strategies. By changing the World script, players may discover and play different games within the basic framework we create.

The game will come with defaults for each of these scripts; the default for the World state will implement the rules as explained above, and there may in fact be multiple default AI scripts, representing different difficulties.

3 Miscellaneous

3.1 Wishlist

- There is no upper limit to how long a game may take, so some sort of save functionality may be desired.
- Some sort of scoring algorithm may be desired for comparison – perhaps "time to victory"?
- Human-vs-human play requires a medium-scale server project to pair opponents. The desire for this capability should be kept in mind and planned for, but it behoves us to create first the capability for playing against a computer opponent, and extend as time and resources permit.
- Since we need a server anyway for human-vs-human gaming, it would be nice to create a website to serve as a repository for interesting AI scripts and World scripts.
- It would be prudent to allow a non-game mode for experimentation (or artistic expression), which implements only a given World script and allows such functions as "pause", "step", "play", "back", and "reverse". To wit: "pause" would halt calculation of the World script and allow the user to change the states of multiple cells; "step" would advance the World state one step; "play" would advance the World state at a given (perhaps user-modifiable) rate; "back" would load the previous World state (if applicable), and "reverse" would rewind previously played states (again, if applicable). (Obviously "back" and "reverse" would only function if "step" or "play" had previously been selected; it is in the general case impossible to derive the previous World state given the current World state and the World script.)

3.2 Concerns

- Multithreading and sockets may not be the most convenient way to implement a local game, although they're the most modular. (I.e., after setup, the code running the player's interface doesn't know or care if the opponent and server are local to the machine, or elsewhere.)