Jason Goldfine-Middleton
932675265

CS 162 – Assignment 1 Reflection

Final Design Description:

main.cpp

```
for each iteration of the game
    Ask user for initial pattern: oscillator, glider, glider gun
    Ask user where upper left corner should be located (give user restrictions)
    Ask user how many (10-1000) generations they want to see
    Ask user how many ms (10-1000) each generation should last for
    Play game*
    Ask user whether to play again
```

* Playing the game consists of calling the Game::play() method on a Game object created with the following arguments: the dimensions of the visible window (40x20), the chosen starting shape, where the upper left corner of the bounding box for that shape will be, number of generations, duration of each generation

struct Cell – represents a location (x, y) on the grid

class Game – contains the grid, rules for changing Cell states on the grid, and a method to run the game

fields:

- width (of grid, including nonvisible buffers)
- height (of grid, including nonvisible buffers)
- buffer (number of extra rows/columns off each edge of visible window)
- generations (number of ticks in this game)
- pauseLength (duration of one tick)
- grid (represents the world with cells being either alive or dead)
- patternCells (stores the user's chosen initial pattern of live cells in the form of coordinates, adjusted with the user's chosen upper left corner)

methods:

- constructor (sets world to initial state with shape included, i.e. tick 0)
- destructor (kills the grid, which is a dynamic 2d array)
- clearGrid (fills the entire grid with dead cells – only run once)
- determineFate (for a cell, apply the rules by counting live neighbors and store its state for the next tick in an array that is passed in)
- initializePattern (using the patternCells field, set relevant cells of grid alive)
- play (runs the game based on the field settings, specifically generations and pauseLength)

- printWholeGrid (for testing purposes, prints the entire grid including buffer region)
- printWindow (print the visible window based on the width, height, and buffer fields)
- setPatternVector (produces the hard-coded base coordinate vectors for the oscillator, glider, and glider gun, setting the patternCells field to an adjusted vector based on the pattern and offsets passed in as arguments)
- tick (applies the rules of the game to current grid, stores the results in a new grid, and after updating every cell, overwrites the original grid **)

** To deal with edge cases, this involves not overwriting the border columns and rows so they always stay dead

Test Plan:

My aim with testing this program was to ensure that every requirement in the "Grading" section of the assignment specifications was complete. I used the approach of incremental development, with testing following the completion of each module produced. My initial module was the user input segment. As I constructed the loops and bounds-checking of user input, I ensured that the game would only accept sane, restricted inputs. The second module of concern was the creation of a struct meant to store the coordinates of a position on the grid. This required only straightforward examination. The third module was the formation of an empty grid filled with dead cells. To test this, I also coded the two functions for printing out the grid, one of which was strictly meant for testing. The fourth module was the ability to place one of the three starting patterns (oscillator, glider, glider gun) on the grid. Testing this consisted of printing the empty grid, applying one of the three patterns in an arbitrary location on the grid, and then printing the grid again.

The hardest part of testing was the final module, which was ensuring that each tick produced correct output in the user-viewable window. This module had a sub-module, which was the handling of the edges and corners. Although I ultimately discovered that corner cases were basically irrelevant due to defined constraints on the user, I had expected to run into issues with the buffer rows/columns I created outside of the viewing window and, more particularly, with artifacts occurring as gliders approached the true (hidden) outer edge of the grid. In order to check that a tick produced correct output, I completed work on coding the loop to run multiple generations, and ensured that, for the oscillator and glider gun, the behavior was cyclical over many generations and, for the glider, that it correctly alternated between four orientations until it collided with the outer edge.

Finally, I focused on the user experience of the Game of Life, verifying that all gliders appeared to disappear off an edge with no disruption to their patterns.

Jason Goldfine-Middleton
932675265

Test Results:

With respect to the testing regimen established above, the following were the results. Some of the results had major implications for the final design of this program.

The user input module worked flawlessly and passed all tests, including when the user attempted to repeatedly enter bad input. (Note: it did not handle the case where the user entered a non-numerical character when an integer was expected.) This test being passed gave me confidence that only sane values would be passed into the Game constructor as arguments, satisfying all the relevant pre-conditions of the class methods.

As stated previously, testing of the Cell struct was minimal due to its simplicity.

Testing for an empty grid was straightforward. Of the two methods for printing the grid, printWholeGrid() and printWindow(), the former was strictly a testing method. Later in the testing process, it would serve the purpose of displaying the behavior of patterns as they collided with the edges of the grid, but at this stage it allowed me to verify that the entire grid array was filled with dead cells.

Testing the creation of the initial life patterns on the grid was straightforward with the two print methods as well. I also ensured that the constellation of coordinates for each pattern was correct by crosschecking them with those in other implementations online. This required manual verification of each hard-coded coordinate pair. I had to do further construction in order to ensure that the patterns evolved as expected and, in the case of the glider gun and glider, that the edge behavior was correct.

I spent the most time testing whether the shapes evolved as expected. This was easy for the oscillator and glider, but difficult for the Gosper glider gun. Since it would be too painstaking to check each individual generation with a known correct sequence of ticks, I decided to look at the state of the cells after a certain number of generations. Once I saw that the state after a given amount of generations matched that of a known correct implementation for a few different numbers of generations, I was satisfied. Initially my code was producing errors with the evolution of the glider gun, leading to its decay. In trying to locate the source of this bug, I removed a large portion of my edge-handling code. Much to my surprise, I found that not only had my edge-handling code caused problems with the glider gun, it was actually completely unnecessary. Removing that chunk of edge-handling code allowed a glider to come to the edge and, when it "collided", create a 2x2 square artifact. Testing revealed that this was not an issue, however, because when the next glider arrived, it and the artifact annihilated one another!

Testing the user experience was fundamentally a case of running the executable. Lo and behold, the expected behavior continued even for very small pause durations and for large numbers of generations.

Problem Resolution:

There were three separate stages of problem resolution in the process of producing this program: 1) the initial contemplation and research, 2) the initial coding attempt, and 3) testing the final implementation plan. These phases will be briefly summarized below.

While I considered the challenges of implementing the assignment, I sought inspiration from various implementations on the Internet. Of primary concern in my mind was the issue of coding an "infinite" world. I knew that there was no way to code an infinite grid, so I looked into ways that the edges could be handled in order to simulate an open world. I found various options, including letting the world "wrap around" like a torus and bordering the grid with permanently dead or alive cells. I started to become concerned the possibility of living patterns exiting the visible grid and later returning, messing up the patterns already visible in the grid. If this were to happen, the illusion of the infinite world would be broken. After more thinking, it dawned on me that in the case of the glider and glider gun for this assignment, if I could just destroy the patterns once they got out of the viewing window, it would have no effect of the illusion of infinity. This is only true because naturally the gliders would never return to their source.

In my initial coding attempt, I created a struct, Point, and two classes, World and Game. Additionally, I had enums to represent aliveness/deadness and each initial pattern. I began running into problems with externally accessing the elements of the member array of the World class. I also just decided that the code was becoming too complex with all the abstraction. I knew that the rules were simple and the grid was simple to implement. The only difficult part was really going to be handling the edge cases, so I decided to start on a new design.

The new design, which ultimately evolved into the completed program, was based on a single struct, Cell, to represent a position in the grid, and a class called Game, which contained the grid and the rules for updating it each tick. Although it may not be as OOP-style as is possible for this assignment, I could not see a benefit to making the code more complex and abstract than necessary. The largest problem was the edge cases, as anticipated. I initially tried an approach where, if a live cell was found to be bordering an edge, an entire 4x4, 4x7, or 7x4 section of grid including that cell would be killed immediately. After implementing this edge "guard" I noticed bugs in the game flow and just commented it out. Surprisingly, at that point my code produced the illusion of being an infinite grid!

My world is stored as a dynamic array. A second local dynamic array stores the next generation's world as it is being determined (according to the four rules of the game). Once all the calculations have been done, the next generation's world is copied to the original array, overwriting it. However, the trick is never overwriting the dead cells on the outer edges. This ultimately has the consequence of causing a stable 2x2 square of live cells to form when a glider hits the edge. I thought this would be an issue when the subsequent glider hits that square of live cells, but actually they mutually annihilate! For the initial single glider pattern, since the buffer is 5 rows/columns on each side, the user will never see the 2x2 square formed when it hits an edge.

In this case, it turns out I was overthinking the assignment. Since we only had to make the Game of Life work for three starting shapes, there was a lot less to think about than I had originally expected.