

Homework 4 — Life

(See Canvas for due dates)

(150 points)

Objectives:

- Implement a GUI with elements that interact with one another
- Understand timer mechanics and how to advance state automatically
- Become more familiar with the observer pattern in general (signals/slots in Qt)

Credit:

- `design.pdf` (**deliverable 1**), your `.h`, `.cpp`, `.pro` files for your Qt project (this should be all `.h` and `.cpp` files and your project's `.pro` file; do not include your `.user.pro` file, **deliverable 2**)

Instructions:

You *may* work as a pair for this assignment (but not more than a pair). If you work as a pair, you **must** use a private repository that you invite Sreesha to as a contributor (Sreesha's github username: SreeshaNath). See the guidelines on the course github in `examples/qt_and_git.md` for tips on dealing with Qt projects and git repositories.

Each partner must make at least **5** meaningful commits to the repository. You do **not** need to connect this repo to a Continuous Integration tool.

You may not post any of your code in a public repository.

[Conway's Game of Life](#) is known as a zero-player game; it has an initial state and evolves from there. It is played on a theoretically infinite 2 dimensional grid (though we will use a rectangular plane that connects to itself, more on that later).

Part 1: Design document (10 points, **deliverable 1**)

First, carefully read the entirety of this design document.

You (and your partner) will make a plan of which classes are in charge of what, how they will communicate (what methods they will have, what signals will they emit, what slots will they have), and what data they will control (what fields will they have). You should also indicate what signals will be emitted by UI elements and which objects will respond to these signals.

1. Use the plot project that we'll be working on in class from Weeks 10, 11 & 12 as well as the Qt documentation to help guide your design.

2. Your design document does not need to be a 100% final plan, but should account for all features that you plan on having.

Then, Fill out the Homework 4 Features [google form](#)



Rules:

A live cell is a grid space that is filled in: 

A dead cell is a grid space that is empty: 

All cells have eight neighbors, the cells that are horizontally, vertically, or diagonally adjacent:

neighbor	neighbor	neighbor
neighbor		neighbor
neighbor	neighbor	neighbor

There are four rules in Conway's Game of Life:

1. Any live cell with fewer than two live neighbors dies. (underpopulation)
2. Any live cell with two or three live neighbors remains alive. (stable)
3. Any live cell with more than three neighbors dies. (overpopulation)
4. Any dead cell with exactly three live neighbors becomes a live cell. (reproduction)

All cells *simultaneously* update each turn to become alive, stay alive, die, or remain dead.

IMPORTANT: You need to calculate the state that each cell will have next turn but not update their current state until the next state for *all* cells have been computed. If you update a cell early, it will affect the status of the other cells waiting to be computed too early and your program will behave incorrectly. To do this you will have to do something like either:


- keeping track of a *next_turn_status_* field in each cell
- OR maintaining a 2nd matrix of updated cell states

Our playing field:

We will play on a 2d rectangular field. All squares still have eight neighbors. The top row “connects” to the bottom row and the right-most column “connects” to the left-most column.

For example:

A live cell on the top row.

neighbor		neighbor		
neighbor	neighbor	neighbor		
neighbor	neighbor	neighbor		

A live cell on the right-most column.



neighbor			neighbor	neighbor
neighbor			neighbor	
neighbor			neighbor	neighbor

A live cell in the top left corner.

	neighbor			neighbor
neighbor	neighbor			neighbor
neighbor	neighbor			neighbor

The playing field in your GUI is populated by cells that are 20 pixel squares. Your playing field must be at least 20 cells wide and 10 cells tall.

When you begin the game, you should randomly initialize each cell with a 50% chance of being alive vs. dead.

Graph:

Underneath our playing field is a graph that tracks the percentage of live cells over time. The top of the y axis is 100% alive and the bottom is 0% alive. The graph is 100 pixels tall. Each bar is 20 pixels wide. Once the graph “fills up”, the oldest bars should no longer be plotted and the most recent bars should appear on the right hand side. Which is to say, the graph will contain the bars corresponding to the n most recent turns where n is equal to the graph width / bar width.

Interactions, Time, & Turns:

The user can click on the playing field. If the user left-clicks on a cell, that cell becomes alive if it was dead or stays alive if it was already alive. If the user right-clicks on a cell, that cell dies if it was alive and remains dead if it was already dead. A mouse click does not trigger the game to advance one turn.

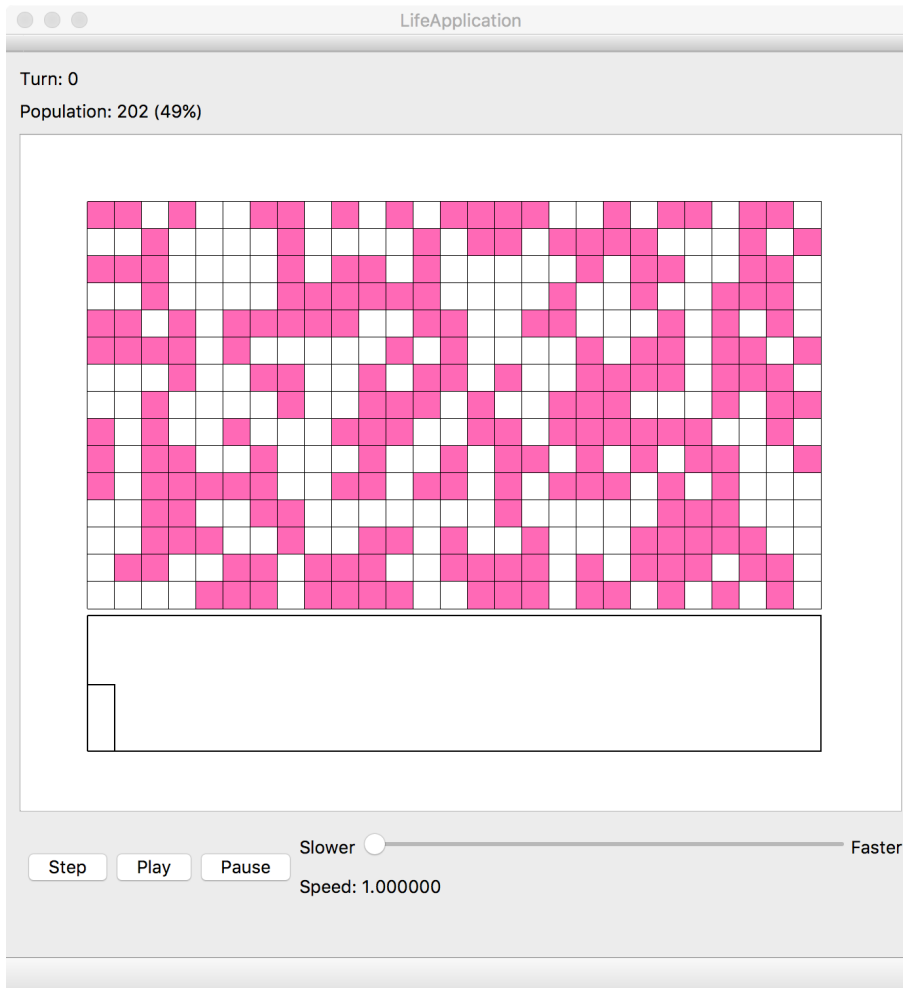
There are **only** three buttons in your GUI. The user can interact with them in the following ways:

- 1) Step—this button advances the game 1 turn.
- 2) Play—this button makes the game play automatically, advancing 1 turn every x seconds, where x is determined by the speed adjustment that the user makes in the GUI.
- 3) Pause—this button stops the game from playing automatically.

There is 1 slider that the user can interact with. This slider determines how quickly the game will advance (how many seconds between turns when in “Play” mode). If the user adjusts the slider while the game is playing, the game should auto-adjust to the new speed after the user stops adjusting the slider. The “Speed” label should update when the user moves the slider.



The graph at the bottom of the screen, the “Turn” label, and the “Population” label should update every time the game advances one turn. They begin with the data based on the initial start state of the game.



Other features (10 points):

You must implement at least 2 extra features. You can add menus, settings, and change the kinds of cells that can populate your playing field. These features must be “meaningful”, as in they must change the user's experience when they interact with the program.

An example of a feature that is not meaningful: change the color of the cells to a new color

An example of a feature that is meaningful: change the color of the cells to a new color based on either user input or on state of the game.

Any questions about what counts/works as meaningful features can be directed to Sreesha asked on Ed.

Object Design:

The design of this program and the underlying objects is entirely up to you.



Tips:

- 1) Items that do not inherit `QObject` cannot emit signals, nor have slots.
- 2) `QGraphicsScene` has an `ItemAt` function that you can use to tell what item was clicked in your scene. The `QGraphicsItem` must implement *both* the `shape()` and the `boundingRect()` methods correctly to be properly detected by `ItemAt()`.
 - a) To do this, you will need to subclass `QGraphicsScene` so that you can override whatever event methods (such as `mousePressEvent`) that are necessary.
- 3) Take a look at the [QTimer](#) documentation. You may choose to use `QTimer`, or you may use one of the alternatives; our game doesn't require single-shot timers or signals (from the timer).
- 4) Use [QDebug](#). The easiest way to use this class is to import it and direct whatever you want to print to `QDebug()`. (e.g. `QDebug() << "click";`) `QDebug` handles correctly routing the output to the console so that you don't end up with weird behavior that can happen with `cout`, such as the console not updating until after you exit the application.
- 5) Watch a few videos/demos of Conway's Life game in action so you are familiar with what the correct implementation looks/behaves like. This will help you identify if there is something wrong with your logic.
- 6) Feel free to use any parts of the plot project as inspiration.
- 7) We highly recommend using only `QGraphicsScenes` and `QGraphicsItems` to visualize your cells/the population tracking graph. We do not recommend using `QTables` or `QGraphs`.

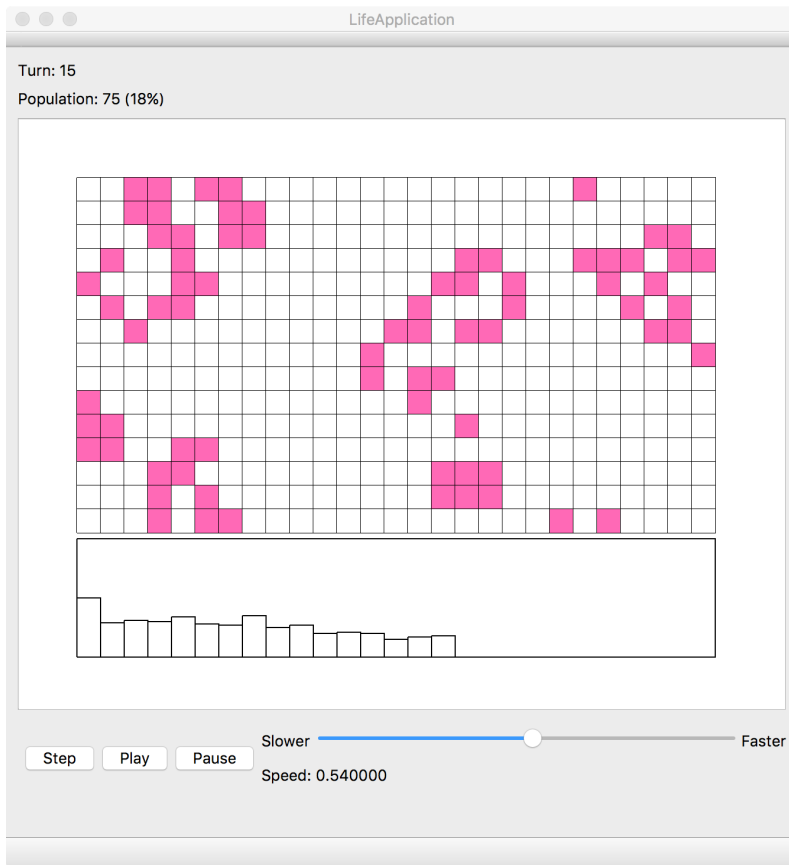
Layout/appearance:

You don't have to match this screenshot pixel-for-pixel, but you should be close.

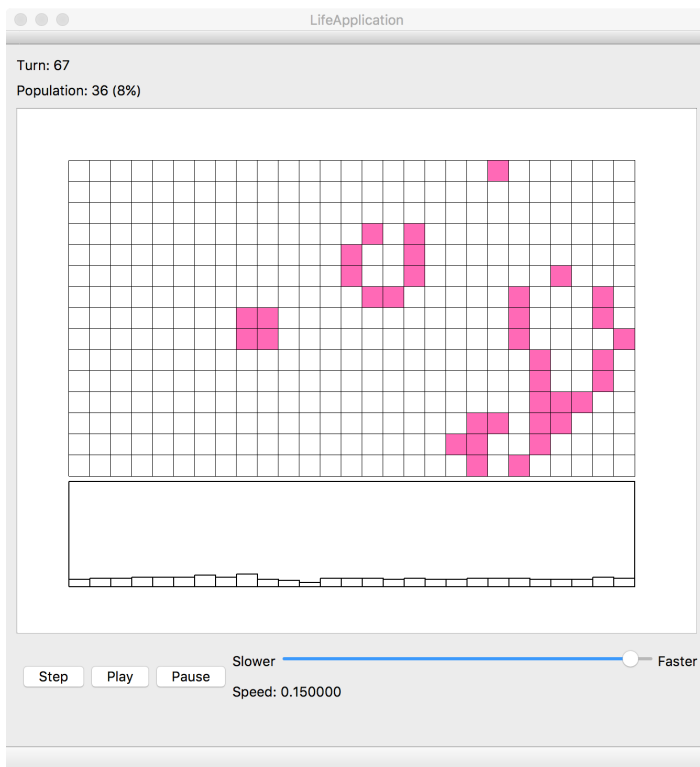
The following picture shows the GUI annotated with the vertical layouts (solid boxes) and horizontal layouts (dashed boxes) used.



The following is an example some turns into the game.



The following is an example screenshot many turns into the game.



Rough Rubric:

	Full credit	Points (total = 150)
design document	as described in instructions	10
Overall Functionality		70
Life game works (model)	has a grid of cells that are alive or dead correct updates for a time step has either a 50% random starting state, or if one of their extras is to implement patterns/something else, has that	35
Life game updates (drawing on the view port)	These points are for the GUI accurately reflecting the underlying state of the cells	15
Population tracking graph	Newest bar always added on the right, graph moves to the left once it is full bar height correctly computed as percentage of live cells	20
GUI interactions		45
Left click makes a cell alive	Not required to work while the game is playing	2.5
Right click kills a cell	Not required to work while the game is playing	2.5
Step button	Advances game 1 step	10
Play button	Advance game until the pause button is clicked. Restarts game if game has been paused.	10
Pause button	Stops the game.	5
Speed slider	Slowest is on the left, fastest is on the right. Game speed can change while it is playing by moving the slider.	10
Text labels	Turn, population, and speed labels update appropriately	5
"Extra" features		10
Feature 1	What they said that they would implement is in the extra features spreadsheet (in this folder) Feature must work correctly and must be ok'd	5
Feature 2		5
Style, Comments, Partner work	as in previous assignments. May follow Qt naming conventions. Partners must contribute equally.	15

