

# Homework 4 — Life

Due Friday, November 8th at 6pm  
(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:

- `life.zip` (this should be your zipped up Qt project)

## 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 Felix to as a contributor. 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).


**Design document:** See PE 10 description.

## 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. (Do not update any cells before you have decided for all of them whether they should be alive or dead).



## 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: change the color of the cells to a new color based on user input or on state of the game.

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

Features need to be chosen by Friday, November 1st at 6pm (see PE 10 instructions).

**Object Design:**

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

**Tips:**

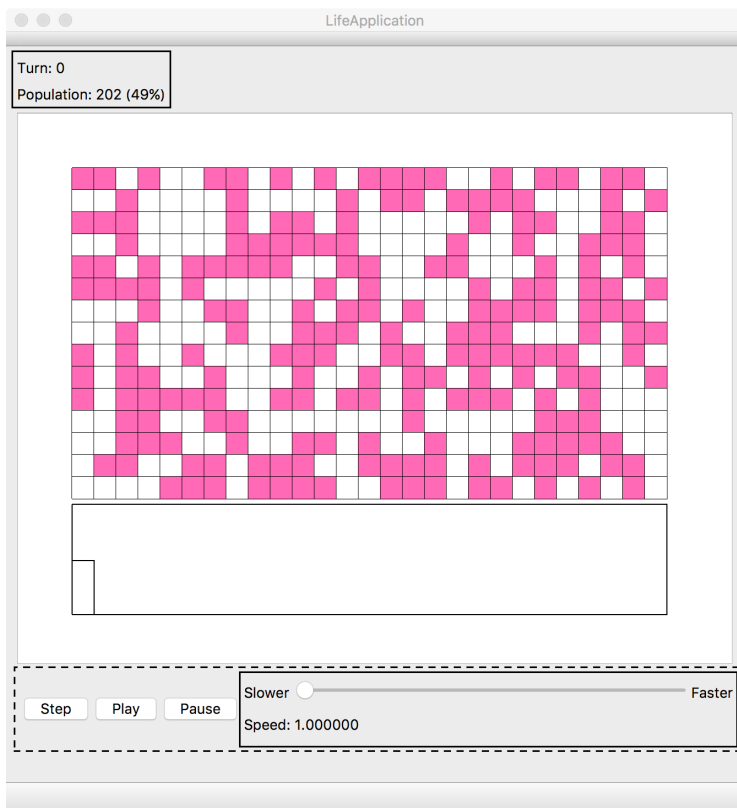
- 1) Items that do not inherit QObject cannot emit signals, nor have slots.
- 2) QGraphicsScene has an ItemAt function that you can use to tell what item was clicked in your scene. The QGraphicsItem must implement *both* the shape() and the boundingRect() methods correctly to be properly detected by 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 the weird behavior that can happen with cout, such as the console not updating until after you exit the application.

**Layout/appearance:**

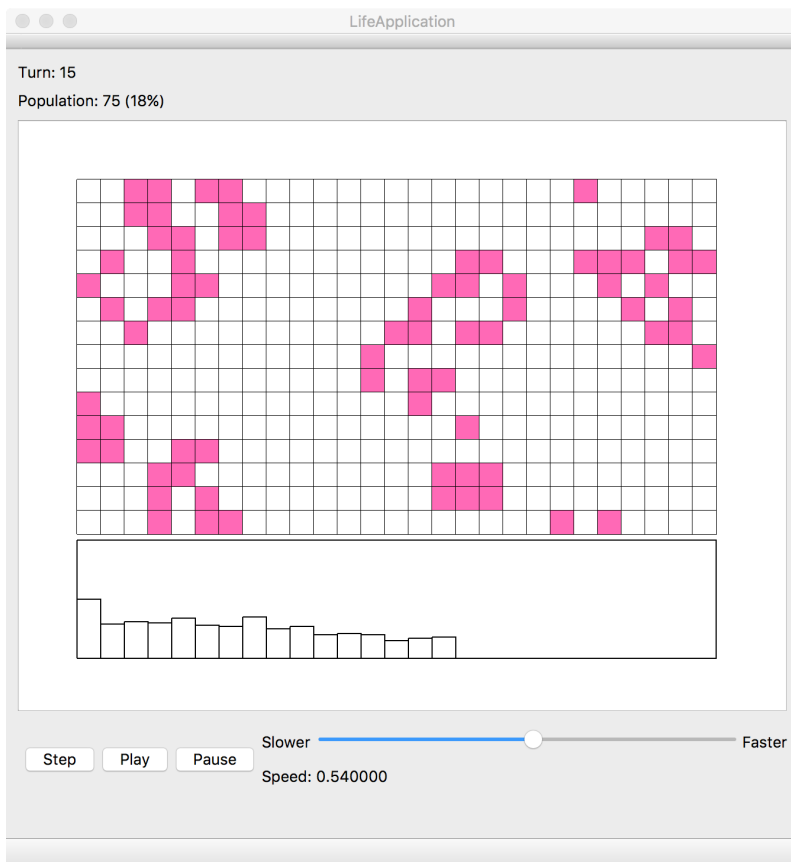
You don't have to match this screenshot pixel-for-pixel, but you should be close.

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.



