Lab 09

Conway's Game of Life (BYOW Intro)

CS61B Spring 2024

Announcements

Project 3A World Generation is due Monday, 4/15.

Build Your Own World (Intro)

Introduction

Build Your Own World or BYOW is Project 3, where our goal is to build a game that generates random, explorable worlds.

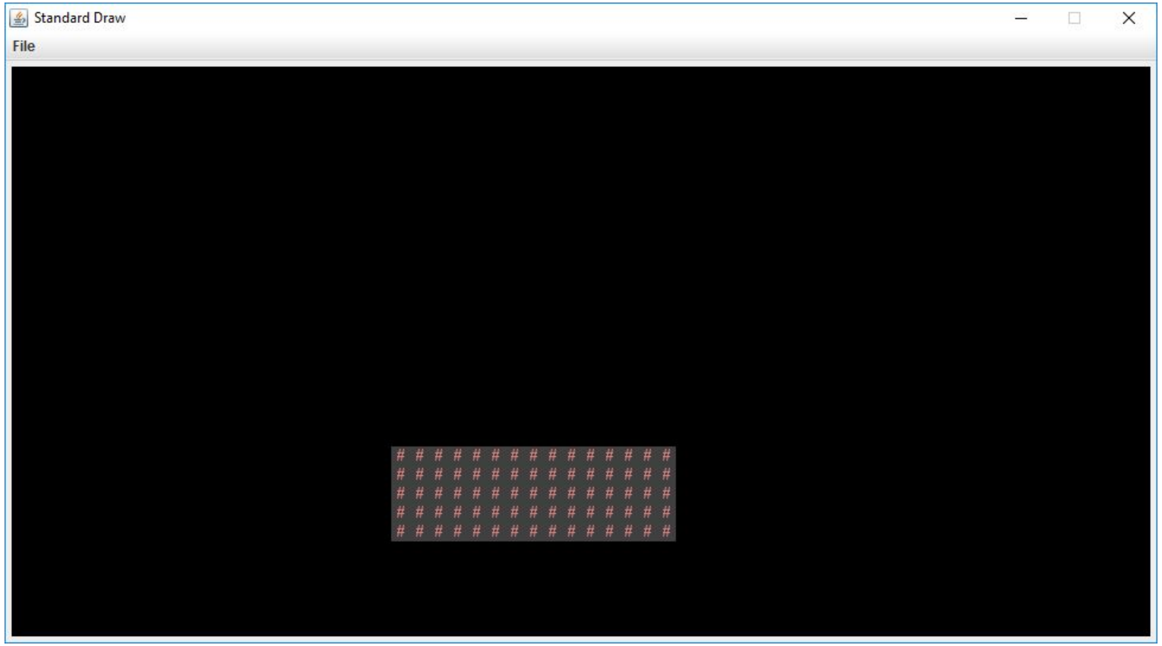
Throughout this project, there are several criterias that you must meet (as detailed in the spec). Lab 09 serves as an introduction to some of the tools you'll want to familiarize yourself with as well as other useful concepts to help you get started on the project.

World Generation

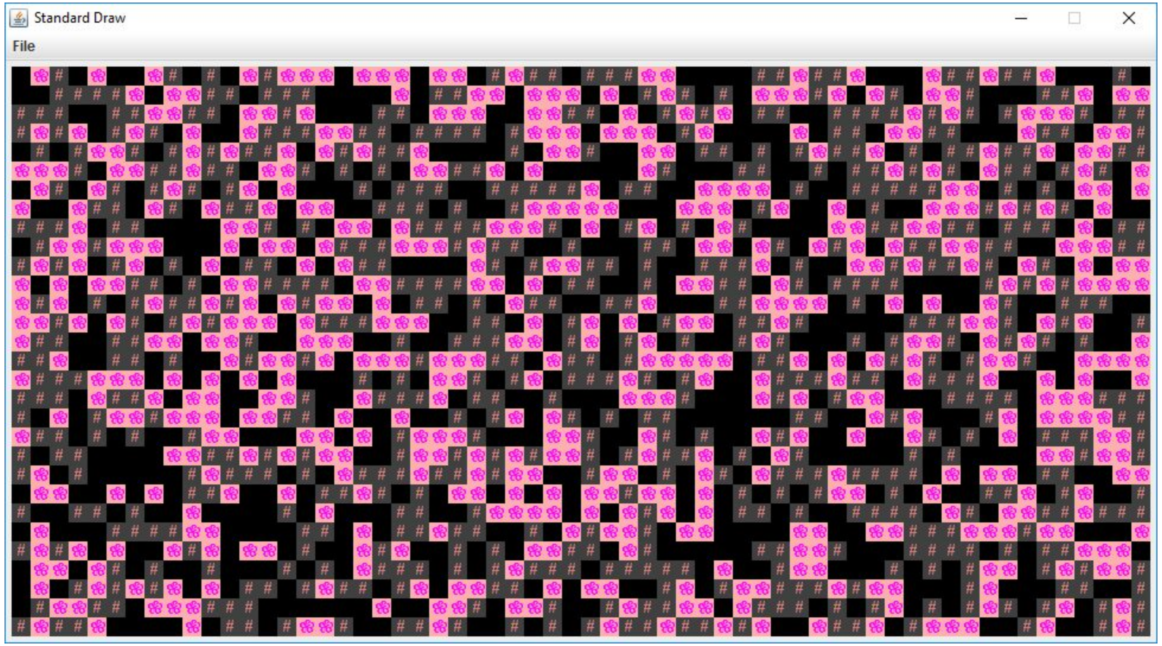
World Generation

Before we get started, let's consider what it looks like to build a world.

Maybe Something Like This (BoringWorldDemo)



Or This (RandomWorldDemo)



World Generation

For this lab and Project 3, we've provided several classes to help build your world:

- A tile rendering engine (**`TERenderer`**)
- A representation of a 2-D tile array (**`TETile`**)
- Some pre-generated tiles (**`Tileset`**)

All of this can be found in the package **`tileengine`**. If you haven't already, go ahead and open up the lab assignment to view these files/classes.

Project 3

Here are some topics we'll cover briefly to help you get started on Project 3:

- Pseudorandomness
- Persistence

Pseudorandomness

In project 3, each of the worlds that are generated should be noticeably distinct from each other. To ensure that they are random (more criteria in Project 3 spec), you'll want to use a random number generator.

More specifically, it'll be a *pseudorandom*. We can do this by creating a `Random` object. If given a seed, the sequence of numbers generated will be deterministic (hence it being pseudorandom).

Pseudorandomness

An example using the `Random` object is provided in the lab spec that we recommend checking out.

The main question to consider is ***how might you also use it to make your worlds noticeably random and distinct from each other***?

- What qualities for each world would you want to randomize? Hallways? Rooms? Location of the rooms?

Persistence

Another part of the project will involve persistence - effectively, how are you going to save the state of your world, so it continues to *persist* after the program ends?

- This involves a later part of the project (not in world generation), but it might help to start thinking about this early.

Persistence - Scenario

Let's set up a hypothetical scenario: Say your game allows players to collect coins that spawn randomly on the map. Collecting these coins adds to a score.

- The players makes a few movements around the map and collects the coins.
- Then, the player saves and quits. When the player loads the world back in, they should expect it to be the same as when they left off and allow them to continue their progress.

Persistence - Questions

Some questions:

- How do you make sure the world is the same (specifically, the rooms and hallways)?
- How do you ensure that the coins *they haven't collected* are still in the same location? Or what about the coins they already collected? Those shouldn't appear anymore.
- What about their score? The player's location?

Persistence - FileUtils

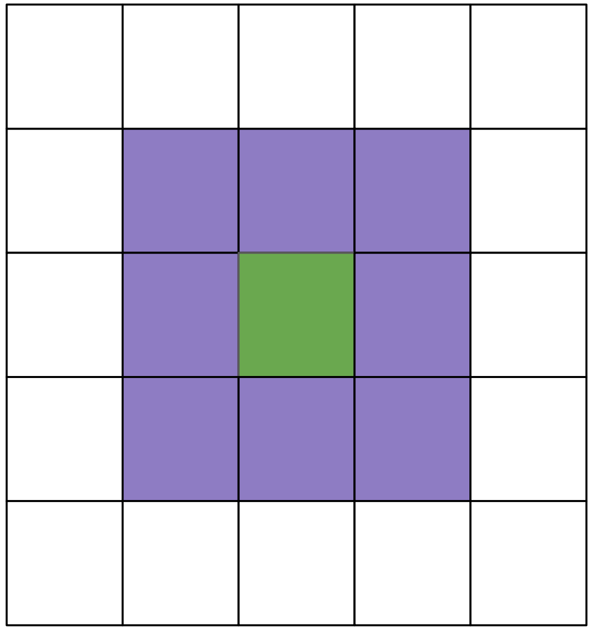
For this lab, you'll learn about storing information in a text file and loading it back into the program. From there, start thinking about what information you might want to actually end up saving and loading back into the game.

Conway's Game of Life

Background

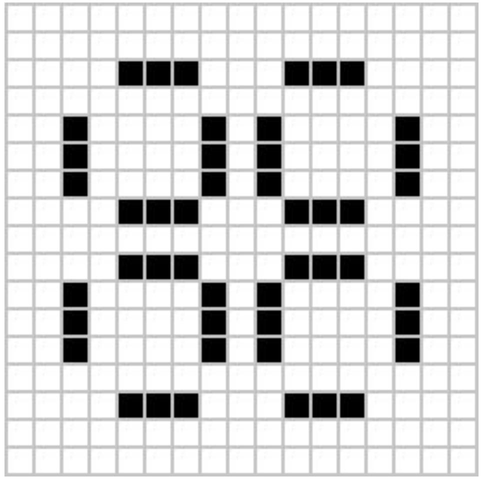
The lab spec goes into a little more detail, but Conway's Game of Life is an example of how cells change over time. It is a zero-player game and each cell is either alive or dead.

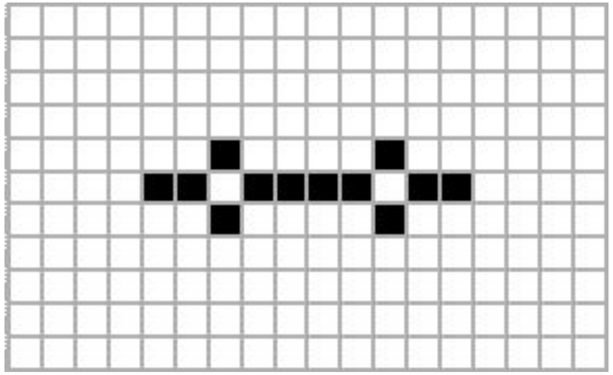
The status of a cell can change depending on its 8 neighbors - right, left, bottom, top, and diagonal (example to the right).

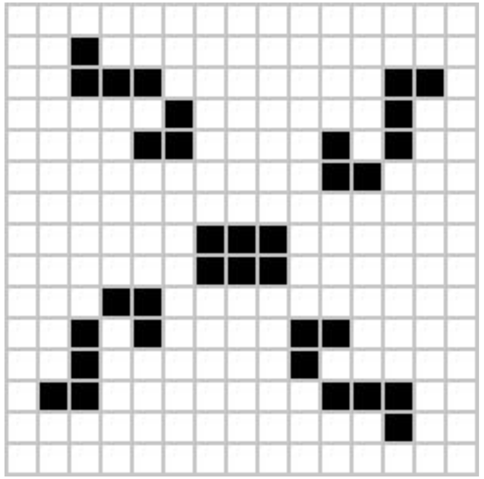


Some Examples

Here are a couple examples of how cells can change over time





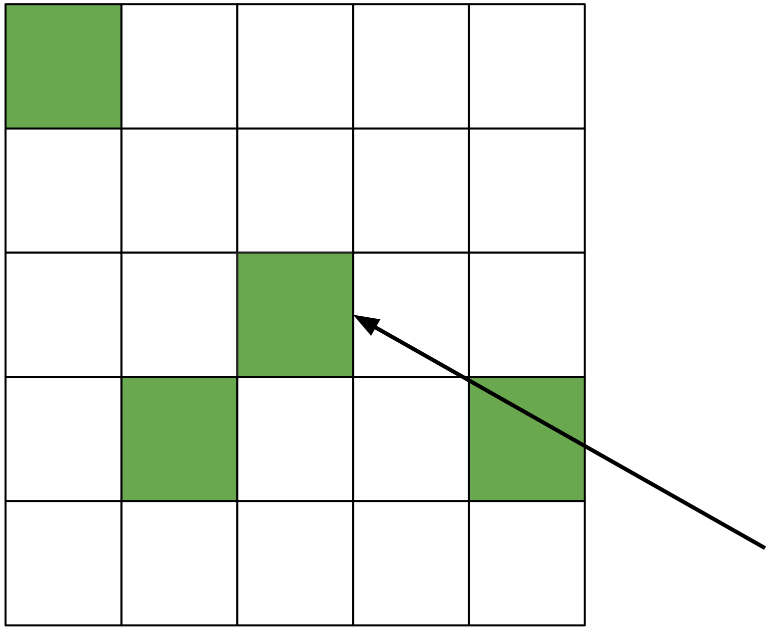


Neighbors and Generations

Earlier, we mentioned how the status of a cell can change based on its 8 neighbors. At each time step, we check if the status of a cell changes, based on the 4 rules:

1. Any live cell with fewer than two live neighbors dies, as if by underpopulation.
2. Any live cell with two or three neighbors lives on to the next generation.
3. Any live cell with more than three neighbors dies, as if by overpopulation.
4. Any dead cell with *exactly three live neighbors* becomes a live cell, as if by reproduction.

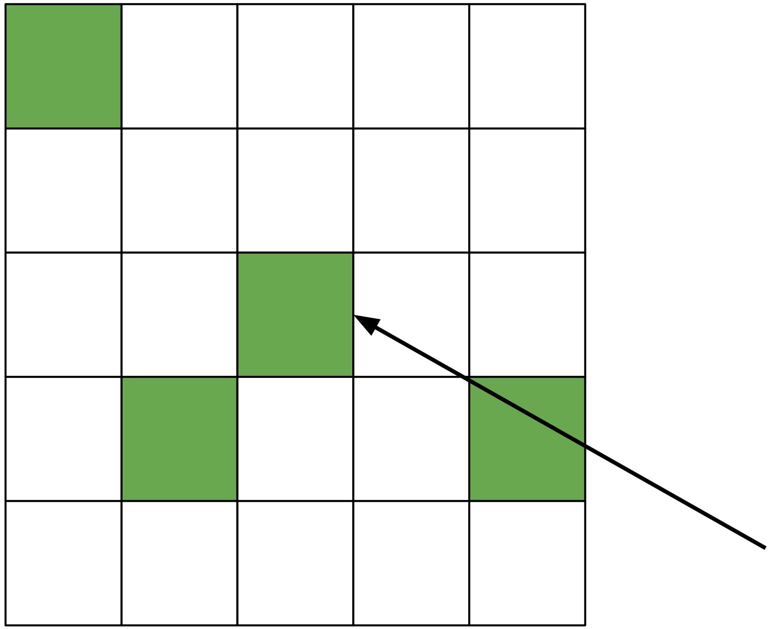
Example



1. Any live cell with fewer than two live neighbors dies, as if by underpopulation.
2. Any live cell with two or three neighbors lives on to the next generation.
3. Any live cell with more than three neighbors dies, as if by overpopulation.
4. Any dead cell with *exactly three live neighbors* becomes a live cell, as if by reproduction.

If we're looking at that cell, how should we expect it to change next time step?

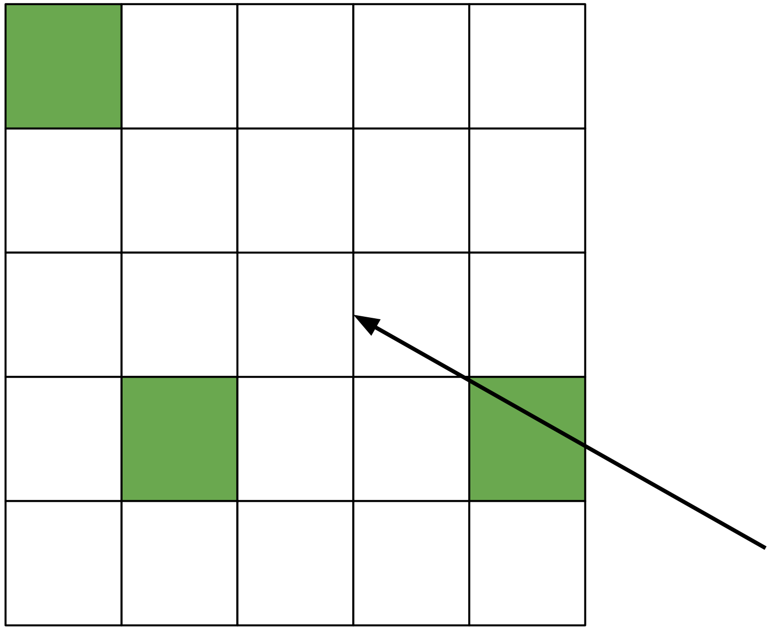
Example



1. **Any live cell with fewer than two live neighbors dies, as if by underpopulation.**
2. Any live cell with two or three neighbors lives on to the next generation.
3. Any live cell with more than three neighbors dies, as if by overpopulation.
4. Any dead cell with *exactly three live neighbors* becomes a live cell, as if by reproduction.

It would cease to exist!

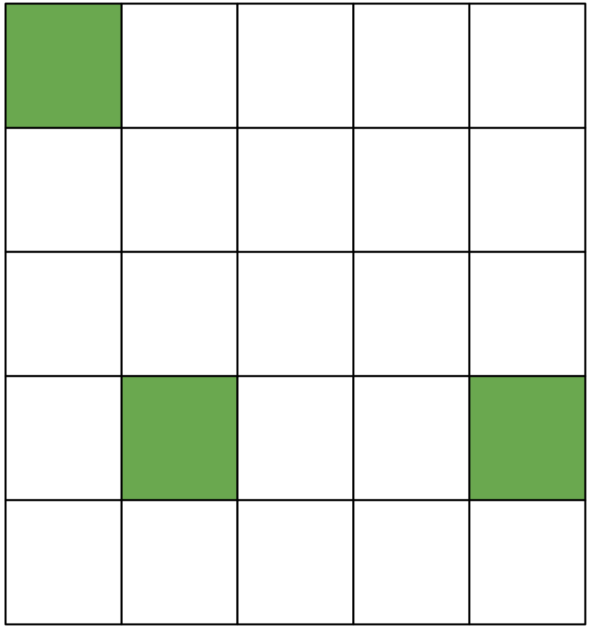
Example



1. **Any live cell with fewer than two live neighbors dies, as if by underpopulation.**
2. Any live cell with two or three neighbors lives on to the next generation.
3. Any live cell with more than three neighbors dies, as if by overpopulation.
4. Any dead cell with *exactly three live neighbors* becomes a live cell, as if by reproduction.

It would cease to exist!

Example



An important thing to note is that you'll have to check this for all cells, since it is possible for a dead cell to become a live cell.

Next Generation

The rules we previously defined is how we create the next generation of cells. For this lab, you'll be be implementing the following method to produce the generation of cells:

```
public TETile[][] nextGeneration(TETile[][] tiles) {
    TETile[][] nextGen = new TETile[width][height];
    // More code below


}
```

The passed in `tiles` will be the current generation/state. The next generation/state should be saved in `nextGen`.

Saving and Loading

We’ll also be implementing two methods to help use save and load the board. Read up on the spec and make sure you understand the requirements and specific format that we’ll be looking for!

Lab Overview

An Overview

Lab 09 is due Friday, 4/05 at 11:59 pm.

Deliverables:

- Complete the following methods in lab09: `nextGeneration`, `saveBoard`, `loadBoard`
- The local tests are not comprehensive - passing them all does not guarantee full score on Gradescope.

Do not modify the files in patterns. The provided local tests will not run if they are changed, and they should remain unmodified for testing purposes.