CS 2112 Fall 2024
Assignment 6
Graphical User Interface

Due: Monday, December 9, 11:59PM
Design document due: Sunday, November 24, 11:59 PM

In this assignment you will use the JavaFX API to build a well-designed graphical visualization of the critter world simulation described in the Project Specification. The visualization will have a responsive graphical user interface (GUI) that displays the positions of critters, rocks, and food. It will connect to the simulation back end developed in previous assignments to permit the user to load and inspect critters, start and stop the simulation, adjust the simulation rate, or advance the world one step at a time. You will also build better critters and explore concurrent programming.

The majority of the work for this assignment focuses on developing new functionality. However, you will also be expected to fix any bugs in your code for Assignments 4 and 5. As this is the final submission of your project, it will be weighted more heavily than previous assignments.

This assignment is quite challenging and complex, but it's quite rewarding. Please read this document in its entirety and start now!

1 Changes

- None so far!

2 Instructions

2.1 Grading

As usual, solutions will be graded on design, correctness, and style. A good design makes the implementation easy to understand and maximizes code sharing while maintaining modularity. Your program should compile without errors or warnings and behave according the requirements given here. Your code should be clear, concise, and easy to read.

We will evaluate your user interface on visual appearance, layout, and design of the controls. We are looking for an attractive and functional interface that offers an enjoyable experience for the user. We have not specified precisely what this means, as we would like you to think it through and come up with your own design.

We also will evaluate the performance of your GUI front end in tandem with your simulation back end. Your front end should be responsive to all user inputs, and to the extent possible, it should always display the most up-to-date state of the world during simulation, Your simulation speed should not be constrained by computational costs from rendering the graphical user interface.

It is a good idea to storyboard your design (draw sketches) and to experiment with different layouts to see what works best. Avoid getting locked into design decisions too early in the process.

2.2 Final project

This assignment is the third and final part of the three-part final project for the course. Consult the Project Specification to find out more about the overall structure of the project.

2.3 Project group

You will work in groups of three or four for this assignment. This should be the same group as in Assignment 5. Make sure any personnel changes are approved by the course staff.

Schedule a check-in with your team, ideally as an in-person meeting. Reflect on what went well (and perhaps didn't go as well) during A4 and A5. Make a plan for when you will meet and how you will communicate during the busy last few weeks of the semester (particularly with the Thanksgiving holiday in the middle). Commit to how you will share code. And finally, do identify duties for each team member and specific tasks, but consider not splitting up the work entirely—pair programming and in-person group meetings can be extremely productive especially when making architectural and interface design decisions.

2.4 Getting Help

As always, the instructor and course staff are available to help with problems you run into. For help, read all Ed posts and ask questions that have not been addressed, attend office hours, or set up team meetings with any course staff member.

2.5 Release

The release files are available on CMSX. We have provided an updated `build.gradle` file that will allow your code to be used with JavaFX. We have also included an interface and class for a data structure that you will implement.

2.6 Restrictions

You will design and build your GUI from scratch. You may use any classes from the standard Java system library. If you would like to use any other third-party library, please create a private post on Ed and wait to receive confirmation from the course staff before using it. You may code the GUI in JavaFX's XML. You may use a GUI builder such as the JavaFX Scene Builder (from Oracle, or an open-source version from Gluon). You may also hand-code your GUI. You may not use any form of AI to generate your code.

You may not change the released AdjustablePriorityQueue interface.

3 Design document

This is a challenging assignment, and we want to steer you on the right path. We require that you submit an early draft of your design overview document before the assignment due date. As this is the final assignment, this preliminary design document will be weighted more than previous design documents. The Overview Document Specification outlines generic expectations. In particular, make sure to explain your plan for safely and efficiently integrating the user interface with the simulation, taking care to explain how you will ensure the user interface is responsive while not constraining simulation speed. We also expect to see design sketches for the GUI and explanations of the different user workflows. Your design and testing strategy might not be complete at that point, but we would still like to see your progress. Finally, please submit a detailed plan for you will implement your algorithm for `ping` as well as the required data structures. As always, you can go over your design document with the course staff and receive feedback in office hours and lab.

4 Requirements

4.1 User Interface

Your program should be able to display all aspects of the current state of the world. It should graphically render hexes and their contents, including food, rocks, and critters. It should be possible to distinguish critters of different species and to see the size and direction of each critter.

The GUI should allow the selection of world files, preferably using a FileChooser. After the user has selected a world file, the program should load the file and initialize the world as done in Assignment 5.

The critters will be controlled by critter programs using the interpreter and simulation engine you built in Assignment 5.

The GUI should allow the user to advance the simulation one step at a time or to let the simulation run continuously. It should be possible to pause and resume a continuously running simulation. The graphical display will be updated continuously as the simulation progresses to reflect the current state of the world.

The total number of time steps taken during the simulation and the total number of critters alive in the world should be displayed. You must also calculate and always display the current frames per second. While the simulation is running, you must also calculate and display the current real simulation steps per second. As in Assignment 5, the user should be able to create a new random world, load a world, or load a specified number of critters.

When loading a critter program file, the user should be able to either specify a number of critters to be randomly placed throughout the world or select a particular hex to place a critter.

The user should be able to control the speed of the simulation. It is up to you how you will allow the user to do so. At minimum, the user should be able to run the simulation as fast as computationally possible given the user's hardware. If the user changes the speed, there must be a noticeable difference for reasonably sized programs. Thus, a user should be able to see a simple world simulate very slowly so they can observe all the changes to the world, and a user should be able to simulate a complex world at around the same speed it would take to do in A5.

When the simulation is paused, the world must always be in a complete state, meaning that all critters in the current turn have completed their action. If you'd like, you are allowed to display intermediate world states where only some of the critters have taken their action *only while the simulation is running*. This may or may not be desired based on how you connect the simulation to the user interface, and this can be challenging to do safely.

Another part of the user interface will allow the user to inspect a single critter somewhere in the world. The user can click on the hex containing a critter to make it the currently displayed critter. The user interface will indicate which critter is currently displayed and will also display the state of the selected critter, corresponding to the 7 initial memory locations, along with the critter program and information about the most recently executed rule. As the simulation progresses, this information will be updated accordingly.

The particulars of the design are up to you. You should strive for an interface that is intuitive, user-friendly, and visually appealing.

4.2 Performance

Your simulation should be able run fairly fast; at least 10 steps per second for a small world. The graphical user interface should interfere minimally with simulation performance. We will evaluate your simulation on consistent hardware with consistent programs. Do not artificially cap the simulation engine to preserve UI responsiveness.

It is possible to speed up your program using concurrency. For example, rendering and simulation can be done in separate threads. In particular, you might think about running the simulation in a separate thread from the JavaFX application thread. However, it is very easy to have these threads interfere if they are accessing common state. You can prevent interference by adding locks, but locks may effectively eliminate useful concurrency. One way to avoid locking is to make read-only copies of data that needs to be shared across threads.

To connect the JavaFX application to a background thread, you may find `Platform.runLater()` useful. It runs some code in the JavaFX application thread, which recall is the only thread that is allowed to access the JavaFX scene graph.

4.3 Responsiveness

The simulation engine should not interfere with GUI responsiveness. Do not artifically limit the GUI framerate to preserve simulation speed. Buttons should perform their action immediately. It is okay if certain

actions, such as stopping the simulation, take their effect only after the current simulation step is is complete. Running the simulation should not make the user interface feel sluggish. Resizing the window, navigating around the world, and zooming should not lag.

First, consider a very small world that can do 1000 simulation steps a second, and assume it takes $\frac{1}{60}$ of a second to draw the world and critters. It is clearly not possible to draw all 1000 states of the world without falling behind.

Now, consider a massive world that takes 2 seconds to simulate one step. While this one time step is simulating, a perfect GUI application would allow the user to perform actions that change the screen, such as zooming and resizing the window. Recall that when the simulation is paused, the world must always be in a complete state, you may display intermediate world states while the simulation is running.

This is extremely challenging and requires careful design, but you should first make sure the rest of your application works well before tackling this.

4.4 A5 Compatibility

Do not delete any of the interfaces that were implemented in A5; otherwise, we will not be able to grade your solution effectively. The code to launch your simulation in the terminal should still be there.

4.5 Adjustable Priority Queue

Java's builtin PriorityQueue does not support efficient updates to the priority of elements already inserted. You will instead make your own priority queue.

Complete the class `BinaryHeap` in `src/main/java/a6` that implements the `AdjustablePriorityQueue` interface. As a hint, you will need an additional data structure (in addition to a binary heap) to be able to look up the index of any item in expected $O(1)$ time. Don't forget to keep this second data structure updated every time you modify the heap; checking the invariant that connects the data structures may save you a lot of debugging. A second hint is to you unit-test your implementation of this interface extensively. Trying to implement algorithms that rely on a buggy data structure is a recipe for disaster.

At the beginning of each method, you must provide a **brief justification** of the method's run-time complexity. A short comment will suffice. The methods in your class must adhere to the required time complexities provided in the interface.

4.6 Improved food sensing

In A5, you implemented smell. This doesn't take into account obstacles, turning, or energy costs. In A6, critters are given a new, more powerful ability, called `ping`. The Project Specification has been updated to include more details on `ping`, as well as examples. You must use Dijkstra's shortest-path algorithm, and you must use the data structure developed in Section 4.5. Duplicates are not allowed on the priority queue, and the time complexity must match that of the algorithm given in lecture.

To get started, begin by updating your tokenizer, parser, and AST to support this extension to the grammar. The tricky part is to use Dijkstra's algorithm correctly. Recall that Dijkstra's single-source shortest path algorithm finds best paths in a graph. In this case you might be tempted to make the graph nodes correspond to map hexes, but this is not the right graph. We can think of a `ping` algorithm as finding the least cost path from the current position and orientation of the critter to any of the final end states that will allow a critter to eat food in front of it. A critter state has two distinguishing components: first, the hex the critter is on, and second, the direction the critter is facing. The weight of a graph edge corresponds to the energy of the action the critter must take to get to the next node. Since different actions take different amounts of energy, the graph edges have different weights. Of course, some nodes should be excluded because of obstacles.

Describe your approach to implementing `ping` in your overview document.

5 GUI Design Choices

Good GUI design can be difficult. It is largely subjective, and there are no hard and fast rules for what makes a good design. That said, there are a few simple strategies you can follow that will enhance the experience for your users.

5.1 Buttons and Control

Users should not need to play a guessing game to find out what buttons do. They should be clearly and succinctly labeled to describe their function. In some cases, an icon can be better than words.

Placement of buttons depends on function and frequency of use. A small button way off on the side of the screen is difficult to access compared to a large button near the focus of the window, and can be annoying if the user needs to use it repeatedly. On the other hand, the close/resize buttons at the top right of windows in Windows and the top left on a Mac are typically used only infrequently. Placing them far away from the central area of the screen makes it unlikely to click them accidentally.

A GUI can provide *keyboard shortcuts*, so that the user doesn't need to move the cursor to initiate an action, or *context menus*, where a user can right-click to display a menu wherever the cursor happens to be, enabling actions that make sense at that particular location. *Tool tips* can be displayed when hovering with the mouse over a component to describe its function.

Consider using some of these features to provide an intuitive and manageable interface.

5.2 Color Schemes

As a general rule, use highly saturated colors sparingly. Saturated colors make sense in the (few) places where you want to draw the user's attention. Avoid having too many colors at once; monochromatic, adjacent, triad, or tetrad schemes work well. A useful site for picking color schemes is paletton.com, or for a more random approach coolors.co

5.3 Navigation

Scrolling and zooming must be implemented to make it easy to view and navigate large worlds. You should also be able to resize the window of your program so that it can be effectively used on screens different than your own.

6 Running your program

If you were to construct a JAR, it should be possible to run your program with the following command,

```
java -jar <your jar>
```

If provided the source code as specified by `critterworld.zip`, it should be possible to run your program with `gradle run`. Both approaches will start up your program in a default world populated by randomly placed rocks, initialize the GUI, and wait for user input. The simulation should not be running initially.

Note that there are no command-line options. All further user interaction should be done through the graphical user interface.

7 Written problems

7.1 Concurrency

Suppose you are sorting arrays of integers on a processor with many cores. You decide to use a **concurrent merge sort** in which different threads work on the different subarrays. Your first try at writing the code looks like this:

```
1  /** Effects: Place elements x[lo..hi) in sorted order.
2   * Requires: ...<1>...
3   */
4  static void sort(final int[] x, final int lo, final int hi, final int[] y) {
5      if (hi == lo + 1) return;
6      final int mid = (lo + hi) / 2;
7      final Barrier barrier = new Barrier();
8      final Thread t = new Thread(() -> {
9          sort(...<2>...);
10         synchronized (barrier) {
11             barrier.notifyAll();
12         }
13     });
14     t.start();
15     sort(...<3>...);
16     synchronized(barrier) { barrier.wait(); }
17     merge(x, lo, mid, hi, y);
18 }
19
20 class Barrier {} // really just an Object so far...
21
22 /** Effect: put the elements x[lo..hi) into sorted (ascending) order.
23  * Modifies: y
24  * Requires: x[lo..mid) and x[mid..hi) are both in sorted order,
25  *   and y is the same size as x.
26  */
27 void merge(int[] x, int lo, int mid, int hi, int[] y) { ... }
```

1. Fill in the three missing parts marked "...<*n*>...", including the Requires clause, to correctly implement the mergesort algorithm, while ignoring any synchronization issues. You can assume that `merge()` is already correctly implemented.
2. Suppose you use your algorithm to sort the entirety of the following array: `(8,0,7,6,1,2,5,4,3)`. Complete the following tree of results from calls to `merge()` showing how these calls arrive at the final sorted result. Put a star on each call that occurs in the original thread. (You do not need to show the contents of `y`).

```
1 (0,1,2,3,4,5,6,7,8) *
2 /                \
```

3. After filling in the missing recursive call arguments in the code, you discover that the `sort()` function often fails to return. Briefly explain the sequence of events that can cause this to happen.
4. Fix the problem identified in the previous part by adding methods and state to the class `Barrier`. Give the new definition of `Barrier` and indicate how to change the uses of `barrier` in the above code. (Hint: With the exception of `Barrier`, you should be able to make the above code look simpler. What is the **condition** the main thread is waiting for?)

7.2 Critter Programs

Writing critter programs and simulating a world is an excellent way to test your entire final project end-to-end. The language you've implemented and simulation you've built are actually really powerful! Let's take advantage of some of the features to do something interesting. Hopefully, this also helps with some of your testing.

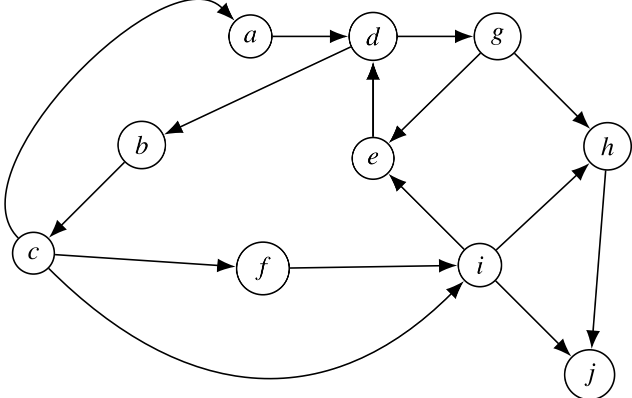


Figure 1: The graph for the written problems

5. **Forager.** Write a forager critter program that finds and eats the food that it can reach using the minimum energy possible. if the energy required to reach the food that costs the least energy exceeds the energy the critter currently has, the critter should wait. the critter should always take the path that will cost the least amount of energy, not the fewest number of actions.

   be sure to test your program works even when the food on the world changes; at every point in time, the critter should be moving in the most energy-efficient way. include sufficient comments in your program or accompanying your program explaining the algorithm you developed.

6. **Prolific.** Write a critter that, starting from a single critter, reproduces as quickly as possible. It may do other things that help it reproduce, like gather food.

7.3 Graphs

Use the graph in Figure 1 in these problems:

7. Starting from vertex a, give the sequence of vertices visited when doing a breadth-first traversal, assuming children are visited in alphabetical order.

8. Do the same for a depth-first traversal.

9. Recall edge classification. Based on the traversal starting from a, label each edge in the graph as a tree edge, forward edge, back edge, or cross edge.

8 Overview of tasks

Determine with your group how to break up the work involved in this assignment. Here is a list of the major tasks involved, in no particular order. Note that this is not exhaustive, and it's likely that some tasks are significantly more challenging than others.

- Fix problems identified in A4 and A5.
- Implement a GUI to display the state of the critter world.
- Connect the simulation engine from Assignment 5 to the display. This means allowing the display to update as the simulation progresses and to start, stop, and step the simulation in a performant manner.
- Make the GUI front end highly responsive to user input and implement the loading of critter and world files and the placement of critters into an existing world.

- Implement the `DynamicPriorityQueue` interface.
- Extend your tokenizer, parser, and AST to support `ping`, then implement `ping` specified above using Dijkstra's shortest-path algorithm so that critters have a new sense to find food.
- Solve the written problems.
- Test!

9 Team evaluation survey

Like previous assignments, we will also ask you to complete a short team evaluation survey—i.e., for your own contributions and performance as well as that of your teammates. This survey will be scored and will contribute to your overall A6 grade. The evaluation will be submitted separately as an online web form that will be provided in the final week of class.

10 Version control

As with the last assignment, you must submit a file `log.txt` containing the commit history of your group. Additionally, you must submit a file showing differences for changes you have made to files you submitted in Assignment 5. You can get the differences by using `git diff` against the appropriate commit hash for your A5 submission.

11 Karma

Karma questions do not affect your raw score for any assignment. They are given as interesting problems that present a challenge.

- Apply your learnings from lab and use LaTeX for your submission.
- Write a critter program that makes the critter walk backward, serving food onto successive hexes it leaves behind, in amounts that form the sequence of prime numbers: 2, 3, 5, 7, 11, ...
- Add animations for actions. Some ideas could be animations for critters being born, becoming food, and eating.
- Add a user interface that allows a user to kill the selected critter. It's wise to restrict user edits to the state to only when the simulation is paused, lest you risk nasty concurrency bugs.
- Add a user interface that allows a user to control the selected critter and decide what moves it performs on each turn. A good way to start this is by storyboarding the user interface with sketches that show how the different components will be placed on the screen. Remember, not all components must be visible at all times.

12 Tips and Tricks

This is your final submission of the project, you want to make the project as a whole work well. This is a significant step up from all previous assignments, and it's truly the culmination of all your hard work and impressive achievementn

We suggest aiming to get the GUI functionality working in advance of the due date so you have a few days to polish the project.

Take care not to entangle your world model with this new user interface. Proper separation of the simulation and GUI is important and something we will be looking for. Think carefully about how you will connect your user interface with your existing simulation framework; there are many approaches, and it's worth trying a few out. The model should not depend on the user interface in any way. As usual, we will be looking for good documentation of your classes and their methods.

Review your lecture notes!

GUI code can become quite long, and you will likely have to make a conscious effort to keep it clean and readable, more so than with previous assignments. In addition to organizing your classes in packages, think about how to organize your resources (fxml files, images, icons). Never access resources using absolute pathnames; they should load properly even if the project's location changes. Instead, use one of the methods in ClassLoader or Class that look for resources in your program's classpath.

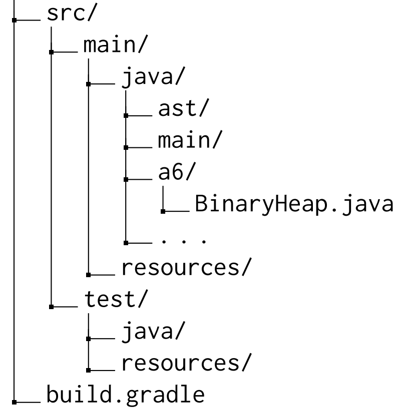
If you are having issues with FXML files, make sure they are located within `src/main/resources` under the same package as your main class. If, for example, your main class resides within a separate package `view`, then you must put your resources within `src/main/resources/view`. You should then be able to access the FXML files using `getClass().getResource()`.

13 Submission

You should submit these items on CMSX:

- `overview.pdf`: Your final design overview for the assignment. It should also include descriptions of any extensions you implemented. Additionally, you should document the different aspects of your GUI. Do not assume that all observable features of your GUI are noticeable to an unfamiliar user. You should also indicate the operating system and the version of Java you use.
- `critterworld.zip`: Your final `critterworld` directory for your project. This is slightly different from previous assignments and more alike to submitting your repository, so please pay special attention. This directory should contain:
  - **Source code**: You should include all source code required to compile and run the project. All source code should reside in `src/main/java` with an appropriate package structure.
  - **Resources**: All resources for your GUI should be under `src/main/resources`.
  - **Tests**: You should include code for all your test cases in `src/test/java` and resources for your tests in `src/test/resources`. You are welcome to create subpackages to keep your tests organized.
  - **Gradle**: As mentioned, you are allowed to include external dependencies for this project. (If you do, please clear it with the course staff in advance). You should submit your `build.gradle` file containing the correct main class name and any dependencies you require. Make sure your `build.gradle` includes everything the original released file contained along with any additional dependencies.

  An example directory structure might look like this:

  We should be able to unzip your `critterworld.zip` into a `critterworld` directory and successfully import it into IntelliJ from an external Gradle model. The provided `build.gradle` should enable us to run your GUI with `gradle run`.
  Do not include unnecessary build files or directories, such as `gradlew.bat` or `gradlew`, `.idea`, or `.gradle` directories. Do not include any hidden files like `.DS_Store` or `__MACOSX`, or compiler outputs such as files ending in `.class` or `.jar` or the `META-INF` folder.
- `screenshots.pdf`: A `.pdf` file containing 3–5 pages of screenshots showcasing your GUI.

- `log.txt`: A dump of your commit log from your version control system.
- `a6.diff`: A text `.diff` file showing the changes to files carried over from Assignments 4 and 5 and used in this assignment. This can be obtained from the version-control system.
- `concurrent_merge_sort.pdf`: Your response to the Concurrent Merge Sort written problem.
- `forager.txt`: An intelligent critter program that finds and eats food using the minimum energy.
- `prolific.txt`: An critter program that reproduces as quickly as possible.
- `graphs.pdf`: Your solution to the graph written problems.