CS 2112 Fall 2025

Assignment 5

Graphical User Interface Design

Due: Thursday, November 20, 11:59PM

Design document due: Tuesday, November 11, 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 3 and 4**.

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

1 Changes

- Clarified that most of the Ring Buffer class needs to be made thread safe, not just put and take.

2 Instructions

2.1 Grading

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 part of the 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 two to three for this assignment. This should be the same group as in Assignment 4. 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 A3 and A4. Make a plan for when you will meet and how you will

communicate. And finally, 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. You can either copy paste the file and replace your old build.gradle, or you can just make the following changes below:

- Underneath the **plugin** block (right under `id 'application'`), add the following:

```
id 'org.openjfx.javafxplugin' version '0.0.14'
```

- Inside the **application** block, change `mainClass` so that it is the same as the directory for whatever your GUI main class ends up being
- Add a new block called **javafx** to allow for your project to use JavaFX

```
javafx {
    version = "21"
    modules = ['javafx.controls', 'javafx.web', 'javafx.fxml']
}
```

Regardless of the method you use, make sure that you change your new `build.gradle` to specify where your main class that handles the GUI is in. The place where you need to make this change is indicated by a comment.

IMPORTANT NOTE: If you decided to copy paste the build.gradle, make sure that you copy pasted the one in the top-level directory, not the one inside of the `ring_buffer` folder

If you are having trouble getting JavaFX set up, check the Tips and Tricks section of this document, ask on the Ed discussion board, or go to office hours.

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 Gluon. You may also hand-code your GUI. You may not use any form of AI to generate your code.

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. 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. As always, you can go over your design document with the course staff and receive feedback in office hours and lab.

4 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 named `a5.diff` showing differences for changes you have made to files you submitted in Assignment 4. As with Assignment 3, we highly recommend you create a tag in your GitHub repo with the state of your final A4 submission, so that you can easily reference it in the future. If you need a refresher on how to do this, from the right hand panel of your GitHub repo, choose the "Create a new release" link. Then, give the release a title, and click the dropdown above that says "Choose a tag". Give your tag a memorable name like a4 and choose "Publish release". This will mark this location in time on your repo with the tag name for easy reference. Now, you can just use `git diff a4 main > a5.diff` to make `a5.diff`.

5 Requirements

5.1 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 4. The critters will be controlled by critter programs using the interpreter and simulation engine you built in Assignment 4.

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 4, 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 A4.

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.

5.2 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.

5.3 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.

5.4 A4 Compatibility

Do not delete any of the interfaces that were implemented in A4; otherwise, we will not be able to grade your solution effectively. The code to launch your simulation in the terminal should still be there.

6 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.

6.1 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. *Tooltips* 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.

6.2 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. For a more random approach you can visit coolors.co.

6.3 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.

7 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.

There is one required command-line option. If the flag `--disable-mutation` is provided, critters should reproduce without any mutation. All further user interaction should be done through the graphical user interface.

8 Testing

Testing GUIs is very different from the way you've been testing your assignments so far, since you can't just write a test suite for it. You have to actually use your GUI and interact with it over and over again in order to find issues. Below are some recommendations and ideas on how to test your GUI:

- Make sure that all the visual cues that we require actually work. For example, critters facing different directions in the model should be visually facing different directions in the GUI.
- Compare the GUI representation of the world to the commandline output that you made for A4
- Test that clicking on a critter produces information for the right critter
- Test different speeds of your simulation and make sure nothing breaks

- Make sure that when you pause your simulation, your GUI does not render an intermediate state of the world
- Run your program repeatedly and try different types of inputs to ensure that you don't have any issues with your implementation of concurrency
- Have your friends and peers play around with the GUI. Since you were the designers of the GUI itself, you know where everything is and you designed it in a way that is intuitive to you. Testing with real users will help you gauge the actual usability of your GUI and help you find bugs since users will use your GUI in ways that you don't expect.

9 Written problems

9.1 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?)

9.2 Critter Program

5. **Prolific.** Write a critter program that, starting from a single critter, reproduces as quickly as possible, resulting in the largest possible world population after some number of steps. Critters may do other things that help it reproduce, like gather food.

10 Coding Problem

10.1 Ring Buffer

A **ring buffer** is a fixed-size FIFO queue implemented using an array and two integer indices called the **head** and **tail**, along with some number of locks or condition variables.

This data structure is commonly used in distributed systems with limited memory. For insertion, the item inserted at the tail end of the array, and the tail index is incremented. If there is insufficient space in the array, as determined by the two indices, then the insertion fails. For removal, the item at the head is returned, and the head index is incremented.

If a **producer** thread tries to insert an item into the queue but the array is full, the thread should block until there is space. Conversely, a **consumer** thread that tries to remove an item from an empty queue should block until a producer thread pushes a new item onto the queue.

If you are opening this project in IntelliJ with the a5release folder, please make sure that you select only the ring buffer folder as the project folder, not any of the top-level folders. If you don't do this, the top-level `build.gradle` for your project will conflict with the ring buffer's `build.gradle`.

10.2 Implementation

We have provided you with a partially broken ring buffer that is not thread-safe. You must update the class to be correct and thread-safe.

Any implementation of concurrent code is likely to be wrong unless all project members have examined the implementation carefully together. While it might be tempting to delegate the job of implementing the ring buffer to one person, we recommend that all project members work together closely on this implementation.

10.3 Testing

Testing concurrent code is challenging because it is **nondeterministic**, meaning that events do not always occur in the same order. The observed behavior depends on thread-scheduling choices that are made by the runtime system and are out of the control of the application programmer. To test the ring buffer, develop a test harness that causes several threads to issue method calls concurrently.

One effective trick for catching concurrency bugs is to inject random delays throughout your code, conditioned on a boolean constant flag so that they can be turned off when not debugging. Random delays will cause your program to explore a wider variety of thread schedules. Generous use of assertions is also highly recommended as a way to catch race conditions and other bugs.

Along with the broken RingBuffer, the release code contains a suite of test cases to test its concurrent and non-concurrent functionality. It is recommended, although not required, that you add your own test cases to practice testing concurrent programs.

11 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 A3 and A4.
- Implement a GUI to display the state of the critter world.
- Connect the simulation engine from Assignment 4 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.
- Solve the written problems.
- Solve the ring buffer coding problem.
- Test!

12 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 A5 grade. The evaluation will be submitted separately as an online web.

13 HARMA

HARMA 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.

14 Tips and Tricks

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. **Don't forget about model–view–controller, as maintaining good MVC separation will be essential for success in A6; A6 will require you to separate out the backend and frontend behind a server/client interface**. 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.

We would highly recommend you use the Canvas node in your GUI for drawing the state of the world.

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()`.

15 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 `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:

```
src/
  main/
    java/
      ast/
      main/
      a5/
      ...
    resources/
  test/
    java/
    resources/
build.gradle
```

  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.
- `a5.diff`: A text `.diff` file showing the changes to files carried over from Assignments 3 and 4 and used in this assignment. This can be obtained from the version-control system.
- `prolific.txt`: A critter program that reproduces as quickly as possible.
- `concurrent_merge_sort.pdf`: Your response to the Concurrent Merge Sort written problem.
- `ring_buffer.zip`: A `zip` file with the `src` folder containing your solution and the `build.gradle`. You may also include the testing folder inside of the `src` folder