Lab 9: Snake Game: Part 2 – Full Game

Goals

To carry forward with the snake game we have been working on, we are going to make a final push toward the end goal. You will notice that the due date is nearly the end of the term, but this does not mean that you can sit around and relax until the last day. This project will take quite a bit of programming to get done. I strongly recommend talking to TAs, tutors, and myself, to hash out any problems that you might have, and to do so early on.

The overall goals of this project are:

- Create a complete working snake game
- Write a larger program that actually does something
- Put the knowledge of many small individual pieces together into a larger system
- Find out how inheritance can really help you, and why it's so neat
- Get some experience writing a GUI program that utilizes mouse and keyboard input
- Master the process of breaking a large problem down into smaller pieces. You do *not* want to write this entire program in a single main method.

Program Description

I've described a basic snake game, but I encourage you to get creative with game design. Go ahead and make a conga line of dancers instead of a snake, just make sure the basic game mechanic stays the same.

Command Line Arguments and Level Configuration File

When the program is started, it should take an optional command line argument giving the name of a level configuration file specifying the size of the game area and location of the walls.

If a file is specified, the program should load the initial configuration for the game from this file.

If no command line argument is given, initialize the game with some default map, which could simply be hardcoded or could be read from a configuration file that is included in the jar.

If your program takes additional command line arguments, document them in your readme file.

If your program can handle additional information in the level configuration files beyond the basic format described in part 1, document it in your readme file.

Moving the Snake

The snake will move automatically in response to a Timer. (You decide how fast the Timer should tick. Make sure it is slow enough to be able to react and play, but not so slow that the game takes forever.)

To make things easier on the graders, all games must use the same basic keyboard controls. The **up**, **down**, **left**, and **right** arrow keys will change the snake's direction to up, down, left, and right, respectively. The snake should keep moving in its current direction if the arrow key is released.

All keyboard controls should be documented in the readme file.

Food and Growth

At least one food item should be placed at a random unoccupied location. If the snake collides with food, the snake eats the food and grows in length. A new food item is placed in at some random unoccupied location.

A snake has a maximum length. When a snake that is at maximum length moves, the tail end of the snake will always be a constant length away from the head. If a snake at less than maximum length moves, the head will move, but the tail end will stay put until the snake has grown to its maximum length. A snake eating food increases the maximum length of the snake by some amount (you can decide this). You can decide how to initialize the snake: starting out as just a head and then growing until it has reached its initial maximum length.

Wall and Snake Collisions

If a snake collides with a wall or its own body, recognize this in the game. This could kill the snake and end the game, or perhaps it causes the snake to lose a life and restart the level. Whatever you choose, document it in your readme file.

Edge of Map

It is your choice whether the world wraps around from one edge to the opposite or if there is effectively a wall just off the edge of the screen that will kill the snake. Whichever you choose, document it in your readme file so we can verify the behavior.

GUI Layout

You have a lot of freedom in designing this GUI, but I require at least the following elements.

- A panel where the "gameboard" is drawn.
- Label(s) for current score information.
- A start button that toggles its text between "Start" and "Pause" as it pauses and unpauses the game. "Start" (i.e., game is paused) should be the default state when the GUI is started.

As in the layout demo lab, I expect you to use Java Swing libraries for your GUI. Also, you should be starting the GUI on the event dispatch thread using `SwingUtilities.invokeLater`, as discussed in class.

Game Play and Scoring

- Game begins when user presses the start/pause button. Text of the button changes to pause. If pause button is pressed, pause the game. When the game is paused, the snake should not keep moving, the gameboard should not respond to arrow key presses, etc. When the game is unpaused, it should resume from where it was paused, not restart the game completely.
- Score increases as the snake eats. This can be as simple as a function of how many pieces of food have been eaten, but feel free to display additional statistics. (Current length of snake, different points for diffent food, level up after some number of pieces, etc.)
- When the game is over, somehow let the user know. The game ends when the snake has collided with the wall or itself and run out of lives (could just be one life). You might also choose to end the game when the user scores a certain number of points or after a certain amount of time has passed.

Extras

Adding extra features can make your game more fun. Just make sure you have the basic game functionality first. Here are a few ideas to get you started.

- Custom background image instead of plain color.
- Nice snake and wall art, rather than just a bunch of rectangles like mine.
- Sound effects and/or background music.

- Extra animation above and beyond
- Fancy game over notification.
- Increase difficulty as game progresses. Speed up snake movement? Add more walls? Require ever more points to get to the next level?
- Add other types of food with additional points/powers.
- Other obstacles? Moving walls?
- Save high scores to a file.
- Game options (Select map or difficulty level from menu? 2 play mode? Computer opponent?)
- Really nice maps. (It's a bit fussy to edit those level files by hand.)
- Easter eggs. (Really important to document those if you want us to find them!)

Suggestions and Hints

Rather than telling you exactly what to do, I will provide a number of hints that you will hopefully benefit from.

- Split things up into smaller pieces! My sample solution that you have seen in class consists of roughly 700 lines of code. Among these lines of code I have at least 15 classes defined. Some are nested, some are anonymous, and some are higher level classes. Again, what I'm trying to say, this is not a problem that you can just write in a single method.
- Build off of previous code. Obviously, I expect you to use the GameManager you wrote for part one, possibly adding additional functionality as you need it. You should be able to use a lot of the work you did for the GUI layout practice to at least get you going on the GUI for the game.
- How to approach the problem…One of the harder things to do when implementing a game like this is to figure out where to start, and what to do first. The first thing you need to understand is that the game isn't going to write itself, and it's not going to be completely done the first time you sit down to write code for it. So the trick is to work on pieces that you can finish and test individually first, then putting them together into a usable system. For example – when writing the snake game, start by just drawing the objects on the screen to make sure they will show up, and then once that is working, figure out how to move the snake.

- Then what…Well, you hopefully know what you want your game to do, how it will work, and what is going to happen as results of something that you do. How many points should be added when the snake eats food, how much should it grow, etc…I encourage you to sit down and think out your own set of rules and policies for the game. I do not want to impose any specific standard in terms of this for your implementation, but if you have a good idea of what you want your program to do, it's easier coming up with the design for that program on your own. I encourage you to come talk to your TA or to me about your design before you start writing a lot of code. I also encourage you to talk about design decisions on the discussion board for the course. Sometimes, it helps venting ideas. Feel free to brainstorm high level ideas (not sharing code!) with your classmates, both in person and on the Learn discussion board.

  If you come up with a design that you think is reasonable, you will likely do well on this assignment as well – that being said, *please* don't hesitate to ask for help, and to pose questions in class. It will benefit everyone.

- What do we need in order for this game to work? Partially it's up to you, but I can list a few things that I used and that you may feel are useful to you as well.

  – Instance variables - I have quite a few in order to keep track of the state of the game. Examples are the game manager, scores, paused state, and such. These typically need to be initialized at the beginning of a game.

  – Private helper methods - I have lots of them, these are methods that do small tasks that you may be performing often, but you don't want to write the code for them over and over again. If you find yourself copying and pasting a lot of code, you should probably be thinking – "Hmmm, I should probably make a method for that!", and then figure out what the method is going to look like, and what parameters it needs, etc.

  – Timer. I use a timer to keep track of how fast the game progresses, but…not every object responds to every tick of the timer. You'll have to decide what class(es) should pay attention to the Timer event.

  – Keyboard Listener – Probably one of the more important things for this game, since that is how the user controls the game. Remember that methods called from listeners should usually not be computation heavy as it may slow down the response time to the next event.

    Only the component with *focus* will be able to listen for keyboard events. When you press a button, generally the button retains focus. If you want some other component to get the focus (which you likely will, since I really don't suggest putting your main key listener inside the start button), you can use the `requestFocusInWindow` method to do so. So, if you wanted a component named `myPanel` to listen for key events, you'd use `myPanel.requestFocusInWindow();` somewhere in your code. Bear in mind that when you click on another component (such a button) that component will gain the keyboard focus.

Turning in your assignment

For this project, I want all the code and resources to be packaged into a self-contained jar file. I **also** expect you to turn in a readme document describing your project (not in the jar file). These are the only two files you will submit.

Jar File

Create a jar file with all the necessary files that you used for your assignment. The jar file must of course include your source files, as well as code from all packages that you used. I.e., the jar should be self-contained and you should be able to run the game completely from the jar.

To make a jar file with a entry point of the `Nibbles` class:

(a) Compile all your classes. (`javac *.java` will compile them if all your source files are in the current directory.)

(b) Use the `jar` command to create a jar with all your files. (This should include both your `.java` source files and your compiled `.class` files.) Use the `e` option to specify the entry point.
```
jar cvfe JarFileName.jar EntryPointName <List of files and directorys to include>
```
So, if all your source files and class files are in the current directory, you can use:
```
jar cvfe Nibbles.jar Nibbles *
```
If you are using any level configuration files, images, sounds, or other files like that, make sure you include them in your jar. You want the jar to contain all the files your program needs to run in the single jar.

(c) Make sure your program runs from the jar file. Use the `-jar` option with java.
```
java -jar Nibbles.jar
```
To properly test this, you should move your jar file to a new location and try running it there to make sure you are not accidentally running from the files you used to make it instead of the jar itself.

README file

You have enough freedom with this project that we'll need some documentation. Submit a readme file that explains how to use your program and any special features we should be aware of.

You have enough freedom with this project that we'll need some documentation. Submit a readme file that explains how to use your program and any special features we should be aware of.

At the very least, your readme should include:

- Game play
  - How to play the game
  - How is the game scored.
- Description of program internals
  - Description of classes. (Where are game logic, data structures, etc.?)
  - Algorithm details, such as:
    * Moving and growing the snake.
    * Detecting collisions
    * Detecting end of game
- Any extras. It is especially important to point out the clever things you do so the grader will know to look for them while testing your program.
- Known bugs and feature requests – I know that no matter how long you have work on this assignment, there will be some bug that you can't quite fix or some feature that you won't quite have time to implement. Tell us about them. What would be your next step?

Submit to Learn

Submit your jar file and readme document to UNM Learn. Make sure that your jar file includes all of your source code!