Project 2 - Feeding Frenzy

Initial Submission Due: February 01, 2009

Final Submission Due: February 16, 2009

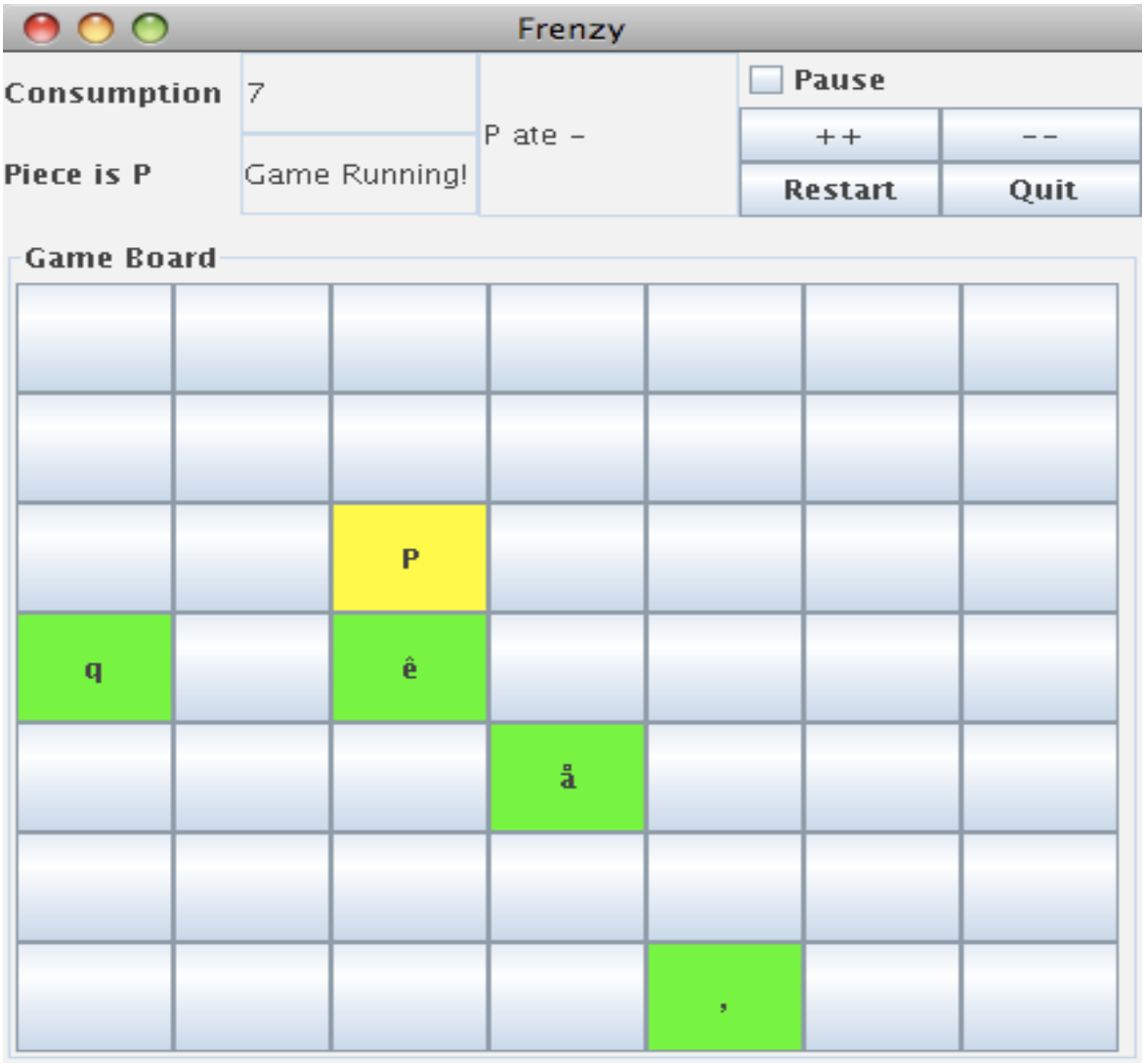
LATE Final Submission Due: February 17, 2009

$Date: 2009/01/26 17:19:29 $

Frenzy is a simple video game played on a grid. Each grid location may contain the user's player piece or an enemy piece. The goal of the game is to eat as many enemy pieces as possible (by landing on enemy grid locations) before being eaten. During a game, the player and enemy pieces race around the board consuming one another.

The user controls their player piece using the arrow keys, and each enemy piece runs as a separate thread controlled by the computer. Each enemy piece moves at the same speed, and each move is one location at a time. The computer creates enemies at regular time intervals, and the user can create an enemy piece by clicking on any grid location.

Here is what the game board looks like (on MacOS):



Objectives

1. Design and develop the design of a more complex system
2. Build a graphical user interface
3. Develop a design based on the Model-View-Controller (MVC) design pattern
4. Create a multi-threaded application
5. Build and play a GUI-based board game

Grading

The grading for this project is as follows:

- **Initial Submission:** 30%
- **Final Submission:** 70%
- Extra Credit: up to 5%; See later in this document.

NOTE: There are no *try* tests for this project. Instructors and graders will run your program to test it. Be sure to test your solution before submitting; that includes command-line-related testing.

Correctness and completeness of the game implementation, as well as your class design and coding style (including proper Javadoc and version control use) will affect your grade for the project.

Programs that do not compile will not be accepted by try. Programs that do not run will receive a grade of zero.

You may not use any GUI builder tools for this project.

Requirements

The game begins with the user's player piece at a random location in a square game grid, and the user controls player piece movement using the arrow keys. The player piece is yellow, with the letter 'P' as a label (see the picture in this document). When the program starts the first time, there are no enemy pieces on the board; *a game restart will have pre-existing enemies on the board*. As shown in the picture, the game window contains the following:

1. The game grid, containing the player and enemy pieces;
2. A Consumption field (showing the number of enemies eaten);
3. An indication of the Player piece ("Piece is P");
4. The game status ("Game Running!" or "GAME IS OVER!" if the player has been eaten);
5. Indication of last "eat" event (e.g. "P ate -" above);
6. ++ and -- buttons, which increase/decrease the speed of all enemy piece movements;
7. A Restart button, which restarts the game, keeping the current state of the enemy pieces;
8. A Quit button, which ends the game, closes the game window and ends the program;
9. A Pause check box; when checked, the checkbox stops movement of ALL pieces and stops enemy pieces from being created.

The green enemy pieces are labeled with a randomly chosen, *printable* ASCII character. When an enemy is created, it chooses one of the four diagonal directions at random, and then repeatedly moves using a step pattern in that direction (e.g. choosing to move top-left, then repeatedly moving one cell to the right, and then one cell up). Both the enemy and player pieces may wrap around the board. For example, moving up past the top of the grid will put the player piece in the same column at the bottom row of the grid.

By default, all enemy pieces move one location every 800ms, and the ++ and -- buttons increase or decrease the frequency of movement. The ++ button makes all enemies move faster, and the -- button makes all enemies move slower.

Enemies are automatically created every 6 seconds unless the "Pause" box is checked. The player may also add a new enemy by clicking on any board location on the grid.

The player and enemy pieces consume other pieces by moving onto a grid location occupied by another piece. When one piece (the eater) moves to a cell already occupied by another piece (the to-be-eaten), the eater consumes the to-be-eaten, which dies and is no longer part of the game.

At the start of a game, the player's "Consumption" number is zero, and increases by one for every piece eaten. Whenever a piece is eaten by the player or an enemy, the "eat" message (see the picture in this document) is updated to show which piece was eaten, and by which other piece.

The game status message will display either "Game Running!" if the game is active, or "GAME IS OVER!" if the game is over. When the game is over, existing enemy pieces stop moving, and the timer no longer continues to create new enemy pieces.

If a **Restart** occurs, the game recreates the player, resets player's consumption to zero, changes the game status message to display "Game Running!", disables the Restart button, allows the existing enemy pieces to resume moving and eating, and resumes the timer that creates new enemy pieces.

Whenever the player dies, the Restart button becomes enabled so that the user can restart the game. When the game restarts, the Restart button becomes disabled.

Specification

Command line arguments

`Frenzy.java` must contain the `main()` method that starts the program. A user shall start the game as follows:

```
java Frenzy [N]
```

The value **N** represents the dimensions of the square board. The [] characters surrounding this command line argument mean that the command line argument is optional. The default board size shall be 15 by 15, the minimum board size is 7 by 7, and the maximum is 30 by 30.

If a user attempts to build a board smaller than 7 by 7 or a board larger than 30 by 30, the program shall issue a usage message to standard error, stating **'Usage: board size N must be between 7 and 30'**, and then gracefully terminate.

Game Board

The game board shall be implemented using a grid of JButtons. Unoccupied grid locations use the default background color for a button. Player and enemy pieces are represented on the board as buttons whose appearance is described above.

Enemy pieces, Threads, and Java GUI Updating

Each enemy piece shall run in a separate thread. The thread must terminate when an enemy is eaten.

When the game ends, the movement of each living enemy piece must be suspended; the pieces remain in existence but cannot move. Some people say the pieces 'freeze'. These pieces will begin moving again after a Restart occurs.

Another problem to resolve is race conditions between different programmer-defined threads. Since multiple pieces compete to move to cells on the board, either or both the piece(s) and the board need synchronization to ensure single-threaded access and change. This is especially important when one piece (the eater) moves to a cell already occupied by another piece (the to-be-eaten).

Design: Model-View-Controller (MVC)

In class you will learn about the model-view-controller GUI architecture. In this project, you will make use of a simple MVC architecture for your program by:

1. Representing the state of the game using objects in a *model*.

   A model is a group of one or more classes that represent the board state, the player's score, the rate at which enemy pieces are created, etc. *Model objects do NOT process events directly or implement the appearance of game elements; model objects simply maintain game state*.

   A single instance of a game model class might have several different forms of display (view) and interaction (control) suitable for a variety of GUIs. While the state representation remains the same, the appearance and mode of interaction may change in various ways. As an extreme example, the display and control could change from using basic text input/output in a terminal to a full-blown GUI interface.

2. Representing the *view* (appearance) and *controller* (user input collection) parts of your program as one or more classes that are **separate from the model**.

   Although separate, the *view* and *controller* interact with the model. For example, the controller part of your program will read arrow key presses from the user, and then invoke the appropriate operations on a game model object to move the player piece; these operations change the state of the game by changing the state of the model. The model object must then notify the view part of your program to inform the view that the game state has changed. The view will then change button attributes on the grid, display scores, etcetera, as appropriate for the new state of the game.

The "controller" part of your program will include a number of event listeners (ActionListener for button and timer events, WindowListener, etc.). Think carefully about what events you will need to handle in your program, and how you wish to design your classes to support this. You will also need to think about how to represent the view for your program, and how the view connects with the underlying model.

See the Liang text Chapter 35 for a more detailed discussion of MVC and some relevant examples.

Game Options for Extra Credit

You may receive up to a 5% bonus for doing one or more of the following in the *final submission*:

- Implement a more visually exciting display for the game. For example, develop and use multiple icons for the self-propelled enemy pieces.
- Design and implement a spiral move algorithm for self-propelled pieces in place of the stair-step move algorithm.
- Design and implement some other form of motion for pieces (see your instructor for prior approval).
- Implement a scrolling list output of the sequence of who ate whom in the middle part of the top panel.

Submission

Initial Submission

Requirements for the initial submission include:

- The program checks the command-line argument for the board size and produces appropriate messages if it is out of range.
- If the size is valid, the program creates a board of the appropriate size.
- The game board appearance is correct and composed of a grid of operable buttons in a *resizable*, center

region of the game's window, and with the game controls at the top of the game window. Closing the window terminates the program correctly and completely.
- Clicking on any board square creates a new enemy piece at that location.
- The program implements a model-view-controller architecture as required. The try submission includes the file README-min, which summarizes the as-built design.
- The user moves the player's piece using the arrow keys.
- Player piece movement 'wraps' around columns and rows of the grid.
- The Quit button is operational.

To repeat: You may not use any GUI builder tools for this project. If you use a GUI builder, you will receive a 0 grade.

Submit your **initial** solution to:

```
try grd-232 project2-1 Frenzy.java README-min [other java files]
```

Final Submission

In the final submission, all requirements must be implemented as described in the Requirements and Specifications sections above. In particular:

- Enemy pieces now move at the same speed, and each is controlled by its own thread.
- The player's piece can eat enemy pieces, and the score updates appropriately.
- The enemy pieces now eat each other or the player piece, and the GUI displays who ate whom.
- Enemy piece movement 'wraps' around columns and rows of the grid.
- The Restart button is operational. When the player dies, the button becomes enabled so that the user can restart the game. When the game resumes running, the button becomes disabled.
- The remaining game buttons (++, --, Pause) operate according to the game requirements and specifications.
- Report the game status ("Game Running!" or "GAME IS OVER!")
- The submission includes a filled-in version of the file README-final , which describes the final MVC design, ideas for design improvements, and a summary of bonus items.

Submit your **final** solution to:

```
try grd-232 project2-2 Frenzy.java README-final [other java files]
```

After you submit, you may check your submission archived by ***try*** to be sure you have submitted all the correct versions of your files, using the try query command like this:

```
try -q grd-232 project2-2
```

Caution

For each submission, only the last submission will be seen and graded. Be sure it's the correct version of your game.

Do not e-mail any files to your instructor. Submit your project ONLY through *try*.