Project 2 - Feeding Frenzy

Initial Submission Due: Sunday, Feb 10, 2008

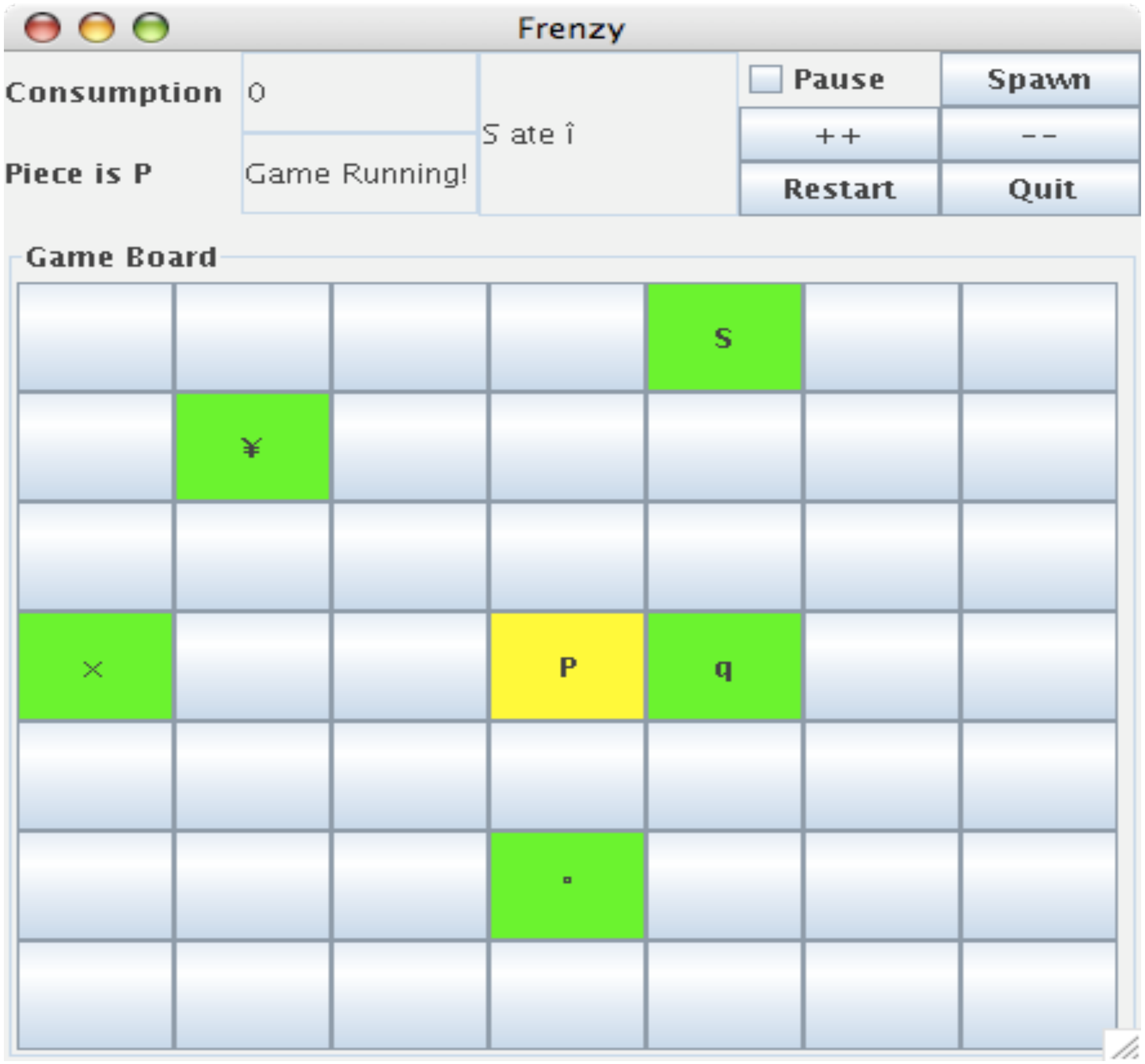
Final Submission Due: Friday, Feb 22, 2008

$Id: writeup.html,v 1.8 2008/01/23 15:36:34 vcss232 Exp $

Overview

Frenzy is a single player board game in which the player must eat other, randomly-moving pieces or be eaten by those pieces. As the player eats other pieces, it gains points based on how many other pieces the player has consumed. The game is over when another piece eats the player.

The human user controls its piece with the arrow keys to move one cell up, down, left or right. The user spawns (creates) the other pieces, and individual threads control these randomly-moving pieces. An optional timer creates and starts new moving pieces at regular intervals. Here is a picture of the program's window.



Objectives

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

Grading

The weights for this project are as follows:

- Initial Submission One: 30%
- Final Submission Two: 70%
- Extra Credit: up to 10%.
  See the Game Options for EXTRA CREDIT section for details.

Note: You must document your design and results in two files: Readme-1.txt and Readme-2.txt. These files will provide design, implementation and status information that belongs with each submission. See the Completing the Readme-1.txt and Readme-2.txt Files section later in this document for details on the Readme files to complete.

Your instructor *may* ask you to work on this project in teams of 2 people. Ask your lecture instructor for details.

A nonworking submission will replace correct work you may have already submitted. That might leave nothing in your submission archive for your instructor to grade!

Since your game runs on a GUI, *try* will not perform any testing for you. Be sure to test your solution before submitting.

Be sure you tend to details -- Follow the coding standards, include the proper class and method documentation, declare constants, properly use version control, follow the naming conventions, keep line length within bounds, use consistent indentation, and so forth. *Be sure you add an `@author` tag in each source code file with your name and CS email address.*

Functionality, design and style are components of your grade. Naturally, your implementation must accurately reflect your design; your design must be reasonable and meet all the indicated requirements.

A non-working project will have no value. Projects submitted late will not be considered for credit. Remember that in addition to correctness, coding standards, style, algorithms and the use of version control count!

Getting Help

You may get help from your instructor(s) and the teaching assistants. Anything else is not allowed and is subject to the penalties listed in the DCS Policy on Academic Dishonesty. This includes but is not limited to:

- Obtaining detailed help from any other people
- Providing detailed help to other people
- Sharing source code with anyone, by any means or medium.

We certainly do not expect there to be absolutely no communication between the students of this class; people tend to learn very well that way. However, we expect that you will limit your discussions to determining the SPECIFICATION of the problem; that means identifying what is required for this project. Crossing the line into what the source code is may mean a zero grade on your project.

The code is analyzed for copying and plagiarism. If you have any concern that what you would like to do might violate our rules, please see your lecture instructor for clarification.

Don't forget that the CS mentoring center is available to you. People there can help you understand the problem and give you pointers on debugging.

Problem Statement: Game Description and Programming Objectives

The program is a single-player board game where the player must eat the other pieces or be eaten by the other pieces. As the user spawns more and more pieces, it becomes a frenzy of many self-propelled pieces eating each other.

With the optional, **extra credit timer feature**, new pieces are born every few seconds, and the player must try to eat the pieces and avoid being eaten by the other pieces as they move about the board. A timer thread generates a new piece, places it on the board, and starts it moving. As the player eats other pieces, the game reports consumption points counting how many other pieces the player has consumed.

The game starts with a board and the player piece displayed in a square grid of buttons. Over the top of the board display is a panel containing game information and buttons to quit and optionally adjust the speed of the game. The player controls movement of the player piece using the standard keyboard arrow keys, and controls the creation of the self-propelled pieces with a button. Each of these automatic pieces is controlled by its own thread of execution.

The programming design objective is to apply the Model View Controller (MVC) design pattern to the game. Java's Swing is a framework based on the ideas of the model view controller. See the Design and Implementation section later in this document for details.

Starting the Program: The Command Line

`Frenzy.java` must contain the `main()` method that starts your 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. This range of sizes has been found to be optimal for game play and reasonable display on an expected typical size computer display.

Program View: What the displayed program looks like

The board is a square, N by N grid of locations in which reside zero or one piece each. Above the board is a panel with some game control buttons and game status information. An empty board location appears as a blank button in the default color of the platform's look and feel. Where a piece occupies a location, the location has a different background color and a single text character in the foreground.

The player piece is labeled with a capital 'P' in a yellow background. The single, human user controls the player piece by using the arrow keys on the keyboard. Other game pieces move on their own, automatically controlled by their own thread of execution. These **self-propelled pieces** have a green background color, and their text display is a randomly-generated, printable character chosen when they are born. The value of each piece is the value of the ASCII code of the character; the String.codePointAt() method provides this value. A non-zero value means that the piece is alive; the value becomes zero when a piece is eaten.

NOTE: Self-propelled pieces must use only printable characters in their names. Otherwise, the view cannot display the piece's name.

The statistics on the player piece are in the left side of the top panel. The middle part of the top panel reports progress as pieces eat each other. In the picture, the game has reported that most recently the 'S' piece ate the 'i-hat' piece. The next feeding will replace this message with the details of who ate whom.

Buttons That Control The Game

The right side of the top panel contains buttons to operate or adjust the game.

The Spawn button spawns a self-propelled piece at a random location within the board. The result is a new, self-propelled piece shown on the button displayed with a randomly-generated character as described earlier.

The Quit button simply quits the game, and the window closes. The game must also end when the human user presses the close button in the window's frame.

Program Model: Game Elements and State

The internal model represents the elements and the state of the game as play progresses. The model includes a player piece, some number of self-propelled pieces, a grid of cells (locations), and some of those cells are occupied by a piece. The model keeps track of where each piece is on the board. At any point in time, each living piece occupies only one cell of the grid. When pieces die, they are removed from the game's model; death of a piece occurs when another piece eats it. As pieces eat each other, the model changes each piece's state as the feeding frenzy proceeds.

Program Control: Operation of the Player Piece

The player piece can move as long as the piece's value is greater than zero, which means the player piece is alive. Movement is from one cell to another, adjacent cell at a time, and the human user presses one of the arrow keys to move the piece up, down, left or right one cell location. If the user presses any other key, the piece must not move.

Movement Wraps Around the Board

If the piece has reached the edge of the board, and the user presses a key that would make the piece move past the edge, then the piece must move by wrapping around the board. For example, if the player piece is at the right edge of row 3, and the user presses the right arrow key, then the player piece must move to the leftmost location on row 3.

Pieces Compete for Cell Locations

When the player piece moves to a new cell location, that new location may already be occupied. If it is empty, the piece simply becomes the resident of the location. If the location is occupied however, the player piece eats the existing, resident piece. This 'meal' increases the eating player's consumption by one, sets the eaten piece's value to zero, and places the player piece in the cell occupied by the eaten piece. Because the eaten piece's value is zero, the eaten piece is now dead and must be removed from the game.

Control of Self-propelled, Automatic Pieces

As long as a piece's value is greater than zero, the piece is alive and can move one cell at a time. Like the player piece, these automatic pieces move one cell at a time up, down, left or right. Each automatic piece has its own time delay that defines the time between each move made by the piece. By default, the delay shall be 600 milliseconds, and the piece controller's constructor allows specifying a different delay.

Each self-propelled piece uses the same algorithm to move. The basic algorithm is known as a **stair-step** move, which means the piece moves in a zig-zag direction diagonally across the board. When the program creates the self-propelled piece, the piece picks at random a **diagonal direction** to move. The choices may be called upper-right, lower-right, lower-left and upper-left. Because the piece moves in only one axis direction at one time, it must move first up or down and second to the left or right. Then it alternates these moves, pausing for the delay period between each move. Movement and delay repeat over and over as long as the piece remains alive. The cycle ends when the piece dies.

As an example, the program may decide that a piece should move in an upper-left, diagonal direction. The piece would first choose to move in the up direction and second in the left direction. The result is a move pattern that steadily moves the piece up and to the left relative to its current location. Note that **each piece instance can decide differently** which diagonal direction to follow.

Be sure to handle interruptions properly in the self-propelled threads. If a thread controlling an automatic piece is interrupted, such as when it is sleeping, the thread should emit a stack trace to standard error and stop execution. The consequence is that the piece will stop moving, die and rot away while the game continues to execute. An interrupted piece must then be removed from the game.

Player Scoring

The player piece has a consumption score. The **consumption** is the count of all pieces it has eaten. Consumption starts at 0 and goes up by 1 whenever it eats a piece.

When the game starts, the player has a consumption score of 0. Each self-propelled piece starts with a value equal to the ASCII value of the first character of their name, which is displayed on the screen. For example, if a piece had a name of the capital letter R, their view would be an R in the cell on the screen, and their value would be 82 because the base 10 ASCII code for the character R is 82. (See www.asciitable.com for a table of codes.)

Game Options for EXTRA CREDIT

You should have noticed that the picture of the game has more buttons than were described. These buttons are part of the **game options** that you may decide to do for **extra credit**.

Each group of options below is worth 3% extra credit. If you implement ALL options successfully, you get 10% extra credit.

Below is a list of extra-credit options, followed by a brief description of the pause, speed-up and slow-down behaviors:

- Game Play Options

  The optional Restart button recreates the player piece to resume the game without killing any self-propelled pieces that may still be running. Restart creates the player, sets the number of pieces to consumed to 0, and places the new player piece on the game board at a randomly selected location. The piece will then be displayed in the view, assuming it is not immediately eaten. Restart does not kill any existing, self-propelled pieces.

  Implement the earlier-described timer mechanism that constructs self-propelled pieces every 2 seconds forever.

  Implement the optional Pause checkbox to pause or resume timer-driven construction of self-propelled pieces. Pause does not stop the user from manually creating self-propelled pieces with the Spawn button or by clicking the mouse in one of the board squares. That allows a user to prevent the runaway creation of pieces.

- Piece Variation Options

  Implement ++ and -- button behaviors to speed up and slow down the movement of self-propelled pieces. The optional ++ and -- buttons speed up or slow down the self-propelled pieces. By changing the delay time between self-propelled movements, the pieces will move slower or faster. Note that speed-up (++) will result from a reduction in delay, while slow-down (--) results from an increase in the game delay time.

  Design, document and implement a spiral move algorithm for self-propelled pieces in place of the stair-step move algorithm.

  Design, document and implement a different diagonal move algorithm that makes self-propelled pieces move.

- Enhanced GUI Options

  Implement a scrolling list output of the sequence of who ate whom in the middle part of the top panel (requires scrollable text updated by Swing but triggered by multiple, self-propelled piece threads).

  Implement a more visually exiting display for the game, such as using icons for the self-propelled pieces.

  OR propose your own, GUI enhancement to your instructor.

Design and Implementation

Graphical programs consist of interconnected components that must fulfill 3 different roles : model, view, and controller.

Model components store application-specific data. For example, a text editor stores the characters that form a document's content. This board game must store a representation of a cell location grid, the player piece, the self-propelled pieces, the information on the placement of each piece within the grid environment and the settings for timing values. In response to a change in state of model components, these model components notify any attached view components to tell them that some specific change has occurred.

View components display the model on a screen. For example, a text editor displays the text content rendered in a specific font with a size in a graphical panel. Swing components such as JPanels and JTextFields will display the player piece and the self-propelled pieces in their positions on the view of the game's board model. The view also displays components (JButtons, JLists, etc.) that allow the human user to control the game.

As the view components receive notification of changes from the model, the view components update their display to show the change. For example, when a self-propelled piece eats another, the view of the eaten piece must be removed from the display, and the eater piece takes its place in the display position where the meal took place.

Controller components manipulate view and model components and handle input events from the mouse and keyboard. Some view components have an associated controller that generates events and handles those events by accessing the model to read or change the model's state. Internally, other controllers such as the self-propelling threads, generate program events that trigger changes in program state.

Model View Controller (MVC) Pattern

You will design a program using the **Model View Controller (MVC)** design pattern. This pattern, developed out of the Smalltalk language community, has many references; for example, http://java.sun.com/blueprints/patterns/MVC.html is a java-specific description.

A major goal of MVC is to separate graphical, user interface code segments, the VIEW, from the central, domain-specific code segments, the MODEL. The third segment is the code that issues requests to the model, and these code segments are the CONTROLLER. The controller segments invoke the system model's behavior.

To illustrate this with Java classes, a JButton in a JPanel represents a VIEW element because it defines a visual appearance on a graphical display screen. Behind the scenes in the Swing framework are the CONTROLLER elements that capture the action of a mouse button press, generate an ActionEvent, and deliver that event to the ActionListener(s) registered with the JButton. The developer must write the ActionListener code so that it calls MODEL class instances to invoke the appropriate behavior of the model. In this way, listeners are the CONTROLLER components in a Java Swing application.

Chapter 30 in Liang's 5th edition textbook is dedicated to the Java Model View Controller approach. You should study appropriate sections of this chapter to learn more and then use the the Liang online MVC self-test to check your understanding.

Threads and the Java GUI Update Problem

One problem with multi-threaded GUI programs is ensuring that GUI element changes are performed by the Swing AWT Thread, not by user-defined threads. Failure to manage GUI updates properly can result in program exceptions or abnormal termination. The design must address this problem with SwingUtilities.invokeLater() or SwingUtilities.invokeAndWait(). When a user-defined thread needs to update GUI objects, it needs to define a Runnable to do the work and pass that Runnable instance to SwingUtilities.

Another problem to resolve is race conditions between different user-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). It is in fact a race for pieces to move to the same adjacent location. The destination location must be secured properly in the face of this race condition.

Outline of Implementation Approach

The model representation has a board, the player piece, and the other pieces that are separate from any graphical user interface infrastructure. As pieces move within the model, the board keeps track of the board state and notifies its observing user interface. Controller elements receive events from the Java Swing framework and respond by invoking methods to handle the event. The model components implement these methods to change their state.

The view is responsible for displaying the visible entities of the model, and the view elements are separate from the program's model parts. Elements in the view observe the model components and change the display in response to update messages sent by the model.

The controller is responsible for accepting user input and issuing messages (making method calls) that control the system's behavior. Control operations include handling the of mouse and keyboard inputs, movements of the automatic pieces and the periodic construction of those pieces.

The basic design of the program needs to separate the representation of the board, the user piece and the self-propelled pieces from the view, or display, of those items in Swing GUI components. The controller elements will include a number of listeners that handle events by manipulating the model elements. These listeners may be somewhat tightly-connected to the GUI components because that is the nature of the Java Swing framework. Also part of the controller infrastructure are the threads that control moves and the optional timer thread that creates new self-propelled pieces.

Submissions

This project has 2 submissions.

Submission One

Submission one must:

- Launch a reasonable GUI to display the board view composed of a grid of operable buttons in a resizable, center region of the game's window, plus the game controls view in the top region of the game's window;
- Limit the size of the board to the minimum and maximum number of squares, and print a usage message if the command line argument is not in range;

- Define a model view controller (MVC) implementation with a Swing-based GUI view and controller apparatus;
- Allow the player piece to move around the board under keyboard control;
- Document the design and implementation in a file named Readme-1.txt, which is a plain text file using 80-character width format;
- Implement the Spawn button in the controls region to create a **non-moving** piece in a random location on the board;
- Implement the behavior for the player to eat another piece;
- And implement the Quit button in the controls region and the close functionality of the window close button.

A 'reasonable' GUI is one that nicely displays the game's board and top panel, and gracefully ends the program when the user quits the game.

The submission one game board consists of these components: an empty cell, a cell occupied by one player piece, or a cell occupied by one non-moving piece. There are no self-propelled pieces for submission one.

The program must accept **at most** one command line argument: a number representing the size of the board (the board will always be built as a square). If a user attempts to build a board smaller than 7 by 7 or a board larger than 30 by 30, issue a usage message to standard error, stating **'Usage: board size N must be between 7 and 30'**, and gracefully terminate the program.

The player piece must move in up, down, left or right directions controlled by the keyboard's arrow keys. This is known as four-way movement. When any moving piece reaches the edge of the board, an attempt to move off the edge will cause the piece to wrap around to the opposite side of the board.

Submission Two

Submission two must:

- Change the Spawn button to create a self-propelled, moving piece;
- Implement the eating behavior, in which each moving piece eats the piece found when it moves to occupy an adjacent cell already containing another piece;
- Record and present the number of pieces consumed by the player piece;
- Report when the game is running and when the game is over;
- Deliver a design, implementation and feedback update in the Readme-2.txt file;
- And document what extra credit options are present, if any, in the Readme-2.txt file.

Submission Instructions

Submit your **initial** working, **submission one solution** to:

```
    try grd-232 project2-1 Frenzy.java Readme-1.txt [additional class and documentation
                                   files]
```

Submit your **final** working, **submission two solution** to:

```
    try grd-232 project2-2 Frenzy.java Readme-2.txt [additional class and documentation
                                   files]
```

Note: The browser may not show the try commands above as a single line of input.

Completing the Readme-1.txt and Readme-2.txt Files

Each `Readme-?.txt` file must describe and document your design, implementation, status and feedback on the project. Here is a Readme file template Readme file template to use for each submission. Using 'save as...' save the template changing the N to 1 or 2 for the respective submissions. Here is the general outline of the Readme-?.txt content:

- Identity of the submission number (1 or 2);
- Your name and user login id;
- Some paragraphs answering questions about your design;
- Some paragraphs answering questions about your implementation and the functionality achieved, including a list of known problems and issues that exist with the version;
- And some paragraphs providing feedback on this submission and the project: what you liked and disliked about it.

After you submit, you may check your submission archived by ***try*** to be sure you have submitted all the correct versions of your files.

Nota Bene

In 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*.