CS 1331 Homework 10

Due Monday November 26th, 2012 8:00 PM

Introduction

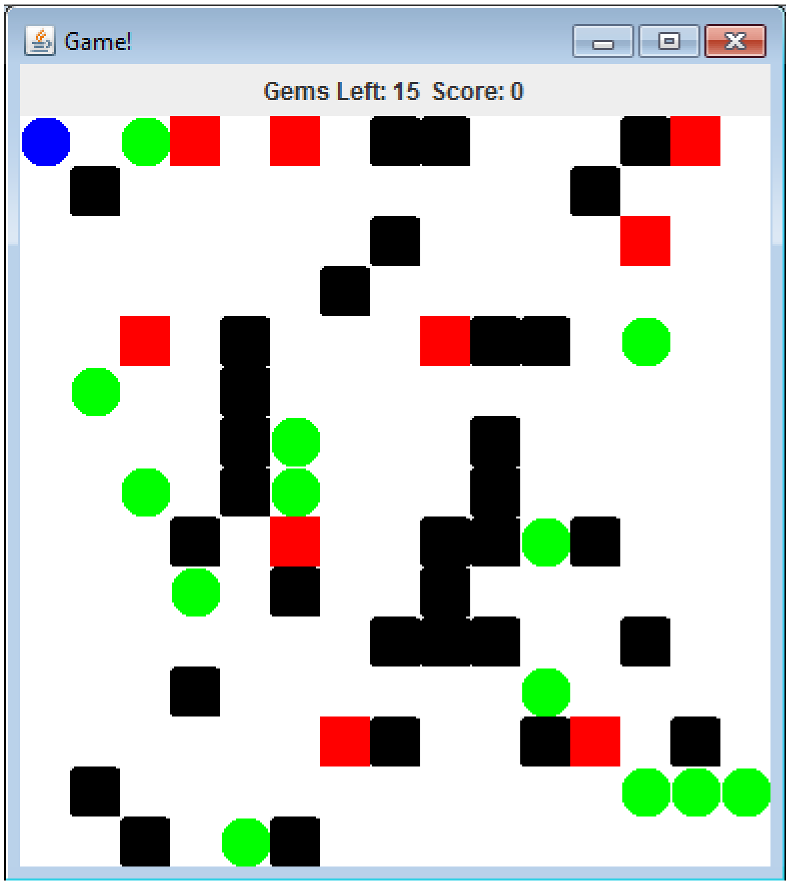
This homework will cover GUIs, graphics, abstract classes, inheritance, polymorphism, dynamic binding, timers, and key listeners.

You will be creating a game where the user controls a character that collects gems while avoiding monsters.

Be sure to name your classes as required by the instructions. Also always use good coding style and indentation, as well as appropriate and descriptive variable names.

This homework will be worth double a regular homework.

A finished implementation might look something like this:



Requirements

For this homework, you are allowed to use any class structure, with any methods you feel are necessary. However, you must meet these requirements:

- The main method is in a class called Game.
- The user is able to control a player that moves with the keyboard.
- The player moves on a level that contains walls (non-passable tiles) and floors (passable tiles).
- The player is not able to move outside the level.
- Some floor tiles contain “gems” which the player is able to pick up.
- There are monsters in the level that the user is trying to avoid.
- Your game should have the capability to contain “simple monsters”, which move left-to-right, up-down, diagonally, or stay still.
- The “simple monsters” are not able to pass through walls or exit the level either.
- The level displays the player, monsters, gems, and walls graphically. In the example above, the player is the blue circle, the gems are the green circles, the simple monsters are the red squares, and the walls are the black squares.
- There is a display that indicates how many gems are left on the board, and the player’s current score.
- When the player hits a monster, either the player loses health or the game ends.
- When the player collects all the gems, either the player moves to the next level or the game ends.
- When the player wins, there is a display to indicate that the player won the game (printing or a JOptionPane is ok here).
- When the player loses, there is a display to indicate that the player lost the game (printing or a JOptionPane is ok here).
- When the player wins or loses and there are no more levels left to play, the monsters stop moving and the player is unable to move.
- Levels’ constructors should be take in a layout based on some arbitrary representation, e.g.

    -
      ```
      ooooxoogoo
      ooxoogooox
      oxoooooogoo
      xxxooogooo
      ```
        - In this example, “o”s represent an available tile, “x”s represent walls, and “g”s represent gems. And could be passed in as 2D char array.
        - You are allowed to set the number of rows and columns in any level to a constant defined in Level.java so you only have to deal fixed level sizes
- You should provide at least two playable levels. You can either choose between them at Game start, or advance to the second after beating the first.
- In addition to having gems, simple monsters, floors, and walls, you must add three extra features to your game via new subclasses. These *could* include: lava tiles, teleporting tiles, “ghost” monsters that can move through walls, monsters that chase the player, pickups that give the player extra health, pickups that give the player invulnerability, etc. Your extra features *cannot* be gems that are worth extra points. You can give pickups

different point values, but they have to have some added component as well.
- Use proper object-oriented programming techniques. For example, use polymorphism when appropriate, and don't put everything in one file!

Additional notes

- You may hard-code the size of the level, the player's starting location, and the name of the level file to read in.
- You may hard-code the number of monsters.
- The implementation of level-layout representations is up to you, but all mechanisms that deal with it should be clear and well-commented (i.e. we should be able to swap in our own level representations with a minimum of fuss).
- You have some freedom in terms of how a user accesses a specific level. You could give them a list to select from, or have them always start at level 1 and level-up to the next.
- You may use your own pictures for the player, pickups, monsters, etc.
- If you're feeling ambitious, this would be a good place to play with File I/O (i.e. the ability to load level representations).

Sample Hierarchy

Note this this hierarchy is one possible design. You do not need to use this hierarchy; however it is a good jumping off point if you don't know where to start. You might not need all methods/variables in these classes, and you will definitely need to add some.

- **Game.java**
  - Description: In addition to containing the main method, it is the gate-keeper of the game, containing Player, Level, and Monster references. Most other classes contain a reference to Game, as it allows them to interact with the player, monsters, and level.
  - Instance variables:
    - A Level
    - A Player
    - An ArrayList of the monsters
    - JLabels for the status and score
    - A LevelPanel
    - A Timer
  - Methods:
    - Game() – sets up the instance variables and starts the timer
    - void win() – called when the player wins. Stops the timer and displays a message.
    - void lose() – called when the player loses. Stops the timer and displays a message.
    - void updateState() - Determines an actual step in the game, responds to the Timer being fired or a Player moving and performs the necessary logic to update the game's state (ie pick up items, take damage).

        - static void main(String[] args) – creates a JFrame and adds the LevelPanel
- **Item.java** (abstract)
    - Description: Items represent the things on the level that the player can pick up.
    - Instance variables:
        - A value for the item (how many points it's worth)
    - Methods:
        - Item(int value) – Takes in the value of the item and initializes the appropriate instance variables
        - abstract void drawItem(Graphics g, int x, int y)
        - int getValue() – getter for the value instance variable
- **Gem.java**
    - Description: A subclass of Item, these represent simple pickups that give the user points. You can hard-code in the point value, or take in as a parameter.
- **Tile.java** (abstract)
    - Description: Tile represents a space on the level. Therefore, the level class will contain a 2D array of Tiles. The exact type of Tile may end up being a Floor, Wall, or some other subclass. Each Tile is responsible for displaying itself.
    - Static variables:
        - Width and height fields for how big each tile is
    - Instance variables:
        - Row and column fields for the location of the tile
    - Methods:
        - Tile() - a constructor that initializes the appropriate instance variables
        - abstract boolean isPassable() – returns true if the tile is passable, false otherwise
        - abstract void playerEnters(Player p) – include if you are planning the Tiles to perform some sort of action on the player
        - abstract void drawTile(Graphics g, int startX, int startY)
- **Floor.java**
    - Description: A Floor tile extends Tile, and represents spaces that the monsters and player can move through.
    - Instance variables:
        - An Item for the Floor tile to hold. Floor tiles not containing an Item contain null instead
    - Methods:
        - Floor(Item item) – a constructor that chains to Tile's constructor and initializes the item
        - Floor() – an overloaded constructor for Floor
        - Item collectItem() – returns the Item that was collected and sets the Floor's Item reference to null.
        - The abstract methods from Tile
- **Wall.java**
    - Description: A Wall tile extends Tile, and represents a wall that the player and monsters cannot pass through.
    - Methods:
        - Wall() – a constructor that chains to the superclass

        - Implements the abstract methods from Tile
- **Character.java** (abstract)
    - Description: Characters represent living objects in the game, such as players and monsters
    - Instance variables:
        - A Game
        - Row and column fields for the location of the character.
    - Methods:
        - Character(Game game, int row, int col) – a constructor that initializes the instance variables
        - abstract void draw(Graphics g) – in the subclasses, this method will draw what character it is.
        - void update() – gets called when the timer is fired. Performs actions like moving a monster
        - boolean runsInto(Character c) – returns true if two Characters are in the same location, false otherwise
- **Player.java**
    - Description: Player represents the player that the user will control. It extends Character.
    - Instance variables:
        - The player's score
    - Player(Game game, int row, int col) – a constructor that chains to Character, and Methods:
        - initializes the score to 0.
        - void move(int drow, int dcol) – moves the player in the inputted direction. First, check that the move is legal, i.e., that the player is moving into a passable tile, and that the player is not moving off the level. Next, actually move the player to that tile, and collect the item that was on the tile. If there was an item, it should update the score. Then, check to see if the player ran into any monsters. Finally, if there are no gems left, the user should win.
        - The abstract methods from Character
- **Monster.java** (abstract)
    - Description: Monster represents the monsters of the game. It extends Character, but since there can be many different types of monsters, Monster is abstract.
    - Methods:
        - Monster(Game game, int row, int col) – a contructor that chains to the super constructor
        - Monster(Game game) – a constructor that assigns the monster to a random row and column.
        - abstract attackAction(Player p) – In the subclasses, this method determines what happens when a monster hits a player. This can be losing health, losing the game, etc.
- **SimpleMonster.java**
    - Description: SimpleMonster is a Monster that can move diagonally, up-down, left-right, or stay still.

- Instance variables:
  - drow and dcol variables to represent how far the SimpleMonster moves each step.
- Methods:
  - SimpleMonster(Game game, int row, int col, int drow, int dcol) – a constructor that chains to the super constructor, and assigns the instance variables
  - SimpleMonster(Game game) – a constructor that chains to the super constructor, and randomly assigns drow and dcol

- **Level.java**
  - Description: The level holds the 2D array of tiles, and determines movement restrictions.
  - Instance variables:
    - 2D array of Tiles
    - A Game
    - The number of items left in the world
  - Methods:
    - Level(Game game, char[][] levelRepresentation) – the constructor. Sets up the instance variables and generates the level's layout based on the given representation.
      - This constructor uses the before proposed representation where the level is passed in as a 2D char array.
    - void draw(Graphics g) – draws all the tiles
    - boolean canMoveToSquare(int row, int col) – determines if the row and column input is within the level, and whether the tile at that location is passable.
    - Item collectItem(int row, int col) – collects the item from that tile (if it is a floor)
- **LevelPanel.java**
  - Description: The LevelPanel represents a single level and the GUI. It extends JPanel
  - Instance variables:
    - A Game
  - Methods:
    - LevelPanel(Game game) – the constructor. Assigns the instance variables and sets up the KeyListener
    - void paintComponent(Graphics g) – calls the level's draw method
  - Inner classes
    - The KeyListener inner class – checks for keyboard input, and moves the player accordingly

Turn-in Procedure

Turn in the following files on T-Square. When you're ready, double-check that you have *submitted* and not just saved as draft.

- Game.java
- readme.txt – Include the three extra features
- Any pictures used in your program
- Any other files needed to run your program

All .java files should have a descriptive javadoc comment.

Don't forget your collaboration statement. You should include a statement with every homework you submit, even if you worked alone.

Verify the Success of Your HW Turn-In

Practice "safe submission"! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.

After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.

Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.

Recompile and test those exact files.

This helps guard against a few things.

It helps insure that you turn in the correct files.

It helps you realize if you omit a file or files.**

(If you do discover that you omitted a file, submit all of your files again, not just the missing one.)

Helps find last minute causes of files not compiling and/or running.

**Note: Missing files will not be given any credit, and non-compiling homework solutions will receive few to zero points. Also recall that late homework (past the grace period of 2 am) will not be accepted regardless of excuse. Treat the due date with respect. The real deadline is 8 pm. Do not wait until the last minute!