Tetris Explanation

By: Teo Ilie

Introduction

Tetris is a Soviet tile-matching strategy video game developed for the first time in 1984 by Alexey Pajitnov. The name comes from the Greek 'tetra' for four, as all pieces are made of four units, and 'tennis', the programmer's favorite sport. Tetris is the best selling video game of all time.

Architectural Overview

Introduction

In order to program Tetris using the java AWT graphics package, we will use the standard set up. There will be four files, all in the same folder. One called `Main.java`, which will be the one that is run to play the game, will construct a *TetrisWindow*: a frame. A second, `TetrisWindow.java`, which extends Frame, will construct the frame and add a panel. This panel, created in the third file called `TetrisPanel.java`, will be responsible for holding the access to painting and to manipulate the game actions and reactions. Said game will be programmed in the fourth and final file `Tetris.java`, in which all necessary methods, including the displaying of the game, will be coded. Note that the *TetrisPanel*, which extends class *Panel*, will implement double-buffering to avoid blinking, as outlined in *Double Buffered Graphics* (Ianine, 2015).

User Input

User interactions will be programmed in the *TetrisPanel* class, implementing the class *keyListener*, which takes in key activities from user input. All methods must be overridden, and in the method *keyPressed*, we will program the following user inputs, each calling its respective method in class Tetris through the object *game*:

1. SPACEBAR – game.hardDrop()
2. LEFT ARROW KEY – game.moveLeft()
3. RIGHT ARROW KEY – game.moveRight()
4. DOWN ARROW KEY – game.moveDown()
5. UP ARROW KEY – game.rotateClockwise()
6. Z KEY – game.rotateCounterclockwise()

All these keys will be processed in a single switch statement.

Program-triggered Piece Dropping

Next it becomes necessary to program the periodic falling of the pieces in Tetris, which becomes faster as the game processes. To accomplish this, we will use the classes *TimerTask* and *Timer*, which runs a process of the programmer's choosing every *n* milliseconds. Because we will be repainting within a *TimerTask* object, this must be running really often, for the display to seem smooth to the gamer. Therefore, we will have *TimerTask* run every millisecond, displaying every time, but only some of these times will it actually shift the piece down. The rest will be 'idle runs.'

As this ‘every so often’ must decrease over time, for the pieces to fall faster, we will have an array *counts* that holds the number of idle runs at the current level, and an integer variable *level* to hold the current level. Then, we will only move a piece down every *counts[level]* runs of *TimerTask*. Counts will hold increasingly smaller numbers, such that higher levels shift pieces down more often, to increase difficulty.

Class *Tetris*

In class *Tetris*, as has been noted, there will be a method *display*, which draws the aesthetic background grid, the current piece, and the landscape. Aside from this, the remaining methods are the following:

1. *hardDrop*() – move piece all the way down
2. *moveDown*() – shift one square down
3. *moveLeft*() – move piece left one square
4. *moveRight*() – move piece right one square
5. *rotateClockwise*() – rotates piece clockwise
6. *rotateCounterClockwise*() – rotate piece counter clockwise

Game Over and Restarting

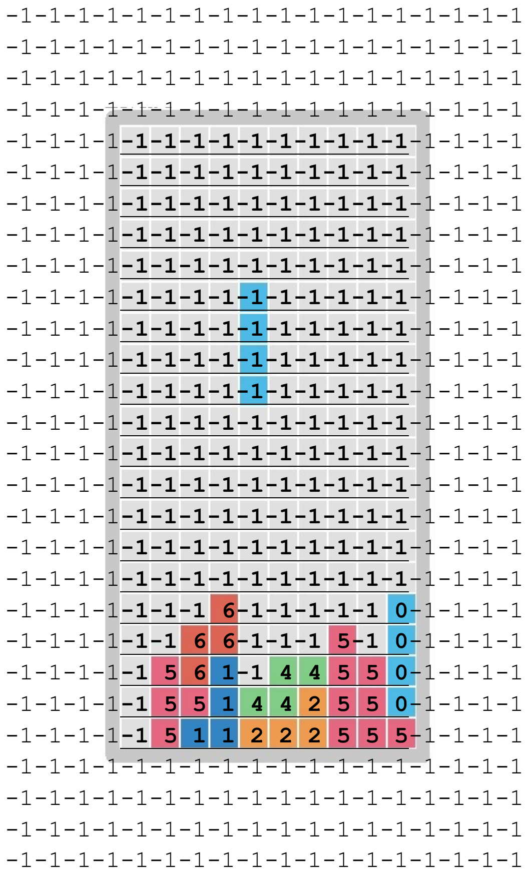
Finally, the last step is to check for game over and to restart the game. This is done by making sure, after every time a piece is added to the landscape, that the top four rows of the fake landscape are empty, that is, the piece is completely visible to the gamer. If not, the game is finished, by setting a boolean variable *gameOver* true. Every time *timertask* is run, it repaints. In *update*, therefore, we will say if the game is not over, display. Else, print “game over, click to restart.” To program the latter, we will use *mouseListener*. In the body of the overridden method *mousePressed*, we will write that if the game is over, and the mouse has been clicked, the landscape is reset, level is reset, curr is reset, and variable *gameOver* is set to false again.

Implementation Details

Tetris is played on a 2D grid, making it the obvious choice to use a 2D array to hold the game status. To simplify the process, we will use an array named *landscape* that is 28 x 18, instead of the regular Tetris 20 x 10, with a border of width four all around that will facilitate checking for borders and displaying the piece falling into the board at the beginning. Note that only the 20 x 10 grid in the center is ever shown to the gamer, as seen in Figure 1 where the landscape array and a screenshot of the gameplay are overlaid. The landscape will hold -1 for empty, and 0 to 5 for the codes of the colors of the landscape, which will be colored using the array of colors, as described below. Note also that the falling piece is not represented in the landscape.

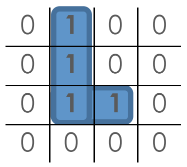
Next, all pieces will be saved as 0 for empty and 1 for full in their own 4 x 4 arrays, 7 in total. All in all, this will be done in a 3D array of 7 x 4 x 4. Their colors will also be saved in an array of 7 colors. Every time a new piece is created, it will be copied from the 3D array of pieces into an array of 4 x 4 called *curr* (seen in Figure 2) within which it can be rotated. Its position will be saved in variables *row* and *col*. Every time the user makes an action unto the piece, *curr* is changed if borders are not being trespassed and the landscape is not being touched, all of which is checked in a separate method *canMove*() also in class *Tetris*. The method *canMove()*

copies the landscape into a temporary array of the same size, as well as the falling piece *curr*, in the position defined by *row* and *col*. Then it checks offline for border overlap and landscape touching. If *canMove*() determines the piece has touched the landscape, it is added to the landscape, and *curr* is reset with the next randomly generated piece, and *row* and *col* are reset from the top as well. Also, the landscape is checked for cleared lines, and the total number of cleared lines saved in a public variable that can be accessed by *TetrisPanel* to change the level based on number of lines cleared, for example, increasing the level every ten lines cleared.



```
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1 6-1-1-1-1-1 0-1-1-1-1
-1-1-1-1-1-1 6 6-1-1-1 5-1 0-1-1-1-1
-1-1-1-1-1 5 6 1-1 4 4 5 5 0-1-1-1-1
-1-1-1-1-1 5 5 1 4 4 2 5 5 0-1-1-1-1
-1-1-1-1-1 5 1 1 2 2 2 5 5 5-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1
```

Figure 1: Visualization of the landscape array



0	1	0	0
0	1	0	0
0	1	1	0
0	0	0	0

Figure 2: Visualization of the current piece array *curr*

References

Ianine, S. (2015). Double Buffered Graphics. *ICS4UE*. Retrieved June 3, 2015.