PP15: ProcessingPuzzle15

Per la documentazione in italiano continuare a scorrere

Documentation (English)

Files

- /: Licences and README
- /doc: documentation in pdf
- /src: Eclipse Project
- /bin: self contained .jar executable

PP15 is an implementation in Processing of the popular game “Puzzle 15”.

The project has been realized using the Eclipse IDE to get an overall better code indentation, auto-completion and library management compared to what the Processing IDE (tries to) offer.

The Processing library has been included in the form of the core.jar library, under the GNU LGPL license (you can find a copy of this license in the root folder of this project). Most of the code is in the Sketch.java class, except for the Main.java class that contains the code needed to run the Processing code properly.

The source code, comments and interface language are all in English. Due to the limited time for the realization of the project there is no localization support for the few strings in the game, and almost all of them are hardcoded.

Classes

All the classes described beloware sub-classes in the Sketch.java class

FileMan

Short for FileManager. This class interacts with the “scores.txt” file containing the high score and the best time respectively per line.

The constructor checks for the extistance of the file before creating it and reading the lines into the **lines** object, which is an object of the **List** class, part of the standard Java Library.

To create the file it’s been used the **createNewFile()** method is called from the **f** object, which is an object of the **File** class, again part of the standard Java Library.

This method creates the file only if it doesn’t exist. This way there is virtually no risk of accidentally overriding an existing file and losing the existing scores.
If the file didn’t previously exist, a new one is created, containing the starting records for the high score and the best time, both initialized from 0.

The functions **getHScore()** and **getBstTime()** return respectively the high score and the best time retrieved from the “scores.txt” file.

The function **updateScores()** takes a new high score and a new best time and save both in the “scores.txt” file. This function is called when exiting the application properly by using the button labelled “Quit”

ImageMan

Short for ImageManager. The role of this class is working with imported images.

More precisely it resizes them to the standard size of 200x200 pixels, and cuts them into a “mosaic” of 50x50 pixels pictures, so that they can overlay the Tiles of the game.

This class stores the original picture in the **pict** object and the mosaic into the **mosaic** array; which are respectively a single **PImage** object and an array of **PImage** objects.
PImage is part of the Processing library.

The method **loadPic()** calls the **selectInput()** method, which is again part of Processing. I couldn’t manage to use a callback function for the beforementioned method inside of the **ImageMan** class, so I just roughly put it outside of it.
While this is, again, pretty rough, due to the simplicity of the application, worked perfectly during the test phase.

I want to make a little sidenote here on the rough nature of this application and its code. I don't want to ignore my responsabilities, but the biggest cause for the mediocre quality of the code is probably the nature of Processing. Over time it proved to be not only inefficient, but also pretty badly documented and overall a big pain to use, since it doesn't seem to respect the most basic principles of the paradigm of Object Oriented Programming. It is pretty handy indeed for simple educational purposes, but in my opinion asking to realize a bigger project like the one proposed with Processing means encouraging bad programming practices that can even be counter productive in the long run. Note that this is just my opinion, and that I don't mean any offense at all, but I wanted to have the possibility to give my personal opinion on this matter, and possibly encourage a change or an improvement in the future.

The beforementioned callback method is **fileSelected()** . It checks if the file selected has one of the most common image extensions and eventually loads them into the **pict** object, resizes it using the **resize()** method and finally calls the **makeMosaic()** method that populates the **mosaic** array with cropped portions of the said picture.

Button

This class implements a simple button in the form of a rectangle of fixed dimension with some text in it.

The constructor takes the button text and position as arguments and transfers them into the object specific variables.

The constructor has also been overloaded to include the possibility to specify a custom color for the button.

The **show()** method uses Processing functions to define how to draw the button.

The core of this class is the **checkPressed()** method. This method has been made to be called inside the Processing **mouseClicked()** function (found in the bottom of the source code). It returns *true* if the mouse is over the button when the function is called, otherwise it returns *false* . It's easy to see how this can be useful inside the **mouseClicked()** function as it's only called when the mouse is pressed.

Tile

This is one of the most important classes of this project.

A *Tile* is defined logically as a single tessel of the "Puzzle 15" game. Therefore it contains its number (from 1 to 16, where 16 is the empty tile), position values for the x and y axes, along with some static final variables like **SIZE** and **STROKESIZE** that clearly have to be consistent across the whole application, and also referenciable from outside of the class itself, expecially when taking measures and making math with them.

A Tile can theorically be initialized without a position, so that it can be assigned later on, therefore two constructors have been made for it.

The **show()** method, similarly to **Button** , uses Processing methods to display the Tile. Note that if the Tile has number 16 it's drawn white and considered as the empty tile.

Score

This class is mainly meant to be used as a simple data structure to hold the **score** and **highScore** values.
It also has a constructor that pulls the value of **highScore** from the "scores.txt" file, if the value isn't 0.

The default value for **highScore** is -1, as it works better when comparing the **score** with the **highScore** and checking if this last one needs to be updated.

Chronometer

This class is similar to intentions to the **Score** class, but more complex due to the nature of time measurement in computer programs.

As the beforementioned **Score** class, it holds the **currentTime** and the **bestTime** , both long integers since when working with timestamps in milliseconds (even when the actual final values are stored in seconds), a long variable is indeed needed.

It also contains a **startTime** variable that holds the absolute time from the *epoch* (in seconds) when the chronometer has been started using the **startChron()** method.
This is a very practical approach to get the **currentTime** since the Chronometer has started, as seen in the **updateTime()** method.
The **updateTime()** method in particular is called in the **Grid.show()** method (described in detail below) that gets cycled when called from the **draw()** method. This ensures a pretty accurate way to update the current time and get up to date values as time passes.

Again, similarly to the **Score** class, the constructor pulls the *bestTime* value from the "scores.txt" file, again if it's not 0.

Grid

> *The Grid. A digital frontier. I tried to picture clusters of information as they moved through the computer.*
>
> *What did they look like? Ships? Motorcycles? Were the circuits like freeways?*
>
> *I kept dreaming of a world, I tought I'd never see. And then, one day - I got in.*

-The Grid, Daft Punk

The **Grid** class is the pulsating core of this application.
It contains instances of various other classes, along with some important other variables and arrays. I will try to summerize the most important ones below.

- **mScore** : the main instance of the **Score** class
- **chron** : the main instance of the **Chronometer** class
- **randTimes** : the number of times the tiles are moved during the randomization process (more info below)
- **gameStarted** : Boolean value needed to check if the game is started and running, used mainly to know when to show the “GAME OVER” text.
- **tileset** : array of **Tile** objects. The tiles are created in order in the array; basically when the game starts it’s virtually already solved, making sure that it can actually be solved without using the provided formula (both for more semplicity in the implementation and for avoiding to actually need to complete complicated games to test the application and lose big amount of times in repeated unnecessary tests)
- **RELX** and **RELY** : for more semplicity, the **Grid** uses a different logic axis origin position from the one of the entire window. This way it’s possible to have a good padding for the game **Grid** and have a lot of clearance to insert various buttons and labels across the interface
- **MOVERULES** : a matrix containing the “rules” on which tiles can move, relatively to the position of the empty tile

The **show()** method contains the code needed to show the **tileset** , **mScore** and **chron** values on screen, along with the picture mosaic from **ImageMan** class, as described above.

Following are some utility methods to get **Tiles** from their numbers, position of **Tiles** with certain numbers and the **Tile** object that has been clicked.

The **getEmpPos()** method is particularly useful for studying the **Tile** moving logic and making use of the **MOVERULES** matrix described above, into the **isMoveable()** method.

The **checkWin()** method is simply a sorting check. If the **tileset** array is correctly sorted, then the puzzle is solved. This was easier thanks to the choice of using a single dimension array for the tileset, instead of a more intuitive, but less efficient matrix.

The **win()** method stops the **chron** , stops the game and if needed updates the **mScore.highScore** and **chron.bestTime** values.

The **randomize()** method initializes the game. It resets the **mScore.score** value, starts the **chron** and tries to move the tiles randomly similarly to how a person would do by clicking randomly on any of the tiles, since a number of tiles greater then **randTimes** is moved in the process.

This seemed like the most effortless way to randomize the **tileset** without having to lose lots of time in continuous trial and fixing phases.

Documentazione (Italiano)

Files

- /: Licenze e README
- /doc: documentazione in pdf
- /src: progetto Eclipse
- /bin: eseguibile .jar self contained

PP15 è un'implementazione in Processing del popolare "Gioco dei 15".

Il progetto è stato realizzato usando l'IDE Eclupse per ottenere una migliore indentazione del codice, un migliore auto-completamento ed una migliore gestione delle librerie, rispetto a cosa l'IDE di Processing (cerca di) offrire.

La libreria di Processing è stata inclusa nel progetto in forma della sua libreria core.jar, sotto la licenza GNU LGPL (una copia della licenza è inclusa nella cartella radice di questo progetto). La maggior parte del codice si trova nella classe Sketch.java, eccetto per la classe Main.java che contiene soltato il codice necessario per eseguire opportunamente il codice Processing.

Il codice sorgente, i commenti e le interfacce sono in inglese. A causa del tempo limitato per la realizzazione del progetto, non c'è alcun supporto per la localizzazione per le poche stringhe contenute nel gioco, e la maggior parte di tali stringhe è "hardcoded".

Classi

Tutte le classi descritte sono sotto classi della classe Sketch.java

FileMan

Abbreviazione per FileManager (gestore file). Questa classe interagisce con il file "scores.txt" contenente i punteggi più alti e i migliori tempi rispettivamente per linea.

Il costruttore controlla l'esistenza del file prima di crearlo e leggendone le linee nell'oggetto **lines** , che è un oggetto di classe **List** , parte della libreria standard di Java.

Per creare il file è stato usato il metodo **createNewFile()** , chiamato dall'oggetto **f** , che è un oggetto di classe **File** , ancora una volta, parte della libreria standard di Java.

Questo metodo crea il file solo se non esiste già. In questo modo non c'è virtualmente alcun rischio di sovrascrivere il file esistente perdendo i punteggi esistenti.

Se il file non esisteva in precedenza, ne viene creato uno nuovo, contenente i valori iniziali per il miglior punteggio ed il miglior tempo, entrambi inizializzati a 0.

Le funzioni **getHScore()** e **getBstTime()** restituiscono rispettivamente il punteggio più alto ed il miglior tempo raccolti dal file “scores.txt”

La funziona **updateScores()** prende come argomenti il nuovo miglior punteggio e il nuovo miglior tempo e li salva entrambi nel file “scores.txt”. Questa funzione viene chiamata quando viene chiuso il programma in maniera corretta usando il tasto indicato come “Quit”.

ImageMan

Abbreviazione di ImageManager (gestore immagini). Il ruolo di questa classe è lavorare con le immagini importate.

Più precisamente, le ridimensiona alla grandezza standard di 200x200 pixel, e le suddivide in un “mosaico” di immagini 50x50, così che possono essere posizionate sopra le tessere del gioco.

Questa classe salva l’immagine originale nell’oggetto **pict** e il mosaico nell’array **mosaic** ; che sono rispettivamente un singolo oggetto **PImage** e un array di oggetti **PImage** .
PImage è parte della libreria di Processing.

Il metodo **loadPic()** chiama il metodo **selectInput()** , che è ancora una volta parte di Processing. Non sono riuscito a usare una funzione di callback per il metodo prima citato dentro la classe **ImageMan** , quindi l’ho grezzamente messa fuori dalla classe.
Anche se questo metodo è, come ho già detto, abbastanza grezzo, grazie alla semplicità dell’applicazione, ha funzionato alla perfezione durante le varie fasi di test.

Vorrei fare una piccola considerazione in merito alla natura grezza di questa applicazione e del suo codice. Non voglio ignorare le mie responsabilità, ma la più importante causa per la qualità mediocre di questo codice è probabilmente la natura stessa di Processing. Nel tempo si è rivelato non solo poco efficiente, ma anche documentato piuttosto male e tutto sommato un gran dolore da usare, dato che non sembra nemmeno rispettare le più basilari regole del paradigma della programmazione a oggetti. È indubbiamente piuttosto comodo per semplici usi didattici, ma a mio avviso chiedere di realizzare un progetto piuttosto grande come quello proposto con Processing significa incoraggiare cattive pratiche di programmazione che potrebbero risultare anche contro producttive nel tempo. Si noti che questa è solamente la mia personale opinione, e che non intendo offendere nessuno in alcun modo, piuttosto volevo semplicemente avere la possibilitù di dare la mia personale opinione a riguardo, e possibilmente incoraggiare un cambiamento o un miglioramento nel futuro.

Il metodo precedentemente menzionato è **fileSelected()** . Esso controlla se il file selezionato ha una delle più comuni estenzioni per le immagini per poi caricarlo nell’oggetto **pict** , lo ridimensiona usando il metodo **resize()** ed infine chiama il metodo

makeMosaic() che popola l'array **mosaic** con porzioni ritagliate da tale immagine.

Button

Questa classe implementa un semplice bottone nella forma di un rettangolo di dimensione fissa con del testo al suo interno.

Il costruttore prende come argomenti il testo del bottone e la sua posizione, per poi trasferirli nelle variabili specifiche dell'oggetto.

È anche disponibile un altro costruttore che include la possibilità di specificare un colore personalizzato per il bottone.

Il metodo **show()** usa le funzioni di Processing per definire come disegnare il bottone.

Il cuore di questa classe è il metodo **checkPressed()** . Questo metodo è stato realizzato per essere chiamato dentro la funzione **mouseClicked()** di Processing (posizionata in fondo al file sorgente). Restituisce *true* se il mouse è sopra il bottone quando la funzione viene chiamata, altrimenti restituisce *false* . È molto facile vedere come questo metodo può essere utile dentro **mouseClicked** dato che verrebbe chiamata solo quando il mouse viene premuto.

Tile

Questa è una delle classi più importanti di questo progetto

Un *Tile* è definito logicamente come un singolo tassello del "Gioco dei 15". Dunque contiene il suo numero (da 1 a 16, dove il 16 è il "tassello vuoto"), il valore della posizione per gli assi x e y, insieme a qualche variabile *static final* come **SIZE** e **STROKESIZE** che chiaramento devono essere consistenti in tutta l'applicazione, e anche referenziabili fuori dalla classe stessa, specialmente quando vengono prese delle misure con le quali vengono fatti dei calcoli.

Un *Tile* può in teoria essere inizializzato senza una posizione, così che gliela si possa assegnare in un secondo momento; per questo sono stati realizzati due costruttori.

Il metodo **show()** , similarmente a come avviene in **Button** , usa le funzioni di Processing per mostrare il *Tile* . Si noti che se il tile ha numero 16 è disegnato di bianco e considerato come uno spazio vuoto.

Score

Questa classe è stata principalmente studiata come una semplice strutturadati per tenere in memoria i valori **score** e **highScore**

Contiene anche un costruttore che prende i valori del punteggio più alto dal file "scores.txt", se il suo valore non è 0.

Il valore predefinito di **highScore** è -1, dato che risulta più comodo quando vengono comparati **score** e **highScore** per controllare se quest'ultimo necessita di essere aggiornato.

Chronometer

Questa classe è simile per intenzioni alla classe **Score** , ma più complessa data la natura della misurazione del tempo nei programmi per computer.

Come la prima menzionata classe **Score** , questa tiene in memoria i valori **currentTime** e **bestTime** , entrambi *long int* in quanto quando si lavora con dei *timestamp* in millisecondi (anche se i valori finali sono memorizzati in secondi), sono senza dubbio richieste delle variabili lunghe.

Contiene anche una variabile **startTime** che contiene il tempo assoluto dall' *epoch* (in secondi) da quando il cronometro è stato azionato usando il metodo **startChron()**

Questo risulta un modo particolarmente pratico per ottenere il **currentTime** da quando il cronometro parte, come è possibile vedere nel metodo **updateTime()** .
Quest'ultimo metodo in particolare è chiamato nel metodo **Grid.show()** (descritto nel dettaglio sotto) che viene ciclato quando chiamato nel metodo **draw()** . Questo assicura un modo abbastanza accurato per aggiornare il tempo corrente e ottenere valori aggiornati col passare del tempo.

Ancora una volta, in modo simile alla classe **Score** , il costruttore legge il valore del miglior tempo dal file "scores.txt", ancora una volta, se diverso da 0.

Grid

> *La Griglia. Una frontiera digitale. Ho cercato di immaginare i cluster di informazioni mentre si muovono attraverso il computer.*
>
> *A cosa assomigliano? Navi? Motociclette? Che i circuiti siano come le superstrade?*
>
> *Continuavo a sognare di un mondo, che non immaginavo avrei mai visto. E poi, un giorno - ci entrai.*

-The Grid, Daft Punk

La classe **Grid** è il cuore pulsante di questa applicazione. Contiene istanze a varie altre classi, insisme ad alcuni array e variabili importanti. Cercherò di riassumere i più importanti di seguito.

- **mScore** : l'istanza principale della classe **Score**
- **chron** : l'istanza principale della classe **Chronometer**
- **randTimes** : il numero di volte che i *Tile* sono mossi durante il processo di

randomizzazione (più informazioni sotto
- **gameStarted** : valore Booleano necessario per controllare se il gioco è partito ed in funzione, usato principalmente per sapere quando mostrare la scritta “GAME OVER”
- **tileset** : array di oggetti di tipo **Tile** . I *Tile* sono creati in ordine all’interno dell’array; quando il gioco parte, è già virtualmente risolto, assicurando che possa essere effettivamente risolto senza implementare la formula fornita (sia per una maggiore semplicità nell’implementazione che per evitare di dover completare casi complicati del puzzle per testare il funzionamento dell’applicazione e perdere così grandi quantità di tempo in test ripetuti e non necessari)
- **RELX** e **RELY** : per una maggiore semplicità **Grid** usa un’origine degli assi logica diversa da quella dell’intera finestra. In questo modo è possibile avere sia un buon margine per la griglia di gioco che spazio a sufficienza per inserire vari bottini e campi di testo nell’interfaccia.
- **MOVERULES** : una matrice contenente le “regole” riguardo quali *Tile* si possono muovere, relativamente alla posizione del *Tile* vuoto

Il metodo **show()** contiene il codice necessario per mostrare i valori di **tileset** , **mScore** e **chron** sullo schermo, insieme al mosaico dell’immagine dalla classe **ImageMan** , come descritto sopra.

Di seguito ci sono alcuni metodi di utilità per ottenere *Tile* dal loro numero, o posizioni di *Tile* con un certo numero e l’oggetto *Tile* che è stato cliccato.

Il metodo **getEmpPos()** risulta particolarmente utile per studiare la logica di movimento dei *Tile* e per usare le regole nella matrice **MOVERULES** come descritto sopra, nel metodo **isMoveable()** .

Il metodo **checkWin()** è semplicemente un controllo di ordinamento. Se l’array **tileset** è ordinato correttamente, allora il puzzle è risolto. Ciò è stato reso più semplice grazie alla scelta di usare un array monodimensionale per il **tileset** , invece di una più intuitiva ma meno efficente matrice.

Il metodo win() ferma **chron** , ferma il gioco e se necessario aggiorna i valori di **mScore.highScore** e **chron.bestTime** .

Il metodo **randomize()** inizializza il gioco. Resetta il valore di **mScore.score** , fa partire **chron** e prova a muovere i *Tile* randomicamente, similarmente a come farebbe una persona cliccando a caso su ogni *Tile* , finché non ha mosso una serie di *Tile* maggiore in numero della variabile **randTimes** .
Questo sembrava il modo migliore e più veloce per randomizzare il tileset, senza dover perdere molto tempo in lunghe fasi di test.