Tetris Java project

Flavian Theurel - Maxime Blanchard

—

LP2A - P2022

Université de Technologie de Belfort Montbéliard

0 - Practical details 2
Game class location 2
Run instruction 2
Key binds 2

1 - Graphical user interface 2
The API (Processing) 2
The interface 3

2 - Program architecture 3
Program's main structure 4
The scenes 4

3 - Game mechanics 5
Grid, lines and tiles 5
Loss condition 5
Wall kicks 5
Lock delay 6
Queue and random generator 7
Tetromino holder 7

0 - Practical details

Game class location

Since we are using the Processing API, the *Game* class had to be placed inside a package so we could give Processing an access to it in the module-info file. **It can be found in the *tetris_game* package.**

Run instruction

In order to run the application from Eclipse, after importing the project, you need to:

- In the Package Explorer, right click on the *core.jar* file, then select Build Path → Add to Build Path
- You can now run the tetris_game/Game.java file

In case there is an issue with the *core.jar* file, here are the instructions to get it:

- Download Processing 3.5.4 (https://processing.org/download)
- Unzip the .zip file
- Find the *core.jar* file under "Processing\processing-3.5.4\core\library"
- Drag and drop the file in the *src* folder of the project in Eclipse's Package Explorer

Key binds

- Rotate 90° clockwise : **Up arrow** and **X**
- Rotate 90° anti-clockwise : **Ctrl** and **W**
- Move left : **Left arrow**
- Move right : **Right arrow**
- Hard drop : **Space**
- Soft drop : **Down arrow**
- Hold : **Shift** and **C**

1 - Graphical user interface

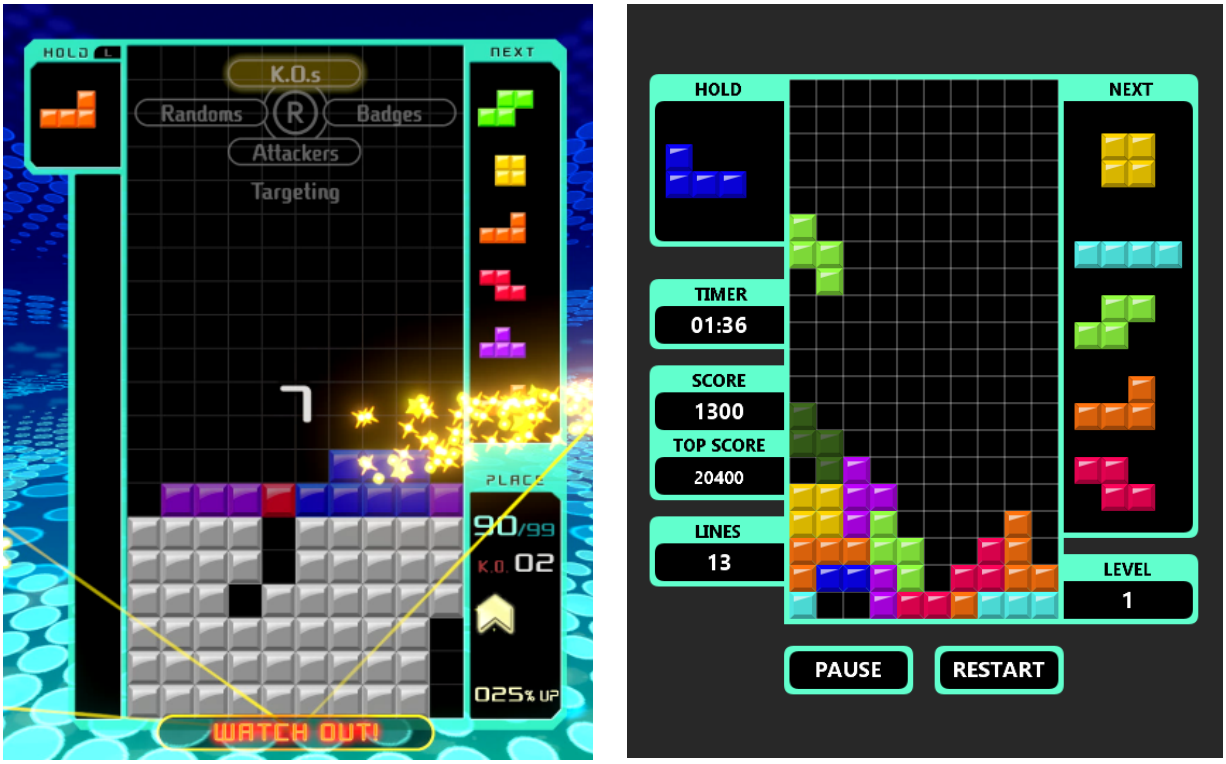
The API (Processing)

Since Swing and AWT were not made for creating games and are quite restrictive in terms of interface design, we decided to use Processing to build our GUI. We chose Processing

because it grants a lot more freedom when designing interfaces, it is fairly easy to use, and also because both of us have already used it in the past.
Processing comes with its own IDE and is meant to be used as a "standalone" language (in a similar way to Arduino). However, since it is a Java library in the first place, it can also be imported and used in a Java project, which is how we used it to build our Tetris project.

The interface

The interface of our game is mostly inspired by Tetris 99. It's the most recent Tetris game playable.



Tetris 99 GUI (left, source: https://www.nintendo.fr/) compared to our game's GUI (right)

2 - Program architecture

Note : please refer to the ***LP2A - Tetris UML.pdf*** file that comes with this report to see all the details about the architecture of our program.

Program's main structure

The entry point of the program is the *main* function of the *Game* class, inside the *tetris_project* package. However, the *main* function only contains one line of code:

```
PApplet.main("tetris_game.Game")
```

This instruction launches the PApplet, which is the base class to create a Processing window. The parameter that is given to the main function is the path to the class in which the PApplet is going to look when trying to call a function. The structure of our program is implicit since most of the function calls are made by the PApplet.

When the PApplet is started, it first calls the *settings()* method in the class passed as an argument (if there is one), and then creates the window. This is why some parameters, such as the size of the window, need to be set in this method. Once the window is created, the PApplet calls once the *setup()* method, and then it is going to call the *draw()* method in a loop until the user closes the window or the exit method of the PApplet is used.

In addition to that, every time a key of the keyboard is pressed, the PApplet will try to call the *keyPressedmethod()*, and when a key is released, it is going to call the *keyReleased()* method. There are also 2 equivalent methods for the mouse buttons (*mousePressed()* and *mouseReleased()*).

The scenes

To better organize the code of the game, we decided to split it into several scenes. A scene combines the elements to display on the screen, the logic that runs in the backend and the processing of the user's input. In terms of code, this translates to a Scene interface which contains 3 abstract methods :

- a *processInput()* method that contains the code to execute if the user presses a key or clicks on a button,
- an *update()* method which is used to keep the objects of the scene up to date (like the Tetrominos[1], the timer, the grid, etc.)
- a *render()* method, which includes all the instructions to display the scene on the window.

These methods are called from the Game class, in the draw function that is itself called by the PApplet. Therefore, the class needed a *setCurrentScene* method (which takes a Scene

[1] "Tetromino" is the official name given to the 7 Tetris shapes

instance as a parameter) to change the scene that is currently loaded by the game when the player loses a game or clicks on a button.

When a scene is set as the current scene, the game will execute the *processInput()*, *update()* and *render()* methods in that order, 60 times per second. While the render() function can be executed as much as we want, some of the code in *processInput()* and *update()* needs to be executed at a given frequency to work properly (like making the Tetrominos fall or move sideways). Therefore, we have to keep track of the time that passes after these parts of the code are executed, so we can make sure that the right amount of time has passed before executing the code again.

3 - Game mechanics

Grid, lines and tiles

We defined the grid as an ArrayList of lines, which are themselves defined as arrays of Tiles.

By default, the grid is filled with tiles of the type *NullTile* (inherited from *Tile)*, which are replaced with tiles of the base *Tile* type when a falling tetromino is locked. To display the grid, we go through each line and call the *display()* function of each tile : if the tile's type is *Tile*, it is going to display a tile as expected, but if it is a *NullTile*, it will display a black square with gray borders (the background of the grid).

The first line in the grid represents the bottom line, while the last line is the one at the top. When a line is filled, it is removed from the ArrayList and a new line filled with *NullTiles* is added at the end.

Loss condition

While only 20 lines of the grid are shown on the screen, the grid contains 3 more lines above the 20th one for technical reasons, notably because new falling tetrominoes are generated above the grid, and then will try to go down two rows.

A game ends when one of these 2 conditions is met:

- There is at least one tile that is not null above the 20th line of the grid,
- A new falling tile doesn't have any room to enter the grid.

Wall kicks

Like most recent Tetris games, our game includes a mechanic called "wall kicks", which allows the player to rotate the falling tetromino in places where it shouldn't be possible because some tiles and/or a wall are obstructing the spot where the tetromino would end up.

To achieve this, the game will try to apply one of four possible translations until an unobstructed spot is found for the tetromino. These translations are defined by two tables (one for the I-shaped tetromino, one for the 6 others tetrominoes). If neither the base rotation or the four alternative rotations are possible, the rotation will be canceled.

	Test 1	Test 2	Test 3	Test 4	Test 5
0>>1	basic rotation	(-1, 0)	(-1, 1)	(0,-2)[1]	(-1,-2)
1>>0	basic rotation	(1, 0)	(1,-1)	(0, 2)	(1, 2)
1>>2	basic rotation	(1, 0)	(1,-1)	(0, 2)	(1, 2)
2>>1	basic rotation	(-1, 0)	(-1, 1)[1]	(0,-2)	(-1,-2)
2>>3	basic rotation	(1, 0)	(1, 1)[1]	(0,-2)	(1,-2)
3>>2	basic rotation	(-1, 0)	(-1,-1)	(0, 2)	(-1, 2)
3>>0	basic rotation	(-1, 0)	(-1,-1)	(0, 2)	(-1, 2)
0>>3	basic rotation	(1, 0)	(1, 1)	(0,-2)[1]	(1,-2)

	Test 1	Test 2	Test 3	Test 4	Test 5
0>>1	basic rotation	(-2, 0)	(1, 0)	(-2,-1)	(1, 2)
1>>0	basic rotation	(2, 0)	(-1, 0)	(2, 1)	(-1,-2)
1>>2	basic rotation	(-1, 0)	(2, 0)	(-1, 2)	(2,-1)
2>>1	basic rotation	(1, 0)	(-2, 0)	(1,-2)	(-2, 1)
2>>3	basic rotation	(2, 0)	(-1, 0)	(2, 1)	(-1,-2)
3>>2	basic rotation	(-2, 0)	(1, 0)	(-2,-1)	(1, 2)
3>>0	basic rotation	(1, 0)	(-2, 0)	(1,-2)	(-2, 1)
0>>3	basic rotation	(-1, 0)	(2, 0)	(-1, 2)	(2,-1)

Rotation tables for the J, L, T, S, Z tetrominoes (left) and for the I tetromino (right). Source : https://tetris.fandom.com/wiki/SRS#Wall_Kicks

Lock delay

When the falling tetromino ends up in a place where it cannot fall anymore, a timer is launched. Once this timer reaches a certain amount of time (2 times the delay it takes for the tetromino to go down one row, which varies according to the difficulty), the tetromino will be locked in place, and the next tetromino in the queue will appear at the top of the grid.

This leaves the player the ability to move the tetromino sideways or to rotate it before it becomes static. If the player moves the tetromino in a position where it is able to fall again, the timer is set back to zero and paused until the tetromino isn't able to fall anymore.

Queue and random generator

The queue is an ArrayList of tetrominoes that contains at any point in time at least 5 tetrominoes. To generate the list of the tetrominoes, we followed the Tetris Guidelines and used a technique called “7-bag generator”. It works by generating a sequence consisting of one copy of each of the 7 tetrominoes in a random order (like if they were drawn from a bag), and then adding this sequence at the end of the queue.

Tetromino holder

We decided to implement in our game a tetromino holder, a common future in recent Tetris games. It allows the player to send the falling tetromino in the “hold box”, while the tetromino that was until now in the box appears at the top of the grid and starts falling. Then, the hold feature cannot be used again until the tetromino is locked into place.

At the beginning of a game, the hold box is empty : the first time it is used, the falling tetromino is replaced by the first tetromino in the queue.