Aalto University

ELEC-A7151 - Object oriented programming with C++

Project plan

Juho Heimonen (473611)
Kalle Alaluusua (585046)
Rustam Latypov (474461)
Visa Lintunen (593009)

November 9, 2018

Introduction and scope

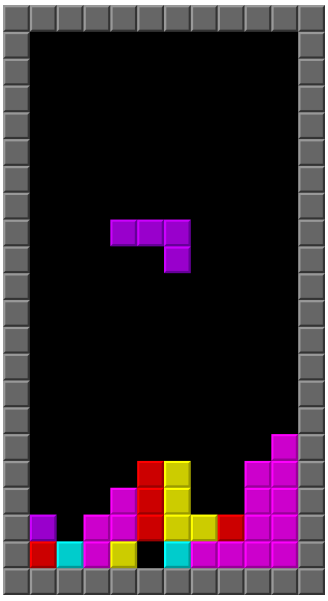
The goal of this Tile-matching project will be to implement Tetris with a special mode of play. Tetris is a well known tile-matching puzzle video game where a players objective is to guide pieces falling from the upper edge of the screen and try to keep the construction as low as possible. When a row is complete with tiles, it disappears. The game becomes more difficult as it progresses, and ends when the construction reaches the top of the screen. The pieces follow arrow-keys and can be dropped instantly by pressing 'space'.

The game rewards the player basic points for successfully landing a block. Elimination points will be awarded based on the size of the eliminations – the more rows eliminated simultaneously, the better. Points will also be awarded for every step a piece falls due to the players actions. The points are calculated as follows:

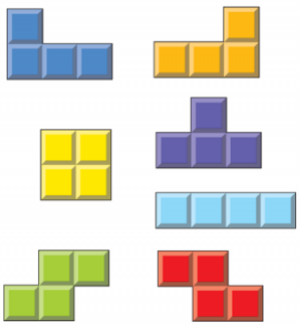
$$\text{points} = a * \text{basic points} + b * \text{elimination size} + c * \text{speed},$$

where a, b, c denote constants that can be tweaked for a desired point-award system.

After each game a player will have the option to submit his/hers high score and view leaderboard standings. This information will be stored locally in a text file. The speed of the game will increase specifically as a function of the amount of eliminations and not of the amount of eliminated rows.



(a) Classic tetris



(b) Possible tetramino shapes

The special mode of play will introduce 1, 2, 3, and 5-tile-size pieces into the game. The drop-rate of these special blocks will be also a function of the amount of eliminations. This variation of the game is called Pentis and it is considered

more challenging. Tetris is faster that Pentis, but only has 7 different blocks. Pentis has 29 different pieces overall. The point-award system and game mechanics will be identical to Tetris. However, the speed-up on Pentis will be slower since it already challenging. A separate leaderboard will be allocated for the Pentis game-mode.

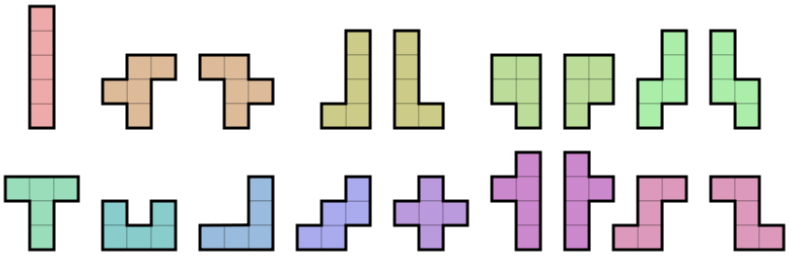


Figure 2: Possible pentomino shapes in Pentis

1 Class hierarchy

This is a preliminary and rough hierarchy of the classes in our game.

1.1 Game

On the highest level we have a class Game. Game objects will have the following functions

- gravity()
- end()
- tryDestroy()
- freeze()

Gravity() is the function for advancing the pieces to the next spot in the vertical axis. This function is called in the main-loop every specific amount of time or increasingly faster depending on the final version. End() function just ends the game once the game area fills up. The tryDestroy() function checks if there are rows to remove, removes them and brings the layers above them down if there are any. Freeze() tries to stop the moving pieces on top of the ones below.

1.2 Grid

Grid objects live in Game objects. Every Game object has exactly one Grid. This class might be scratched as the information of the cells and their positions and qualities can be stored in the game objects. Grid will have the following functions:

- checkEnd()
- isTaken(coords)
- add(coords)

The checkEnd() function checks if the game area is full. This will be called in the main-loop before every Game objects gravity() function. The main purpose of Grids is to keep track of what is and where it is. The isTaken() function tells what if anything exists in given coordinates. Add(coords) and delete(coords) functions together are the interface in which game objects modify the game area.

1.3 Cell

Cells represent the squares in the grid. Cells contain the information of what type they are i.e. wall, empty or tile. In addition to the type they also contain a reference to the color or picture of the cell. Cells will have the following functions:

- pic()
- type()

Pic() function returns the information of how the cell looks like. Type() function returns one of the types mentioned above.

1.4 Pentamino

Pentaminos represent the moving pieces regardless of the amount of pieces they are made of. Pentaminos store the information of what shape they are. Pentaminos aren't made of cells in the underlying logic but in the gui they look the same they are turned into full cells once they are frozen into a place. Pentaminos will have the following functions:

- turn()
- shape()
- position()
- move(dir)

Turn() function changes the orientation of the pentamino by 90 degrees. Shape() returns the shape of the pentamino. Position() returns the position of the pentamino and move(dir) moves the pentamino in the specified direction.

2 Pentamino logic

2.1 Pentamino representation

The pentamino pieces (and trivially the pieces composed of less than five tiles) can be represented in a 5 by 3 binary array where a cell is given a value of 1 if it contains a tile. The empty cells are given a value of 0. Such array is depicted in Table 1. For example, when the array cells are enumerated as in Table 1, filling in the cells 1, 4, 7, 10 and 13 yields a 5-tile line piece. The table also depicts the default rotational axis, cell 7, which is viable for the majority of the pieces in game. Thus, the pieces can alternatively be stored using each tile's offset from the default rotational axis.

Table 1: A grid representing the array that holds the individual pentomino shapes. The cells are enumerated for demonstration purposes and the colored cell 7 depicts the default rotational axis.

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

When the in-game-location of the rotational axis and the orientation of a certain piece are known, the whole piece can be uniquely reconstructed on the grid. The in-game-location is an ordered pair, such as (x, y), where x represents the horizontal displacement and y the vertical displacement of the piece. The orientation is an element belonging to a ring of integers modulo 4 $\mathbb{Z}/4\mathbb{Z}$, where 0 represents the default orientation and addition of 1 a 90 degree clockwise turn. The horizontal displacement of the active piece is increased by the clock and the player, and the vertical displacement by the player. The rotational value of the piece is altered by the player.

When either the location or the orientation of the piece is to be altered, an altered copy of the original piece is reconstructed on the grid. If none of the active tiles is obstructed by a static tile, the location and rotation of the copy are stored as the current location and rotation of the piece. Otherwise, the location and the rotation are unaltered.

2.2 Exceptions

When the active piece drops on something, i.e. the vertical displacement of the piece is altered such that it would have collided with a static tile or the floor of the grid, the active piece is converted as a collection of static tiles.

When the player attempts to rotate a piece, but the position it would normally occupy after basic rotation is obstructed, (either by the wall or floor of the grid, or by a static tile), the game will attempt to "kick" the piece into an alternative position nearby.

2.3 End of turn

When the active piece drops on the stack of static tiles or on the floor of the grid, a turn ends. This triggers the following turn of events: first, the grid is inspected for complete horizontal lines. If complete lines are found, they are removed. Then, the contents of each line above the removed lines are lowered accordingly. Finally, a new turn begins and a new active piece starts its descend.

Roles in group

Roles in the group are not strictly settled and they may and will vary as the project progresses. The initial roles are chosen based on personal intrests of the participants.

Due to the size of the assignment and the group, the roles are more than probably going to mix up.

Kalle Alaluusua

- **Team Lead**
- **Software Architect**

Juho Heimonen

- **Software Engineer**
- **Tester**

Rustam Latypov

- **Software Engineer**
- **UX Designer**

Visa Lintunen

- **UI Designer**
- **Tester**

Preliminary schedule

Schedule is planned to get a rough outline for the progression of the project. Estimated weekly work hours per group member are 6-8 hours. In case some harder problems occur, the work hours can be expanded.

- **Week 1 (12.11.-18.11.):**

Designing the basics of the game. Deciding the main functionalities in-game and options the player is able to affect.

- **Week 2 (19.11.-25.11.):**

Final decisions about the mechanics, player controls and the object interaction. Rough visuals and pseudo-code/code. Structure of the code written as comments in plain English.

- **Week 3 (26.11.-02.12.):**

Finishing rough alpha-phase version of the game. Debugging and refining the game mechanics. Implementing the scoring system and the menus.

- **Week 4 (03.12.-09.12.):**

Designing and implementing the final visuals of the game. Debugging. Beta-phase version out for testing with people outside of the group.

- **Week 5 (10.12.-14.12.):**

Polishing the code and visuals. Final version 1.0 out.