

Report

Tetris like Game

Abstract

This report presents the process of developing a “Tetris” like game in python explaining the main steps of the project. It also gives some possible improvement that could be done to the game to add more enjoyable possibilities for the player.

Introduction:

The idea of this project was to develop a Tetris like game to assess the skills learned during Python Programming course and to continue learning by practicing.

There have been a few steps for this project:

| | |
|---|----------|
| Introduction: | 1 |
| Development: | 1 |
| Definition of necessary class | 1 |
| Generating board and shapes | 2 |
| Creating the Board: | 2 |
| Defining the Shapes: | 2 |
| Creating a New Shape: | 2 |
| Shapes movement and Collisions: | 3 |
| Updating the Current Shape: | 3 |
| Drawing the game board and interface | 3 |
| Counting and storing points | 4 |

Development:

Definition of necessary class

Two different classes are used for this project. The first one is the **Tetris** class that serves as the central component for managing the game board and its associated functionalities. It encompasses various key aspects of the Tetris game, including the creation and maintenance of the game board itself. Additionally, the Tetris class is responsible for defining and handling the different Tetris shapes that appear during gameplay. The shapes are objects of the class **Shape**. This class is used to generate a random shape that can be placed on the game board.

Generating board and shapes

Creating the Board:

To create the game board, the **createBoard** method in the **Tetris** class is used. It initializes the **board** variable as a 2D list with dimensions defined in the **settings** dictionary. Each cell on the board is initially set to 0, representing an empty cell. The board serves as a grid where the Tetris shapes will be displayed during gameplay.

Defining the Shapes:

The Tetris shapes that are available are defined as matrices in the **shapes** list. Each shape is represented by a 2D list, where 0 represents an empty cell and a different number indicates a filled cell. Each type of shape uses a different number to define a filled cell, this situation was decided to make it easier to differentiate each shape in the board and to have a way to easily give the same color to all the shapes of the same

type. These matrices define the initial configuration of each shape on the board. The shapes can also be rotated to change their orientation during the game.

Creating a New Shape:

When a new shape is needed, the `newShape` method is responsible for generating a random shape. It selects a shape randomly from the `shapes` list and sets its initial position at the top center of the board. Each shape has its own coordinates that can change during the game. Additionally, the `next_shape` attribute is updated to indicate the shape that will appear in the next move. This attribute of the Tetris class will be used later to draw the preview of the next shape as in the Tetris game.

Shapes movement and Collisions:

As in the Tetris game, every game tick the current shape will drop by one cell until it can't due to reaching the end of the board or colliding with an already fixed cell. The user can also make the cell go on the left or the right to position the shape to his will. However, collisions have to be implemented for the game to work correctly. There are two types of collisions:

- Collision with the board edges, that can be managed using the length of the piece, the direction it wants to go and its position. Computing this information allows the game engine to detect if the shape tries to go out of bounds.
- Collision with a fixed shape, which is managed by looking at the direction of the shape and if there is at least one free cell for each "side cell" of the shapes.

The detection of collisions is managed using the `check_collision` method. Every time the shape tries to move (game tick or user input) or to rotate the method checks if the movement can be done and then allows or rejects it.

Updating the Current Shape:

To update the current shape on the board, the `updateCurrentShape` method is used. It uses the `lockedboard` attribute as a base for the current state of the game and then positions the current shape on the grid accordingly. This method is the last method that is called before dealing with the printing of the game interface.

Drawing the game board and interface

Drawing the game is separated between drawing the board and the menu of the game. For the board the engine bases the drawing on the `board` attribute since it represents the current state of the game. Each cell is drawn as a square of color, black for an empty cell and a specific color with white square inside for the cell of a shape. For the menu we need to print a few things but the idea remains the same. The game draws the preview of the next shape using the same system as to draw the board. It also draws the name of the game and the current top 3 high scores. Since the game is updated every tick we render the game every time (at a 30 fps frequency) to have a fluid looking game.

Counting and storing points

As a scoring system the game uses the basic Tetris scoring system described in the Tetris wiki.

Every time a shape is fixed due to collisions the engine calls the `clean_lines` method that checks if a row of the board is full of shape cells. If so the row gets destroyed and the upper rows are moved down by 1. The methods return the number of rows that were destroyed in one iteration to compute the number of points that the player earned with this move. At the end of each game the 3 best scores are stored into a file to let player track his best score of all time.

Dealing with user input

To manage user input inside the game I used the implementation of user input available in Pygame. The Pygame instance detect every keyboard/mouse event the user do and we can use this to launch specific event depending on which key is pressed.

Conclusion:

As a conclusion, we can say that this little project was a success because I was able to use the knowledge I gained from the course to build my game. Still it would be possible to improve some features like the GUI design or the scoring system for the game to be more enjoyable. We could also create new functionalities like a player versus player mode using sockets and threads (or asynchronous functions). I didn't have time to implement this due to the project deadline and other projects but that's definitely possible. However in this case I think the code should be splitted in different modules to keep a better visibility over all the functionalities. I didn't do it for this first version as the amount of code was still acceptable in one file but the architecture using Class would allow us to split the code.

Bibliography:

<https://tetris.wiki/Scoring>

<https://www.pygame.org/docs/> : Pygame documentation

Inspiration: "quadrapassel" game on Linux Ubuntu