Tetris Game

Final Project Report

Richard Hans (2802516384)
L1BC
Jude Joseph Lamug Martinez MCS

Algorithm and Programming
Binus International University
Jakarta

Abstract

This report was created as documentation of the Algorithm and Programming final project used to showcase a students coding skills within the language Python. This specific project written in this report will refer to a simple hangman game with an american sign language detector as the letter inputs from the user.

Introduction

This final project extends beyond the topics covered in class to provide a meaningful challenge and an opportunity to apply Python programming skills in a creative and practical way. The goal of the project is to develop a fully functional Tetris game, a classic puzzle game that encourages logical thinking and quick decision making.

Project Inspiration

The idea for the tetris project came from wanting a fun & classic game with the challenge of applying my python programming skills, why I choose tetris? Because it's the game that has been loved for years, and it offers a great between being simple to play but complex to code. That's why I decided it's the perfect choice for a project that tests problem solving capability and creativity.

The inspiration started with a simple question, What makes a game fun and also a good way to learn programming? Tetris was the answer because it has its interesting challenges like grids, handling shapes, and making everything work smoothly with the player inputs. It's also a game that most of the population in the world knows, which is more exciting to create.

On a personal level, Tetris reminds me of my childhood when I used to play it with family and friends. This project gave me the chance to relive those memories while building something from scratch.I also wanted to push myself beyond what I learned in class, exploring new ideas like designing with object oriented programming and making the game interactive and user friendly.

This project is not just about creating a game, it's about learning how to plan, adapt, and solve problems along the way. Just like Tetris requires quick thinking to fit the blocks together, I had to figure out how to bring all the parts of the code together to make a complete, working game, and it's been a fun and rewarding journey.

Development Plan

The tetris game is made to make sure that the modularity and maintainability while delivering a seamless user experience. The solution is to build around *python and pygame*, with separate modules oto handle the different aspects of the game, such as grid management, Tetrimino behavior, user interaction, and scoring. This modular approach makes sure that each component functions independently while working together to make a cohesive game system.

Grid system is the main backbone of the system where it is implemented in the grid.py, it would track placement of blocks, clear rows when they are full, and shift the remaining blocks downward. Function as *is_row_full, clear_row,* and *clear_full_rows* makes sure that the grid behaves as expected during the gameplay The grid system also handles all the boundary checks and makes sure that new blocks will spawn in valid position.

The Tetromino management system, found in *blocks.py*, defines each Tetromino's shape, rotation states, and movement behavior through dedicated subclasses of the Block class, such as *LBlock, TBlock, IBlock, and more*. Each shape is represented using a dictionary of rotation states (self.cells), where the keys correspond to the rotation index, and the values contain a list of Position objects representing the block's cells in the grid. Some shapes, like the *OBlock*, do not rotate and have a single state, while others, like the *IBlock or LBlock,* have multiple rotation states to account for their unique behaviors. Each subclass also initializes its starting position using the move method. The modular design allows for precise control over how shapes rotate, move, and interact with the grid. Combined with the inherited methods like *rotate, move, and undo_rotation* from the Block class, the system make sure smooth movement, accurate rotations, and proper alignment of blocks within grid boundaries, avoiding collisions and making sure gameplay consistency.

The game loop, implemented in *main.py*, manages the overall flow of the game. It continuously updates the screen, processes user inputs, and moves blocks downward at timed intervals. Real-time inputs allow players to move blocks left, right, or down and rotate them with ease. The loop also handles game-over conditions by checking whether new blocks can be placed on the grid.

- Game Components

- The Tetris game features a modular structure that ensures smooth gameplay.
- The main game loop manages updates to the grid, processes user inputs, and moves Tetrominoes dynamically.
- A grid system tracks the positions of blocks, clears completed rows, and handles falling blocks effectively.
- Tetromino pieces are designed with unique attributes for rotation, movement, and interaction with the grid.
- Player controls allow responsive and straightforward interactions, ensuring an enjoyable experience.

- Tetromino Features

- Each Tetromino has distinct rotation configurations, enabling seamless movements within the grid.
- Pieces are visually unique, with specific colors assigned for easy identification.
- The interaction between Tetrominoes and the grid ensures compliance with game rules, maintaining smooth gameplay.
- Collision detection prevents overlaps and invalid movements, allowing proper alignment of blocks.

- Player Interaction
- The game provides simple controls for moving blocks left, right, and downward.
- Rotation mechanics enable players to adjust the orientation of Tetrominoes strategically.
- Responsive inputs ensure quick and accurate actions, maintaining the flow of the game.

- Design Challenges
  - Developing a dynamic grid management system to handle row clearing and shifting blocks efficiently was a key challenge.
  - Implementing rotation logic for Tetrominoes while respecting grid boundaries and avoiding overlaps required meticulous planning.
  - Ensuring real-time responsiveness to player inputs without lag needed optimization of input handling and processing.
  - Designing an end-game condition that reliably detects when no new blocks can be placed added to the complexity.

- Future Improvements

Difficulty progression could be added to increase the speed of falling blocks as the game time goes longer and more points being added. **Power-Ups** could also be introduced for unique abilities, such as clearing multiple rows at once. A team **leaderboard system** would track and display all the high scores. Adding **animation** could also be an option.

Discussion

The main game loop, located in *main.py,* is the central hub for running the tetris game. It begins by starting Pygame using the *pygame.init()* function to load all requirements . The game window is created using *pygame.display.set_mode(),* setting the display size to support the grid and Tetrominoes.A Pygame *Clock* object is used to manage the frame rate, making sure that the game runs smoothly at 60 frames per second. . The loop itself handles user input, updates the grid and active Tetromino, and redraws the screen on every loop. The program terminates smoothly using sys.exit() when the player closes the window. To manage the gameplayflow, the main loop would coordinate with other modules to get the grid current status, determine scoring updates, and check for game over conditions. Input handling is a key part of the loop, capturing player actions such as moving or rotating Tetrominoes in real time.

The system is implemented in *grid.py* where it is the backbone of the tetris game where it manages block placement, row clearing, and grid updates in real time to ensure smooth gameplay. It has a size of 20x10 two-dimensional list where each cell represents a grid space, starting as empty (value 0) and becoming filled when a Tetromino is placed. The system has key methods like *is_inside* to verify boundary conditions*, is_empty* to check for unoccupied cells, and *is_row_full* to identify fully occupied rows eligible for clearing. The *clear_row* and *clear_full_rows* methods work together to remove completed rows and shift blocks above downward, making sure that the grid remains organized and dynamic. The *move_row_down* method handles these shifts, preserving gameplay flow even when multiple rows are cleared simultaneously.The *draw method* visualizes the grid by dynamically updating cell colors using Pygame, making each Tetromino's placement visible. It makes collision detection closely coupled such that, besides disallowing impossible movements or rotations of Tetrominoes, all the game rules have been strictly followed. Resetting the grid with the reset method resets all cells to their default state, preparing the board for a new game. The grid's modularity allows it to interact smoothly with other game components like Tetromino classes and the main game loop, supporting score calculation and game over conditions. This detailed yet efficient design highlights the complexity of managing dynamic gameplay while making sure that the performance remains optimal, even at higher speeds or under intensive player inputs.

The tetromino management system at blocks.py, shows the behavior, structure, and interaction of each unique Tetris piece, making it an important component of the game's functionality. Seven subclasses *LBlock, JBlock, IBlock, OBlock, SBlock, TBlock, and ZBlock,* containing a shared functionality like movement, rotation, and setup. Each subclass is coded with specific attributes, including rotation states stored in dictionaries, where each key represents a rotation index and the value is a list of Position objects that define the shape's configuration on the grid. This modular design simplifies the management of different shapes while making sure that there is precise control over how each Tetromino rotates and moves. The rotate method cycles through these states, updating the piece's orientation dynamically, while the move method adjusts its position within the grid. The undo_rotation makes sure that no invalid rotations are reverted, particularly when collisions or boundary violations occur. Colors for each Tetromino are assigned in *colors.py*, using RGB. The system interacts closely with the grid, using methods like *is_empty* and *is_inside* to validate movements and rotations, making sure that all actions follow the game's rules. The starting position of each Tetromino is adjusted during the start to account for its size and shape, making a seamless spawning at the top of the grid. The dynamic design of the Tetromino management system not only maintains smooth interaction with the grid but also modularity, making it easier to test, debug, and extend the functionality for future features like new shapes or unique block mechanics.

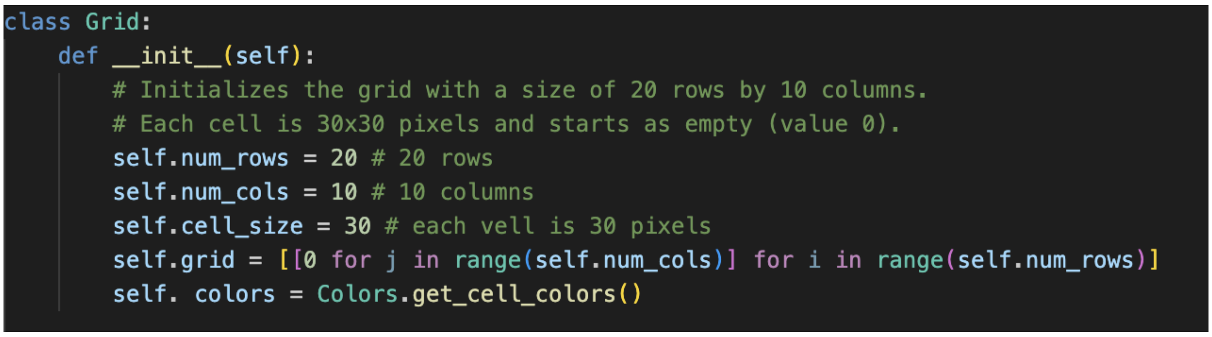
The game logic and scoring system is placed at *game.py* form the foundation of the Tetris gameplay by coordinating player actions, enforcing rules, and tracking progress. Central to the game logic is the integration of collision detection, which ensures that Tetrominoes interact properly with the grid, preventing movements or rotations into occupied or out of bounds cells. The centre of the game logic is the integration of collision detection which ensures that Tetrominoes interact properly with the grid, preventing movements or rotations into occupied or out-of-bounds cells. The game also incorporates mechanics for placing Tetrominoes, *clearing full rows*, and *spawning new pieces*, creating a continuous flow of gameplay. The scoring system is designed to reward players for their efficiency, with points awarded based on the number of rows cleared in a single move. More points could also be gained when accelerating block descent (Pushing down the down button), rewarding players to take calculated risks for higher scores.Game over conditions are implemented by checking whether new Tetrominoes can spawn at the top of the grid,if not, the game ends, showing that its the end of the game. The integration of these elements ensures a seamless balance between challenge and progression, while the scoring system provides a real measure of player performance and improvement. This cohesive system not only defines the pace and flow of the game but also lays the groundwork for potential upgrades to the game, such as level progression or competitive leaderboards, to further giving more of the player experience.

The multiple modules that are working with Pygame, they have clear and interactive features of the Tetris game which makes the game really alive and energetic. The visual contents of the game are controlled by functions such as draw which are manipulated in the Grid and Block classes to dynamically render what game state is at the screen. Each grid cell is represented like a certain rectangle, through which colors depending on its occupation with a Tetromino or lack thereof provide excellent feedback to the player. The Tetriminoes are recognized clearly as distinguishable colored ones in colors.py. The system gets updated in real time to reflect every player action such as moving or rotating a piece as well as to visually signal cleared rows immediately on the grid.

Pygame's event handling for player input appeals to interactive and fluid gameplay. The arrow keys make possible the movement of Tetrominoes toward the left side, toward the right, and downward, while upward arrow invokes a rotation on the Tetromino. This input method would give precision and fast responses so that the player can feel comfortable making strategic decisions during fast-paced gameplay. The game loop also continually listens for inputs and integrates them seamlessly in the system's game logic without interruption or lag.

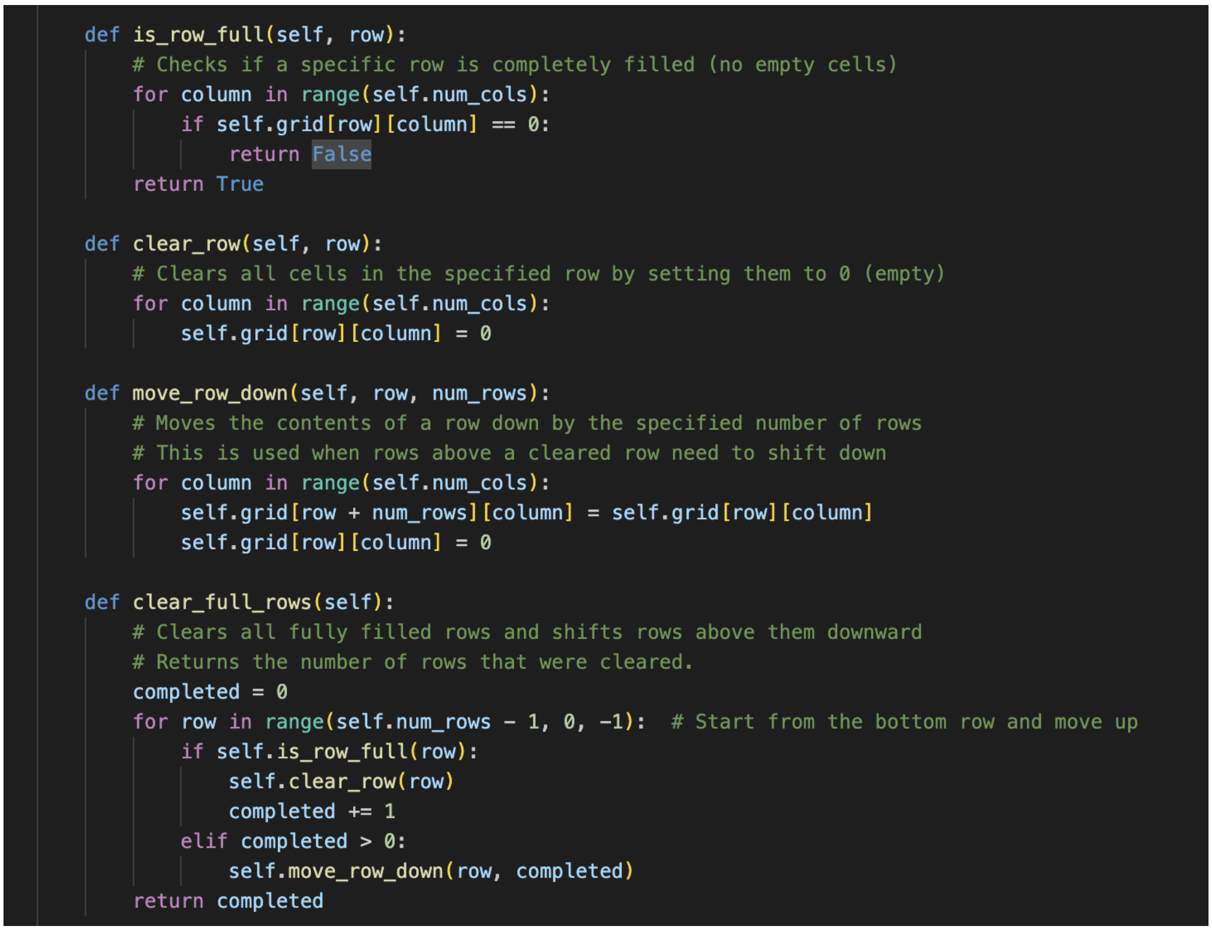
Screenshots

1. Grid Initialization (grid.py)



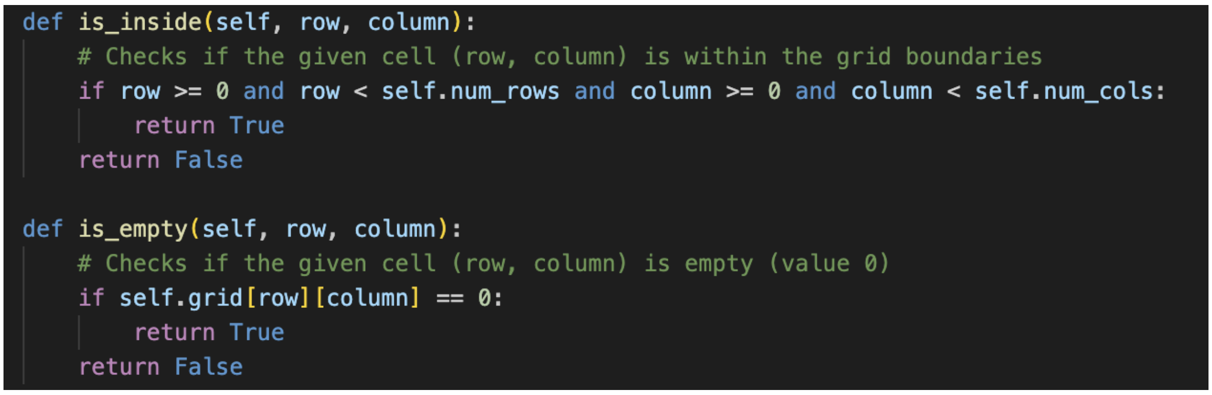
```python
class Grid:
    def __init__(self):
        # Initializes the grid with a size of 20 rows by 10 columns.
        # Each cell is 30x30 pixels and starts as empty (value 0).
        self.num_rows = 20 # 20 rows
        self.num_cols = 10 # 10 columns
        self.cell_size = 30 # each vell is 30 pixels
        self.grid = [[0 for j in range(self.num_cols)] for i in range(self.num_rows)]
        self. colors = Colors.get_cell_colors()
```

2. Row Clearing Logic (grid.py)



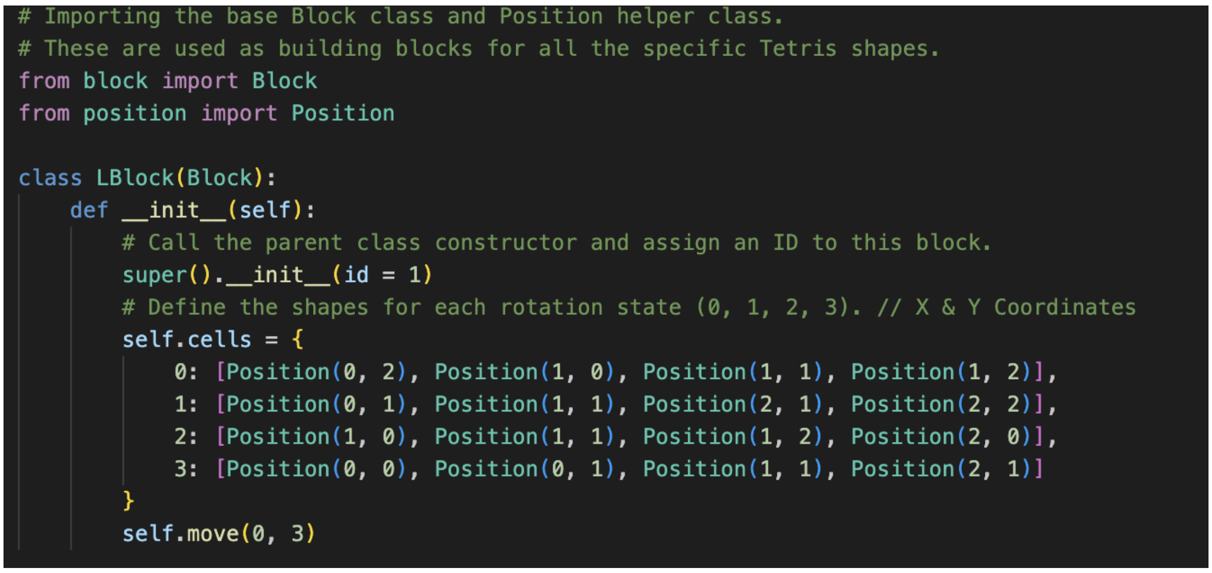
```python
    def is_row_full(self, row):
        # Checks if a specific row is completely filled (no empty cells)
        for column in range(self.num_cols):
            if self.grid[row][column] == 0:
                return False
        return True

    def clear_row(self, row):
        # Clears all cells in the specified row by setting them to 0 (empty)
        for column in range(self.num_cols):
            self.grid[row][column] = 0

    def move_row_down(self, row, num_rows):
        # Moves the contents of a row down by the specified number of rows
        # This is used when rows above a cleared row need to shift down
        for column in range(self.num_cols):
            self.grid[row + num_rows][column] = self.grid[row][column]
            self.grid[row][column] = 0

    def clear_full_rows(self):
        # Clears all fully filled rows and shifts rows above them downward
        # Returns the number of rows that were cleared.
        completed = 0
        for row in range(self.num_rows - 1, 0, -1):  # Start from the bottom row and move up
            if self.is_row_full(row):
                self.clear_row(row)
                completed += 1
            elif completed > 0:
                self.move_row_down(row, completed)
        return completed
```

3.Collision Detection (grid.py)



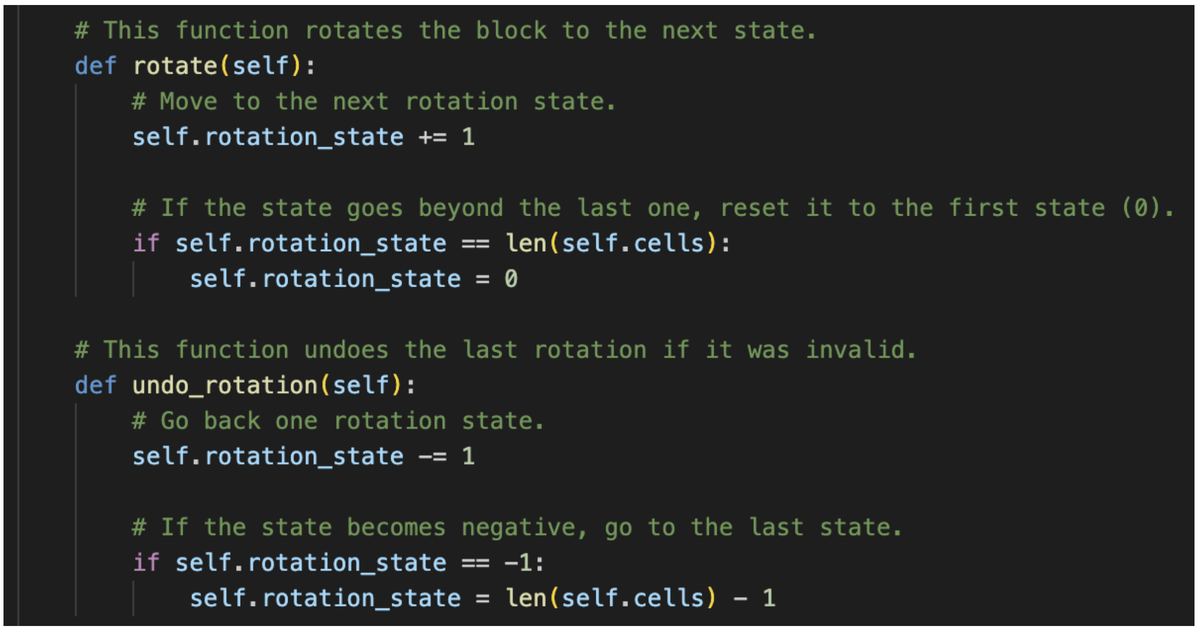
```python
def is_inside(self, row, column):
    # Checks if the given cell (row, column) is within the grid boundaries
    if row >= 0 and row < self.num_rows and column >= 0 and column < self.num_cols:
        return True
    return False

def is_empty(self, row, column):
    # Checks if the given cell (row, column) is empty (value 0)
    if self.grid[row][column] == 0:
        return True
    return False
```

4. Tetromino Class Definitions (Example of the L block) (blocks.py)

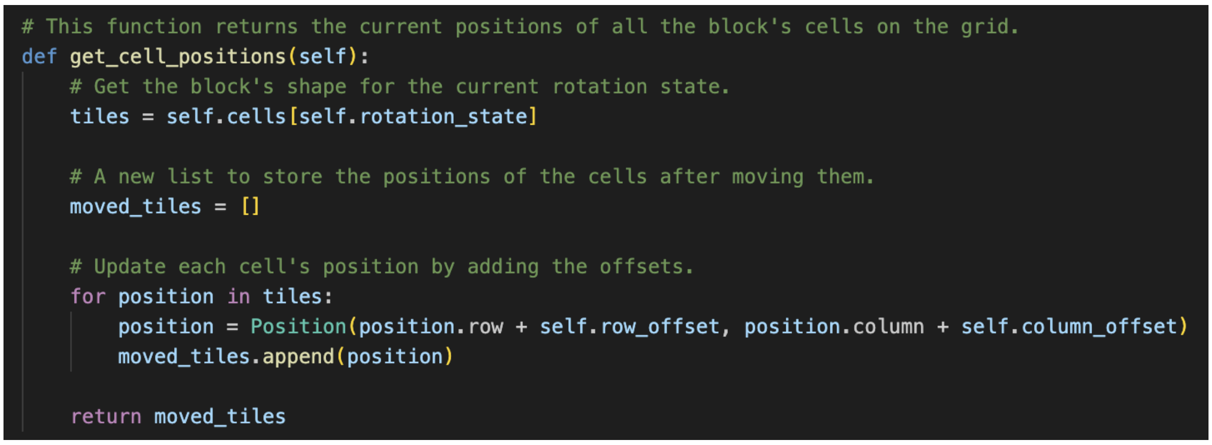


```python
# Importing the base Block class and Position helper class.
# These are used as building blocks for all the specific Tetris shapes.
from block import Block
from position import Position

class LBlock(Block):
    def __init__(self):
        # Call the parent class constructor and assign an ID to this block.
        super().__init__(id = 1)
        # Define the shapes for each rotation state (0, 1, 2, 3). // X & Y Coordinates
        self.cells = {
            0: [Position(0, 2), Position(1, 0), Position(1, 1), Position(1, 2)],
            1: [Position(0, 1), Position(1, 1), Position(2, 1), Position(2, 2)],
            2: [Position(1, 0), Position(1, 1), Position(1, 2), Position(2, 0)],
            3: [Position(0, 0), Position(0, 1), Position(1, 1), Position(2, 1)]
        }
        self.move(0, 3)
```

5. Tetromino Rotation Logic & Cell Position Update (block.py)

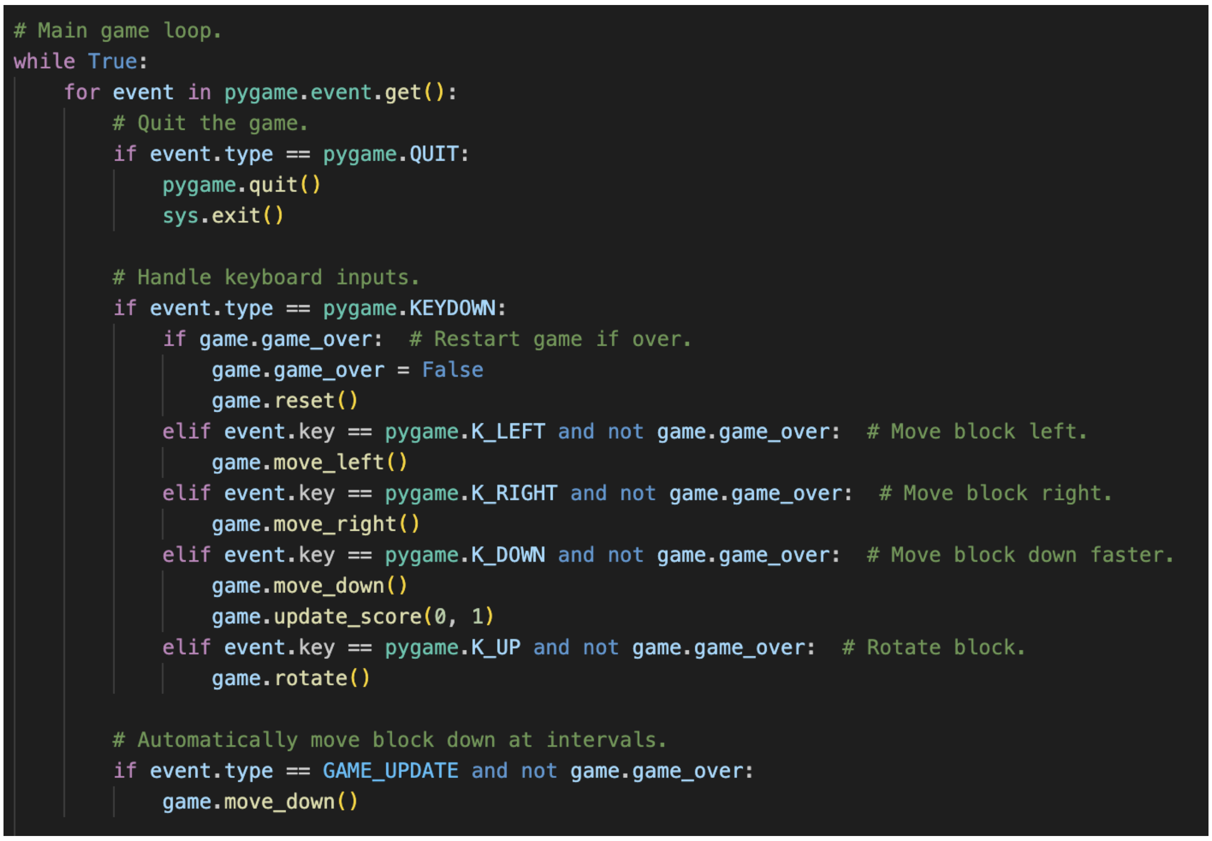


```python
    # This function rotates the block to the next state.
    def rotate(self):
        # Move to the next rotation state.
        self.rotation_state += 1

        # If the state goes beyond the last one, reset it to the first state (0).
        if self.rotation_state == len(self.cells):
            self.rotation_state = 0

    # This function undoes the last rotation if it was invalid.
    def undo_rotation(self):
        # Go back one rotation state.
        self.rotation_state -= 1

        # If the state becomes negative, go to the last state.
        if self.rotation_state == -1:
            self.rotation_state = len(self.cells) - 1
```

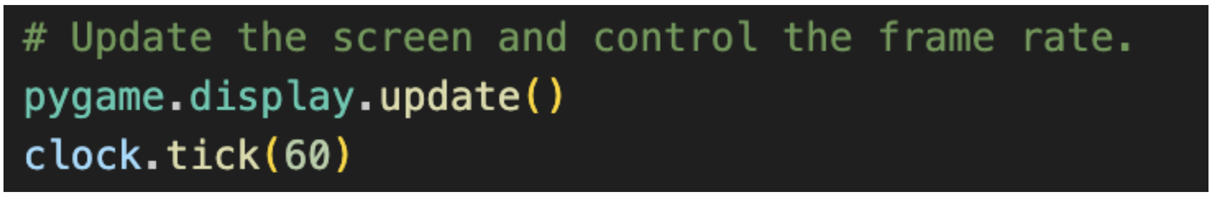


```python
# This function returns the current positions of all the block's cells on the grid.
def get_cell_positions(self):
    # Get the block's shape for the current rotation state.
    tiles = self.cells[self.rotation_state]

    # A new list to store the positions of the cells after moving them.
    moved_tiles = []

    # Update each cell's position by adding the offsets.
    for position in tiles:
        position = Position(position.row + self.row_offset, position.column + self.column_offset)
        moved_tiles.append(position)

    return moved_tiles
```

6. Main Game Loop (main.py)

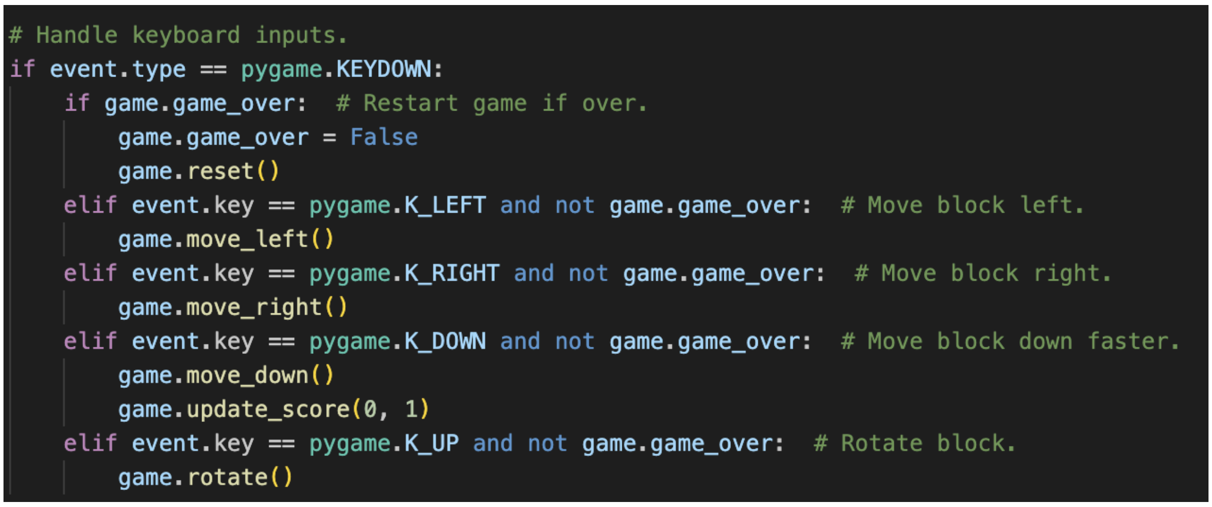


```python
# Main game loop.
while True:
    for event in pygame.event.get():
        # Quit the game.
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

        # Handle keyboard inputs.
        if event.type == pygame.KEYDOWN:
            if game.game_over:  # Restart game if over.
                game.game_over = False
                game.reset()
            elif event.key == pygame.K_LEFT and not game.game_over:  # Move block left.
                game.move_left()
            elif event.key == pygame.K_RIGHT and not game.game_over:  # Move block right.
                game.move_right()
            elif event.key == pygame.K_DOWN and not game.game_over:  # Move block down faster.
                game.move_down()
                game.update_score(0, 1)
            elif event.key == pygame.K_UP and not game.game_over:  # Rotate block.
                game.rotate()

        # Automatically move block down at intervals.
        if event.type == GAME_UPDATE and not game.game_over:
            game.move_down()
```



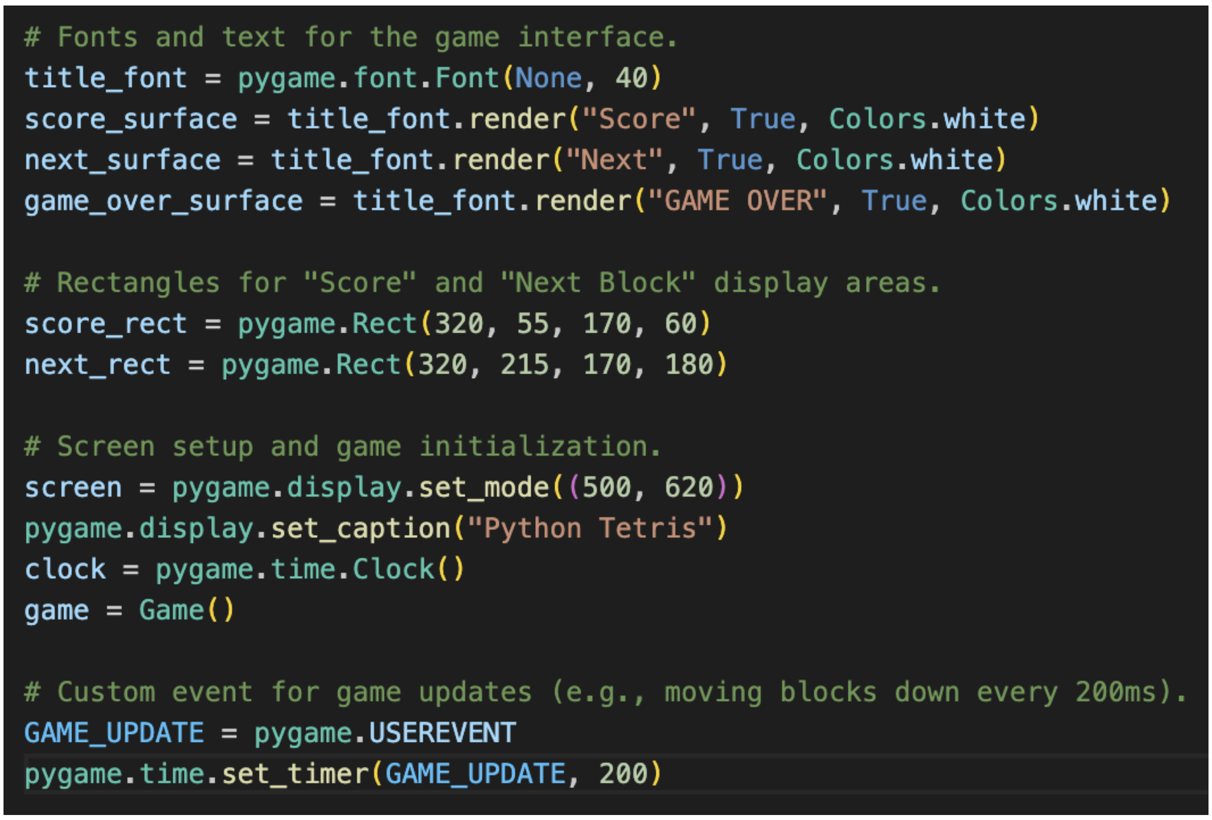
```python
# Update the screen and control the frame rate.
pygame.display.update()
clock.tick(60)
```

7. Player input handling (main.py)



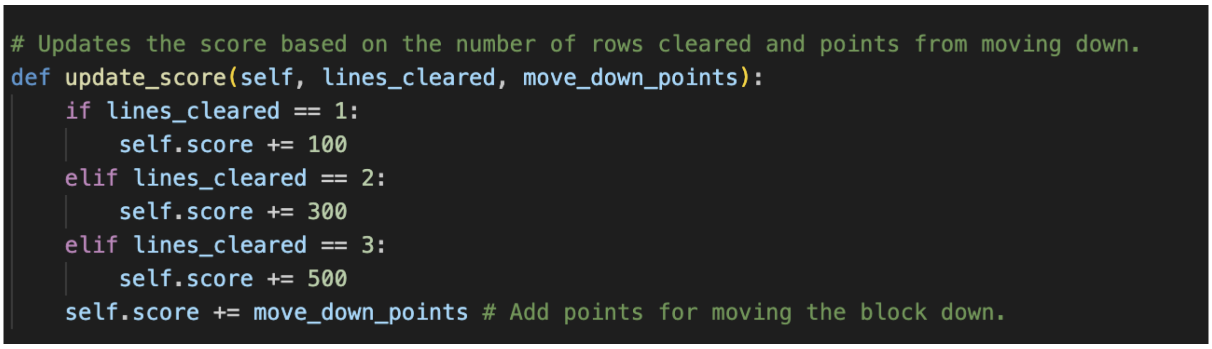
```
# Handle keyboard inputs.
if event.type == pygame.KEYDOWN:
    if game.game_over:  # Restart game if over.
        game.game_over = False
        game.reset()
    elif event.key == pygame.K_LEFT and not game.game_over:  # Move block left.
        game.move_left()
    elif event.key == pygame.K_RIGHT and not game.game_over:  # Move block right.
        game.move_right()
    elif event.key == pygame.K_DOWN and not game.game_over:  # Move block down faster.
        game.move_down()
        game.update_score(0, 1)
    elif event.key == pygame.K_UP and not game.game_over:  # Rotate block.
        game.rotate()
```

8. Game Initialization

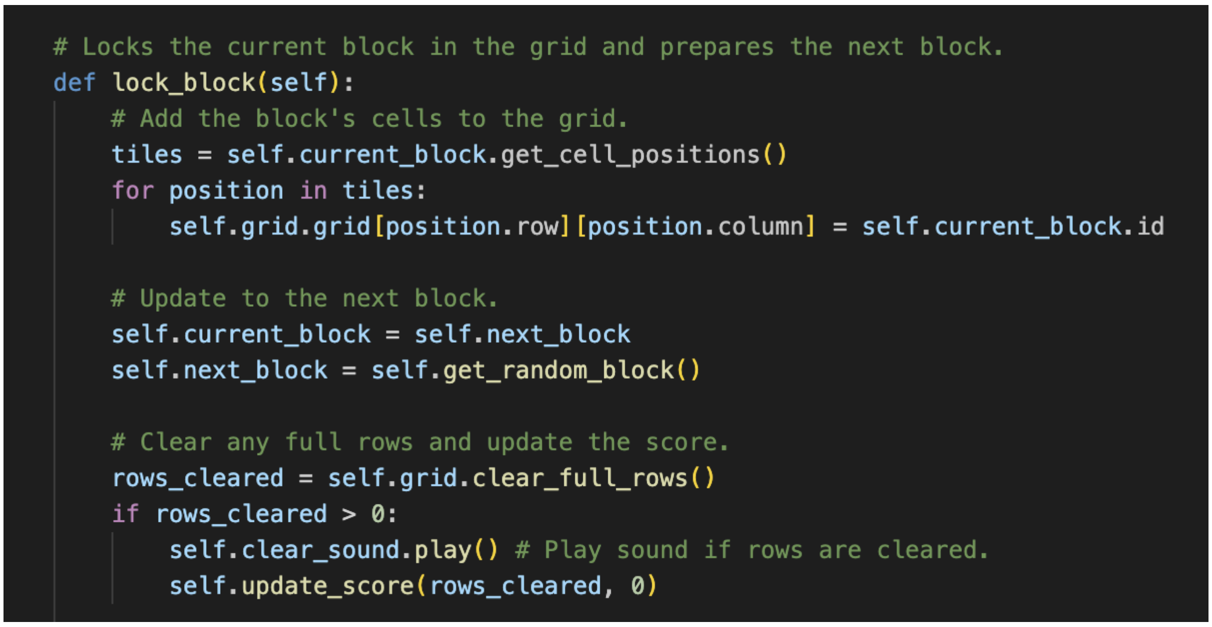


```
# Fonts and text for the game interface.
title_font = pygame.font.Font(None, 40)
score_surface = title_font.render("Score", True, Colors.white)
next_surface = title_font.render("Next", True, Colors.white)
game_over_surface = title_font.render("GAME OVER", True, Colors.white)

# Rectangles for "Score" and "Next Block" display areas.
score_rect = pygame.Rect(320, 55, 170, 60)
next_rect = pygame.Rect(320, 215, 170, 180)

# Screen setup and game initialization.
screen = pygame.display.set_mode((500, 620))
pygame.display.set_caption("Python Tetris")
clock = pygame.time.Clock()
game = Game()

# Custom event for game updates (e.g., moving blocks down every 200ms).
GAME_UPDATE = pygame.USEREVENT
pygame.time.set_timer(GAME_UPDATE, 200)
```

9. Scoring system (game.py)

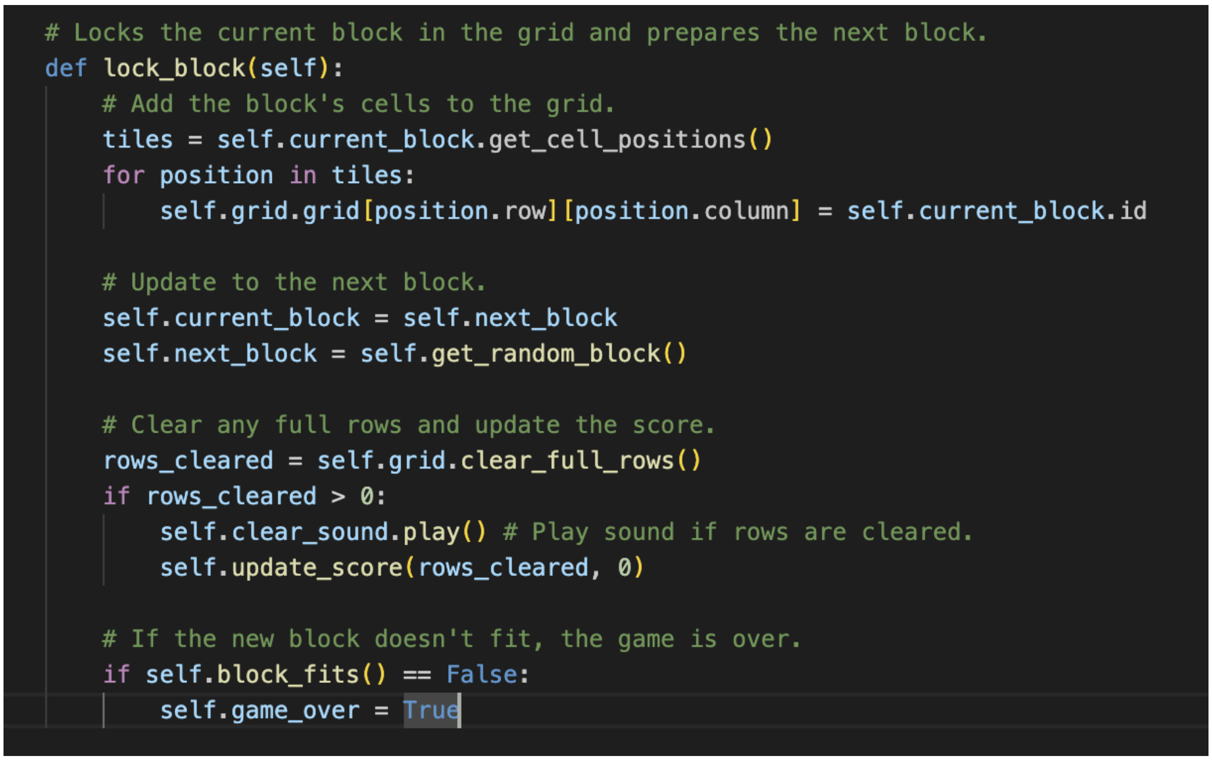


```python
# Updates the score based on the number of rows cleared and points from moving down.
def update_score(self, lines_cleared, move_down_points):
    if lines_cleared == 1:
        self.score += 100
    elif lines_cleared == 2:
        self.score += 300
    elif lines_cleared == 3:
        self.score += 500
    self.score += move_down_points # Add points for moving the block down.
```



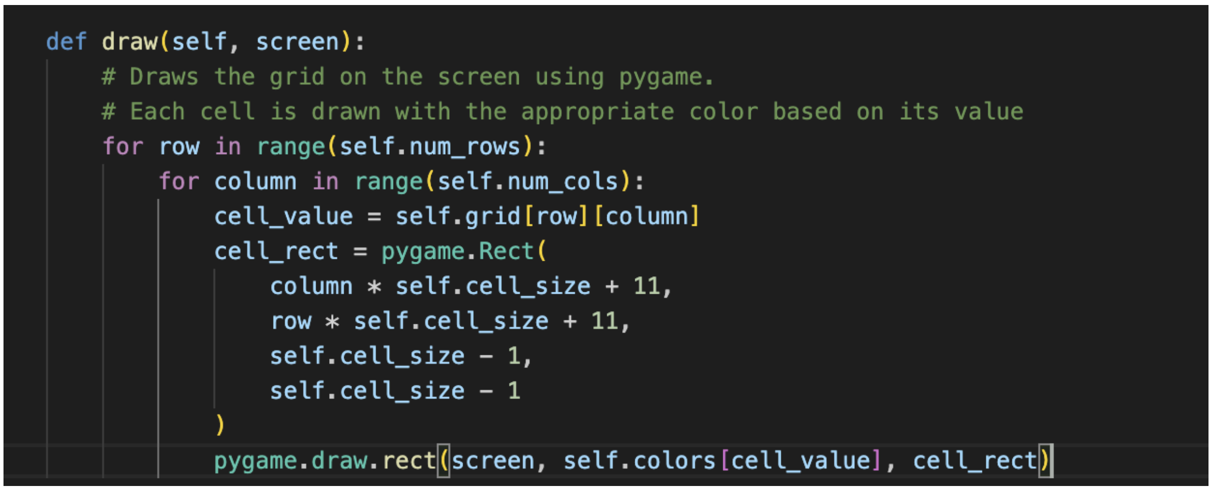
```python
    # Locks the current block in the grid and prepares the next block.
    def lock_block(self):
        # Add the block's cells to the grid.
        tiles = self.current_block.get_cell_positions()
        for position in tiles:
            self.grid.grid[position.row][position.column] = self.current_block.id

        # Update to the next block.
        self.current_block = self.next_block
        self.next_block = self.get_random_block()

        # Clear any full rows and update the score.
        rows_cleared = self.grid.clear_full_rows()
        if rows_cleared > 0:
            self.clear_sound.play() # Play sound if rows are cleared.
            self.update_score(rows_cleared, 0)
```

10. Game over logic (game,py)



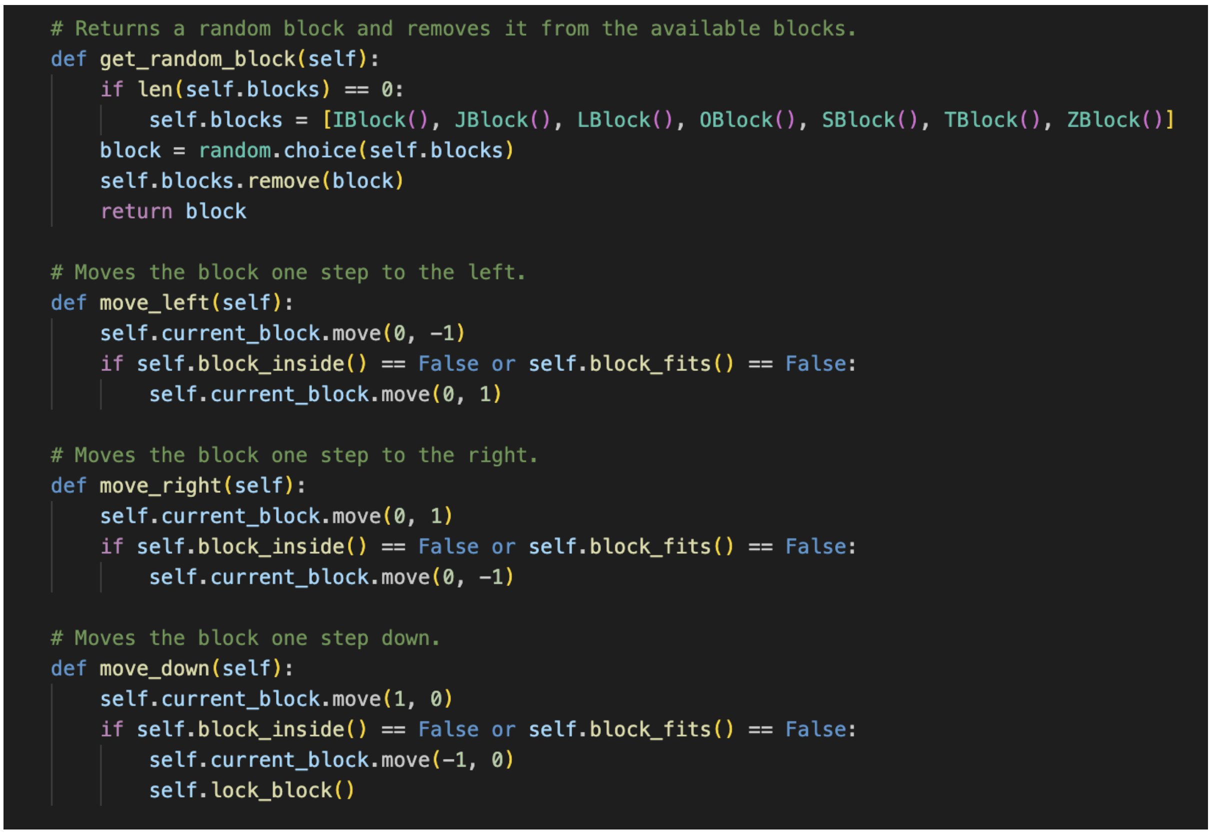
```python
# Locks the current block in the grid and prepares the next block.
def lock_block(self):
    # Add the block's cells to the grid.
    tiles = self.current_block.get_cell_positions()
    for position in tiles:
        self.grid.grid[position.row][position.column] = self.current_block.id

    # Update to the next block.
    self.current_block = self.next_block
    self.next_block = self.get_random_block()

    # Clear any full rows and update the score.
    rows_cleared = self.grid.clear_full_rows()
    if rows_cleared > 0:
        self.clear_sound.play() # Play sound if rows are cleared.
        self.update_score(rows_cleared, 0)

    # If the new block doesn't fit, the game is over.
    if self.block_fits() == False:
        self.game_over = True
```

11, Draw the grid (grid.py)



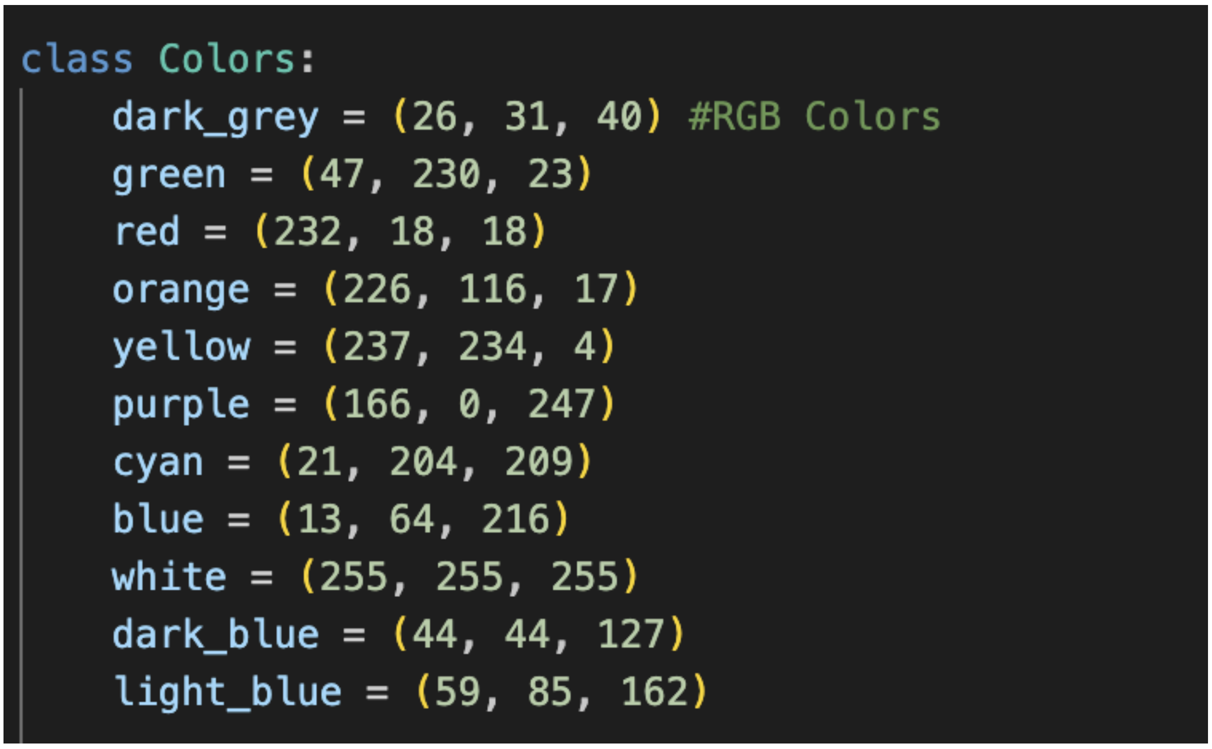
```python
def draw(self, screen):
    # Draws the grid on the screen using pygame.
    # Each cell is drawn with the appropriate color based on its value
    for row in range(self.num_rows):
        for column in range(self.num_cols):
            cell_value = self.grid[row][column]
            cell_rect = pygame.Rect(
                column * self.cell_size + 11,
                row * self.cell_size + 11,
                self.cell_size - 1,
                self.cell_size - 1
            )
            pygame.draw.rect(screen, self.colors[cell_value], cell_rect)
```

12. Tetromino Movement and Placement (game.py)



```python
# Returns a random block and removes it from the available blocks.
def get_random_block(self):
    if len(self.blocks) == 0:
        self.blocks = [IBlock(), JBlock(), LBlock(), OBlock(), SBlock(), TBlock(), ZBlock()]
    block = random.choice(self.blocks)
    self.blocks.remove(block)
    return block

# Moves the block one step to the left.
def move_left(self):
    self.current_block.move(0, -1)
    if self.block_inside() == False or self.block_fits() == False:
        self.current_block.move(0, 1)

# Moves the block one step to the right.
def move_right(self):
    self.current_block.move(0, 1)
    if self.block_inside() == False or self.block_fits() == False:
        self.current_block.move(0, -1)

# Moves the block one step down.
def move_down(self):
    self.current_block.move(1, 0)
    if self.block_inside() == False or self.block_fits() == False:
        self.current_block.move(-1, 0)
        self.lock_block()
```

13. Color management (color.py)



```python
class Colors:
    dark_grey = (26, 31, 40) #RGB Colors
    green = (47, 230, 23)
    red = (232, 18, 18)
    orange = (226, 116, 17)
    yellow = (237, 234, 4)
    purple = (166, 0, 247)
    cyan = (21, 204, 209)
    blue = (13, 64, 216)
    white = (255, 255, 255)
    dark_blue = (44, 44, 127)
    light_blue = (59, 85, 162)
```

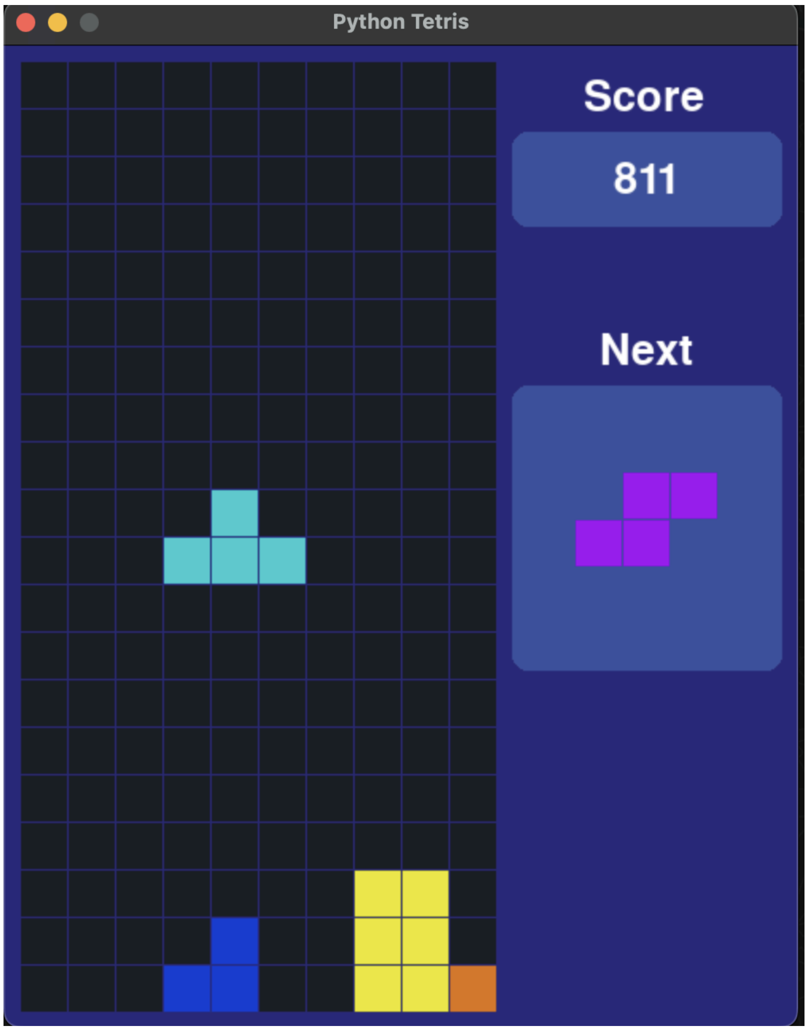
14. Row Shifting Logic (grid.py)

```
def move_row_down(self, row, num_rows):
    # Moves the contents of a row down by the specified number of rows
    # This is used when rows above a cleared row need to shift down
    for column in range(self.num_cols):
        self.grid[row + num_rows][column] = self.grid[row][column]
        self.grid[row][column] = 0
```

15. Position Class (position.py)

```
class Position:
    def __init__(self, row, column):
        # Represents the position of a cell in the grid using row and column.
        self.row = row  # The row index of the cell.
        self.column = column  # The column index of the cell.
```

Gameplay :



Reference :

- https://www.youtube.com/watch?v=nF_crEtmpBo&t=3203s
- https://www.youtube.com/watch?v=uoR4ilCWwKA&t=72s
- https://www.youtube.com/watch?v=JkjiFPNH0Ng
- https://www.youtube.com/watch?v=ROElF_BlUJI