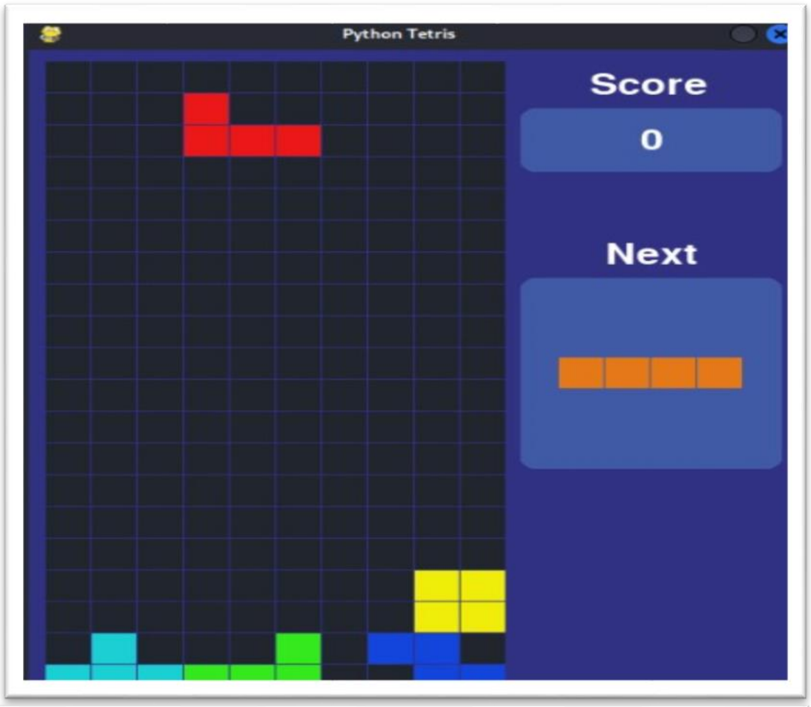
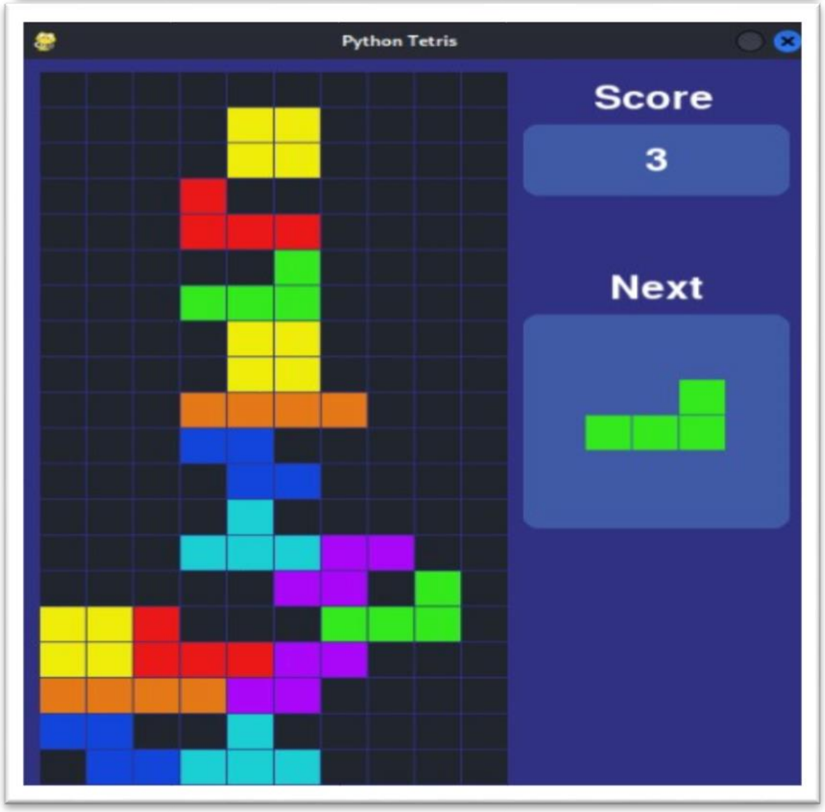
GROUP	**MALIK SAIF ISLAM (35748)** **NASIR HUSSAIN (43913)** **SYED RAFFAY ALI (46894)**
PROJECT REPORT	**TETRIS-STYLE GAME**
TEACHER	**SIR USMAN SHAREEF**
FACULTY	**COMPUTING**
COURSE	**ANALYSIS OF ALGORITHMS**
DEPARTMENT	**CYBER SECURITY**
DATED	**12-Nov-2024**

TETRIS-STYLE GAME

Graphical Representation:





OVERVIEW OF PROJECT:

1. Objective

This project aims to create a functional and visually appealing Tetris game using the Pygame library in Python. The game includes multiple types of blocks, a scoring system, music, and sound effects for an engaging user experience.

2. Technologies Used

- **Python**: Main programming language.
- **Pygame**: For graphical rendering and handling game interactions.

3. Components

A. Class Descriptions

- **Block Class**:
  - Represents the general structure and behavior of any Tetris block.
  - Methods include move (), rotate (), undo_rotation (), and draw ().
  - The get_cell_positions () method determines each cell's position after rotation and translation.
- **Block Subclasses (LBlock, JBlock, IBlock, OBlock, SBlock, TBlock, ZBlock)**:
  - Each subclass represents a specific Tetris block type.
  - Contains unique configurations of self. Cells to define each rotation state.
- **Colors Class**:
  - Defines color constants for various elements (e.g., blocks, grid).
  - Method get_cell_colors () returns a list of colors associated with each block type.
- **Grid Class**:
  - Manages the grid where blocks fall and checks for full rows.
  - Contains methods to clear rows, check if rows are full, reset the grid, and draw the grid onto the game screen.
- **Game Class**:
  - Manages game state, including current and next blocks, score, sound effects, and game-over conditions.
  - Methods include update_score (), move_left (), move_right(), move down(), lock_block(), rotate(), and reset().
  - Draw () method to render current and next blocks, as well as other game elements, onto the screen.

B. Sound Effects and Music

- Sounds for block rotation and row clearance enhance the gameplay experience.
- Background music that loops indefinitely throughout the game.

4. How the Game Works

- **Block Movement**: Players control the blocks' horizontal movement, rotation, and vertical drop.
- **Row Clearing**: Full rows disappear, and the blocks above them shift down.
- **Scoring System**:
  - Points are awarded based on the number of rows cleared at once.
- **Game Over**: Occurs when blocks reach the top of the grid without any more possible moves.

5. User Interface

- The game displays the current score and a preview of the next block.
- The grid's color scheme and responsive block design allow easy gameplay understanding.

6. Future Improvements

- **Difficulty Levels**: Increase block drop speed with score increments.
- **Power-Ups**: Special blocks with unique effects, like clearing multiple rows.
- **Pause Functionality**: Allow players to pause and resume the game.

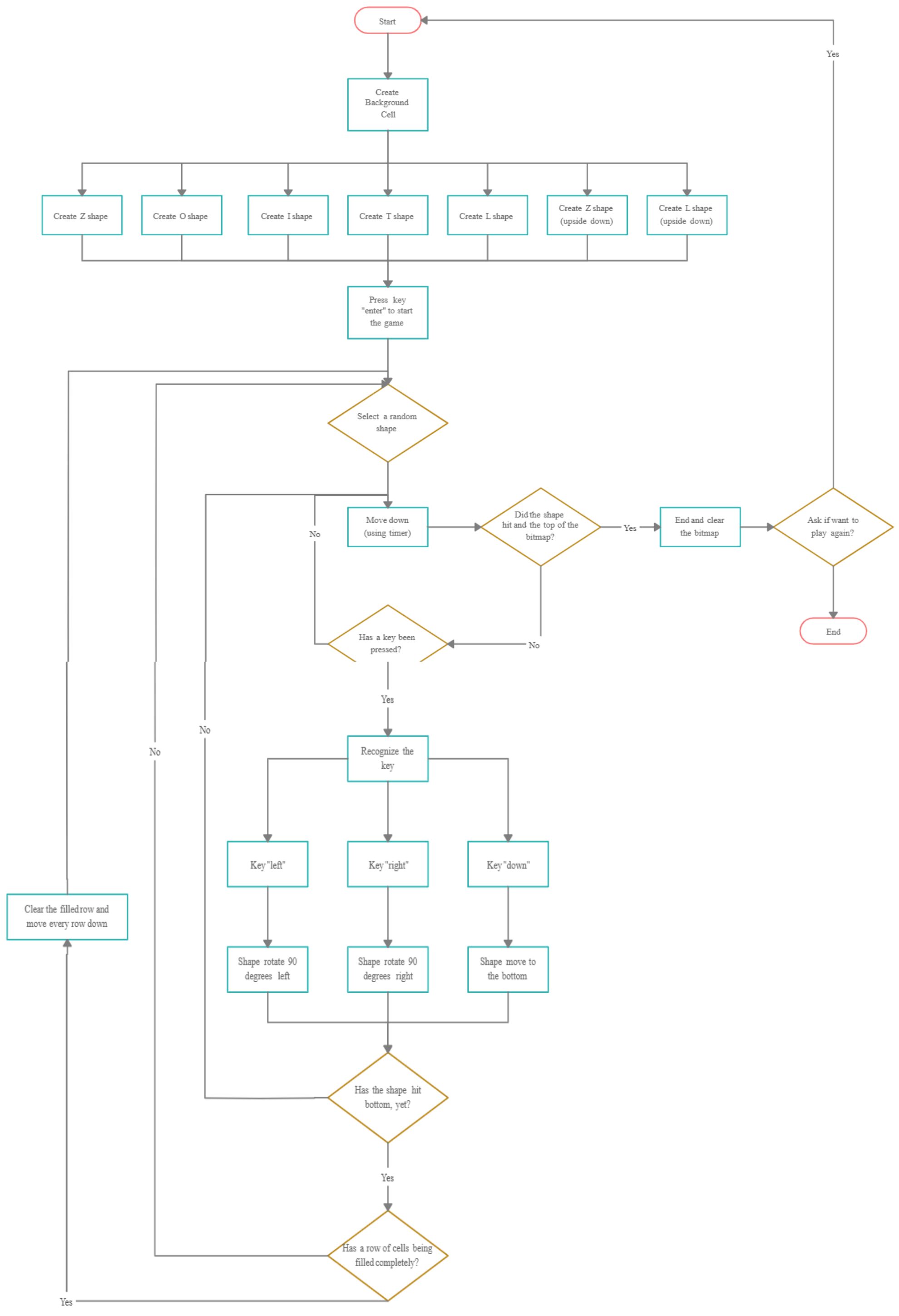
OVERVIEW OF THE GAME:

This game is a Tetris-like puzzle game where players control falling blocks to arrange them within a grid. The objective is to complete horizontal rows without gaps, which will then clear from the grid, allowing more space for additional blocks. The player scores points by clearing rows, and the game ends when blocks stack to the top of the grid.

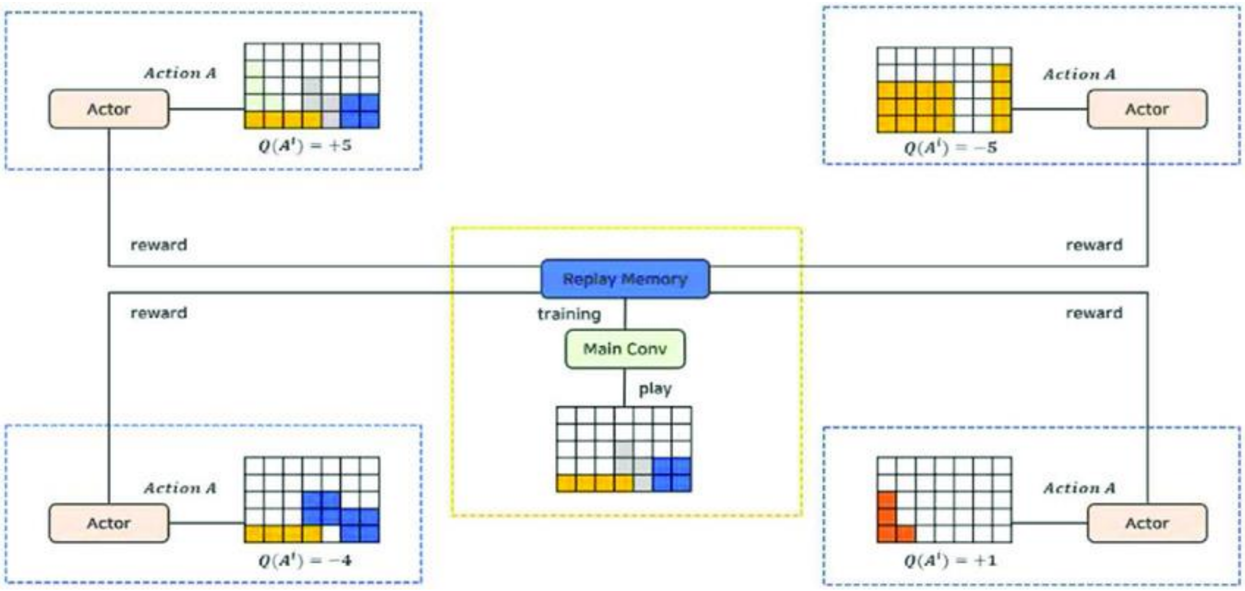
KEY FUNCTIONALITIES:

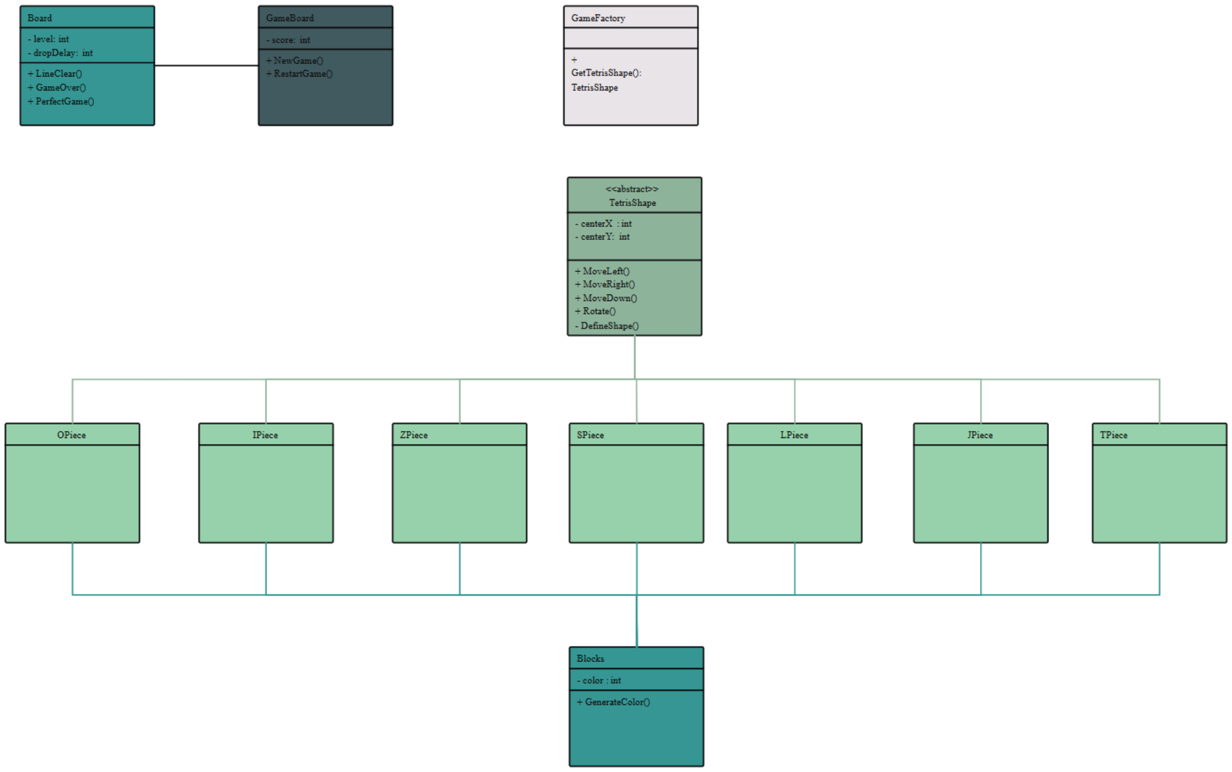
1. **Grid Management**:
   - A grid tracks the placement of blocks and handles row-clearing when rows are completely filled.
2. **Block Creation and Control**:
   - Different block shapes are randomly generated, each with unique rotation states.
   - Blocks can move left, right, and down and can rotate within grid boundaries.
3. **Collision Detection and Locking**:
   - Blocks stop moving if they reach the bottom or another block, at which point they "lock" into place.
4. **Row Clearing and Scoring**:
   - Completed rows are cleared from the grid, and the player earns points for each cleared row (increasing with multiple rows at once).
5. **Game Over Condition**:
   - The game ends if a new block cannot fit at the top of the grid.
6. **Visual Display**:
   - The game displays the grid, blocks, score, and the next block preview.

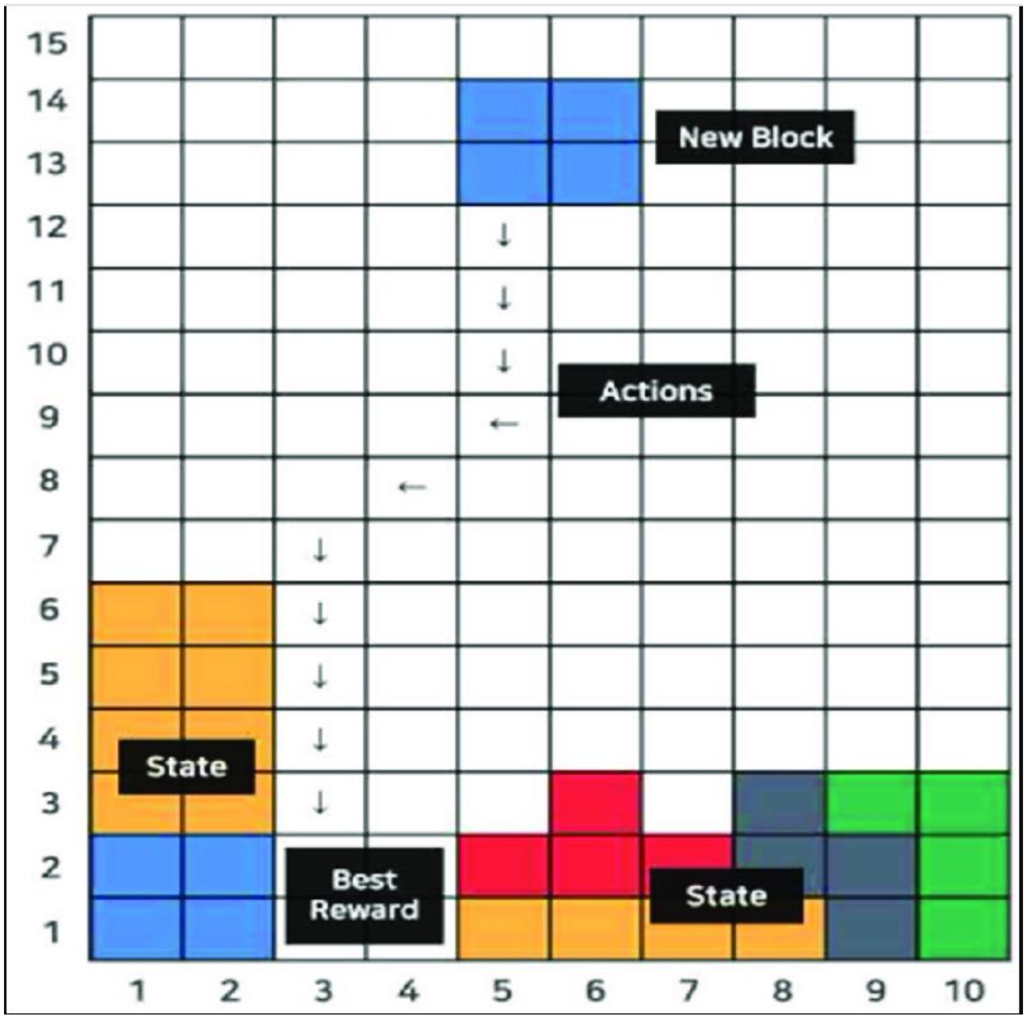
FLOW CHART:



Start
Create Background Cell
Create Z shape
Create O shape
Create I shape
Create T shape
Create L shape
Create Z shape (upside down)
Create L shape (upside down)
Press key "enter" to start the game
Select a random shape
Move down (using timer)
Did the shape hit and the top of the bitmap?
Yes
End and clear the bitmap
Ask if want to play again?
Yes
End
No
No
Has a key been pressed?
Yes
Recognize the key
Key "left"
Key "right"
Key "down"
Shape rotate 90 degrees left
Shape rotate 90 degrees right
Shape move to the bottom
Has the shape hit bottom, yet?
No
Yes
Has a row of cells being filled completely?
No
Yes
Clear the filled row and move every row down


Action A
Actor
$Q(A^i) = +5$
reward
Action A
Actor
$Q(A^i) = -5$
reward
Replay Memory
training
Main Conv
play
reward
Action A
Actor
$Q(A^i) = -4$
reward
Action A
Actor
$Q(A^i) = +1$


Board
- level: int
- dropDelay: int
+ LineClear()
+ GameOver()
+ PerfectGame()
GameBoard
- score: int
+ NewGame()
+ RestartGame()
GameFactory
+ GetTetrisShape(): TetrisShape
<<abstract>> TetrisShape
- centerX : int
- centerY: int
+ MoveLeft()
+ MoveRight()
+ MoveDown()
+ Rotate()
- DefineShape()
OPiece
IPiece
ZPiece
SPiece
LPiece
JPiece
TPiece
Blocks
- color : int
+ GenerateColor()



ANALYSIS OF CODE:

1) BLOCK CLASS:

```
from colors import Colors
import pygame
from position import Position

class Block:
    def __init__(self, id):
        self.id = id
        self.cells = {}
        self.cell_size = 30
        self.row_offset = 0
        self.column_offset = 0
        self.rotation_state = 0
        self.colors = Colors.get_cell_colors()
```

```
def move(self, rows, columns):
    self.row_offset += rows
    self.column_offset += columns

def get_cell_positions(self):
    tiles = self.cells[self.rotation_state]
    moved_tiles = []
    for position in tiles:
        position = Position(position.row + self.row_offset, position.column + self.column_offset)
        moved_tiles.append(position)
    return moved_tiles

def rotate(self):
    self.rotation_state += 1
    if self.rotation_state == len(self.cells):
        self.rotation_state = 0

def undo_rotation(self):
    self.rotation_state -= 1
    if self.rotation_state == -1:
        self.rotation_state = len(self.cells) - 1

def draw(self, screen, offset_x, offset_y):
    tiles = self.get_cell_positions()
    for tile in tiles:
        tile_rect = pygame.Rect(offset_x + tile.column * self.cell_size,
            offset_y + tile.row * self.cell_size, self.cell_size -1, self.cell_size -1)
        pygame.draw.rect(screen, self.colors[self.id], tile_rect)
```

Class Attributes and Constructor

1. **__init__(self, id)**:
    - This is the initializer method that sets up a new block instance.
    - Id serves as a unique identifier for the block, which is later used to assign a color.
    - Other attributes:
        - self.cells: A dictionary storing block shapes for different rotation states. Each key represents a rotation state, and each value contains the block's relative positions in that state.
        - self.cell_size: Specifies the size of each cell that forms the block, set to 30 pixels by default.
        - Self.row_offset and self.column_offset: Track the block's position offset in rows and columns, effectively allowing it to move around the grid.
        - self.rotation_state: Tracks the current rotation state of the block, initially set to 0.
        - self.colors: Loads color values for blocks from the Colors module, using the block's id to determine which color to assign.

Movement and Rotation Methods

2. **move(self, rows, columns)**:
    - Moves the block in the grid by adjusting the row_offset and column_offset.
    - rows and columns specify the amount to move in each direction.

    - This function is essential for moving blocks left, right, or down during gameplay.
3. **get_cell_positions(self)**:
    - Returns the current, actual grid positions of each cell in the block, taking the block's rotation and offset into account.
    - It fetches the positions of cells in the block for the current rotation_state, adds the offsets to each cell's position, and stores these modified positions in moved_tiles.
    - This method is useful when checking if the block collides with other blocks or if it's within bounds.
4. **rotate(self)**:
    - Rotates the block to the next rotation state by incrementing rotation_state.
    - If rotation_state reaches the end of available states, it wraps back to 0, enabling cyclic rotation.
5. **undo_rotation(self)**:
    - Reverses a rotation by decrementing rotation_state.
    - If rotation_state becomes negative, it wraps around to the last state, undoing the rotation.

Drawing Method

6. **draw(self, screen, offset_x, offset_y)**:
    - Renders the block on the screen using pygame.
    - Uses get_cell_positions to determine where each cell in the block should be drawn, taking into account offsets for both block position and screen position (offset_x and offset_y).
    - Creates pygame.Rect objects representing each cell and then draws them with pygame.draw.rect, using the color from self.colors [self.id].
    - The self.cell_size - 1 ensures a small gap between cells, giving a clear boundary between them for visibility.

Overall Usage and Flow

- This Block class represents a single game piece, likely a Tetris-like shape, that can rotate, move, and be drawn on the screen.
- Each block has multiple rotation states, and methods are provided to manage rotation, movement, and rendering.
- The draw function leverages the pygame library to render each block visually.

Potential Improvements

- **Collision Detection**: This class does not include collision handling, so integrating methods to detect if the block overlaps with other blocks or the game boundary could be beneficial.
- **Additional Customization**: The cell_size could be passed as a parameter to allow different sizes for blocks if needed.
- **Error Handling**: rotate and undo_rotation methods handle rotation cycling, but adding explicit checks for invalid rotation states would improve robustness.

Time Complexity:

1. **__init__(self, id):**
    - This initializer method assigns values to instance variables and retrieves colors using Colors.get_cell_colors ().
    - All assignments and list/dictionary creations are O(1).
    - **Overall Complexity**: **O(1)** (constant time)
2. **move(self, rows, columns):**
    - This method only updates row_offset and column_offset by adding the parameters rows and columns.
    - These are simple arithmetic operations with constant time complexity.
    - **Overall Complexity**: **O(1)**
3. **get_cell_positions(self):**
    - The method retrieves tiles from self.cells based on self.rotation_state.
    - It iterates over tiles (a list of positions) and modifies each position based on row_offset and column_offset.
    - Let's say n is the number of tiles (positions) in the block.
    - **Overall Complexity**: **O(n)** where n is the number of tiles in the current rotation state.
4. **rotate(self):**
    - This method increments rotation_state by 1 and checks if it equals the length of self.cells.
    - Since this is just an increment and a comparison, it's constant time.
    - **Overall Complexity**: **O(1)**
5. **undo_rotation(self):**
    - Similar to rotate, this method decrements rotation_state by 1 and checks if it equals -1, cycling it back to the end if necessary.
    - **Overall Complexity**: **O(1)**
6. **draw(self, screen, offset_x, offset_y):**
    - The method first calls get_cell_positions(), which has complexity **O(n)** (as previously calculated).
    - After obtaining the positions, it iterates over tiles to draw each cell, and since there are n tiles, this takes **O(n)**.
    - **Overall Complexity**: **O(n)** (due to the loop over tiles).

Summary

- __init__: **O(1)**
- move: **O(1)**
- get_cell_positions: **O(n)**
- rotate: **O(1)**
- undo_rotation: **O(1)**
- draw: **O(n)**

Overall Complexity of Using the Class

In typical usage, methods like move, rotate, and undo_rotation are **O(1)**, but get_cell_positions and draw are **O(n)** due to their dependence on the number of cells n in the block's current rotation state. So, the most time-consuming operations here are **O (n)** for **get_cell_positions** and draw.

2) BLOCKS CLASS OF GAME:

```
from block import Block
from position import Position

class LBlock(Block):
    def __init__(self):
        super().__init__(id = 1)
        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)

class JBlock(Block):
    def __init__(self):
        super().__init__(id = 2)
        self.cells = {
            0: [Position(0, 0), Position(1, 0), Position(1, 1), Position(1, 2)],
            1: [Position(0, 1), Position(0, 2), Position(1, 1), Position(2, 1)],
            2: [Position(1, 0), Position(1, 1), Position(1, 2), Position(2, 2)],
            3: [Position(0, 1), Position(1, 1), Position(2, 0), Position(2, 1)]
        }
        self.move(0, 3)

class IBlock(Block):
    def __init__(self):
        super().__init__(id = 3)
        self.cells = {
            0: [Position(1, 0), Position(1, 1), Position(1, 2), Position(1, 3)],
            1: [Position(0, 2), Position(1, 2), Position(2, 2), Position(3, 2)],
            2: [Position(2, 0), Position(2, 1), Position(2, 2), Position(2, 3)],
            3: [Position(0, 1), Position(1, 1), Position(2, 1), Position(3, 1)]
        }
        self.move(-1, 3)

class OBlock(Block):
    def __init__(self):
        super().__init__(id = 4)
        self.cells = {
            0: [Position(0, 0), Position(0, 1), Position(1, 0), Position(1, 1)]
        }
        self.move(0, 4)
```

```
class SBlock(Block):
    def __init__(self):
        super().__init__(id = 5)
        self.cells = {
            0: [Position(0, 1), Position(0, 2), Position(1, 0), Position(1, 1)],
            1: [Position(0, 1), Position(1, 1), Position(1, 2), Position(2, 2)],
            2: [Position(1, 1), Position(1, 2), Position(2, 0), Position(2, 1)],
            3: [Position(0, 0), Position(1, 0), Position(1, 1), Position(2, 1)]
        }
        self.move(0, 3)

class TBlock(Block):
    def __init__(self):
        super().__init__(id = 6)
        self.cells = {
            0: [Position(0, 1), Position(1, 0), Position(1, 1), Position(1, 2)],
            1: [Position(0, 1), Position(1, 1), Position(1, 2), Position(2, 1)],
            2: [Position(1, 0), Position(1, 1), Position(1, 2), Position(2, 1)],
            3: [Position(0, 1), Position(1, 0), Position(1, 1), Position(2, 1)]
        }
        self.move(0, 3)

class ZBlock(Block):
    def __init__(self):
        super().__init__(id = 7)
        self.cells = {
            0: [Position(0, 0), Position(0, 1), Position(1, 1), Position(1, 2)],
            1: [Position(0, 2), Position(1, 1), Position(1, 2), Position(2, 1)],
            2: [Position(1, 0), Position(1, 1), Position(2, 1), Position(2, 2)],
            3: [Position(0, 1), Position(1, 0), Position(1, 1), Position(2, 0)]
        }
        self.move(0, 3)
```

1. Block Class Analysis

First, let's look at the Block class:

-
  ```
  python
  Copy code
  class Block:
      def __init__(self, id):
          self.id = id
          self.cells = {}
          self.cell_size = 30
          self.row_offset = 0
          self.column_offset = 0
          self.rotation_state = 0
          self.colors = Colors.get_cell_colors()
  ```
- **Initialization (__init__ method):**
    - The __init__ method is executed when a new block object is created. It initializes several attributes (e.g., id, cells, row_offset, column_offset, etc.).

- Calling Colors.get_cell_colors () could have its own internal complexity. Assuming this is a simple method returning a list, we can assume it runs in constant time O (1) unless the method itself has a more complex operation.
- **Time Complexity of __init__:** O(1) (constant time)

2. Move Method Analysis

- python
  Copy code
```python
def move(self, rows, columns):
    self.row_offset += rows
    self.column_offset += columns
```
- **Move Method:**
  - The move method updates the row and column offsets by adding the provided rows and columns values to the current offsets. This involves two simple arithmetic operations and is independent of the size of the input, meaning the method runs in constant time.
  - **Time Complexity of move:** O(1) (constant time)

3. Get Cell Positions Method Analysis

- python
  Copy code
```python
def get_cell_positions(self):
    tiles = self.cells[self.rotation_state]
    moved_tiles = []
    for position in tiles:
        position = Position(position.row + self.row_offset, position.column + self.column_offset)
        moved_tiles.append(position)
    return moved_tiles
```
- **Get Cell Positions Method:**
  - The method first fetches the list of tiles for the current rotation_state from the self.cells dictionary, which is a dictionary lookup (self.cells[self.rotation_state]). Dictionary lookups are typically O(1) in Python.
  - It then iterates over the list of tiles, which are objects of Position, and for each tile, it updates its position and appends it to the moved_tiles list. If there are n positions in tiles, the loop will run n times.
  - **Time Complexity of get_cell_positions:**
    - The dictionary lookup is O(1).
    - The loop runs n times, where n is the number of tiles in self.cells[self.rotation_state].
    - **Time Complexity:** O(n) where n is the number of tiles in the current rotation state.

4. Rotate Method Analysis

- python
  Copy code
```python
def rotate(self):
    self.rotation_state += 1
    if self.rotation_state == len(self.cells):
        self.rotation_state = 0
```

- **Rotate Method:**
    - The method increments the rotation_state by 1, then checks if it exceeds the number of rotation states in self.cells. If it does, it resets the rotation_state to 0.
    - Both operations (incrementing and comparison) are constant-time operations.
    - **Time Complexity of rotate:** O(1) (constant time)

5. Undo Rotation Method Analysis

- python
  Copy code

```python
def undo_rotation(self):
    self.rotation_state -= 1
    if self.rotation_state == -1:
        self.rotation_state = len(self.cells) - 1
```

- **Undo Rotation Method:**
    - This method decrements the rotation_state and checks if it is -1 (indicating that the block has rotated past its first state). If so, it resets the rotation_state to the last index of self.cells.
    - The operations are simple arithmetic and comparison checks.
    - **Time Complexity of undo_rotation:** O(1) (constant time)

6. Draw Method Analysis

- python
  Copy code

```python
def draw(self, screen, offset_x, offset_y):
    tiles = self.get_cell_positions()
    for tile in tiles:
        tile_rect = pygame.Rect(offset_x + tile.column * self.cell_size,
            offset_y + tile.row * self.cell_size, self.cell_size -1, self.cell_size -1)
        pygame.draw.rect(screen, self.colors[self.id], tile_rect)
```

- **Draw Method:**
    - The get_cell_positions() method is called first, which we have already analyzed to have a time complexity of O(n) where n is the number of tiles.
    - Then, for each tile, a rectangle is created and drawn using pygame.draw.rect. Assuming this operation is O(1) for each tile, the loop iterates over all n tiles.
    - **Time Complexity of draw:** O(n) where n is the number of tiles in the current rotation state.

Time Complexity of the Block Class

- The time complexity of the operations in the Block class is mostly constant-time operations O(1) except for get_cell_positions() and draw(), which depend on the number of tiles.
- If the block has n tiles in its current rotation state, the time complexity of these methods will be O(n).
- **Overall Time Complexity of Block:**
    - The most complex operations in terms of time complexity are get_cell_positions() and draw(), which are O(n) where n is the number of tiles in the block.

7. Time Complexity of the Block Subclasses

Each of the subclasses (e.g., LBlock, JBlock, IBlock, etc.) initializes its cells attribute with a predefined set of positions.

For example:

-
  ```
  python
  Copy code
  class LBlock(Block):
      def __init__(self):
          super().__init__(id = 1)
          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)
  ```
- **Initialization of the cells dictionary**: The initialization of the cells dictionary is a constant-time operation because the size of the dictionary is fixed. Even though each block may have a different number of states (rotations), the overall dictionary construction is done in constant time relative to the size of the fixed state count.
- **Move Method Call (self.move(0, 3))**: This is a constant-time operation (O(1)).

Time Complexity of Subclasses:

- The time complexity for initializing each subclass is O(1) for constructing the cells dictionary.
- The overall complexity of using the LBlock, JBlock, IBlock, etc., would still depend on the operations invoked on them (mainly the get_cell_positions() and draw() methods, both of which depend on the number of tiles, n).

Overall Time Complexity for All Blocks:

- **Initialization:** O(1) for each block class.
- **Operations on Block (e.g., get_cell_positions, draw):** O(n) where n is the number of tiles in the current rotation state.

Conclusion

- The **time complexity of the Block class operations** (e.g., move, rotate, undo_rotation, get_cell_positions, draw) is generally O(1) for constant-time operations, with O(n) complexity for methods that involve iterating through the tiles (get_cell_positions, draw).
- The time complexity of initializing each block (LBlock, JBlock, etc.) is O(1) since the dictionary of positions is predefined and does not change based on input size.

3) COLOR CLASS OF THE GAME:

```
class Colors:
    dark_grey = (26, 31, 40)
    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)

    @classmethod
    def get_cell_colors(cls):
        return [cls.dark_grey, cls.green, cls.red, cls.orange, cls.yellow, cls.purple, cls.cyan, cls.blue]
```

Time Complexity Analysis of the Colors Class

The Colors class contains color definitions as tuples and a class method get_cell_colors that returns a list of colors. Let's break down the code:

1. Class Variables

The class defines several color variables, which are tuples representing RGB values:

-
  ```
  python
  Copy code
  dark_grey = (26, 31, 40)
  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)
  ```
- Each of these assignments is a simple operation where a tuple is created and assigned to a class variable.
- **Time Complexity of Assignments:** O (1) for each variable. Since there are 11 color variables, the total time complexity of creating these assignments is O(11) which simplifies to **O(1)** as the number of variables is constant.

2. get_cell_colors Method

-
  ```
  python
  Copy code
  @classmethod
  def get_cell_colors(cls):
      return [cls.dark_grey, cls.green, cls.red, cls.orange, cls.yellow, cls.purple, cls.cyan, cls.blue]
  ```
- This method returns a list containing a subset of the color class variables. The list contains 8 elements, and each element is a reference to the respective color tuple.
- **Time Complexity of get_cell_colors:**
  - The method involves creating a new list of size 8 by accessing class variables. Accessing each class variable is a constant-time operation (O(1)), and creating the list also takes constant time since the number of elements is fixed.
  - Therefore, the time complexity of creating and returning the list is **O(1)** because the size of the list is fixed and does not change with input size.

Overall Time Complexity of the Colors Class

- **Class Variable Initialization:** O(1) for each color variable.
- **get_cell_colors Method:** O(1) for creating and returning a list of fixed size.

Conclusion

- The **overall time complexity** of the Colors class, including its variables and methods, is **O(1)**, as all operations are constant-time operations, regardless of the size of input or data.

4) GAME CLASS:

```
from grid import Grid
from blocks import *
import random
import pygame

class Game:
    def __init__(self):
        self.grid = Grid()
        self.blocks = [IBlock(), JBlock(), LBlock(), OBlock(), SBlock(), TBlock(), ZBlock()]
        self.current_block = self.get_random_block()
        self.next_block = self.get_random_block()
        self.game_over = False
        self.score = 0
        self.rotate_sound = pygame.mixer.Sound("Sounds/rotate.ogg")
        self.clear_sound = pygame.mixer.Sound("Sounds/clear.ogg")

        pygame.mixer.music.load("Sounds/music.ogg")
        pygame.mixer.music.play(-1)

    def update_score(self, lines_cleared, move_down_points):
        if lines_cleared == 1:
            self.score += 100
        elif lines_cleared == 2:
```

```python
            self.score += 300
        elif lines_cleared == 3:
            self.score += 500
        self.score += move_down_points

    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

    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)

    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)

    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()

    def lock_block(self):
        tiles = self.current_block.get_cell_positions()
        for position in tiles:
            self.grid.grid[position.row][position.column] = self.current_block.id
        self.current_block = self.next_block
        self.next_block = self.get_random_block()
        rows_cleared = self.grid.clear_full_rows()
        if rows_cleared > 0:
            self.clear_sound.play()
            self.update_score(rows_cleared, 0)
        if self.block_fits() == False:
            self.game_over = True

    def reset(self):
        self.grid.reset()
        self.blocks = [IBlock(), JBlock(), LBlock(), OBlock(), SBlock(), TBlock(), ZBlock()]
        self.current_block = self.get_random_block()
        self.next_block = self.get_random_block()
        self.score = 0

    def block_fits(self):
        tiles = self.current_block.get_cell_positions()
        for tile in tiles:
            if self.grid.is_empty(tile.row, tile.column) == False:
                return False
        return True

    def rotate(self):
        self.current_block.rotate()
        if self.block_inside() == False or self.block_fits() == False:
            self.current_block.undo_rotation()
        else:
            self.rotate_sound.play()

    def block_inside(self):
        tiles = self.current_block.get_cell_positions()
        for tile in tiles:
            if self.grid.is_inside(tile.row, tile.column) == False:
                return False
        return True

    def draw(self, screen):
        self.grid.draw(screen)
        self.current_block.draw(screen, 11, 11)

        if self.next_block.id == 3:
            self.next_block.draw(screen, 255, 290)
        elif self.next_block.id == 4:
            self.next_block.draw(screen, 255, 280)
        else:
            self.next_block.draw(screen, 270, 270)
```

Time Complexity Analysis of the Game Class

This Game class involves various methods that control the flow of a Tetris-like game. The class utilizes a grid and a set of block objects and updates the score, manages block movement, rotation, locking, and row clearing. Let's break down the time complexity for key methods in this class:

1. __init__(self) Method

The initialization method sets up several objects:

- self.grid = Grid(): Initializes a new Grid object. Assuming the Grid class initialization is O (1), or based on its internal initialization, we can analyze it separately.
- self. Blocks = [...]: Creates a list of block objects (7 blocks). Creating each block object (such as IBlock (), JBlock ()) may have an internal time complexity based on its own initialization, but we can assume its O (1) for each block since the block configuration seems fixed.
- **Time Complexity:** This method has a time complexity of O (1) since all operations are constant-time operations, with no loops or recursion.

2. Get_random_block (self) Method

This method:

- Chooses a random block from the list of available blocks.

- Removes the chosen block from the list and returns it.
- **Operations:**
    - Random. Choice (self. Blocks): Choosing a random item from the list takes constant time, O (1).
    - self.blocks.remove (block): Removing an item from a list can take O (n) time where n is the length of the list. Since self. Blocks has a fixed length of 7, this operation is O (1) in practice.
    - **Time Complexity:** The time complexity of this method is **O (1)**.

3. Move_left (self) and move_right(self) Methods

Both methods perform similar operations:

- They move the current block by 1 unit in the left/right direction.
- They then check if the block is inside the grid and if it fits. This involves checking whether the block's cells are within the grid and not overlapping existing blocks.
- **Operations:**
    - self.current_block.move (...): This operation is a constant-time move operation, i.e., O (1).
    - self.block_inside () and self.block_fits (): Both of these methods loop over the cells of the current block to check if they are inside the grid or fit in the grid, respectively. If the block has k cells (depending on the block type), these checks are O (k). Since k is constant for each block type (maximum 4 cells), we treat this as **O (1)**.
- **Time Complexity:** Both methods have a time complexity of **O (1)**.

4. Move down (self) Method

This method moves the current block down and checks for collisions or boundaries:

- If the block doesn't fit or isn't inside the grid, it moves the block back up and locks the block.
- **Operations:**
    - self.current_block.move (1, 0): Moving the block is O (1).
    - self.block_inside () and self.block_fits (): Each of these methods is O (1) as explained above.
    - self.lock_block (): Locks the block by updating grid positions, and clears full rows.
        - **lock_block()** involves:
            - Iterating over the block's cells to lock them in the grid, which is O (k) where k is constant.
            - Calling self.grid.clear_full_rows(), which involves checking all rows of the grid. If the grid size is m (say, 20 rows), this operation has time complexity O(m).
- **Time Complexity:** The total time complexity for move down () is **O (1) + O (m)**, which simplifies to **O (m)** (if the grid has m rows).

5. lock_block(self) Method

The method locks the current block in place and updates the grid:

- **Operations:**

    - Iterating over the block's cells to lock them in the grid, which is O (k).
    - Clearing full rows, which O (m) is as explained above.
- **Time Complexity:** This method is **O (k + m)**, where k is the number of cells in the block (constant) and m is the number of rows in the grid. In practice, since k is small, this simplifies to **O (m)**.

6. block_fits(self) Method

This method checks whether the current block fits inside the grid without colliding with other blocks:

- **Operations:**
    - Iterating over each tile of the current block (k cells) and checking if the corresponding grid position is empty. Since k is constant, this is **O (k)**, which simplifies to **O (1)**.
- **Time Complexity:** This method is **O (1)**.

7. Rotate (self) Method

This method rotates the current block:

- If the rotated block doesn't fit or is outside the grid, it undoes the rotation.
- **Operations:**
    - self.current_block.rotate (): Rotation is a constant-time operation, O (1).
    - self.block_inside () and self.block_fits (): Each of these methods is O (1) as explained before.
    - **Time Complexity:** This method is **O (1)**.

8. Draw (self, screen) Method

This method draws the grid and the current block on the screen:

- **Operations:**
    - self.grid.draw(screen): This method would iterate over the grid to draw the elements. If the grid is of size m x n, this operation takes **O (m * n)**.
    - self.current_block.draw(screen, ...): Drawing the current block is a constant-time operation **O(1)** since the block has a fixed number of cells.
- **Time Complexity:** The total time complexity is **O(m * n)** where m and n are the grid dimensions.

Overall Time Complexity Summary

- **Initialization (__init__)**: O(1)
- **Block randomization (get_random_block)**: O(1)
- **Movement methods (move_left, move_right, move_down)**: O (1) for most part, but O (m) for move_down due to row clearing.
- **Locking (lock_block)**: O(m) for row clearing.

- **Block fitting checks (block_fits)**: O(1)
- **Rotation (rotate)**: O(1)
- **Drawing (draw)**: O (m * n) where m and n are the grid dimensions.

In summary, the overall complexity for most of the methods is **O(1)**, except for grid operations involving row clearing or drawing, which are **O(m)** and **O(m * n)**.

5) GRID CLASS OF THE GAMING:

```
import pygame
from colors import Colors

class Grid:
    def __init__(self):
        self.num_rows = 20
        self.num_cols = 10
        self.cell_size = 30
        self.grid = [[0 for j in range(self.num_cols)] for i in range(self.num_rows)]
        self.colors = Colors.get_cell_colors()

    def print_grid(self):
        for row in range(self.num_rows):
            for column in range(self.num_cols):
                print(self.grid[row][column], end = " ")
            print()

    def is_inside(self, row, column):
        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):
        if self.grid[row][column] == 0:
            return True
        return False

    def is_row_full(self, row):
        for column in range(self.num_cols):
            if self.grid[row][column] == 0:
                return False
        return True

    def clear_row(self, row):
        for column in range(self.num_cols):
            self.grid[row][column] = 0

    def move_row_down(self, row, num_rows):
        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):
        completed = 0
        for row in range(self.num_rows-1, 0, -1):
            if self.is_row_full(row):
                self.clear_row(row)
                completed += 1
            elif completed > 0:
                self.move_row_down(row, completed)
        return completed

    def reset(self):
        for row in range(self.num_rows):
            for column in range(self.num_cols):
                self.grid[row][column] = 0

    def draw(self, screen):
        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)
```

Time Complexity Analysis of the Grid Class

The Grid class manages a grid structure for the Tetris game, providing functionalities for grid initialization, row checking, clearing rows, and rendering the grid to the screen. Let's break down the time complexity of each method:

1. __init__ (self) Method

This method initializes the grid and sets up the colors:

- self.grid = [[0 for j in range(self.num_cols)] for i in range(self.num_rows)]: This is a list comprehension to create a 2D grid of size num_rows x num_cols. If num_rows = 20 and num_cols = 10, the total number of cells is 200, so this operation has a time complexity of **O (m * n)**, where m is the number of rows and n is the number of columns.
- self.colors = Colors.get_cell_colors (): Assuming that get_cell_colors () returns a fixed list of colors, this is an O (1) operation.
- **Time Complexity:** The overall time complexity of the initialization is **O(m * n)**.

2. Print_grid (self) Method

This method prints the grid to the console:

- **Operations:**
    - Two nested loops iterate over num_rows and num_cols, so the time complexity of printing all cells is **O (m * n)**.
- **Time Complexity: O (m * n)**.

3. Is_inside (self, row, column) Method

This method checks if the given position is inside the grid:

- **Operations:**
  - The method checks if row and column are within valid bounds (0 <= row < num_rows and 0 <= column < num_cols).
  - This operation involves just two comparisons, making it a constant-time operation.
- **Time Complexity: O (1)**.

4. Is_empty (self, row, column) Method

This method checks if a specific cell is empty:

- **Operations:**
  - The method checks if the value in self. Grid [row] [column] is 0.
  - This is a single comparison, making the operation constant-time.
- **Time Complexity: O (1)**.

5. Is_row_full (self, row) Method

This method checks if a specific row is full (no empty cells):

- **Operations:**
  - A loop iterates over each column in the specified row to check if any column is empty (self.grid[row][column] == 0).
  - This operation takes linear time relative to the number of columns, i.e., **O(n)**.
- **Time Complexity: O (n)** where n is the number of columns.

6. Clear_row (self, row) Method

This method clears a specific row (sets all cells in the row to 0):

- **Operations:**
  - A loop iterates over each column in the row and sets the value to 0.
  - This operation also takes linear time relative to the number of columns, i.e., **O (n)**.
- **Time Complexity: O (n)**.

7. Move_row_down (self, row, num_rows) Method

This method moves a row down by num_rows rows:

- **Operations:**
  - A loop iterates over all columns in the row, copying values from the row row to row + num_rows and setting the original cells to 0.
  - This operation also takes linear time relative to the number of columns, i.e., **O (n)**.
- **Time Complexity: O (n)**.

8. Clear_full_rows (self) Method

This method clears full rows and moves down any rows above them:

- **Operations:**
  - The method iterates over all rows in the grid (in reverse order) and checks if the row is full. This involves calling is_row_full() for each row, which is **O (n)** per row.
  - If a row is full, it clears the row using clear_row (), which is **O (n)**, and then moves rows above it down with move_row_down (), which is also **O (n)**.
  - The outer loop iterates over m rows, and the inner operations for each row involve **O (n)** work.
- **Time Complexity:** The total complexity for this method is **O (m * n)**, where m is the number of rows and n is the number of columns.

9. reset(self) Method

This method resets the grid by setting all cells to 0:

- **Operations:**
  - A nested loop iterates over all rows and columns, setting each cell to 0.
  - The time complexity is **O(m * n)** because it iterates through the entire grid.
- **Time Complexity: O(m * n)**.

10. Draw (self, screen) Method

This method draws the grid on the screen using the Pygame library:

- **Operations:**
  - Two nested loops iterate over num_rows and num_cols, and for each cell, a rectangle is drawn using Pygame's pygame.draw.rect() method.
  - Each call to pygame.draw.rect() is a constant-time operation, so the total time complexity is proportional to the number of cells in the grid.
- **Time Complexity: O(m * n)**.

Overall Time Complexity Summary

- **__init__(self)**: O(m * n)
- **print_grid(self)**: O(m * n)
- **is_inside(self, row, column)**: O(1)
- **is_empty(self, row, column)**: O(1)
- **is_row_full(self, row)**: O(n)
- **clear_row(self, row)**: O(n)
- **move_row_down(self, row, num_rows)**: O(n)
- **clear_full_rows(self)**: O(m * n)
- **reset(self)**: O(m * n)
- **draw(self, screen)**: O(m * n)

6) MAIN CLASS OF THE GAME:

```
import pygame,sys
from game import Game
from colors import Colors

pygame.init()

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)

score_rect = pygame.Rect(320, 55, 170, 60)
next_rect = pygame.Rect(320, 215, 170, 180)

screen = pygame.display.set_mode((500, 620))
pygame.display.set_caption("Python Tetris")

clock = pygame.time.Clock()

game = Game()

GAME_UPDATE = pygame.USEREVENT
pygame.time.set_timer(GAME_UPDATE, 200)

while True:
   for event in pygame.event.get():
      if event.type == pygame.QUIT:
         pygame.quit()
         sys.exit()
      if event.type == pygame.KEYDOWN:
         if game.game_over == True:
            game.game_over = False
            game.reset()
         if event.key == pygame.K_LEFT and game.game_over == False:
            game.move_left()
         if event.key == pygame.K_RIGHT and game.game_over == False:
            game.move_right()
         if event.key == pygame.K_DOWN and game.game_over == False:
            game.move_down()
            game.update_score(0, 1)
         if event.key == pygame.K_UP and game.game_over == False:
            game.rotate()
      if event.type == GAME_UPDATE and game.game_over == False:
         game.move_down()

   #Drawing
   score_value_surface = title_font.render(str(game.score), True, Colors.white)

   screen.fill(Colors.dark_blue)
   screen.blit(score_surface, (365, 20, 50, 50))
   screen.blit(next_surface, (375, 180, 50, 50))

   if game.game_over == True:
      screen.blit(game_over_surface, (320, 450, 50, 50))
```

```
pygame.draw.rect(screen, Colors.light_blue, score_rect, 0, 10)
screen.blit(score_value_surface, score_value_surface.get_rect(centerx = score_rect.centerx,
   centery = score_rect.centery))
pygame.draw.rect(screen, Colors.light_blue, next_rect, 0, 10)
game.draw(screen)

pygame.display.update()
clock.tick(60)
```

Time Complexity Analysis of the Tetris Game Loop

Let's break down the key operations in the Tetris game loop and analyze their time complexities:

1. Initialization

-
  ```
  python
  Copy code
  pygame.init()
  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)
  score_rect = pygame.Rect(320, 55, 170, 60)
  next_rect = pygame.Rect(320, 215, 170, 180)
  screen = pygame.display.set_mode((500, 620))
  pygame.display.set_caption("Python Tetris")
  clock = pygame.time.Clock()
  game = Game()
  ```
- **pygame.init()**: Initializes all Pygame modules. This operation is **O(1)** in the context of setting up the game, though it may involve internal setup that's generally constant.
- **Font rendering (title_font.render)**: Renders the "Score," "Next," and "Game Over" text. Rendering text is **O (k)**, where k is the number of characters. In this case, it's a constant operation, so it's **O (1)**.
- **Creating the game object (game = Game())**: This is the creation of the Game object, which will likely have its own initialization, depending on the game logic.
- **Time Complexity: O (1)** for initialization.

2. Event Handling

-
  ```
  python
  Copy code
  for event in pygame.event.get():
     if event.type == pygame.QUIT:
        pygame.quit()
        sys.exit()
     if event.type == pygame.KEYDOWN:
        if game.game_over == True:
  ```

```
            game.game_over = False
            game.reset()
        if event.key == pygame.K_LEFT and game.game_over == False:
            game.move_left()
        if event.key == pygame.K_RIGHT and game.game_over == False:
            game.move_right()
        if event.key == pygame.K_DOWN and game.game_over == False:
            game.move_down()
            game.update_score(0, 1)
        if event.key == pygame.K_UP and game.game_over == False:
            game.rotate()
    if event.type == GAME_UPDATE and game.game_over == False:
        game.move_down()
```

- **pygame.event.get()**: Retrieves all pending events in the event queue. If n is the number of events in the queue, this function is **O (n)**. However, the number of events is typically small and constant, so in practice, it can be considered **O(1)**.
- **Key event handling**: Inside the loop, the game checks for specific key events (left, right, down, up) and performs certain actions (move_left (), move_right(), move_down (), rotate()).
    - These actions typically interact with the Game class methods. Let's assume these operations are **O (1)** because they typically perform simple state changes.
    - **game.update_score (0, 1)**: This operation likely updates the score and could be considered **O (1)**.
- **Time Complexity:** For the event handling section, assuming a small number of events and fast keypress handling, it is **O(1)**.

3. Game Loop Update

```
• python
  Copy code
  if event.type == GAME_UPDATE and game.game_over == False:
      game.move_down()
```

- **game.move_down()**: This method moves the current Tetris block down. If the operation involves checking or updating the grid, its time complexity depends on the grid size and block movements. Assuming it involves checking a small number of cells (e.g., 10 columns per row), this could be **O(n)**, where n is the number of columns in the grid.
- **Time Complexity: O(n)**, where n is the number of columns in the grid.

4. Drawing Operations

```
• python
  Copy code
  score_value_surface = title_font.render(str(game.score), True, Colors.white)

  screen.fill(Colors.dark_blue)
  screen.blit(score_surface, (365, 20, 50, 50))
  screen.blit(next_surface, (375, 180, 50, 50))

  if game.game_over == True:
      screen.blit(game_over_surface, (320, 450, 50, 50))

  pygame.draw.rect(screen, Colors.light_blue, score_rect, 0, 10)
  screen.blit(score_value_surface, score_value_surface.get_rect(centerx = score_rect.centerx,
      centery = score_rect.centery))
  pygame.draw.rect(screen, Colors.light_blue, next_rect, 0, 10)
  game.draw(screen)
```

- **title_font.render(str(game.score), True, Colors.white)**: Rendering the score text is **O(1)** as the score is a simple integer.
- **screen.fill(Colors.dark_blue)**: This operation fills the entire screen with a background color. Filling the entire screen with pixels is an **O (A)** operation, where A is the area of the screen, i.e., **O(width * height)**. In this case, **O (500 * 620)**.
- **screen.blit(...)**: Blitting images (e.g., score, next blocks) is typically **O (1)** for each individual image.
- **pygame.draw.rect(...)**: Drawing a rectangle is **O (1)**.
- **Game. Draw (screen)**: This method likely draws all elements of the game (such as the grid, the falling block, etc.). Assuming it involves iterating through the grid and drawing cells, this could be **O (m * n)** where m is the number of rows and n is the number of columns.
- **Time Complexity: O (m * n)** for drawing the game state (grid).

5. FPS Control

Python
Copy code
```
pygame.display.update ()
clock.tick (60)
```

- **pygame.display.update()**: This updates the entire screen to render the new frame. It's an **O (1)** operation.
- **clock.tick (60)**: This controls the frame rate, ensuring the game runs at 60 FPS. This is also **O (1)**.

Overall Time Complexity

- **Event Handling**: **O(1)**
- **Game Update**: **O(n)** (for game.move_down())
- **Drawing**: **O(m * n)** (for drawing the grid)
- **FPS Control**: **O(1)**

In the worst case, the drawing and game update steps dominate, making the overall time complexity for each frame **O (m * n)**, where m is the number of rows and n is the number of columns in the grid.

Thus, **O (m * n)** is the time complexity of the game loop, which is the most significant factor in the game's performance.

CODE POSTION OF THE GAME:

```
class Position:
   def __init__(self, row, column):
     self. Row = row
     self. Column = column
```

Explanation:

- **__init__ (self, row, column)**: This is the constructor method. It initializes a new Position object with a specified row and column.
    - Row: The row index (vertical position) of the grid.
    - Column: The column index (horizontal position) of the grid.

Time Complexity:

The time complexity for creating an instance of the Position class is **O(1)** since the constructor simply assigns values to two attributes.

Example:

Python
Copy code
position = Position (5, 10) # Creates a Position object at row 5, column 10
Print(position.row) # Outputs: 5
Print (position.column) # Outputs: 10

OVERALL TIME COMPLEXITY:

1. Block Movement and Rotation:

- **move()**: The move() function adjusts the block's position based on the rows and columns specified. This is a constant-time operation O (1) O (1) O (1).
- **rotate()**: The rotate() function updates the block's rotation state, and the undo_rotation() function undoes the rotation. Each of these operations is also O (1) O (1) O (1), as it involves simple state manipulation.
- **get_cell_positions ()**: This function iterates through the block's cells and calculates the new positions based on the offsets. This is an O (n) O (n) O (n) operation, where nnn is the number of cells (constant for each block type).

2. Grid Operations:

- **clear_full_rows()**: This function checks and clears full rows from the grid. It involves iterating through each row to check if it's full (which takes O (m) O (m) O (m), where mmm is the number of rows) and potentially moving rows down. The complexity here is O (m·n) O (m \cdot n) O (m·n), where mmm is the number of rows and nnn is the number of columns.
- **is_row_full(), clear_row(), move_row_down()**: These functions are called for each row and are O(n)O(n)O(n) operations, where nnn is the number of columns.

3. Collision Detection:

- **block_fits()**: This function checks if the block fits within the grid. It iterates through all the cell positions of the block, so the time complexity is O (n) O (n) O (n), where nnn is the number of cells in the block.
- **block_inside()**: Similar to block_fits(), this checks if the block's cells are inside the grid's boundaries, and the complexity is also O(n)O(n)O(n).

4. Drawing and Rendering:

- **Draw ()**: The draw () function in both the Block and Grid classes involves iterating over the grid's cells to render the blocks. This takes O (m·n) O (m \cdot n) O (m·n), where mmm the number of rows and nnn is is the number of columns, because each cell is drawn individually.
- The number of tiles (cells) drawn for each block is constant for all block types, so this part's complexity is determined by the grid size.

5. Main Loop Operations:

- **Game Logic**: The game loop typically runs once per frame, checking for user input, updating the grid, moving blocks, rotating them, and drawing the updated state. Each operation, like block movement, collision detection, and drawing, is constant-time per frame, so the loop overall runs in O(1)O(1)O(1) per frame, but the time complexity of rendering and grid checks depends on the grid size.

Overall Time Complexity:

- **For a single operation (like moving or rotating the block)**: O (1) O (1) O (1) for basic block movement, rotation, and user input handling.
- **For grid-related checks (e.g., clearing rows or checking if blocks fit)**: O (m·n) O (m \cdot n) O (m·n), where mmm is the number of rows and nnn is the number of columns.
- **For the game loop**: The game loop will run continuously, and most operations per frame involve a combination of O(1)O(1)O(1) and O(m·n)O(m \cdot n)O(m·n) operations.