Programming a Testris Game

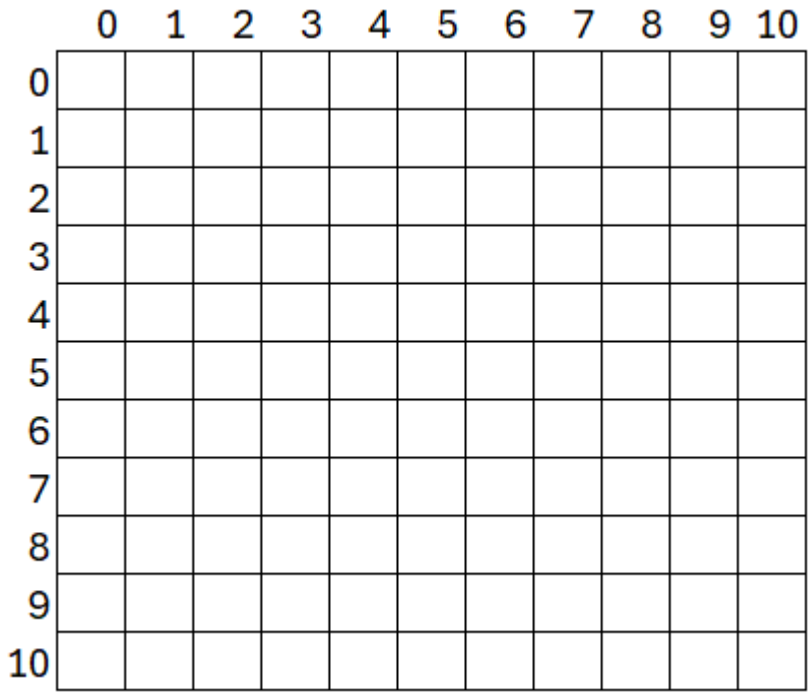
May 29, 2024 Uncategorized

SplashKit is a comprehensive open-source game development library providing extensive range of tools and functionalities. In this article, we will go through a tutorial on using SplashKit and building one of most classic puzzle game: Tetris!

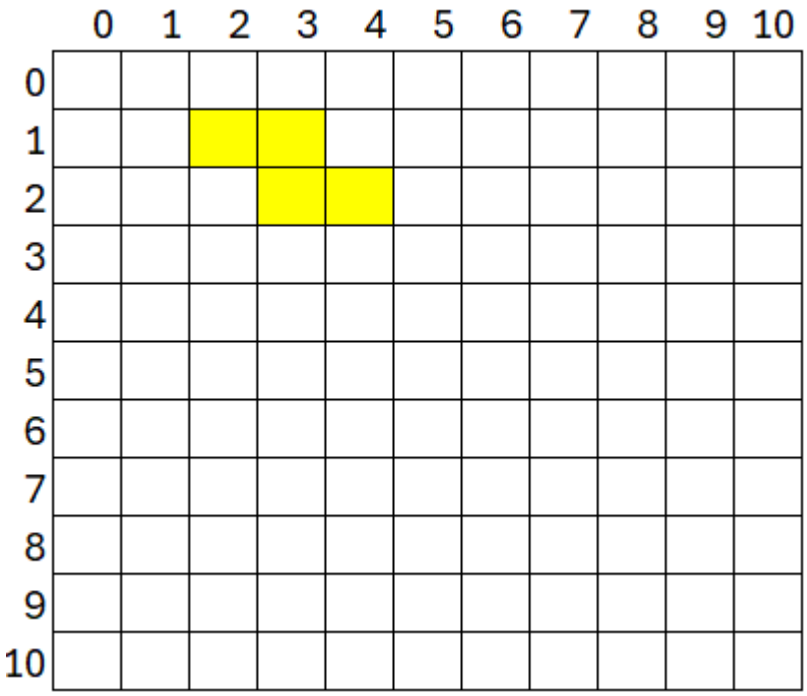
1. **Designing game logic**

The Tetris Game is constructed from a game board with gridlines. A block is moved across the board through the commands of player. When the block touches the board it is locked within its position. When there is a completed row, the row is then deleted and player gets higher score.

Therefore, we can think of the gameboard as 2D arrays with rows and columns. Each cell in the gameboard can be referred to by its own coordinate.



With this logic, each block can also be referred to by a list of coordinates. We can create properties to store the coordinates of the current block, then update its coordinate as the player moves the shape across the game board. When a block touches the bottom of the gameboard, it is locked to its coordinate. We can assign these locked cells to value = 1.

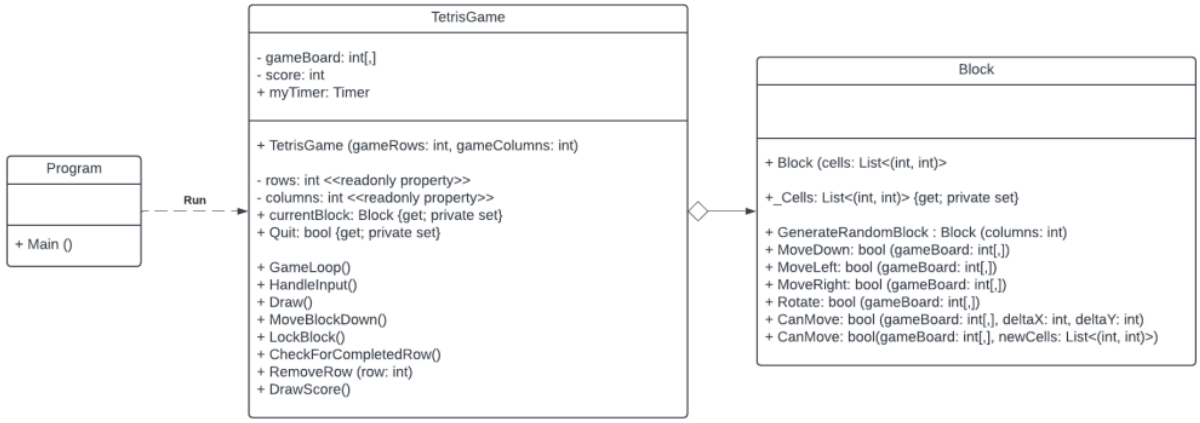


When the game is initiated, it will create a loop through each cell in the 2D array game board and check its value, if equal to 0 then it only draws a blank rectangle, if equal to 1 then it fills

the cell with blue color to indicate locked cell, if a cell's coordinates match the coordinates of the current block then that cell is filled with red color.

A game loop is created to move the current block by 1 row as 1 second pass by, and check for completed rows. When a row is completed (all cells in the row = 1) then that row is deleted and player's score increases by 100.

2. Drawing UML class diagram



With the above game logic, we can start sketching how we can implement the program using an UML class diagram. So 2 classes can be created to initiate the game board and the block.

a. Block class

We should start first with the Block class, which initiates each Block using a constructor with parameter is a list of tuples. But why do we use list of tuples instead of list of arrays to store the shape's cells coordinates? Well mostly it's because of the syntax of tuples that clearly denotes the coordinates as (int, int), so when we refer to the x, y coordinate of a cell, we use Cells[i].Item1 and Cells[i].Item2. With arrays, we need to remember the index positions, which can be more error-prone.

Then we need to create property to store the updated coordinates of the current block. Now moving on to the methods – what our block can do. We create different shapes of our blocks and initiate their coordinate through a GenerateRandomBlock method. Then there's method to move our block down, left, right and also to rotate the block by 90 degrees. To identify if the block has reached the last row or has reached a locked row, we create CanMove method

to return a boolean value if the block can still move by deltaX or deltaY vertically or horizontally. If the block cannot move further, the CanMove method should also update the value of the block's coordinate in the game board to 1 to indicate a locked cell.

b. TetrisGame class

Our game board is initiated by a constructor with number of rows and columns. A game board field is created to store a two-dimensional array of integers (int[,]). Key characteristics of `int[,]`:

- The dimensions of the array (number of rows and columns) are fixed and cannot be changed
- All rows have the same number of columns, creating a rectangular structure.
- All elements initialized to the default value of int (which is = 0)

Some read-only properties within the class include rows, columns, coordinates of the current block, Quit boolean. Other fields include score and timer.

Some methods in our TetrisGame class are:

- Draw: To clear screen and loop through each cell in the game board and draw its color based on the above mentioned game logic
- HandleInput: Handle player's key input to call MoveLeft, MoveRight, Rotate methods from the Block class and the MoveBlockDown method from this class
- MoveBlockDown: Call the MoveDown method in the Block class. If the MoveDown method return a false, then lock the current block's coordinates and generate a new block
- LockBlock: retrieve the current coordinates of the block to update the cells' values to 1 in the game board
- CheckForCompletedRow: loop through each row in the gameboard, if all cells in the row equal 1 then call the RemoveRow method and increase player's score by 100
- RemoveRow: pass in the completed row, then update the value of all above rows in the gameboard, then clear the top row of the game
- GameLoop: Start the timer and call MoveBlockDown method for every second passes by then reset timer again; and also call CheckForCompletedRow method
- DrawScore: Present the player score into the screen

3. Start coding

Now we have a good idea of how to implement the program, we can move to the fun part where our idea can be implemented through C# code.

a. Block class

Step 1. Initiating constructor and properties

```
12 references
public class Block
{
    20 references
    public List<(int, int)> Cells { get; private set; }

    7 references
    public Block(List<(int, int)> cells)
    {
        Cells = cells;
    }
```

Step 2. Visualizing our blocks though GenerateRandomBlock method

```
public static Block GenerateRandomBlock(int columns)
{
    Random rand = new Random();
    int blockType = rand.Next(0, 7); // 7 types of Tetris blocks

    switch (blockType)
    {
        case 0: // I block
            return new Block(new List<(int, int)> { (0, columns / 2 - 1), (0, columns / 2), (0, columns / 2 + 1), (0, columns / 2 + 2) });
        case 1: // O block
            return new Block(new List<(int, int)> { (0, columns / 2), (0, columns / 2 + 1), (1, columns / 2), (1, columns / 2 + 1) });
        case 2: // T block
            return new Block(new List<(int, int)> { (0, columns / 2), (1, columns / 2 - 1), (1, columns / 2), (1, columns / 2 + 1) });
        case 3: // S block
            return new Block(new List<(int, int)> { (0, columns / 2), (0, columns / 2 + 1), (1, columns / 2 - 1), (1, columns / 2) });
        case 4: // Z block
            return new Block(new List<(int, int)> { (0, columns / 2 - 1), (0, columns / 2), (1, columns / 2), (1, columns / 2 + 1) });
        case 5: // J block
            return new Block(new List<(int, int)> { (0, columns / 2 - 1), (1, columns / 2 - 1), (1, columns / 2), (1, columns / 2 + 1) });
        case 6: // L block
            return new Block(new List<(int, int)> { (0, columns / 2 + 1), (1, columns / 2 - 1), (1, columns / 2), (1, columns / 2 + 1) });
        default:
            throw new Exception("Invalid block type");
    }
}
```

Make use of Random class in .NET Framwork's System to create random id for our blocks. When create a new block, we can assign it to its own list of coordinates within the game board. Remember that all blocks are generated from the top of the window, so we only need

to pass in the number of columns in our gameboard to calculate the Y coordinates, while X coordinates are 0.

Step 3. Check if the block can move

```
public bool CanMove(int[,] gameBoard, int deltaX, int deltaY)
{
    List<(int, int)> newCells = new List<(int, int)>();
    foreach (var cell in Cells)
    {
        newCells.Add((cell.Item1 + deltaX, cell.Item2 + deltaY));
    }
//using method overloading
    return CanMove(gameBoard, newCells);
}

2 references
public bool CanMove(int[,] gameBoard, List<(int, int)> newCells)
{
    foreach (var cell in newCells)
    {
        if (cell.Item1 < 0 || cell.Item1 >= gameBoard.GetLength(0) ||
            cell.Item2 < 0 || cell.Item2 >= gameBoard.GetLength(1) ||
            gameBoard[cell.Item1, cell.Item2] == 1)
        {
            return false;
        }
    }
    return true;
}
```

The CanMove method updates the coordinates of the block if moved by deltaX and deltaY to newCells. If the coordinates of these newCells <0, or >the height and width of the game board, or equal to coordinate of a locked cell, then this method return a boolean value of false, else it returns true.

Step 4. Move the block

```
public bool MoveDown(int[,] gameBoard)
{
    if (CanMove(gameBoard, 1, 0))
    {
        for (int i = 0; i < Cells.Count; i++)
        {
            Cells[i] = (Cells[i].Item1 + 1, Cells[i].Item2);
        }
        return true;
    }
    return false;
}

1 reference
public void MoveLeft(int[,] gameBoard)
{
    if (CanMove(gameBoard, 0, -1))
    {
        for (int i = 0; i < Cells.Count; i++)
        {
            Cells[i] = (Cells[i].Item1, Cells[i].Item2 - 1);
        }
    }
}
```

If the block is moved down, then the block's cells coordinates are updated by +1 row, same column. Similarly, when the block is moved left, the cells' row stay the same, but columns are updated by -1. Call the CanMove method to check if the block can move by the appropriate deltaX and deltaY, then update the coordinates of the block accordingly. Now do the same to create a MoveRight method. 🙂

b. TetrisGame class

Step 1. Initiating constructor and properties

```
namespace TetrisGame
{
    3 references
    public class TetrisGame
    {
        4 references
        private readonly int rows;
        8 references
        private readonly int columns;
        11 references
        private int[,] gameBoard;
        5 references
        public SplashKitSDK.Timer myTimer;
        8 references
        public Block currentBlock { get; private set; }
        3 references
        private int score;
        2 references
        public bool Quit {get; private set; }

        1 reference
        public TetrisGame(int gameRows, int gameColumns)
        {
            rows = gameRows;
            columns = gameColumns;
            gameBoard = new int[rows, columns];
            currentBlock = Block.GenerateRandomBlock(columns);
            myTimer = new SplashKitSDK.Timer("My Timer");
            myTimer.Start();
            score = 0;
        }
```

Step 2. Draw our game board

```
public void Draw()
{
    int cellSize = 30;
    SplashKit.ClearScreen(SplashKitSDK.Color.White);
    DrawScore();
    // Draw the game board
    for (int r = 0; r < rows; r++)
    {
        for (int c = 0; c < columns; c++)
        {
            SplashKit.DrawRectangle(SplashKitSDK.Color.Gray, c * cellSize, r * cellSize, cellSize, cellSize);
            if (gameBoard[r, c] == 1)
            {
                SplashKit.FillRectangle(SplashKitSDK.Color.Blue, c * cellSize, r * cellSize, cellSize, cellSize);
                SplashKit.DrawRectangle(SplashKitSDK.Color.Black, c * cellSize, r * cellSize, cellSize, cellSize);
            }
        }

        // Draw the current block
        foreach (var cell in currentBlock.Cells)
        {
            SplashKit.FillRectangle(SplashKitSDK.Color.Red, cell.Item2 * cellSize, cell.Item1 * cellSize, cellSize, cellSize);
            SplashKit.DrawRectangle(SplashKitSDK.Color.Black, cell.Item2 * cellSize, cell.Item1 * cellSize, cellSize, cellSize);
        }

        // Refresh the screen to show the new drawings
        SplashKit.RefreshScreen();
    }
}
```

Determine the cell size to draw, then loop through each cell in the gameboard to draw its color according to the value of that cell in the gameBoard (0 or 1). Draw the current block by referring to the coordinates of currentBlock property. Then refresh the screen.

Step 3. Implement HandleInput

```
public void HandleInput()
{
    SplashKit.ProcessEvents();
    if (SplashKit.KeyDown(KeyCode.EscapeKey)) Quit = true;
    else if ( SplashKit.KeyDown(KeyCode.UpKey) ) currentBlock.Rotate(gameBoard);
    else if ( SplashKit.KeyDown(KeyCode.DownKey) ) MoveBlockDown();
    else if ( SplashKit.KeyDown(KeyCode.LeftKey) ) currentBlock.MoveLeft(gameBoard);
    else if ( SplashKit.KeyDown(KeyCode.RightKey) ) currentBlock.MoveRight(gameBoard);
}
```

User SplashKit to acknowledge player's input key, then call methods of the Block class accordingly.

Step 4. Initiate MoveBlockDown and LockBlock methods

```
private void MoveBlockDown()
{
    if (!currentBlock.MoveDown(gameBoard))
    {
        LockBlock();
        currentBlock = Block.GenerateRandomBlock(columns);
    }
}


1 reference
private void LockBlock()
{
    foreach (var cell in currentBlock.Cells)
    {
        gameBoard[cell.Item1, cell.Item2] = 1;
    }
}
```

Call the MoveDown method from the Block class, if it returns false then lock that block's coordinates and generate new block. Lock the current block by assigning that cell in the game board to 1.

Step 5. Check for completed row and remove row

```
private void CheckForCompletedRows()
{
    for (int r = 0; r < rows; r++)
    {
        bool rowComplete = true;
        for (int c = 0; c < columns; c++)
        {
            if (gameBoard[r, c] == 0)
            {
                rowComplete = false;
                break;
            }
        }
        if (rowComplete)
        {
            RemoveRow(r);
            score += 100; // Increment score for each completed row
        }
    }
}

1 reference
private void RemoveRow(int row)
{
    for (int r = row; r > 0; r--)
    {
        for (int c = 0; c < columns; c++)
        {
            gameBoard[r, c] = gameBoard[r - 1, c];
        }
    }

    // Clear the top row
    for (int c = 0; c < columns; c++)
    {
        gameBoard[0, c] = 0;
    }
}
```

Loop through each cell in the gameboard, if one cell in that row is =0 then the row is not completed. If row is completed, remove the row by updating all of its cell value equal to the above row, then clear the top row. Remember to also increase the player's score by 100.

Step 6. Make a GameLoop method to handle the game loop logic

```
public void GameLoop()
{
    if (myTimer.Ticks > 1000)
    {
        MoveBlockDown();
        myTimer.Reset();
        myTimer.Start();
    }
    CheckForCompletedRows();
}
```

Call MoveBlockDown method by every second by start then reset timer when it passed by 1 second. Call the CheckForCompletedRows method.

Step 7. Draw Score

```
1 reference
public void DrawScore()
{
    SplashKit.DrawText($"SCORE: {score}", SplashKitSDK.Color.HotPink, 10, 10);

}
```

Use SplashKit's DrawText to present the player's score onto the game screen.

c. Program class

Now that we have created a complete TetrisGame class, initiate the game by creating an object TetrisGame, then loop while the game is still running to call the game's methods

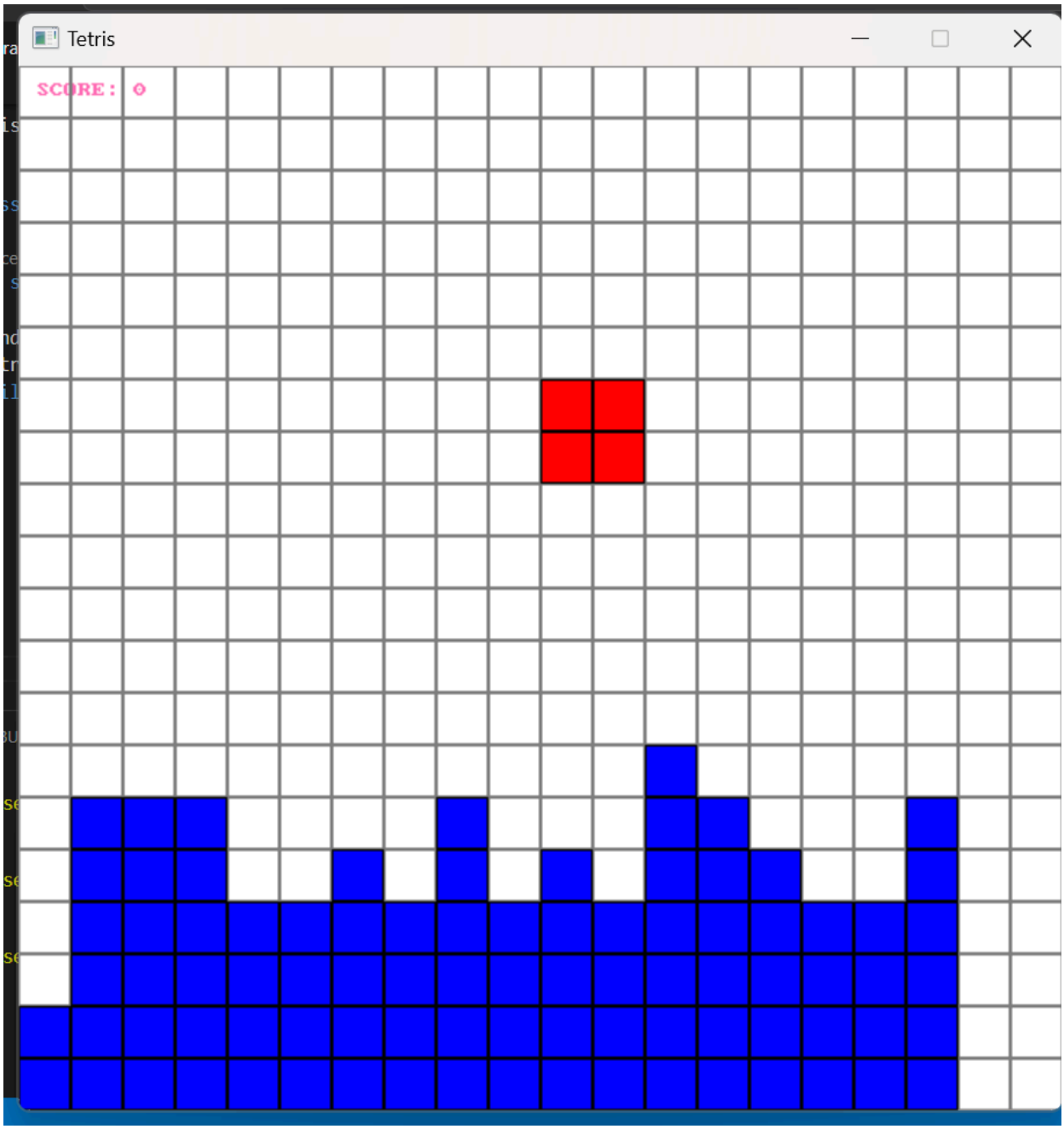
```
public class Program
{
    0 references
    public static void Main()
    {
        Window gameWindow = new Window("Tetris",600,600);
        TetrisGame game = new TetrisGame(20,20);
        while ( game.Quit != true )
        {
            game.HandleInput();
            game.GameLoop();
            game.Draw();
            SplashKit.Delay(20);
        }
    }
}
```

4. Running our Game

```
$ skm dotnet build
```

```
$ skm dotnet run
```

Run these steps on Terminal to see our program comes to life



Feel free to comment or suggest further improvements on my Tetris Program. Have fun coding! ^^

← Previous

Leave a comment