MEF UNIVERSITY
FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

COMP 204

Tetris 2048

Ali Mert Gök - 042001030
Cem Alp Özer - 042301097
Tolga Neslioğlu - 042301100

Instructor: Muhittin Gökmen (Prof. Dr.)

2025

ABSTRACT

This project involves creating a Python-based game that hybridizes Tetris and 2048 mechanics. The goal was to develop a falling-block puzzle where each Tetris piece's tiles carry numbers that merge like in 2048. The game was built with an object-oriented design, featuring a graphical interface, interactive controls, and full implementation of both Tetris rules (shape movement/rotation, line clearing) and 2048 rules (combining equal numbers into doubles). Key challenges included managing the grid updates when lines clear and tiles merge in chains, handling collision detection and rotation for complex shapes, and integrating a scoring system that recognizes merges up to the 2048 tile. The final solution successfully demonstrates a playable Tetris-2048 hybrid with smooth visuals, responsive controls, and a robust game loop, providing both a fun gameplay experience and insights into integrating two classic game algorithms.

Tetris 2048

Ali Mert Gök - 042001030

Cem Alp Özer - 042301097

Tolga Neslioğlu - 042301100

MEF UNIVERSITY
Faculty of Engineering
Department of Computer Engineering

Instructor: Prof. Dr. Muhittin Gökmen

APRIL, 2025

Keywords: Tetris, tetromino, tile, game grid, 2048

1. PROJECT CONTENT

1.1. Description of the Project

Tetris-2048 is a puzzle game that combines the well-known gameplay of Tetris with the numeric merging goals of 2048. In classic Tetris, geometric pieces called *tetrominoes* fall into a grid; the player moves and rotates them to form full horizontal lines, which clear from the board. In 2048, the player slides numbered tiles on a grid so that tiles with the same number collide and merge into higher-value tiles, aiming to create a tile with the number 2048. This project's concept merges these two ideas: Tetris tetrominoes fall in real-time, but each block of a tetromino has a number (2 or 4). When blocks of equal number stack on top of each other, they merge into a single block of double value (just like 2048). The objective for us is to keep clearing lines and merging tiles to eventually produce a 2048 score, without letting the stack reach the top of the grid. The game also keeps track of the highest score of the user.

The game is implemented on a grid (20 rows × 12 columns) and includes all fundamental features of both Tetris and 2048. The key features implemented are:

- **Merging:** If two tiles with the same number become vertically adjacent, they will merge into one tile with the next power-of-two value (e.g., two 4s merge into an 8). The merged tile inherits the position of the lower tile. The game supports chain merging – if a merge causes another set of identical tiles to align, subsequent merges happen in sequence automatically during that cycle. Each merge yields points equal to the new tile's value, which are added to the score.
- **Tetromino Movement and Rotation:** The seven standard Tetris piece shapes (I, O, Z, S, L, J, T) are present. The player can move the current piece left, right, or down, and rotate it clockwise or counter-clockwise. A hard drop function instantly drops the piece to the lowest possible position for quick placement.
- **Game Grid and Line Clearing:** The game grid holds fallen tiles. Whenever a horizontal line is completely filled with tiles, that line is cleared (removed), and any tiles above it fall down to fill the gap (as in normal Tetris). This helps prevent the stack from reaching the top. Clearing a line adds to the score (summing the values of tiles in that line).

- **Tetromino Generation and Preview:** New tetrominoes are generated randomly from the 7 types. Each new piece's individual tiles start with number 2 (or occasionally 4). The next upcoming tetromino is shown in a preview area, allowing the player to plan ahead.
- **Scoring System:** The score increases whenever merges or line clears occur. For merges, the value of the resulting merged tile is added; for a cleared line, the sum of all tile values in that line is added. The current score is continuously displayed, and a high score is tracked and saved between sessions (stored in a file) so that the game remembers the best score achieved.
- **Win and Lose Conditions:** The player wins by achieving a score of 2048. The game is lost if a new tetromino can no longer enter the grid because the tiles have stacked up to the top, triggering a game over. A game-over screen then notifies the player and shows the final score.
- **Graphical User Interface:** The game is rendered with a GUI library, displaying the grid, tiles, and interface panels. Each numbered tile is drawn as a colored square – the background color of the tile changes depending on its number (using the familiar 2048 color scheme). The grid has clearly drawn cell boundaries. Text is shown for score, high score, "Next" piece preview, and other messages.
- **User Interaction and Extra Features:** The game includes a start menu and pause menu for better user control. At launch, a start menu allows selection of difficulty (for example, choosing piece fall speed: Easy, Medium, Hard) before starting the game. A background image and music are played on the menu for polish. During gameplay, pressing "P" pauses the game and brings up a pause menu with options to resume or restart. The interface also shows a ghost piece (a semi-transparent projection of the current tetromino's shape at the position it would land if dropped straight down) to assist the player in positioning pieces. A restart button (or menu option) can reset the game at any time. Additionally, background music plays during the game, and sound effects could be added for merges or line clears (if enabled). All these extras enhance the overall experience beyond the basic requirements.

1.2. Description of the solution

Overall Architecture

The Tetris-2048 game uses a modular object-oriented design to combine elements of Tetris and 2048. The main function sets up the display, background music, and a menu for difficulty selection, then enters the game loop. It initializes a GameGrid to track board state and handles creating and switching Tetromino pieces (the current and next pieces). Tetromino and Tile classes model the moving pieces and their numbered blocks, while GameGrid applies the rules of the hybrid game. After each piece locks, the game first merges matching tiles, then clears any full rows. Finally, it uses an algorithm to remove floating tiles; this sequence of operations is handled by GameGrid. Overall, the architecture separates concerns cleanly, with each component focusing on its role, all within a structured object-oriented framework.

Tile class (tile.py):

Tile is a simple class for each numbered square in the game. It stores a value (2, 4, etc.) and automatically selects foreground and background colors based on that value. The Tile.draw method takes a grid coordinate and size, then renders a filled square with a border and displays the number in the center. This allows Tetromino and GameGrid to place tiles on the board without worrying about the drawing details. Because Tile encapsulates its appearance and number, the main classes can treat it as a unit.

Tetromino class (tetromino.py):

Tetromino represents a falling Tetris piece. It is created with one of seven shapes (I, O, Z, S, J, L, T), each defined by a small tile matrix indicating which cells are occupied. In the constructor, tiles in this matrix are created with values (usually 2, with 10% chance of 4) and positioned randomly above the top of the grid. The Tetromino has methods to move left, right, or down; each move first checks with can_be_moved to ensure the new position is valid (no overlap with existing tiles or walls). Rotation is handled by rotating the tile matrix and using can_be_rotated to prevent invalid rotations, undoing the change if needed. The draw method simply loops through the matrix and draws each occupied tile at its board position. In gameplay, pressing space makes

the piece drop straight down, and when it settles it provides its occupied cells to GameGrid for locking into the board. Each of the occupied cells in a Tetromino is a Tile object.

GameGrid class (game_grid.py):

GameGrid is the core logic for the board. It stores a 2D array (matrix) of Tile objects for all locked squares and tracks the current, next, and predicted tetrominos. The display methods draw the empty grid, the boundary box, and any occupied tiles, and also render a side panel showing the score and next piece. GameGrid keeps the score and provides reset functionality for a new game. It also has methods to lock a tetromino's tiles into the matrix and check for game-over if any tile is placed above the top. GameGrid also provides methods for merging, clearing, and cleaning up tiles. After a Tetromino lock, merge_tiles() scans each column, combining adjacent equal tiles (doubling the bottom tile's value and adding it to the score) and collapsing tiles downward. Next, clear_full_rows() removes any completely filled row, adding those values to score and shifting rows above down. Finally, handle_free_tiles() uses a flood-fill from the bottom to identify and remove any tiles not connected to the bottom (adding their values to score). This strict order—merging before clearing—is applied repeatedly until no more changes occur. The update_grid method locks falling pieces into the matrix and sets a game-over flag if any tile ends up above the top.

Main module (main.py):

Orchestrates the game, handling the game loop, user input, creating new tetrominoes, and high-level game events (start menu, pause and restart, game over, win).

Supporting Classes:

Supporting classes include simple utilities such as Point, which represents (x, y) coordinates for tile and tetromino positions. Point provides methods to move or translate coordinates, which is used when calculating tile placements after rotations or shifts. The project also uses library classes like Picture (for menu images) and Color to manage colors for tiles and text. These helper classes handle basic data or display tasks so that the main game logic classes remain focused on gameplay.

The **gameplay loop** follows the traditional Tetris cycle enhanced with 2048 logic:

The game begins with a start menu where the player selects a difficulty (Easy/Medium/Hard) to set the drop speed and clicks to start the game.

1. **Spawn Tetromino:** A new tetromino is created with a random shape and random initial tile numbers. The piece starts at a random horizontal position at the top of the grid. For example, a tetromino might spawn containing tiles [2, 2, 4, 2] in its 4 blocks.
2. **Player Input:** While the tetromino falls, the player can press keys:
    - Left/Right arrow keys to move the piece horizontally by one cell.
    - Down arrow for a *soft drop*, moving the piece down faster.
    - Space for a *hard drop* (instantly drop the piece to the lowest possible position).
    - C / Z keys to rotate the piece clockwise or counter-clockwise.
    - P key to pause/resume the game.
3. **Automatic Fall:** The piece continuously drops one cell at a fixed interval. Each iteration of the game loop, if no input prevents it, the tetromino moves down by one.
4. **Lock and Merge:** When the tetromino can no longer move down (either it reached the bottom or it landed on top of placed tiles), it is "locked" into the grid. At this point, its individual tiles transfer into the GameGrid. Then the 2048 merge check runs: any vertically adjacent tiles with the same number will merge into one (e.g., two 4's merge into an 8) . Merging is done column by column, from the bottom up, allowing chain reactions (a merge can cause another merge above if the same numbers align after the collapse).
5. **Clear Lines:** After merging, the game checks for full horizontal lines. Each full line is cleared, awarding points equal to the sum of the numbers that were in that line . When a line is cleared, all lines above it drop down by one. Multiple lines can be cleared at once if several rows become full.
6. **Remove Free Tiles:** After clears, any tile not supported by a tile underneath (directly or via a connected stack to the floor) is removed . This prevents any odd "floating" tiles that could result from the merges and clears, effectively simulating gravity for any disjoint tiles.

7. **Score Update:** The score is updated throughout the above steps. Each merge adds the value of the resulting merged tile to the score, and each cleared line adds the sum of its tiles' values to the score. The current score is continually displayed on the screen (typically at the top of the grid).
8. **Next Piece:** A new tetromino is spawned and the cycle repeats. The game continues until a new tetromino cannot be placed because the pile of tiles has reached the top of the grid (Game Over), or until the player creates a 2048 tile (optional win condition).

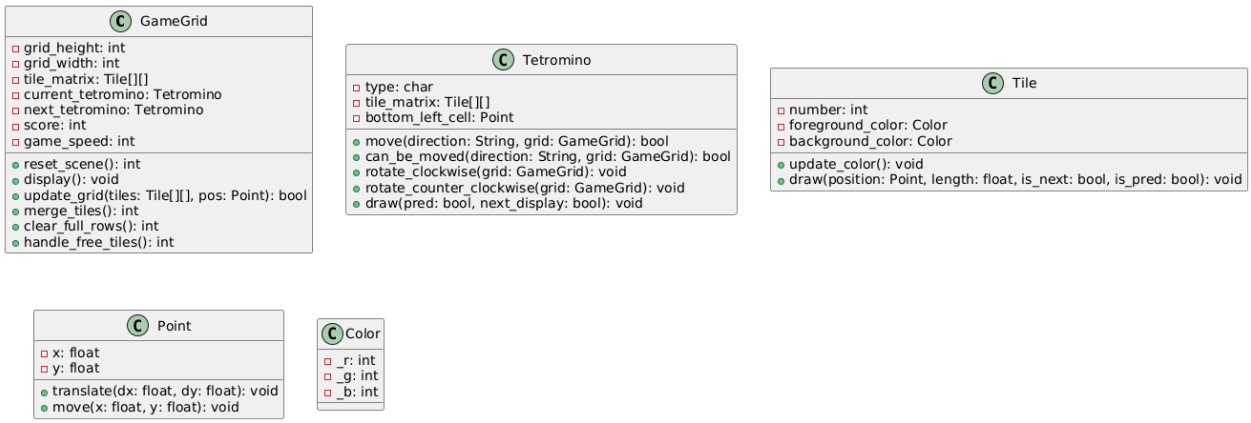

Figure 1. UML Diagrams.

2. DIVISION OF RESPONSIBILITIES

Cem Alp Özer	Tile Numbers, Tile Colors, New Random Piece, Update Grid (Landed Tetromino), Merging Tiles, Chain Merging, Handling Free Tiles, Game Music.
Tolga Neslioğlu	Tetrominoes (Different Shapes), Hard Drop, New Random Piece, Clear Full Lunes, Game Cycle, Game Difficulty.
Ali Mert Gök	Rotate, show next piece, computing and showing the score, win and loss, pause and restart, GUI design, show high score, tetromino drop prediction (ghost piece)

3. EXAMPLES

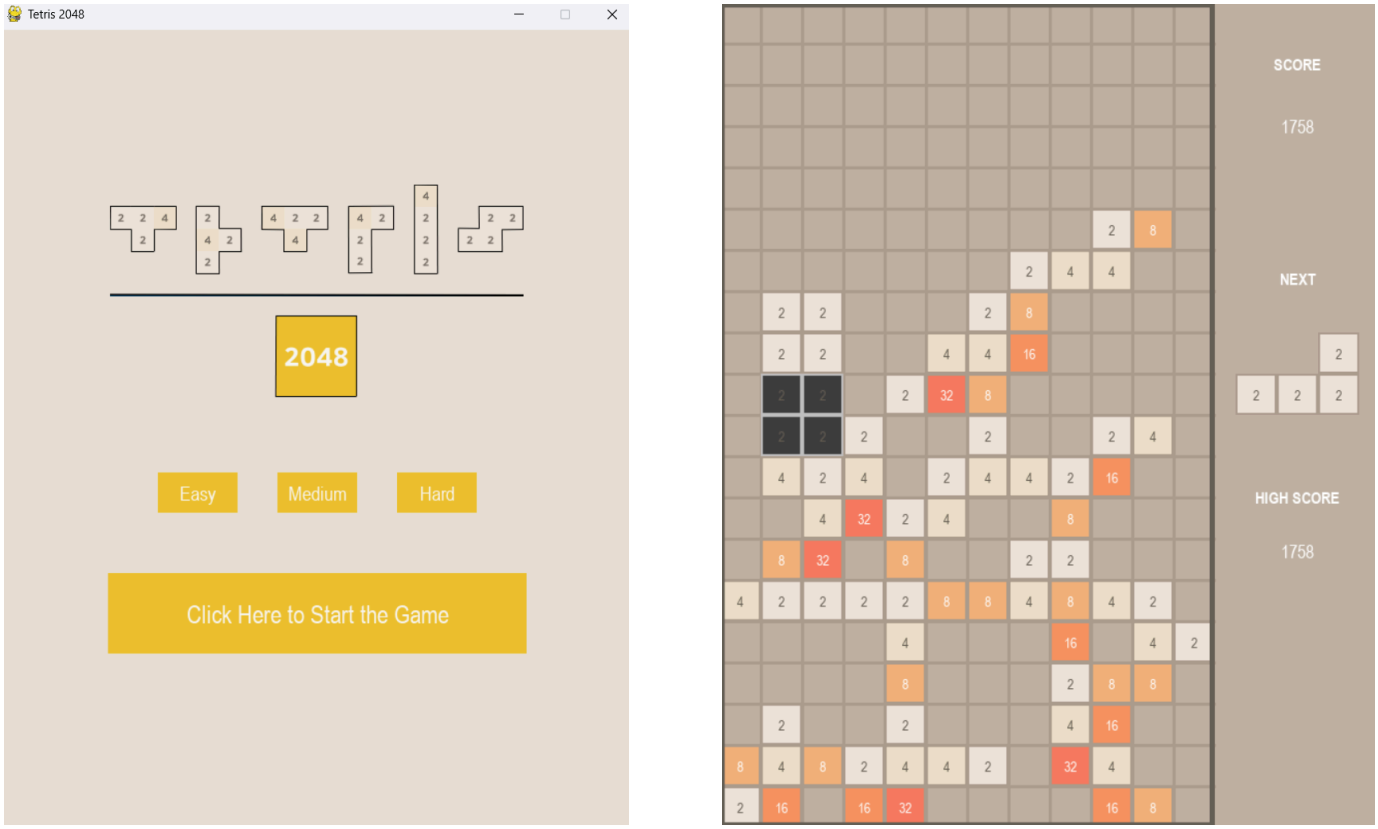

Figure 2. Game interface elements: menus, ghost piece, score, high score, and next piece.

Figure 3. Win, Game Over, and Pause screens.

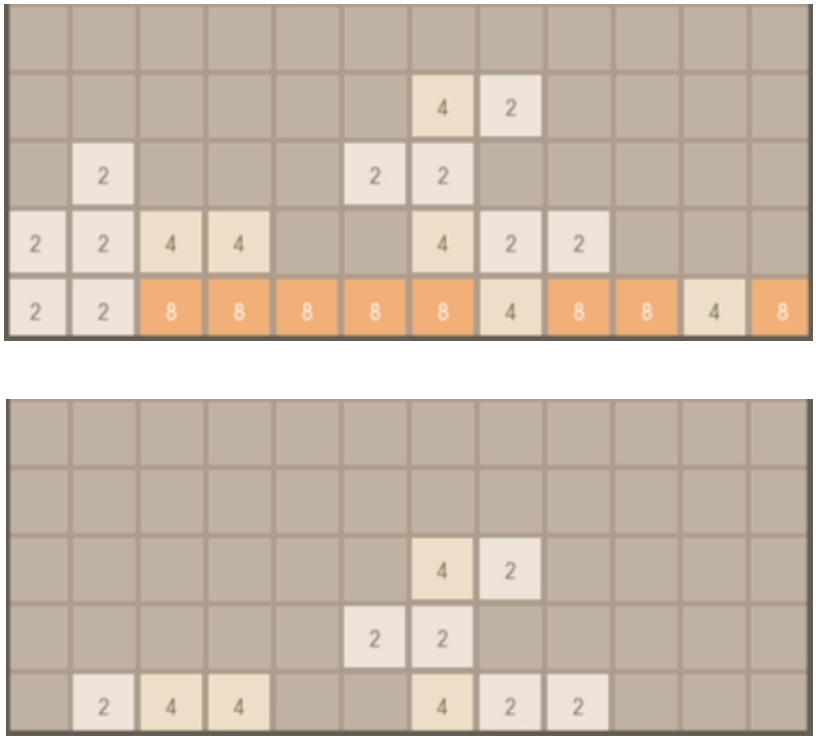

Figure 4. Clearing Full Lines

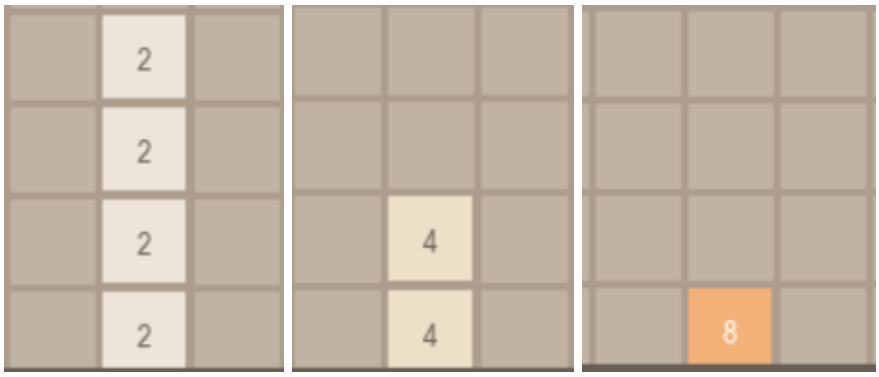

Figure 5. Merging

Figure 6. Handling Free Tiles

After a piece locks, the grid first runs **merge_tiles()**, which scans each column bottom-up to combine matching values and collapse the column, repeating this loop until no further merges occur—this handles any chain-merge reactions in a single drop. Each merge doubles the lower tile's value, updates its color, and adds to the score. Once merging stabilizes, **clear_full_rows()** removes any filled rows and shifts the remaining tiles down, which can create gaps. Finally, **handle_free_tiles()** performs a flood-fill from all bottom-row cells to mark supported tiles; any unmarked "floating" tiles are removed (their values added to score) to ensure only physically connected tiles remain. By interleaving these steps—merge, clear, flood-fill—in a loop, the game resolves every chain reaction and clears all unsupported tiles before the next piece falls.

4. CHALLENGES AND ACHIEVEMENTS

Working on this project was both challenging and rewarding. One major challenge was integrating two distinct game mechanics into a cohesive experience. Initially, the Tetris portion (handling falling pieces, movement, rotation, and line clearing) was familiar, but adding the 2048 merging rules introduced new complexity. We had to ensure that whenever identical-numbered tiles stacked, they merged correctly without breaking the Tetris physics. Debugging the merge algorithm required careful thought – for example, handling multiple merges in the same column and preventing a tile from merging twice in one drop. We overcame this by implementing a clear step-by-step merge procedure and testing it with various scenarios (dropping pieces that would cause one merge vs. chain merges). Seeing the first successful chain reaction (e.g., two 2's merging into 4, then two 4's immediately merging into 8) was a big milestone in the project.

Another challenge was managing the game state updates when many events could occur at once. For instance, a single piece lock-in might trigger a merge, which could then complete a line, which then causes tiles to drop and possibly merge again. Ensuring the code handled these in the correct order without errors taught us a lot about controlling program flow and edge cases. We learned to use techniques like flood-fill (for detecting floating tiles) and file I/O (for saving high scores) as part of the game logic.

Another major challenge we faced was in the area of interface design; it had to be clean and intuitive. The key features we needed to implement included the next piece preview, score and high score display, and ensuring proper functional alignment of the pause, restart, and win/lose screens. Additionally, we had to incorporate both the rotation mechanics and the ghost piece functionality to enhance gameplay strategy. Despite these challenges, we managed to maintain a smooth user experience, allowing players to track their progress and interact with the game effortlessly. It was truly rewarding to see how we successfully integrated dynamic UI elements with the core game logic, all while preserving responsiveness and clarity.

Throughout the development, we learned a great deal about object-oriented design and working with GitHub. Working with separated classes like Grid, Tetromino, etc. clarified our thinking and made debugging easier, since we could isolate whether a problem was in movement, collision detection, or merging. We also improved our skills in handling user input and real-time events.

The project reinforced the importance of incremental development and testing: we built the game in stages (first basic Tetris movement like rotating, then line clearing, then merging, etc.), verifying each part before adding the next. This iterative approach helped catch issues early.

In terms of achievements, we are proud that we successfully implemented all the required features and even added polished extras like a pause menu, a next-piece preview, ghost piece indicator, and music – these taught us how to manage state (pausing and resuming the game loop). By the end, we had a fully functional Tetris-2048 game that is fun to play and demonstrates complex interactions between game rules. Completing this project deepened our understanding of game development and algorithms, and it was very satisfying to see the final product working as intended, combining the best of both classic Tetris and 2048.

References

[1] M. Gökmen, "COMP 204 – Project 2: Tetris 2048," lecture slides, Computer Engineering Dept., MEF University, Istanbul, Turkey, Apr. 2025.

[2] MEF University, "Tetris 2048 base code documentation," course handout, COMP 204 Programming Studio, Istanbul, Turkey, Apr. 2025.