Object Oriented Programming Project

~~Sand Tetris~~ Tetris

Developed By

6704062660027 Wachirawit Kosawanichakit

Present to

—

This report is a part of the Object Oriented Programming Course

Semester 1/2025

King Mongkut's University of Technology North Bangkok

Class Preface

This project report is part of the Object-Oriented Programming (OOP) course for the first semester of the academic year 2025 (1/2568). The purpose of this project is to apply OOP concepts in a practical way by developing a playable and functional game — ~~**Sand Tetris**~~ **Tetris**

Class Introduction

Project Title {

~~Sand Tetris~~ Tetris

}

Type of Project {

Java game applet

}

Project Significance {

Tetris is one of the most classic puzzle games, based on logic and quick decision-making. It serves as an excellent candidate for implementing Object-Oriented Programming (OOP), as the game can be separated into independent objects such as blocks, shape, the game board, collision system and the graphical user interface (GUI).

}

Expected Benefits {

- Gain practical experience in applying OOP concepts in a real project.
- Understand how GUI applications are structured using Java Swing.
- Be able to extend the knowledge gained to develop other games or applications in the future.

}

Scope of the project {

This project is a Java-based puzzle game, specifically a “Tetris MVP” (Minimum Viable Product)

Core gameplay {

The core involves the player controlling falling shapes, called “**Tetrominoes**” the player can move the shapes left and right using **A/D** or arrow keys and rotate them using the **W** or up arrow key. The objective is to form complete horizontal lines, which are then cleared from the board, causing the blocks above to fall. The game ends (Game Over) when the stack of blocks reaches the top of the top of the screen. The application supports a restart feature, allowing the player to begin a new game by pressing **Enter** or **left-clicking** the mouse.

}

The application is built entirely as a Java Application using the javax.swing library for its graphical user interface.

}

Project Proposal {

Project Proposal: Sand Tetris

1. หัวข้อ

Sand Tetris – เกม Tetris เวอร์ชันดัดแปลงที่บล็อกเมื่อถึงพื้นจะแตกออกเป็นชิ้นเล็ก ๆ หรือเม็ดทราย (Sand) ซึ่งจะไหลและกองทับกันแทนที่จะคงเป็นบล็อกดั้งเดิม

2. เนื้อหาเกี่ยวกับเกม

Sand Tetris เป็นเกมแนว Puzzle ที่พัฒนาต่อยอดจากเกม Tetris แบบดั้งเดิม
ผู้เล่นควบคุมบล็อกที่ตกลงมาจากด้านบนเช่นเดียวกับ Tetris ปกติ แต่เมื่อบล็อกตกถึงพื้น มันจะสลายตัวกลายเป็นชิ้นเล็ก ๆ หรือเม็ดทราย (cell 1x1) ที่จะไหลลงมาตามแรงโน้มถ่วง
เป้าหมายของผู้เล่นคือทำให้แถวเต็มเพื่อเคลียร์และทำคะแนน
หากเม็ดทรายกองสูงจนเต็มด้านบนของจอ เกมจะจบลง

3. Storyboard คร่าว ๆ

1. เริ่มเกม – หน้าจอแสดงกระดานว่าง
2. บล็อกสุ่มปรากฏด้านบนและเริ่มตกลงมา
3. ผู้เล่นควบคุมบล็อก (ซ้าย/ขวา/หมุน) ระหว่างการตก
4. เมื่อบล็อกถึงพื้น → แตกออกเป็นเม็ดทราย
5. เม็ดทรายกองทับกัน ถ้าเต็มแถว → เคลียร์
6. ถ้าทรายกองถึงด้านบน → เกมจบ

4. ประโยชน์

- ฝึกทักษะการเขียนโปรแกรมเชิงวัตถุ (OOP) โดยแบ่งคลาสเช่น Block, Particle, Board, GameEngine
- ฝึกคิดเชิงตรรกะและอัลกอริทึมสำหรับการจำลองการตกของวัตถุ (gravity simulation)
- สร้างความคิดสร้างสรรค์โดยการนำเกมคลาสสิกมาดัดแปลงเป็นเวอร์ชันใหม่
- เสริมทักษะการแก้ปัญหาและการออกแบบระบบ

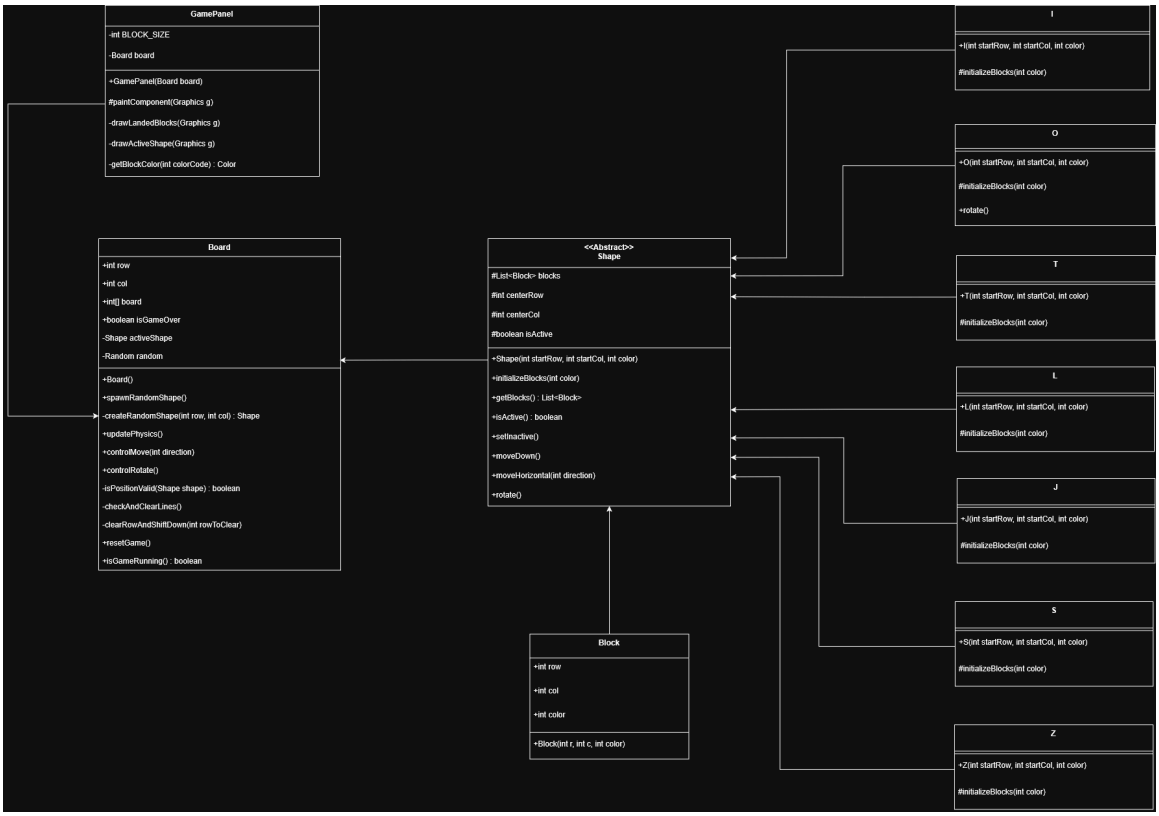
5. ตารางแผนงาน (2 เดือน)

สัปดาห์ 1–2: ออกแบบ class, สร้าง grid และระบบ block ตกแบบ Tetris ปกติ
สัปดาห์ 3: ทำระบบแตก block → กลายเป็น cell 1x1
สัปดาห์ 4: ทำ gravity simulation ให้ cell ไหลลง/ซ้าย/ขวาได้
สัปดาห์ 5–6: ทำระบบเคลียร์แถวและการนับคะแนน
สัปดาห์ 7: ปรับปรุง UI เบื้องต้น, ใส่ระบบคะแนน
สัปดาห์ 8: ทดสอบและปรับปรุงขั้นสุดท้าย, เตรียมส่งงาน

}

Class Development

Class Diagram {



Main {

This is the main launcher class. It is responsible for creating the game window (JFrame), the game screen (GamePanel), and the primary Game Loop. It also manages all Event Handling (KeyListener and MouseListener). **}**

GamePanel {

A class that extends JPanel to act as the "View" of the game. It overrides the paintComponent method to render the current game state (both landed blocks and the active shape) onto the screen.

}

Board {

This is the "brain" or "Game Engine". It holds the state of the grid (board[]) and all core logic, including movement (controlMove), rotation (controlRotate), line clearing (checkAndClearLines), and the game-over state (isGameOver).

}

Shape {

An abstract class that serves as a blueprint for the seven tetrominoes (e.g., I, O, T). It contains shared methods like rotate() and moveDown().

}

Block {

The smallest data class, used to store the coordinates (row, col) and color of a single square.

}

}

Application of Object-Oriented Principles {

The project's architecture is fundamentally built upon core OOP concepts.

Inheritance and Abstraction {

Inheritance and Abstraction are most evident in the **Shape** class structure. **Shape.java** is an abstract class that serves as a blueprint for all seven Tetrominoes. It provides shared functionalities like `moveDown()` and `Rotate()`. Critically, it contains an abstract method, `initializeBlocks()`, which forces each concrete subclass (like **I.java**, **T.java, O.java, etc.**) to define its own unique shape.

}

Polymorphism {

Is clearly demonstrated through method **Overriding**. While most shapes inherit the standard `rotate()` logic from the parent **Shape** class, the **O.java** class (the square) overrides this method with an empty one, as it does not need to rotate. Similarly, our **GamePanel** class overrides the `paintComponent()` method from JPanel to implement custom rendering logic.

}

Composition {

Composition is used throughout. A **Shape** object is composed of a `List<Block>`. The Board object has-a **Shape** (`activeShape`). Finally, our `GamePanel(Board board)`, which accepts the game engine upon creation. **}**

Encapsulation {

Encapsulation is central to the **Board.java** class, which acts as the "brain" of the game. It hides all complex game logic—such as physics updates (`updatePhysics()`), collision detection (`controlMove()`), and line clearing (`checkAndClearLines()`)—from the **Main** class. The **`Main`** class only needs to call high-level methods like `board.updatePhysics()` without needing to know the implementation details.

}

Constructors {

Constructors are used in all classes to initialize their state, such as `Block(r, c, color)` and `GamePanel(Board board)`, which accepts the game engine upon creation.

}

}

GUI and Event Handling {

The Graphical User Interface (GUI) is built with javax.swing. A JFrame acts as the main application window. Inside it, we place a custom GamePanel (which extends JPanel), serving as the canvas where all game graphics are rendered.

Event Handling {

Event Handling is managed through two listeners attached to the JFrame. A `KeyListener` (using `KeyAdapter`) listens for keyboard input. The `keyPressed()` method checks the key code if it matches A, D or W, it calls the corresponding `board.controlMove()` or `board.controlRotate()` methods.

A `MouseListener` (using `MouseAdapter`) is also implemented. Both listeners check if the game is in a “Game Over” state; if so, an ENTER key press or a `MouseEvent.BUTTON1` (left-click) will trigger the `board.resetGame()` method to restart.

}

}

Key Algorithms {

Line Clearing Algorithm {

This algorithm, found in checkAndClearLines(), iterates through the board in a for-loop starting from the bottom row upwards (r--). When a "full" row (one with no empty spaces) is detected, it calls clearRowAndShiftDown(). This method copies all data from every row above (r-1) and shifts it down to the current row (r), clearing the topmost row. A crucial step is to then call r++,

forcing the loop to "re-check" the same row index (which now contains new data) in case of multi-line clears.

}

Rotation Algorithm {

To solve the problem of shapes rotating through walls or other blocks, a "Rotate-and-Revert" algorithm is used in `controlRotate()`. It follows these steps

Try Rotate {

The algorithm first applies the rotation optimistically (`activeShape.rotate()`). **}**

Check {

It then calls `isPositionValid()` to check if the shape's new position is colliding with walls, the floor, or other locked blocks. **}**

Decide {

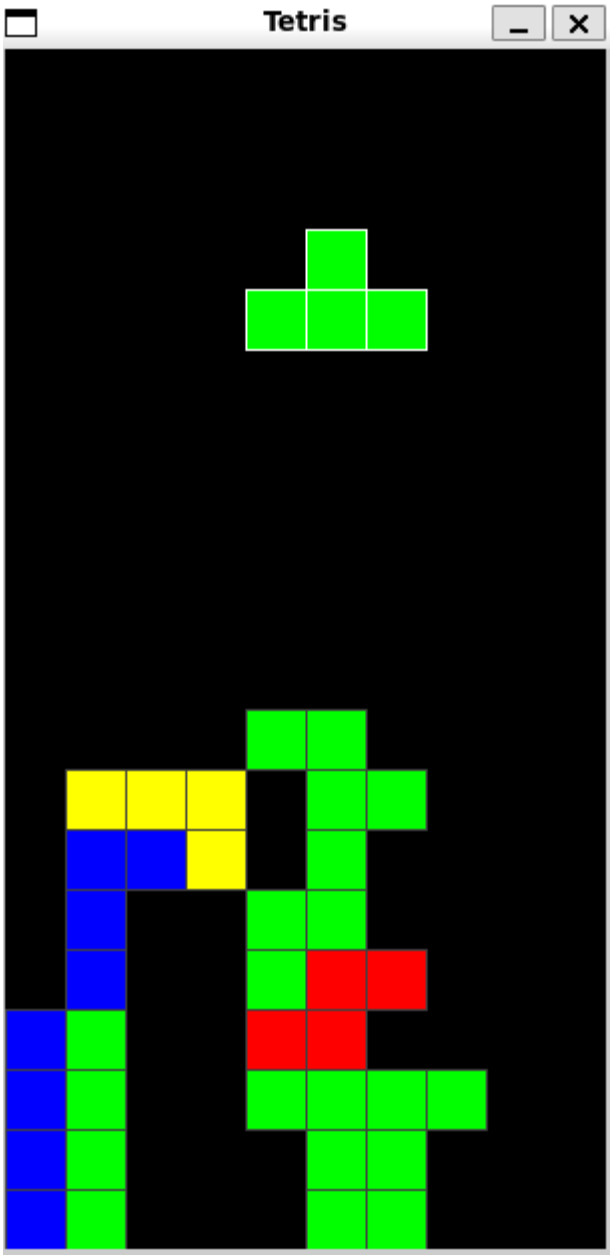
If **`true`** (no collision): The rotation is valid and allowed to complete.

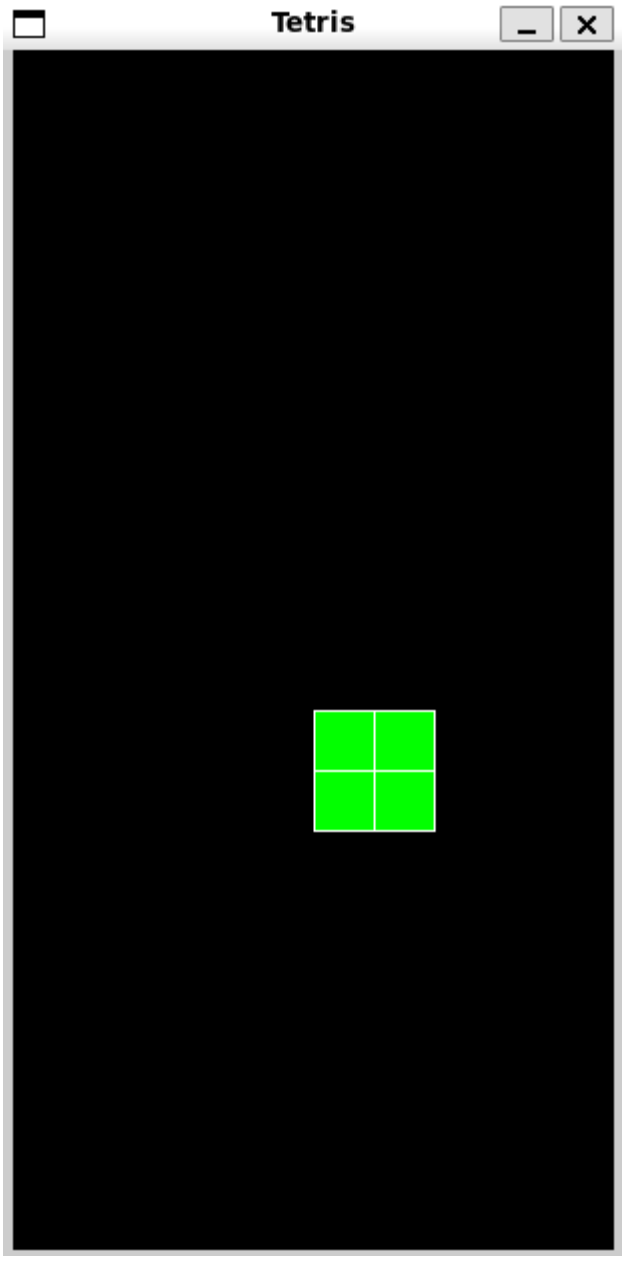
If **`false`** (collision): The rotation is invalid. The algorithm calls `activeShape.rotate()` three more times. Since the rotation is a 90-degree counter-clockwise turn, rotating four times (1+3) equals 360 degrees, effectively reverting the shape to its original position before the collision occurred. **}**

}

}

Screenshot of the game running




Tetris

Summary

Obstacles and Challenges {

A significant initial challenge was the Spaghetti Code from the first version, which was complex and buggy.

Another major technical challenge was optimizing the core data structure for performance, specifically regarding **Cache Locality**.

Initial Design (Array of Objects) {

The first design used `Block board[] = new Block[row * col];`, where Block was an object containing a Position and a Color. While this is a classic OOP approach, it resulted in poor performance because the array stored *references* to objects scattered across the heap. Iterating this array during updatePhysics caused a massive 191 million cache misses (16.19%), as the CPU constantly had to fetch data from main memory.

}

Refactored Design (Primitive Array) {

This was refactored to use a simple primitive array: `int board[] = new int[row * col];`. In this design, 0 represented an empty space, and integers 1-4 represented colors. This single change guaranteed that all board data was stored in a single, contiguous block of memory.

}

The Result {

This new structure provided excellent cache locality. When the updatePhysics loop accessed `board[i]`, the adjacent `board[i-1]` was already loaded into the CPU's high-speed cache. This optimization cut cache misses by almost 50% (down to 96 million) and nearly halved the total processor cycles.

}

Finally, several other critical gameplay bugs were identified and resolved, including an ArrayIndexOutOfBoundsException caused by incorrect collision-check ordering, a rotation bug that allowed shapes to "tunnel" through walls, and a rendering bug that caused active shapes to be drawn incorrectly.

}

Project Highlights {

The project's main success was the Refactoring effort. The architecture was successfully converted from a complex console application into a clean, stable GUI application using a modern Game Loop. The final Game Loop successfully decouples Framerate (FPS) from Game Speed (Physics), allowing for a smooth 60 FPS visual update while the game logic runs at a consistent, separate pace. The project serves as a strong, practical application of all major OOP principles and demonstrates a deep understanding of data-oriented design for performance (Cache Locality).

}

Future Recommendations {

For future development, Encapsulation could be further improved by making `board.board[]` private and providing a getter method. New features like a scoring system, increasing difficulty, or a "Next Piece" preview could be added. Finally, the rotation system could be upgraded from the current "rotate-and-revert" logic to a more standard "Wall Kick" or Super Rotation System (SRS) for a smoother player experience.

}