Old School Tetris
DESIGN DOCUMENT

Introduction

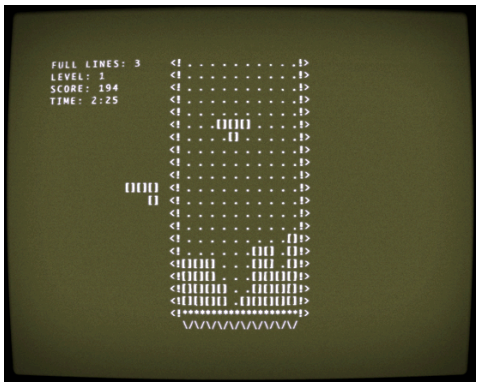
Game Summary Pitch

Old School Tetris is a puzzle game taking great inspiration from the already well-established block-puzzle game Tetris. It aims to take the look and feel of the original Tetris released in 1984 while adding newer features from modern iterations of the title.

Inspiration

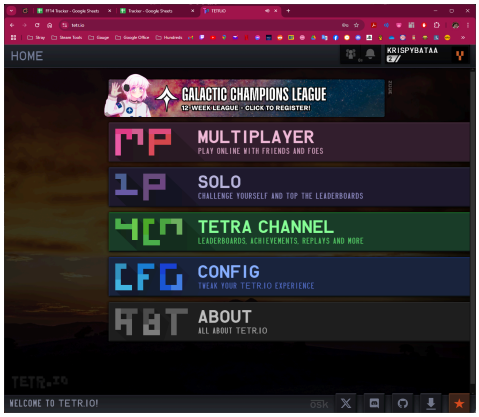
Original Tetris

The original Tetris was released in 1984 from Moscow. **Tetris** at its core was the a vanilla puzzle game where the player fits **falling blocks into a line** with the aim to **clear the bottom of the board**. Given its release window, it sports a **vintage design style** that is well-suited to the team's artistic capabilities.



Tetr.io

A modern, online rendition of the Tetris formula with online multiplayer and other **game modes** that enhance the player experience alongside other evolutions



Player Experience

Players will experience a fast-paced, mentally engaging challenge that tests their quick decision-making and spatial awareness. The increasing speed as the game progresses keeps the tension and excitement high, while the addition of a bomb piece introduces a strategic element to clear the board.

Platform

To be released primarily on Windows

Development Software

- IntelliJ IDEA | Main Programming

Genre

Singleplayer, Puzzle, Casual

Target Audience

Blockmates, Casual Players

Concept

Gameplay overview

After interacting with the **main menu** the player is greeted with an empty screen. The game requires the player to fit the falling blocks together like a puzzle. There are **seven shapes** with **four blocks** each, with each block falling at a fixed speed (varies per level). The overall **goal** is to rotate and form the shapes until they form a solid row without gaps. (Description borrowed from Temple University).

Primary Mechanics

Mechanic
Blocks | Block movement The player's main task, is to rotate a falling block to fit into the grid
Row Clearing When a row is perfectly filled with blocks, it is 'cleared' and the player earns a score
Score and Difficulty With a total of 3 levels (the varying factor being block falling speed), the player will aim to clear 40 rows per level

Audio

Music and Sound Effects

Using publicly available audio, the team is considering importing them into the game that would be made.

Game Experience

UI

Acknowledging the team's limited digital art capabilities, the team decided to be faithful to the original game's ASCII-based based art-style, what would otherwise be called retro in feel. Utilizing as few extraneous art resources as possible, potentially gives the look that the game is being run in the console, emphasizing the 'Old School' part of the title.

Controls

Keyboard

Arrow keys, Spacebar

GitHub Repository

- Embedded here is the link to the GitHub Repository

OS CONCEPTS APPLIED

- Thread Implementation present in the Game.java

```
//Gameplay Threads
private Thread threadAnimation;
private Thread threadAutoDown;
private Thread threadLoaded;
private long playTime;
private long lTimeStep;
final static int INPUT_DELAY = 40;
private boolean bMuted = true;
private boolean isRestarting = false;
```

- Concurrency Control via implementing the restartGame() method that prevents from restarting too soon

```
public void restartGame() {
        long currentTime = System.currentTimeMillis();
        if (currentTime - lastRestartTime < RESTART_COOLDOWN ||
isRestarting) {
            return; // Ignore restart if too soon or already restarting
        }

        isRestarting = true;
        lastRestartTime = currentTime;

        try {
            // Stop all existing threads first
            stopThreads();

            // Reset the game state
            GameLogic.getInstance().clearBoard();
            GameLogic.getInstance().initGame();
            GameLogic.getInstance().setbPlaying(true);
```

```
            GameLogic.getInstance().setbPaused(false);
            GameLogic.getInstance().setbGameOver(false);
            GameLogic.getInstance().setbRestarted(true);

            // Reset the screen
            gmsScreen.resetGame();
```

- The run() method sets the thread priority to Thread.MIN_PRIORITY to manage CPU allocation among threads.

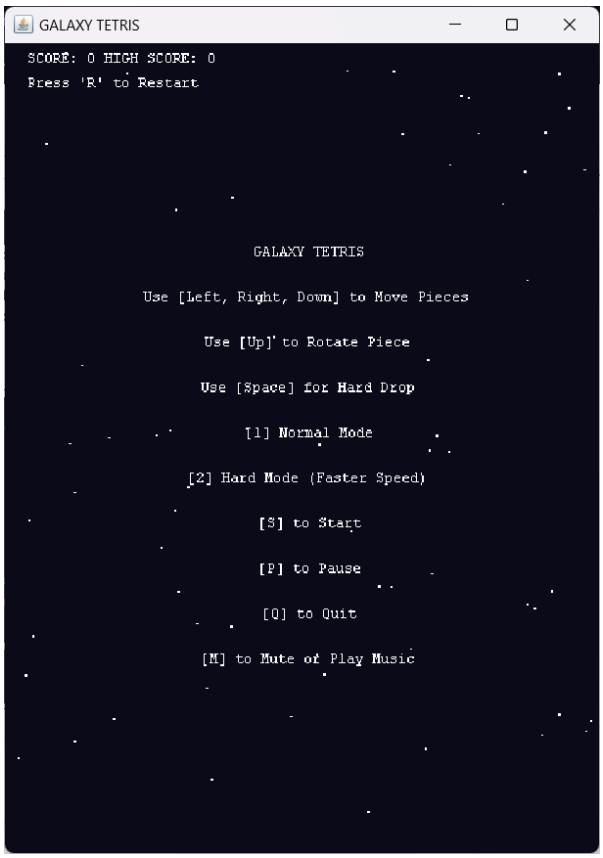
THREADS IN DETAIL

```
public void run(){
        Thread.currentThread().setPriority(Thread.MIN_PRIORITY);

        long lStartTime = System.currentTimeMillis();
        if(!GameLogic.getInstance().isbLoaded() && Thread.currentThread() ==
threadLoaded){
            GameLogic.getInstance().setbLoaded(true);
        }

        while(Thread.currentThread() == threadAutoDown){
            if(!GameLogic.getInstance().isbPaused() &&
GameLogic.getInstance().isbPlaying()){
                tryMovingDown();
            }
            gmsScreen.repaint();
            try{
                lStartTime += nAutoDelay;
                Thread.sleep(Math.max(0, lStartTime -
System.currentTimeMillis()));
            } catch (InterruptedException e){
                break;
            }
        }
```

```
        while (Thread.currentThread() == threadAnimation){
            if(!GameLogic.getInstance().isbPaused() &&
GameLogic.getInstance().isbPlaying()){
                updateGrid();
            }

            gmsScreen.repaint();

            try{
                lStartTime += ANIMATION_DELAY;
                Thread.sleep(Math.max(0, lStartTime -
System.currentTimeMillis()));
            } catch (InterruptedException e){
                break;
            }
        }
    }
```

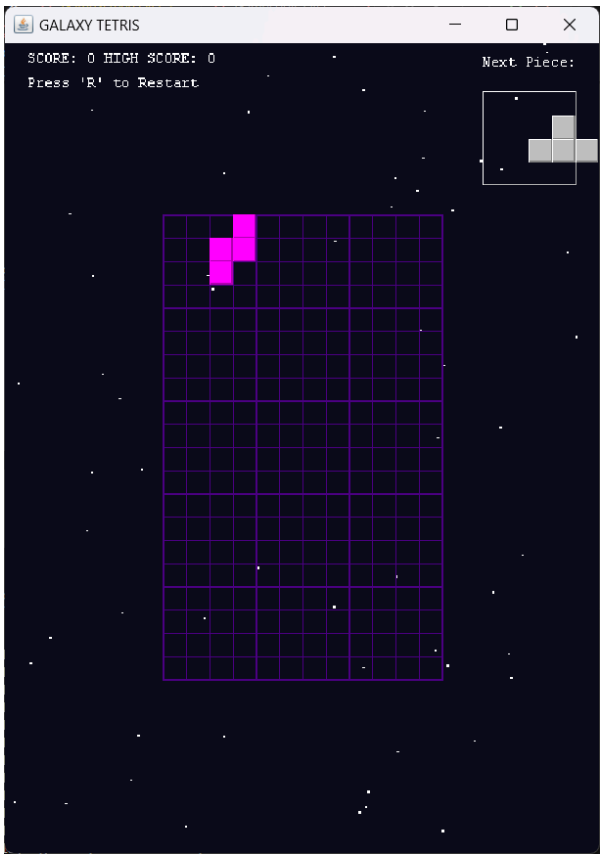
- The code above ensures that the three main threads (threadAnimation, threadAutoDown, threadAutoload) run in sync with each other. In the table below, the code produces different errors whenever one thread is commented out since these three threads are the main methods that make the game run properly.

```
public void callThreads(){
        // if(threadAnimation ==
null){
        //     threadAnimation = new
Thread(this);
        //
threadAnimation.start();
        // }
        if(threadAutoDown == null){
            threadAutoDown = new
Thread(this);
            threadAutoDown.start();
        }


if(!GameLogic.getInstance().isbLoaded(
) && threadLoaded == null){
            threadLoaded = new
Thread(this);
            threadLoaded.start();
        }
    }
```



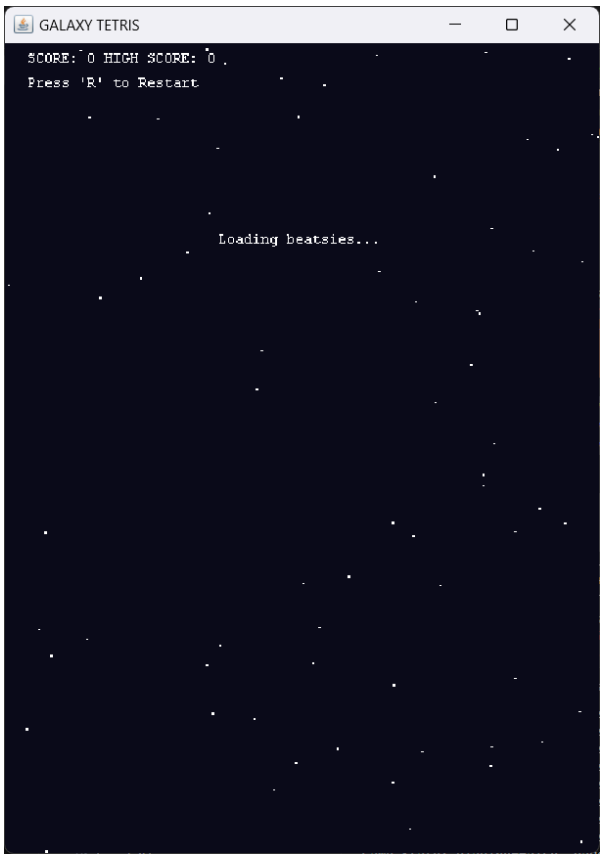
The game state freezes and the animation isn't rendering properly if threadAnimation is removed.

```
public void callThreads(){
        if(threadAnimation == null){
            threadAnimation = new
Thread(this);
            threadAnimation.start();
        }
        // if(threadAutoDown == null){
        //     threadAutoDown = new
Thread(this);
        //     threadAutoDown.start();
        // }


if(!GameLogic.getInstance().isbLoaded(
) && threadLoaded == null){
            threadLoaded = new
Thread(this);
            threadLoaded.start();
        }
    }
```



The tetronimoes (tetris pieces) are not falling properly when threadAutoDown is removed.

```
public void callThreads(){
        if(threadAnimation == null){
            threadAnimation = new
Thread(this);
            threadAnimation.start();
        }
        if(threadAutoDown == null){
            threadAutoDown = new
Thread(this);
            threadAutoDown.start();
        }

        //
if(!GameLogic.getInstance().isbLoaded(
) && threadLoaded == null){
        //      threadLoaded = new
Thread(this);
        //      threadLoaded.start();
        // }
    }
```



The game is not switching from loading state to game start when threadLoaded is removed.

- The code uses flags such as isRestarting and bMuted to manage the state of the game and ensure that operations like restarting the game or playing music are performed correctly and without conflict.

```
public void restartGame() {
        long currentTime =
System.currentTimeMillis();
        if (currentTime -
lastRestartTime < RESTART_COOLDOWN
|| isRestarting) {
            return; // Ignore
restart if too soon or already
restarting
```

```
// Reset music if not muted
            if(!bMuted){

if(clipBGM.isRunning()) {
                    clipBGM.stop();
                }

clipBGM.setFramePosition(0);
```

```
        }

        isRestarting = true;
        lastRestartTime =
currentTime;
```

```
clipBGM.loop(Clip.LOOP_CONTINUOUSLY
);
            }
```

- The class extends Panel is responsible for the game elements, which includes the grid, tetrominoes, and the score.

```
public class GameScreen extends Panel {
    private Dimension dimOff;
    private Image imgOff;
    private Graphics grpOff;
    public Grid grid = new Grid();
    private GameFrame gmf;
    private Font font = new Font("Monospaced", Font.PLAIN, 12);
    private Font fontBig = new Font("Monospaced", Font.PLAIN +
Font.ITALIC, 36);
    // Galaxy theme colors
    private final Color spaceBackground = new Color(0x0B0B1A); // Deep
space blue
    private final Color starColor = new Color(0xFFFFFF); // White for
stars
    private final Color gridLineColor = new Color(0x4B0082); // Indigo
for grid lines
    private final Color[] galaxyColors = {
        new Color(0xFF69B4), // Hot pink
        new Color(0x9370DB), // Medium purple
        new Color(0x00CED1), // Dark turquoise
        new Color(0x7FFFD4), // Aquamarine
        new Color(0xFF6347), // Tomato red
        new Color(0x98FB98), // Pale green
        new Color(0xDDA0DD)  // Plum
    };
    private FontMetrics fontMetrics;
    private int nFontWidth;
```

```
    private int nFontHeight;
    private String strDisplay = "";
    public Tetronimo tetronimoOnDeck;
    public Tetronimo tetronimoCurrent;
    private Timer timer;
```

- Includes a KeyAdapter to listen for key events, such as pressing 'R' to restart the game. This is an example of handling asynchronous user input events, which is a common concept in GUI applications.

```
this.addKeyListener(new KeyAdapter() {
            @Override
            public void keyPressed(KeyEvent e) {
                if (e.getKeyCode() == KeyEvent.VK_R) {
                    restartGame();
                }
            }
        });
```

- A Timer is used to manage periodic actions, which could be related to game updates or animations. While not explicitly shown in the viewed lines, timers are

often used in games to handle regular updates without blocking the main thread.

```
public class GameScreen extends Panel {
    private Dimension dimOff;
    private Image imgOff;
    private Graphics grpOff;
    public Grid grid = new Grid();
    private GameFrame gmf;
    private Font font = new Font(name:"Monospaced", Font.PLAIN, size:12);
    private Font fontBig = new Font(name:"Monospaced", Font.PLAIN + Font.ITALIC, size:36);
    // Galaxy theme colors
    private final Color spaceBackground = new Color(rgb:0x0B0B1A); // Deep space blue
    private final Color starColor = new Color(rgb:0xFFFFFF); // White for stars
    private final Color gridLineColor = new Color(rgb:0x4B0082); // Indigo for grid lines
    private final Color[] galaxyColors = {
        new Color(rgb:0xFF69B4), // Hot pink
        new Color(rgb:0x9370DB), // Medium purple
        new Color(rgb:0x00CED1), // Dark turquoise
        new Color(rgb:0x7FFFD4), // Aquamarine
        new Color(rgb:0xFF6347), // Tomato red
        new Color(rgb:0x98FB98), // Pale green
        new Color(rgb:0xDDA0DD)  // Plum
    };
    private FontMetrics fontMetrics;
    private int nFontWidth;
    private int nFontHeight;
    private String strDisplay = "";
    public Tetronimo tetronimoOnDeck;
    public Tetronimo tetronimoCurrent;
    private Timer timer;
```

Sources

- https://tetris.com/history-of-tetris
- https://tetris.com/article/128/what-was-the-world-like-the-year-tetris-was-born#:~:text=Find%20out%20a%20little%20bit,audiences%20all%20across%20the%20globe.
- https://news.temple.edu/news/2023-03-22/falling-place-piecing-together-tetris-enduring-legacy-0#:~:text=Tetris%20begins%20with%20an%20empty,complete%20solid%20rows%20without%20gaps.