НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. ІГОРЯ СІКОРСЬКОГО» ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАНОЇ ТЕХНІКИ КАФЕДРА ОБЧИСЛЮВАНОЇ ТЕХНІКИ

КУРСОВА РОБОТА

з дисципліни «Інженерія програмного забезпечення» на тему: «Веб-сайт для гри в тетріс»

Студента II курсу групи ІО-14
напряму підготовки: 123 – Комп'ютерна інженерія Лупащенко Артема Андрійовича

Керівник: Русінов Володимир Володимирович

Національна оцінка: _____
Кількість балів: _____

Науковий керівник _____
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

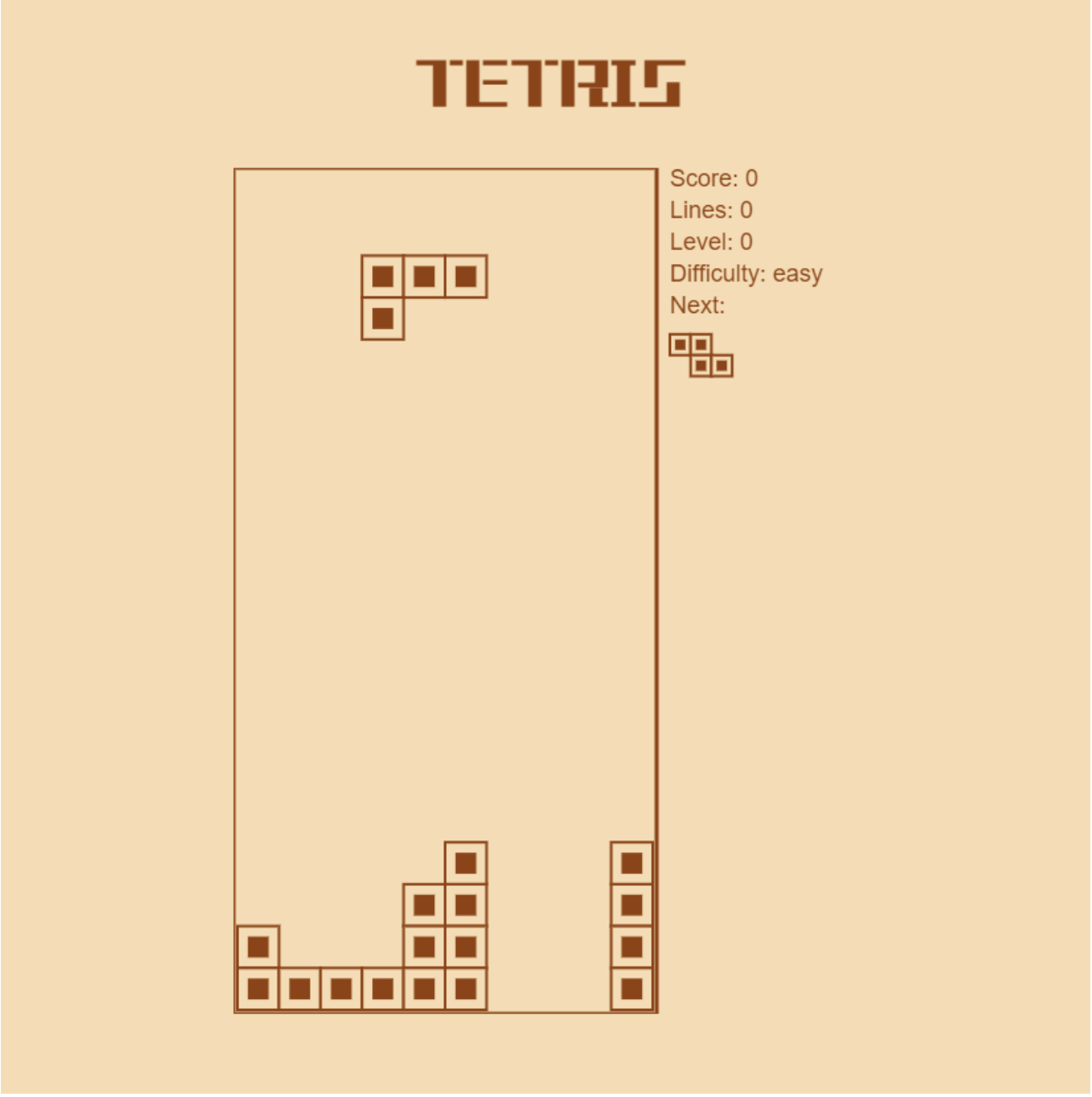
Члени комісії _____
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

_____ _____
(підпис) (вчене звання, науковий ступінь, прізвище та ініціали)

КИЇВ-2023

Вступ

У курсовій роботі описана логіка роботи, тестування та демонстрація роботи гри в тетріс зробленого на веб-сайті. У цій грі вашою основною метою буде вчасно розставляти падаючі фігури, видаляючи рядки і заробляючи рахунок. Програма, яка описана на сайті, дозволяє грати з комфортом. Вона виконана в стильному та легкому для освоєння інтерфейсі.



Скріншот роботи гри

Технічне завдання

Загальне завдання

Завданням курсової роботи є реалізація гри в *тетріс на веб-сайті*. Програма написана на мові *JavaScript* з використанням мов верстки HTML та CSS. Розробка відбувалась у середовищі *Visual Studio Code*.

Можливості програми
Гра повинна мати такі можливості:

- Управління з клавіатури;
- Зручний та інтуїтивний геймплей;
- Вибір складності;
- Перезапуск гри та пауза;
- Прискорення гри з плином часу для ускладнення;
- Можливість спостерігати результат гри у вигляді очок

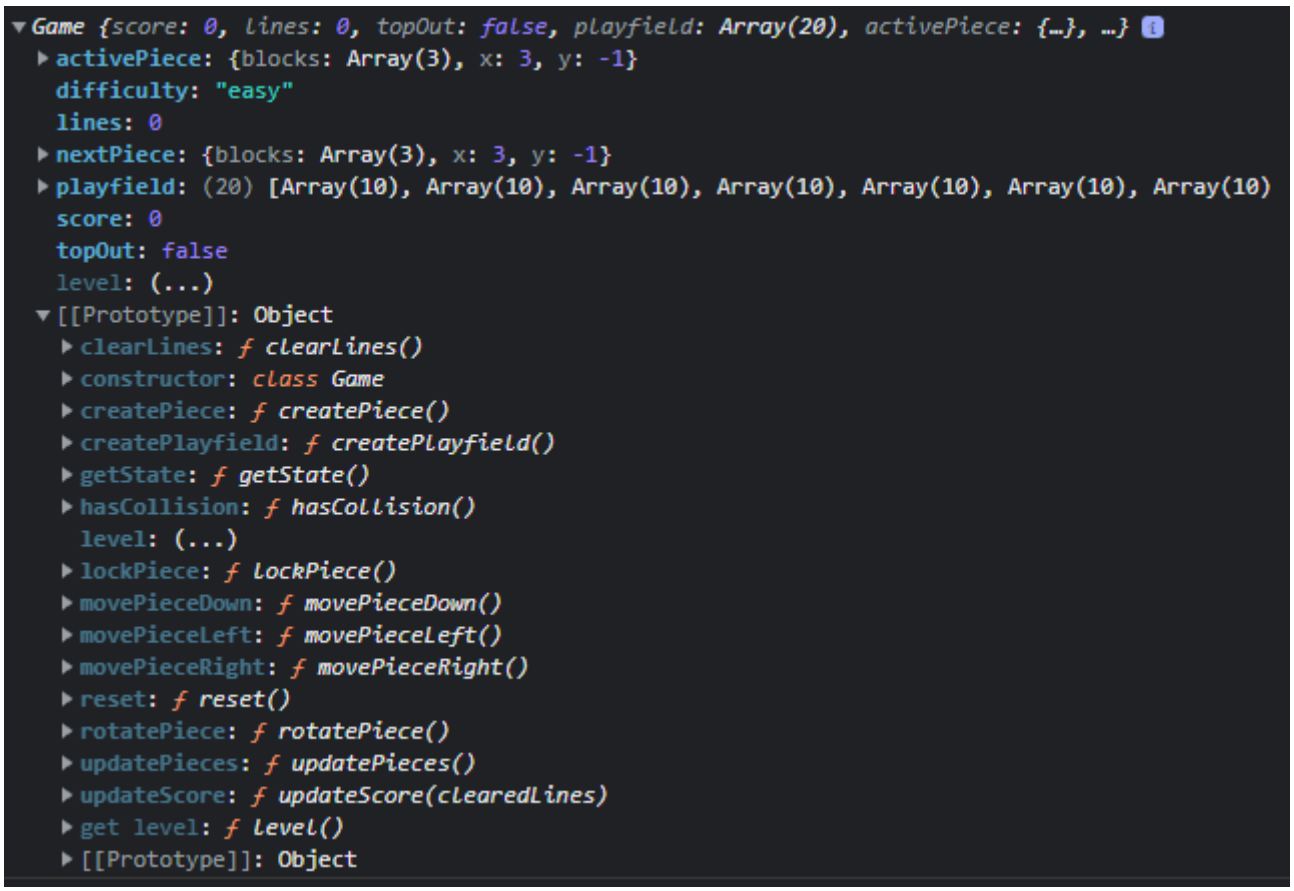
Логіка гри

Загальне завдання

Має бути реалізований клас *Game,* що міститиме усі необхідні методи для функціонування гри.

Детальний опис класу

- Статичне поле points містить бали, які гравець отримує за очищення рядків.
- Конструктор класу ініціалізує початкові значення для рівня складності (difficulty) та викликає метод reset().
- Геттер level обчислює рівень гравця на основі кількості ліній, які він видалив.
- Метод getState() повертає поточний стан гри, включаючи рахунок, рівень, складність, кількість видалених ліній, масив зі станом ігрового поля, наступний елемент (фігура) та прапор, що вказує на кінець гри.
- Метод reset() скидає гру до початкового стану, встановлюючи рахунок, кількість ліній, прапор topOut на false, очищає ігрове поле, створює початкову фігуру та наступну фігуру.
- Метод createPlayfield() створює двовимірний масив для представлення ігрового поля.
- Метод createPiece() випадковим чином вибирає тип фігури та повертає об'єкт фігури зі відповідними блоками, координатами та іншими властивостями.
- Методи movePieceLeft(), movePieceRight(), movePieceDown(), rotatePiece() виконують відповідні рухи фігурою вліво, вправо, вниз та обертання.
- Метод hasCollision() перевіряє, чи є зіткнення фігури з межами ігрового поля або з вже розміщеними блоками.
- Метод lockPiece() розміщує блоки фігури на ігровому полі при зіткненні.
- Метод clearLines() перевіряє, чи є повні рядки на ігровому полі, видаляє їх та зсуває решту рядків вниз.
- Метод updateScore() оновлює рахунок та кількість видалених ліній на основі кількості очищених рядків.
- Метод updatePieces() оновлює активну фігуру та наступну фігуру.



Перелік властивостей та методів класу Game

Графічний інтерфейс

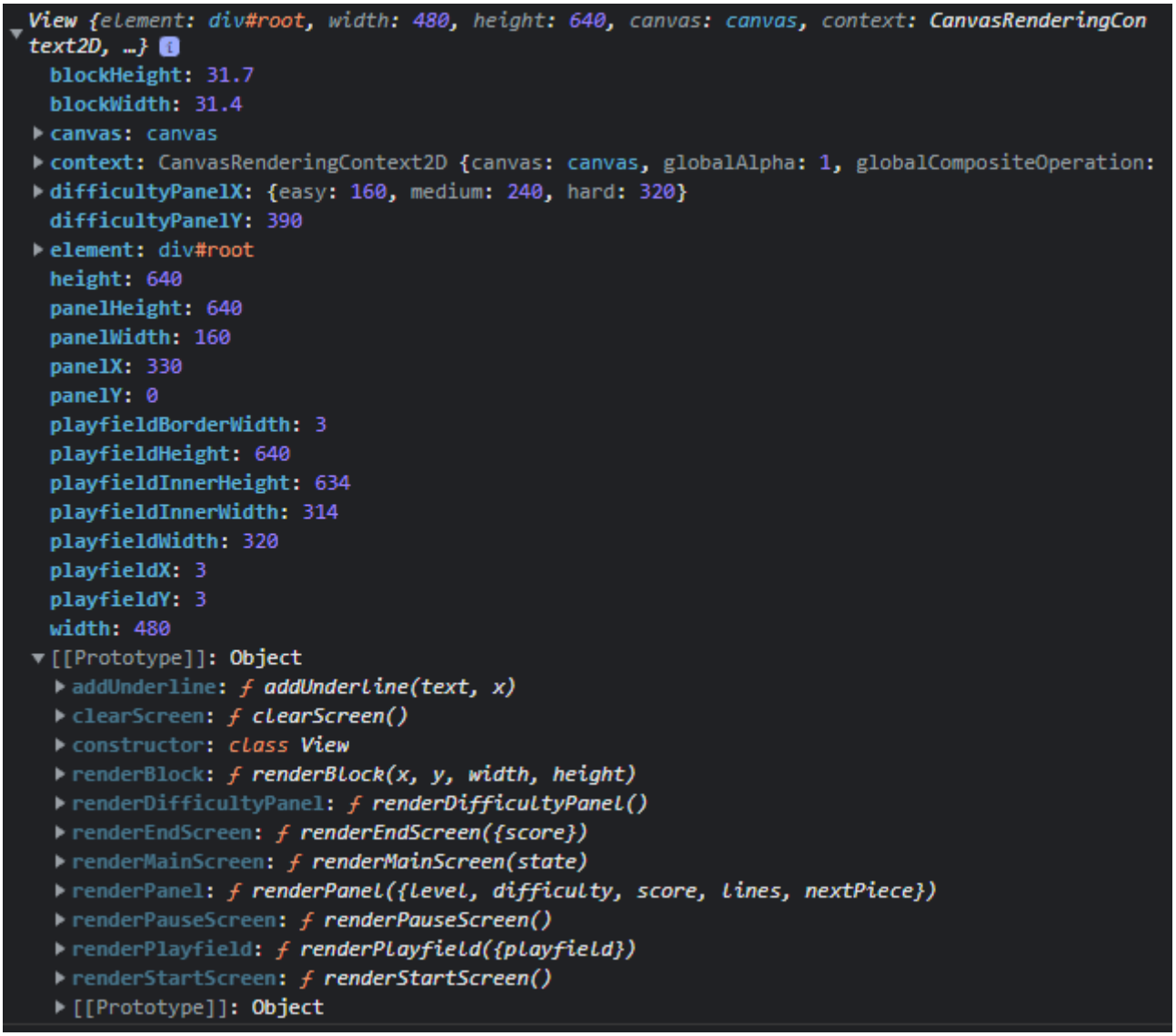
Загальне завдання

Має бути реалізований клас *View*, що міститиме усі необхідні методи для відображення екранів початку, гри, паузи, кінця.

Детальний опис класу

- Конструктор класу приймає елемент DOM (element), ширину та висоту канвасу, а також кількість рядків та стовпців ігрового поля.
- Метод renderMainScreen(state) відображає головний екран гри, який складається з ігрового поля та панелі з інформацією. Він очищує екран, викликає метод renderPlayfield() для відображення ігрового поля та метод renderPanel() для відображення панелі з інформацією гравця.
- Метод renderStartScreen() відображає стартовий екран гри, де гравець може побачити повідомлення про початок гри та інструкції щодо зміни складності. Він очищує екран, встановлює необхідні стилі тексту та викликає метод renderDifficultyPanel() для відображення панелі з варіантами складності.
- Метод renderPauseScreen() відображає екран паузи гри, де гравець може побачити повідомлення про паузу та інструкції щодо продовження гри або повернення назад. Він встановлює необхідні стилі та відображає текстові повідомлення.
- Метод renderEndScreen({score}) відображає екран завершення гри, де гравець може побачити свій рахунок та повідомлення про кінець гри та інструкції щодо перезапуску гри. Він очищує екран, встановлює необхідні стилі та відображає повідомлення про кінець гри та рахунок гравця.
- Метод clearScreen() очищує канвас.
- Метод renderPlayfield({playfield}) відображає ігрове поле гри. Він проходиться по кожному елементу масиву playfield та викликає метод renderBlock() для відображення блоків, які займають клітини на полі. Також він малює межу ігрового поля.
- Метод renderPanel({level, difficulty, score, lines, nextPiece}) відображає панель з інформацією про рівень, складність, рахунок, кількість ліній та наступну фігуру. Він використовує метод renderBlock() для відображення блоків на панелі.
- Метод renderBlock(x, y, width, height) відображає блок на канвасі. Він малює прямокутник та внутрішню частину блоку з встановленими стилями.
- Метод renderDifficultyPanel() відображає панель з варіантами складності. Він виводить текст з варіантами "EASY", "MEDIUM" та "HARD".
- Метод addUnderline(text, x) додає підкреслення під заданим текстом на панелі складності. Він очищує частину канвасу, на якій розташована

панель складності, та малює лінію під текстом.



Перелік властивостей та методів класу View

Управління

Загальне завдання

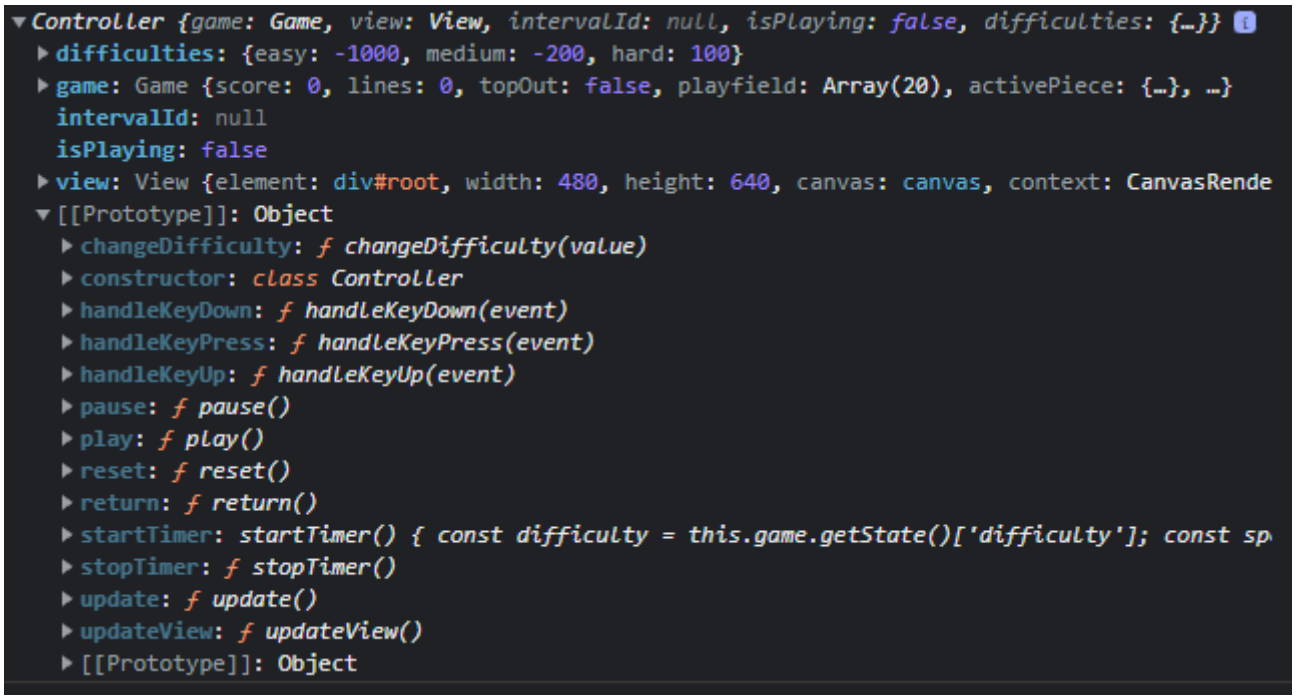
Має бути реалізований клас *Controller*, що міститиме усі необхідні методи для реагування на дії гравця.

Детальний опис класу

- Конструктор (constructor) приймає екземпляри гри (game) та відображення (view). Він ініціалізує внутрішні змінні, встановлює обробники подій клавіатури та викликає методи відображення стартового екрану і встановлення складності гри.
- Метод update() виконує оновлення стану гри та оновлення відображення. Він викликає метод movePieceDown() гри для переміщення фігури вниз і потім оновлює відображення викликом методу updateView().
- Метод play() встановлює флаг isPlaying в значення true, запускає таймер і оновлює відображення.
- Метод pause() встановлює флаг isPlaying в значення false, зупиняє таймер і оновлює відображення.
- Метод reset() скидає гру до початкового стану, запускає гру і оновлює відображення.
- Метод return() повертається до стартового екрану гри, зберігаючи поточний рівень складності, скидає гру до початкового стану і оновлює відображення.
- Метод changeDifficulty(value) змінює складність гри на задане значення. Він встановлює внутрішню змінну difficulty гри на задане значення та викликає метод addUnderline() відображення для підкреслення вибраної складності на панелі.
- Метод updateView() оновлює відображення гри залежно від поточного стану гри. Він перевіряє, чи гра закінчена (isGameOver), і відображає відповідний екран (renderEndScreen()), або чи пауза (!isPlaying), і відображає екран паузи (renderPauseScreen()), або відображає головний екран гри (renderMainScreen()).
- Метод startTimer() встановлює таймер для автоматичного оновлення гри. Він визначає швидкість оновлення (speed) залежно від поточного рівня складності та внутрішньо заданих значень. Якщо інтервал таймера ще не встановлено (intervalId дорівнює null), то він створює інтервал і викликає метод update() для оновлення гри з встановленою швидкістю. Якщо швидкість виявляється меншою або рівною 0, то встановлюється швидкість 100 мс.
- Метод stopTimer() зупиняє таймер. Він перевіряє, чи інтервал таймера встановлено (intervalId не дорівнює null), і якщо так, то він очищає інтервал і змінну intervalId встановлює в null.
- Метод handleKeyDown(event) обробляє подію натискання клавіші. Він перевіряє натиснуту клавішу (event.key) і виконує відповідну дію залежно від поточного стану гри (isPlaying). Для кожної клавіші виконується відповідна дія, така як скидання гри, пауза, рух фігури вліво, обертання

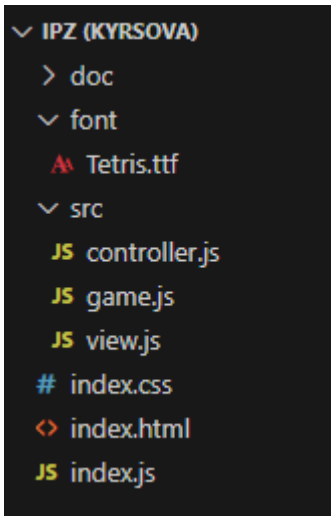
фігури, рух фігури вправо, або прискорення фігури вниз.

- Метод handleKeyUp(event) обробляє подію відпускання клавіші. Він перевіряє відпущену клавішу (event.key) і якщо це клавіша зі стрілкою вниз, то він запускає таймер для встановлення швидкості падіння фігури.
- Метод handleKeyPress(event) обробляє подію натискання клавіші. Він перевіряє натиснуту клавішу (event.key) і якщо це клавіша 'e', 'h' або 'm', то він змінює складність гри на 'easy', 'hard' або 'medium' відповідно.



Перелік властивостей та методів класу Controller

Структура коду



Програмний код

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Tetris</title>
    <link rel="stylesheet" href="index.css">
</head>
<body>
    <div id="root">
        <h1>
            <span style="color: saddlebrown">TETRIS</span>
        </h1>
    </div>

    <script src="index.js" type="module"></script>
</body>
</html>
```

index.css

```
@font-face {
    font-family:"tetris";
    src:url("./font/Tetris.ttf");
    font-weight:normal;
    font-style:normal;
}


body {
    margin: 0;
    background-color: wheat;
    font-family: "tetris";
```

```
}

#root {
    padding: 3rem;
    text-align: center;
}

#root h1 {
    font-size: 50px;
    color: wheat;
}

#root canvas {
    display: block;
    margin: 0 auto;
}
```

index.js

```
import Game from "./src/game.js";
import View from "./src/view.js";
import Controller from "./src/controller.js";

const root = document.querySelector('#root')

const game = new Game();
const view = new View(root, 480, 640, 20, 10);
const controller = new Controller(game, view);


window.game = game;
window.view = view;
window.controller = controller;
```

game.js

```
export default class Game {
    static points = {
        '1': 40,
        '2': 100,
        '3': 300,
        '4': 1200
    };

    constructor() {
        this.difficulty;
        this.reset();
    }

    get level() {
        return Math.floor(this.lines * 0.1);
    }

    getState() {
        const playfield = this.createPlayfield();
```

```
    const {y: pieceY, x: pieceX, blocks} = this.activePiece;

    for (let y = 0; y < this.playfield.length; y++) {
        playfield[y] = [];

        for (let x = 0; x < this.playfield[y].length; x++) {
            playfield[y][x] = this.playfield[y][x];
        }
    }

    for (let y = 0; y < blocks.length; y++) {
        for (let x = 0; x < blocks[y].length; x++) {
            if (blocks[y][x]) {
                playfield[pieceY + y][pieceX + x] = blocks[y][x];
            }
        }

    }

    return {
        score: this.score,
        level: this.level,
        difficulty: this.difficulty,
        lines: this.lines,
        nextPiece: this.nextPiece,
        playfield,
        isGameOver: this.topOut
    };
}

reset() {
    this.score = 0;
    this.lines = 0;
    this.topOut = false;
    this.playfield = this.createPlayfield();
    this.activePiece = this.createPiece();
    this.nextPiece = this.createPiece();
}

createPlayfield() {
    const playfield = [];

    for (let y = 0; y < 20; y++) {
        playfield[y] = [];

        for (let x = 0; x < 10; x++) {
            playfield[y][x] = 0;
        }
    }

    return playfield;
}
```

```
createPiece() {
    const index = Math.floor(Math.random() * 7);
    const type = 'IJLOSTZ'[index];
    const piece = {};

    switch (type) {
        case 'I':
            piece.blocks = [
                [0,0,0,0],
                [1,1,1,1],
                [0,0,0,0],
                [0,0,0,0]
            ];
            break;
        case 'J':
            piece.blocks = [
                [0,0,0],
                [1,1,1],
                [0,0,1]
            ];
            break;
        case 'L':
            piece.blocks = [
                [0,0,0],
                [1,1,1],
                [1,0,0]
            ];
            break;
        case 'O':
            piece.blocks = [
                [0,0,0,0],
                [0,1,1,0],
                [0,1,1,0],
                [0,0,0,0]
            ];
            break;
        case 'S':
            piece.blocks = [
                [0,0,0],
                [0,1,1],
                [1,1,0]
            ];
            break;
        case 'T':
            piece.blocks = [
                [0,0,0],
                [1,1,1],
                [0,1,0]
            ];
            break;
        case 'Z':
            piece.blocks = [
                [0,0,0],
```

```
                [1,1,0],
                [0,1,1]
            ];
            break;
        default:
            throw new Error("Unknown figure type")
    }

    piece.x = Math.floor((10 - piece.blocks[0].length) / 2);
    piece.y = -1

    return piece;
}

movePieceLeft() {
    this.activePiece.x -= 1;

    if (this.hasCollision()) {
        this.activePiece.x += 1;
    }
}

movePieceRight() {
    this.activePiece.x += 1;

    if (this.hasCollision()) {
        this.activePiece.x -= 1;
    }
}

movePieceDown() {
    if (this.topOut) return;

    this.activePiece.y += 1;

    if (this.hasCollision()) {
        this.activePiece.y -= 1;
        this.lockPiece();
        const clearedLines = this.clearLines();
        this.updateScore(clearedLines);
        this.updatePieces();
    }

    if (this.hasCollision()) {
        this.topOut = true;
    }
}

rotatePiece() {
    const blocks = this.activePiece.blocks;
    const length = blocks.length;

    const temp = [];
```

```
        for (let i = 0; i < length; i++) {
            temp[i] = new Array(length).fill(0);
        }

        for (let y = 0; y < length; y++) {
            for (let x = 0; x < length; x++) {
                temp[x][y] = blocks[length - 1 - y][x]
            }
        }

        this.activePiece.blocks = temp;

        if (this.hasCollision()) {
            this.activePiece.blocks = blocks;
        }
    }

    hasCollision() {
        const {y: pieceY, x: pieceX, blocks} = this.activePiece;

        for (let y = 0; y < blocks.length; y++) {
            for (let x = 0; x < blocks[y].length; x++) {
                if (
                    blocks[y][x] &&
                    ((this.playfield[pieceY + y] === undefined ||
this.playfield[pieceY + y][pieceX + x] === undefined) ||
                    this.playfield[pieceY + y][pieceX + x])
                    ) {
                    return true;
                }
            }
        }

        return false;
    }

    lockPiece() {
        const {y: pieceY, x: pieceX, blocks} = this.activePiece;

        for (let y = 0; y < blocks.length; y++) {
            for (let x = 0; x < blocks[y].length; x++) {
                if (blocks[y][x]){
                    this.playfield[pieceY + y][pieceX + x] = blocks[y][x];
                }
            }
        }
    }

    clearLines() {
        const rows = 20;
        const columns = 10;
        let lines = [];
```

```
        for (let y = rows - 1; y >= 0; y--) {
            let numberOfBlocks = 0;

            for (let x = 0; x < columns; x++) {
                if (this.playfield[y][x]) {
                    numberOfBlocks += 1;
                }
            }

            if (numberOfBlocks === 0) {
                break;
            } else if (numberOfBlocks < columns) {
                continue;
            } else if (numberOfBlocks === columns) {
                lines.unshift(y);
            }
        }

        for (let index of lines) {
            this.playfield.splice(index, 1);
            this.playfield.unshift(new Array(columns).fill(0));
        }

        return lines.length;
    }

    updateScore(clearedLines) {
        if (clearedLines > 0) {
            this.score += Game.points[clearedLines] * (this.level + 1);
            this.lines += clearedLines;
        }
    }

    updatePieces() {
        this.activePiece = this.nextPiece;
        this.nextPiece = this.createPiece();
    }
}
```

view.js

```
export default class View {
    constructor(element, width, height, rows, columns){
        this.element = element;
        this.width = width;
        this.height = height;

        this.canvas = document.createElement('canvas');
        this.canvas.width = this.width;
        this.canvas.height = this.height;
        this.context = this.canvas.getContext('2d');

        this.playfieldBorderWidth = 3;
        this.playfieldX = this.playfieldBorderWidth;
```

```
        this.playfieldY = this.playfieldBorderWidth;
        this.playfieldWidth = this.width * 2 / 3;
        this.playfieldHeight = this.height;
        this.playfieldInnerWidth = this.playfieldWidth - this.playfieldBorderWidth *
2;
        this.playfieldInnerHeight = this.playfieldHeight - this.playfieldBorderWidth
* 2;

        this.blockWidth = this.playfieldInnerWidth / columns;
        this.blockHeight = this.playfieldInnerHeight / rows;

        this.panelX = this.playfieldWidth + 10;
        this.panelY = 0;
        this.panelWidth = this.width / 3;
        this.panelHeight = this.height;

        this.difficultyPanelX = {
            'easy': this.width / 2 - 80,
            'medium': this.width / 2,
            'hard': this.width / 2 + 80
        }
        this.difficultyPanelY = this.height / 2 + 70

        this.element.appendChild(this.canvas);
    }

    renderMainScreen(state) {
        this.clearScreen();
        this.renderPlayfield(state);
        this.renderPanel(state);
    }

    renderStartScreen() {
        this.clearScreen();
        this.context.fillStyle = 'saddlebrown';
        this.context.font = '36px "Arial"';
        this.context.textAlign = 'center';
        this.context.textBaseline = 'middle';
        this.context.fillText('Press ENTER to Start', this.width / 2, this.height /
2);
        this.context.font = '12px "Arial"';
        this.context.fillText('Press (E, M, H) to Change Difficulty', this.width / 2,
this.height / 2 + 30);

        this.renderDifficultyPanel();
    }

    renderPauseScreen() {
        this.context.fillStyle = 'rgba(245,222,179,0.75)';
        this.context.fillRect(0, 0, this.width, this.height);

        this.context.fillStyle = 'saddlebrown';
        this.context.font = '36px "Arial"';
```

```
        this.context.textAlign = 'center';
        this.context.textBaseline = 'middle';
        this.context.fillText('Press ENTER to Resume', this.width / 2, this.height /
2);
        this.context.fillText('Press ESCAPE to Return', this.width / 2, this.height /
2 + 40);
    }

    renderEndScreen({score}) {
        this.clearScreen();

        this.context.fillStyle = 'saddlebrown';
        this.context.font = '36px "Arial"';
        this.context.textAlign = 'center';
        this.context.textBaseline = 'middle';
        this.context.fillText('GAME OVER', this.width / 2, this.height / 2 - 48);
        this.context.fillText(`Score: ${score}`, this.width / 2, this.height / 2);
        this.context.fillText('Press ENTER to Restart', this.width / 2, this.height /
2 + 48);
    }

    clearScreen() {
        this.context.clearRect(0, 0, this.width, this.height);
    }

    renderPlayfield({playfield}) {
        for (let y = 0; y < playfield.length; y++) {
            const line = playfield[y];

            for (let x = 0; x < line.length; x++) {
                const block = line[x];

                if (block) {
                    this.renderBlock(
                        this.playfieldX + (x * this.blockWidth),
                        this.playfieldY + (y * this.blockHeight),
                        this.blockWidth,
                        this.blockHeight
                    );
                }
            }
        }
        this.context.strokeStyle = "saddlebrown";
        this.context.lineWidth = this.playfieldBorderWidth;
        this.context.strokeRect(0, 0, this.playfieldWidth, this.playfieldHeight);

    }

    renderPanel({level, difficulty, score, lines, nextPiece}) {
        this.context.textAlign = 'start';
        this.context.textBaseline = 'top';
        this.context.fillStyle = 'saddlebrown';
        this.context.font = '18px "Arial"';
```

```
        this.context.fillText(`Score: ${score}`, this.panelX, this.panelY + 0);
        this.context.fillText(`Lines: ${lines}`, this.panelX, this.panelY + 24);
        this.context.fillText(`Level: ${level}`, this.panelX, this.panelY + 48);
        this.context.fillText(`Difficulty: ${difficulty}`, this.panelX, this.panelY +
72);
        this.context.fillText('Next:', this.panelX, this.panelY + 96);

        for (let y = 0; y < nextPiece.blocks.length; y++) {
            for (let x = 0; x < nextPiece.blocks[y].length; x++) {
                const block = nextPiece.blocks[y][x];

                if (block) {
                    this.renderBlock(
                        this.panelX + (x * this.blockWidth * 0.5),
                        this.panelY + 110 + (y * this.blockHeight * 0.5),
                        this.blockWidth * 0.5,
                        this.blockHeight * 0.5
                    );
                }
            }

        }
    }

    renderBlock(x, y, width, height) {
        const innerWidth = width / 2;
        const innerHeight = height / 2;;
        const innerX = x + (width - innerWidth) / 2;
        const innerY = y + (height - innerHeight) / 2;

        this.context.fillStyle = 'wheat';
        this.context.strokeStyle = 'saddlebrown';
        this.context.lineWidth = 2;

        this.context.fillRect(x, y, width, height);
        this.context.strokeRect(x, y, width, height);

        this.context.fillStyle = 'saddlebrown';
        this.context.fillRect(innerX, innerY, innerWidth, innerHeight);
    }

    renderDifficultyPanel() {
        this.context.font = '16px "Arial"';

        this.context.fillText('EASY', this.difficultyPanelX['easy'],
this.difficultyPanelY);
        this.context.fillText('MEDIUM', this.difficultyPanelX['medium'],
this.difficultyPanelY);
        this.context.fillText('HARD', this.difficultyPanelX['hard'],
this.difficultyPanelY);
    }
```

```
    addUnderline(text, x) {
        text = text.toUpperCase();
        const underlineOffset = 10;
        const textWidth = this.context.measureText(text).width;
        const textBoundingBoxDescent =
this.context.measureText(text).fontBoundingBoxDescent;
        const startX = x - (textWidth / 2);
        const endX = startX + textWidth;

        this.context.clearRect(0, this.difficultyPanelY + textBoundingBoxDescent,
this.width, this.height);
        this.context.beginPath();
        this.context.lineWidth = 1;
        this.context.strokeStyle = 'saddlebrown';
        this.context.moveTo(startX, this.difficultyPanelY + underlineOffset);
        this.context.lineTo(endX, this.difficultyPanelY + underlineOffset);
        this.context.stroke();
    }
}
```

controller.js

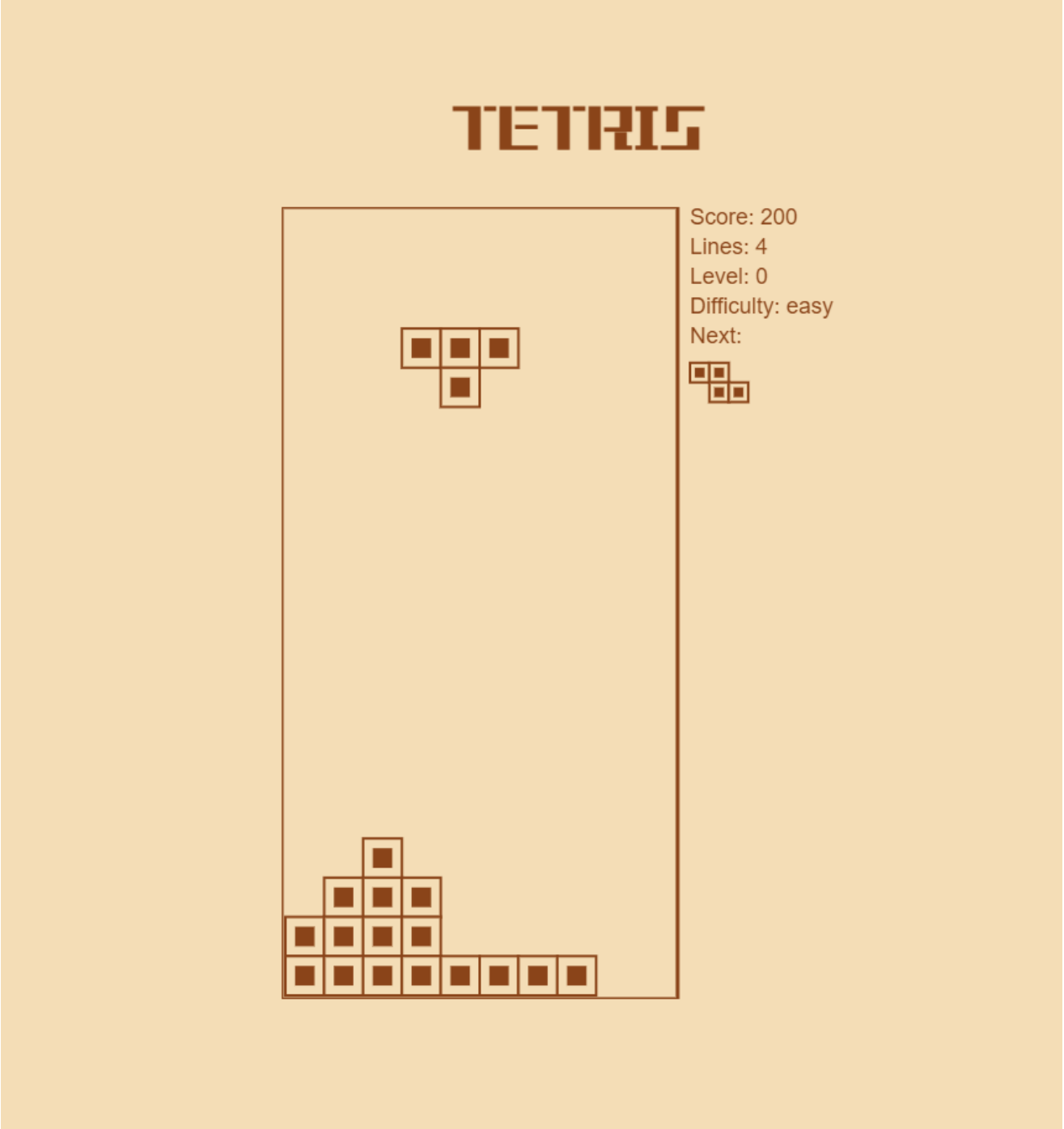
```
export default class Controller {
    constructor(game, view) {
        this.game = game;
        this.view = view;
        this.intervalId = null;
        this.isPlaying = false;
        this.difficulties = {
            'easy': -1000,
            'medium': -200,
            'hard': 100
        };

        document.addEventListener('keydown', this.handleKeyDown.bind(this));
        document.addEventListener('keyup', this.handleKeyUp.bind(this));
        document.addEventListener('keypress', this.handleKeyPress.bind(this));

        this.view.renderStartScreen();
        this.changeDifficulty('easy');
    }

    update() {
        this.game.movePieceDown();
        this.updateView();
    }

    play() {
        this.isPlaying = true;
        this.startTimer();
        this.updateView();
    }
```

```
    pause() {
        this.isPlaying = false;
        this.stopTimer();
        this.updateView();
    }

    reset() {
        this.game.reset();
        this.play();
    }

    return() {
        const difficulty = this.game.getState()['difficulty'];

        this.game.reset();
        this.view.renderStartScreen();
        this.changeDifficulty(difficulty);
    }

    changeDifficulty(value) {
        this.game.difficulty = value;
        this.view.addUnderline(value, this.view.difficultyPanelX[value]);
    }

    updateView() {
        const state = this.game.getState();

        if (state.isGameOver) {
            this.view.renderEndScreen(state)
        } else if (!this.isPlaying) {
            this.view.renderPauseScreen();
        } else {
            this.view.renderMainScreen(state);
        }

    }

    startTimer() {
        const difficulty = this.game.getState()['difficulty'];
        const speed = 1000 - this.difficulties[difficulty] -
this.game.getState().level * 100;

        if (!this.intervalId) {
            this.intervalId = setInterval(() => {
                this.update();
            }, speed > 0 ? speed : 100);
        }
    }

    stopTimer() {
        if (this.intervalId) {
            clearInterval(this.intervalId)
```

```
        this.intervalId = null;
    }
}

handleKeyDown(event) {
    const state = this.game.getState();

    switch (event.key) {
        case 'Enter':
            if (state.isGameOver) {
                this.reset();
            } else if (this.isPlaying) {
                this.pause();
            } else {
                this.play();
            }
            break;
        case 'Escape':
            if (!this.isPlaying) {
                this.return();
            }
            break;
        case 'ArrowLeft':
            if (this.isPlaying) {
                this.game.movePieceLeft();
                this.updateView();
                break;
            }
        case 'ArrowUp':
            if (this.isPlaying) {
                this.game.rotatePiece();
                this.updateView();
                break;
            }
        case 'ArrowRight':
            if (this.isPlaying) {
                this.game.movePieceRight();
                this.updateView();
                break;
            }
        case 'ArrowDown':
            if (this.isPlaying) {
                this.stopTimer();
                this.game.movePieceDown();
                this.updateView();
                break;
            }
    }
}

handleKeyUp(event) {
    switch (event.key) {
        case 'ArrowDown':
```

```
                if (this.isPlaying) {
                    this.startTimer();
                    break;
                }
            }
        }

        handleKeyPress(event) {
            switch (event.key) {
                case 'e':
                    if (!this.isPlaying) {
                        this.changeDifficulty('easy')
                        break;
                    }
                case 'h':
                    if (!this.isPlaying) {
                        this.changeDifficulty('hard')
                        break;
                    }
                case 'm':
                    if (!this.isPlaying) {
                        this.changeDifficulty('medium')
                        break;
                    }
            }
        }
    }
```

Стартове вікно програми



Ігровий процес

Вікно паузи гри

Вікно завершення гри

Висновки

В ході роботи було успішно реалізовано гру - "Тетріс". Результатом курсової роботи стала повноцінна програма з графічним інтерфейсом, яка забезпечує захоплюючий ігровий процес. В процесі розробки були здійснені пошук рішень, що дозволили успішно відтворити оригінальний геймплей "Тетрісу", а також закріпити теоретичні знання та практичні навички з проектування, моделювання, розробки та тестування програмного забезпечення з графічним інтерфейсом.

Список використаних джерел

Для вирішення проблем використовувалися такі джерела:

1. MSDN Library
2. Форум Stackoverflow
3. https://harddrop.com/wiki/Category:Rotation_Systems