Modern Tetris Implementation: Object-Oriented Design & Documentation - Final Version

By CS23B1023 Athidathan K

December 7, 2024

Contents

1 Overview **3**
1.1 Changes from Mid-Semester Version . . . 3

2 Project Structure **4**

3 Classes and Files **4**
3.1 `Position` Class (`src/position.h`, `src/position.cpp`) . . . 4
3.2 `Block` Class Hierarchy (`src/block.h`, `src/block.cpp`, `src/blocks.cpp`) . . . 5
3.2.1 Base `Block` Class . . . 5
3.2.2 Derived Block Classes (`src/blocks.cpp`) . . . 5
3.3 `Grid` Class (`src/grid.h`, `src/grid.cpp`) . . . 6
3.4 `Game` Class (`src/game.h`, `src/game.cpp`) . . . 6
3.5 `GameMode` Abstract Class (`src/game_mode.h`) . . . 7
3.6 `SinglePlayer` Class (`src/single_player.h`, `src/single_player.cpp`) . . . 7
3.7 `TwoPlayer` Class (`src/two_player.h`, `src/two_player.cpp`) . . . 8

4 OOP Concepts Incorporated **9**
4.1 Encapsulation . . . 9
4.2 Abstraction . . . 9
4.3 Inheritance . . . 10
4.4 Polymorphism . . . 10
4.4.1 Compile-Time Polymorphism . . . 11
4.4.2 Run-Time Polymorphism . . . 11
4.5 Exception Handling . . . 13
4.5.1 Main Function . . . 13
4.5.2 Raylib's Automatic Exception Handling . . . 14

5 Example Code Snippets **14**
5.1 Drawing a Block . . . 14
5.2 Handling Block Rotation . . . 15
5.3 Moving Blocks Down Automatically . . . 15
5.4 Handling Input for Two Players . . . 16

5.5 Drawing the Game Using `Game::Draw(int offsetX, int offsetY)` 17
5.6 Drawing the Game Over Screen 18
5.7 Clearing Full Rows in the Grid 18
5.8 Updating the Score 19

6 Screenshots **19**
6.1 Title Screen 20
6.2 Single-Player Mode 21
6.3 Two-Player Mode 22
6.4 Game Over Screen 22

7 Building the Project **23**
7.1 Requirements 23
7.2 Instructions 23

8 Conclusion **23**

1 Overview

This documentation provides an overview of the Tetris project, explaining its structure, specifications, and how various Object-Oriented Programming (OOP) concepts are incorporated, including encapsulation, abstraction, inheritance, polymorphism, operator overloading, and plans for future use of templates and virtual functions.

The Tetris project is implemented in C++ using the Raylib graphics library. The game simulates the classic Tetris gameplay, where different shaped blocks fall into a grid, and the player manipulates them to form complete rows.

1.1 Changes from Mid-Semester Version

- **Polymorphism**:
  - Introduced an abstract class `GameMode` to facilitate common interface for different game modes.
  - Created derived classes `SinglePlayer` and `TwoPlayer`.
- **Main Function**:
  - Added a title screen for game mode selection.
  - Reworked main logic to support different game modes using polymorphism.
  - Implemented exception handling.
  - Added background music.
  - Introduced game over logic and a game over screen.
- **Game Class**:
  - Moved UI drawing logic to the `Game` class's `Draw` function.
  - Added support for offset parameters in the `Draw` function.
  - Added Rotate and Clear Sounds.
- **Grid Class**:
  - Added support for offset parameters in the `Draw` function for multiple game grids.

These changes significantly enhance the flexibility, maintainability, and functionality of the Tetris project, providing a robust foundation for future development and additional game modes.

2 Project Structure

```
.vscode/
build/
Font/
    Handjet.ttf
lib/
    raylib.h
Screenshots/
Sounds/
    clear.mp3
    music.mp3
    rotate.mp3
src/
    block.cpp
    block.h
    blocks.h
    colors.cpp
    colors.h
    game_mode.h
    game.cpp
    game.h
    grid.cpp
    grid.h
    main.cpp
    position.cpp
    position.h
    single_player.cpp
    single_player.h
    two_player.cpp
    two_player.h
Makefile
Documentation.tex
main.code-workspace
```

3 Classes and Files

3.1 `Position` Class (`src/position.h`, `src/position.cpp`)

Represents the position of a block in the grid.

```
class Position {
public:
    int row;
    int column;
    Position(int row, int column);
    Position operator+(const Position& other) const;
};
```

3.2 Block Class Hierarchy (src/block.h, src/block.cpp, src/blocks.cpp)

3.2.1 Base Block Class

Encapsulation: The Block class encapsulates the properties and behaviors of a Tetris block.

```
class Block {
public:
    Block();
    void Draw(int offsetX, int offsetY);
    void Move(int rows, int columns);
    void Rotate();
    void UndoRotation();
    std::vector<Position> GetCellPositions();
protected:
    int id;
    int rotationState;
    int rowOffset;
    int columnOffset;
    std::map<int, std::vector<Position>> cells;
    int cellSize;
    std::vector<Color> colors;
};
```

Example: The Block class manages its state internally and provides public methods for interaction, demonstrating encapsulation.

3.2.2 Derived Block Classes (src/blocks.cpp)

Inheritance: Multiple classes (LBlock, JBlock, IBlock, OBlock, SBlock, TBlock, ZBlock) inherit from the Block class.

Example:

```
// src/blocks.cpp

#include "block.h"

// LBlock class inheriting from Block
class LBlock : public Block {
public:
    LBlock() {
        id = 1;
        cells[0] = {Position(0, 2), Position(1, 0), Position(1, 1), Position(1, 2)};
        cells[1] = {Position(0, 1), Position(1, 1), Position(2, 1), Position(2, 2)};
        cells[2] = {Position(1, 0), Position(1, 1), Position(1, 2), Position(2, 0)};
        cells[3] = {Position(0, 0), Position(0, 1), Position(1, 1), Position(2, 1)};
        Move(0, 3);
    }
};

// Other block classes (JBlock, IBlock, OBlock, SBlock, TBlock, ZBlock) defined
    similarly
```

Each derived class initializes its specific shape by setting up the cells member variable inherited from the Block class.

3.3 Grid Class (src/grid.h, src/grid.cpp)

Represents the game grid where the blocks are placed. Manages the grid state, drawing, and clearing full rows.

```cpp
// src/grid.h

#pragma once
#include <vector>
#include <raylib.h>

class Grid {
public:
    Grid();
    void Initialize();
    void Print();
    void Draw();
    bool IsCellOutside(int row, int column);
    bool IsCellEmpty(int row, int column);
    int ClearFullRows();
private:
    int grid[20][10];
    int numRows;
    int numCols;
    int cellSize;
    std::vector<Color> colors;
    bool IsRowFull(int row);
    void ClearRow(int row);
    void MoveRowDown(int row, int numRows);

friend class Game;
};
```

3.4 Game Class (src/game.h, src/game.cpp)

Controls the overall game logic.

Abstraction: Provides simplified methods to handle game mechanics like input handling and block movement.

```cpp
// src/game.h

#pragma once
#include "grid.h"
#include "block.h"
#include "raylib.h"

class Game {
public:
    Game(int playerNumber = 1);
    ~Game();
    void Draw(int offsetX = 0, int offsetY = 0);
    void HandleInput();
    void HandleInputPlayer2();
    void MoveBlockDown();
    void DrawGameOverScreen(Game& winner);
    void Reset();
    bool gameOver;
    int score;
    int playerNumber;
    Font font;
private:
    void MoveBlockLeft();
    void MoveBlockRight();
    Block GetRandomBlock();
    std::vector<Block> GetAllBlocks();
    bool IsBlockOutside();
    void RotateBlock();
    void LockBlock();
    bool BlockFits();
    void UpdateScore(int linesCleared, int moveDownPoints);
    Grid grid;
    std::vector<Block> blocks;
    Block currentBlock;
    Block nextBlock;
    Sound rotateSound;
    Sound clearSound;
};
```

3.5 `GameMode` Abstract Class (`src/game_mode.h`)

Defines a common interface for different game modes. **Polymorphism**: Uses virtual functions to allow derived classes to provide specific implementations for different game modes.

```cpp
// src/game_mode.h

#pragma once
#include "game.h"

class GameMode {
public:
    virtual void HandleInput() = 0;
    virtual void Update() = 0;
    virtual void Draw() = 0;
    virtual bool IsGameOver() const = 0;
    virtual ~GameMode() = default;
    virtual Game* GetWinner() = 0;
    virtual int getWinnerScore() = 0;
    virtual void Reset() = 0;
};
```

3.6 `SinglePlayer` Class (`src/single_player.h`, `src/single_player.cpp`)

Implements the single-player game mode.

Inheritance: Inherits from `GameMode` and provides specific implementations for single-player mode.

```cpp
// src/single_player.h

#pragma once
#include "game_mode.h"

class SinglePlayer : public GameMode {
public:
    SinglePlayer();
    void HandleInput() override;
    void Update() override;
    void Draw() override;
    bool IsGameOver() const override;
    ~SinglePlayer() override = default;
    Game* GetWinner() override;
    int getWinnerScore() override { return GetWinner()->score; }
    void Reset() override { game.Reset(); }
private:
    Game game;
};
```

3.7 TwoPlayer Class (src/two_player.h, src/two_player.cpp)

Implements the two-player game mode.

Inheritance: Inherits from `GameMode` and provides specific implementations for two-player mode.

```cpp
// src/two_player.h

#pragma once
#include "game_mode.h"

class TwoPlayer : public GameMode {
public:
    TwoPlayer();
    void HandleInput() override;
    void Update() override;
    void Draw() override;
    bool IsGameOver() const override;
    Game* GetWinner() override;
    int getWinnerScore() override { return GetWinner()->score; }
    void Reset() override {
        player1Game.Reset();
        player2Game.Reset();
    }
private:
    Game player1Game;
    Game player2Game;
};
```

```cpp
// src/two_player.cpp

#include "two_player.h"

TwoPlayer::TwoPlayer() : player1Game(1), player2Game(2) {
    // Initialize the two-player games
}

void TwoPlayer::HandleInput() {
    player1Game.HandleInput();
    player2Game.HandleInputPlayer2();
}

void TwoPlayer::Update() {
    player1Game.MoveBlockDown();
    player2Game.MoveBlockDown();
}

```

```cpp
void TwoPlayer::Draw() {
    player1Game.Draw(0, 0);
    player2Game.Draw(500, 0);
}

bool TwoPlayer::IsGameOver() const {
    return player1Game.gameOver || player2Game.gameOver;
}

Game* TwoPlayer::GetWinner() {
    if (player1Game.score > player2Game.score)
        return &player1Game;
    else
        return &player2Game;
}
```

4 OOP Concepts Incorporated

4.1 Encapsulation

Classes manage their own data and expose functionality through public methods. Access specifiers `public` and `private` are used to enforce access control, separating the interface from the implementation.

Example:

In the `Block` class (`src/block.h`), data members are declared as `protected` or `private`, while member functions that form the interface are declared as `public`.

```cpp
// src/block.h

class Block {
public:
    Block();
    void Draw(int offsetX, int offsetY);
    void Move(int rows, int columns);
    void Rotate();
    void UndoRotation();
    std::vector<Position> GetCellPositions();
protected:
    int id;
    int rotationState;
    int rowOffset;
    int columnOffset;
    std::map<int, std::vector<Position>> cells;
    int cellSize;
    std::vector<Color> colors;
};
```

The implementation of these member functions is separated in the corresponding `.cpp` file (`src/block.cpp`), keeping the interface clean and promoting encapsulation.

4.2 Abstraction

Abstraction is achieved by using header files to define class interfaces. The implementation details are hidden in the `.cpp` files, allowing users of the classes to interact with them without needing to understand the internal workings.

Example:

The `Game` class interface is defined in `src/game.h`, abstracting the game mechanics and providing a clear interface for interaction.

```cpp
// src/game.h

class Game {
public:
    Game();
    ~Game();
    void Draw();
    void HandleInput();
    void MoveBlockDown();
    bool gameOver;
    int score;
private:
    // Private member functions and variables
};
```

4.3 Inheritance

Inheritance is used to create specialized block types from a base `Block` class. This allows for code reuse and establishes a hierarchical class structure.

Class Hierarchy:

- `Block` (base class)
  - `IBlock`
  - `JBlock`
  - `LBlock`
  - `OBlock`
  - `SBlock`
  - `TBlock`
  - `ZBlock`

Example:

```cpp
// src/blocks.cpp

class TBlock : public Block {
public:
    TBlock() {
        id = 6;
        cells[0] = {Position(0, 1), Position(1, 0), Position(1, 1), Position(1, 2)};
        // other rotation states...
        Move(0, 3);
    }
};
```

4.4 Polymorphism

Polymorphism allows objects to be treated as instances of their base class rather than their actual derived class. This project demonstrates both compile-time and run-time polymorphism.

4.4.1 Compile-Time Polymorphism

Function Overloading (used):

Function overloading is utilized in the project to allow functions with the same name to have different parameter lists.

```cpp
// src/block.h
class Block {
public:
    void Move(int rows, int columns);

    // Overloaded Move function
    void Move(const Position& pos);
};
```

Operator Overloading (used):

Operator overloading is implemented in the `Position` class to simplify position arithmetic.

```cpp
// src/position.cpp

Position Position::operator+(const Position &other) const{
    return Position(row + other.row, column + other.column);
}
```

This allows positions to be added using the `+` operator.

Templates (not currently used):

Templates are not explicitly used in this project as the data types are consistent, and there's no immediate need for generic programming. However, templates could be implemented for generic data structures or algorithms if needed in future enhancements.

4.4.2 Run-Time Polymorphism

Virtual Functions (used):

Run-time polymorphism is achieved through the use of virtual functions in the ‘GameMode‘ abstract class. This allows derived classes to provide specific implementations for different game modes.

Example:

The `GameMode` Abstract class defines a common interface for different game modes using pure virtual functions. The `SinglePlayer` and `TwoPlayer` classes inherit from `GameMode` and provide specific implementations for these virtual functions.

```cpp
// src/game_mode.h

#pragma once
#include "game.h"

class GameMode {
public:
    virtual void HandleInput() = 0;
    virtual void Update() = 0;
    virtual void Draw() = 0;
    virtual bool IsGameOver() const = 0;
    virtual ~GameMode() = default;
    virtual Game* GetWinner() = 0;
    virtual int getWinnerScore() = 0;
    virtual void Reset() = 0;
};
```

Derived Classes:

```cpp
// src/single_player.h

#pragma once
#include "game_mode.h"

class SinglePlayer : public GameMode {
public:
    SinglePlayer();
    void HandleInput() override;
    void Update() override;
    void Draw() override;
    bool IsGameOver() const override;
    ~SinglePlayer() override = default;
    Game* GetWinner() override;
    int getWinnerScore() override { return GetWinner()->score; }
    void Reset() override { game.Reset(); }
private:
    Game game;
};
```

```cpp
// src/two_player.h

#pragma once
#include "game_mode.h"

class TwoPlayer : public GameMode {
public:
    TwoPlayer();
    void HandleInput() override;
    void Update() override;
    void Draw() override;
    bool IsGameOver() const override;
    Game* GetWinner() override;
    int getWinnerScore() override { return GetWinner()->score; }
    void Reset() override {
        player1Game.Reset();
        player2Game.Reset();
    }
private:
    Game player1Game;
    Game player2Game;
};
```

Usage in Main Function:

In the main function, a `GameMode` pointer is used to refer to either a `SinglePlayer` or `TwoPlayer` object, demonstrating run-time polymorphism. This pointer is then used to execute the rest of the gameplay loop logic.

```cpp
// src/main.cpp

int main() {
    try {
        // .. Game initialization

        GameMode* game = nullptr;
        if (mode == 1) {
            game = new SinglePlayer();
        } else if (mode == 2) {
            SetWindowSize(1000, 620); // Adjust window size for two-player mode
            game = new TwoPlayer();
        } else {
            throw std::runtime_error("Invalid game mode selected.");
        }
        // Game logic done using GameMode pointer
        while (!WindowShouldClose()) {
            if (game->IsGameOver()) {
                // Determine the winner
                Game* winner = game->GetWinner();
                winner->DrawGameOverScreen(*winner);
                // Restart mechanism
                if (IsKeyPressed(KEY_ENTER)) {
                    game->Reset();
                    SeekMusicStream(music, 0);
            }
            } else {
                UpdateMusicStream(music);
                game->HandleInput(); // Run-Time Polymorphism
                if (EventTriggered(0.2))
                    game->Update();

                BeginDrawing();
                ClearBackground(BLACK);
                game->Draw();
                EndDrawing();
            }
        }
        UnloadMusicStream(music);
        game->Reset();
        CloseAudioDevice();
        CloseWindow();
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

4.5 Exception Handling

Exception handling is an important aspect of robust software development. In this Tetris project, exception handling is incorporated to manage errors gracefully and ensure the program can recover from unexpected situations. Below are some key areas where exception handling is implemented:

4.5.1 Main Function

The main function includes a try-catch block to handle any exceptions thrown during initialization or game execution. This ensures that any errors are caught and reported, preventing the program from crashing unexpectedly.

```
// src/main.cpp
int main() {
    try {

        // game logic

        GameMode* game = nullptr;
        if (mode == 1) {
            game = new SinglePlayer();
        } else if (mode == 2) {
            SetWindowSize(1000, 620); // Adjust window size for two-player mode
            game = new TwoPlayer();
        } else {
            throw std::runtime_error("Invalid game mode selected.");}

        // game logic

    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

4.5.2 Raylib's Automatic Exception Handling

Raylib provides automatic exception handling for resource loading, such as fonts, sounds, and textures. When resources are not loaded properly, Raylib functions return default values or handle the errors internally, reducing the need for extensive manual exception handling in the code.

For example, when loading a font or sound, Raylib will handle any issues internally and provide a default value if the resource fails to load. This simplifies the code and ensures that the program can continue running even if some resources are not available.

Example:

```
// src/game.cpp
Game::Game(int playerNumber) {
    grid = Grid();
    blocks = GetAllBlocks();
    currentBlock = GetRandomBlock();
    nextBlock = GetRandomBlock();
    gameOver = false;
    score = 0;
    this->playerNumber = playerNumber;

    // Raylib handles errors internally and provides default values
    font = LoadFontEx("Font/Handjet.ttf", 64, 0, 0);
    rotateSound = LoadSound("Sounds/rotate.wav");
    clearSound = LoadSound("Sounds/clear.wav");
}
```

By leveraging Raylib's built-in error handling, the code remains clean and concise, while still ensuring robustness and stability.

5 Example Code Snippets

5.1 Drawing a Block

The `Block::Draw` function is responsible for rendering a Tetris block on the screen. It takes two parameters, `offsetX` and `offsetY`, which allow for positioning the block at different locations.

```cpp
// src/block.cpp

void Block::Draw(int offsetX, int offsetY) {
    std::vector<Position> tiles = GetCellPositions();
    for (const Position& item : tiles) {
        DrawRectangle(
            item.column * cellSize + offsetX,
            item.row * cellSize + offsetY,
            cellSize - 1, cellSize - 1,
            colors[id]
        );
    }
}
```

Explanation:

- The `GetCellPositions()` function retrieves the positions of the block's cells.
- The `DrawRectangle` function is used to draw each cell of the block at the specified offset.
- The `offsetX` and `offsetY` parameters allow the block to be drawn at different positions on the screen.

Demonstrates encapsulation by handling drawing internally.

5.2 Handling Block Rotation

The `Block::Rotate` function updates the rotation state of a Tetris block. This function ensures that the block cycles through its available rotation states.

```cpp
// src/block.cpp

void Block::Rotate() {
    rotationState = (rotationState + 1) % cells.size();
}
```

Explanation:

- The `rotationState` variable keeps track of the current rotation state of the block.
- The rotation state is updated by incrementing it and taking the modulus with the number of rotation states.
- This ensures that the rotation state cycles through the available states.

Shows how the block manages its rotation state.

5.3 Moving Blocks Down Automatically

The following code snippet demonstrates how the game logic moves the current block down at regular intervals, simulating gravity.

```cpp
// src/main.cpp

if (EventTriggered(0.2)) { // Game speed is 0.2 seconds
    game->MoveBlockDown();
}
```

Explanation:

- The `EventTriggered` function checks if the specified interval (0.2 seconds) has passed.
- If the interval has passed, the `MoveBlockDown` function is called to move the current block down.
- This ensures that the block moves down at regular intervals, simulating gravity.

Demonstrates how the game logic moves the current block down at regular intervals.

5.4 Handling Input for Two Players

The `TwoPlayer::HandleInput` function is responsible for handling input separately for two players in a two-player game mode. Player 1 uses the WASD keys for control, while Player 2 uses the arrow keys.

```cpp
// src/two_player.cpp

void TwoPlayer::HandleInput() {
    player1Game.HandleInput();
    player2Game.HandleInputPlayer2();
}
```

```cpp
// src/game.cpp

void Game::HandleInput() {
    // Player 1 controls
    if (IsKeyPressed(KEY_A)) MoveBlockLeft();
    if (IsKeyPressed(KEY_D)) MoveBlockRight();
    if (IsKeyPressed(KEY_W)) RotateBlock();
    if (IsKeyPressed(KEY_S))  {
        MoveBlockDown();
        MoveBlockDown();
        UpdateScore(0, 2);}
}

void Game::HandleInputPlayer2() {
    // Player 2 controls
    if (IsKeyPressed(KEY_LEFT)) MoveBlockLeft();
    if (IsKeyPressed(KEY_RIGHT)) MoveBlockRight();
    if (IsKeyPressed(KEY_UP)) RotateBlock();
    if (IsKeyPressed(KEY_DOWN)) {
        MoveBlockDown();
        MoveBlockDown();
        UpdateScore(0, 2);}
}
```

Explanation:

- The `HandleInput` function for `player1Game` handles input for Player 1 using the WASD keys.
- The `HandleInputPlayer2` function for `player2Game` handles input for Player 2 using the arrow keys.
- This ensures that input is handled separately for each player in a two-player game.

Demonstrates how input is handled separately for two players, with Player 1 using the WASD keys and Player 2 using the arrow keys.

5.5 Drawing the Game Using `Game::Draw(int offsetX, int offsetY)`

The `Game::Draw` function is responsible for rendering the game state on the screen. It takes two parameters, `offsetX` and `offsetY`, which allow for positioning the game grid and UI elements at different locations. This is particularly useful for multiplayer modes where multiple game grids need to be displayed side by side.

```cpp
// src/game.cpp

void Game::Draw(int offsetX, int offsetY) {
    grid.Draw(offsetX, offsetY);

    // Draw the current block
    currentBlock.Draw(offsetX + 11, offsetY + 11);

    // Draw the UI elements
    DrawTextEx(font, "Score", {(float)(365 + offsetX), (float)(15 + offsetY)}, 38, 2,
         WHITE);
    char playerText[10];
    sprintf(playerText, "Player %d", playerNumber);
    DrawTextEx(font, playerText, {(float)(365 + offsetX), (float)(500 + offsetY)},
        38, 2, WHITE);
    DrawTextEx(font, "Next", {(float)(370 + offsetX), (float)(175 + offsetY)}, 38, 2,
         WHITE);
    DrawRectangleRounded({(float)320 + offsetX, (float)55 + offsetY, 170, 60}, 0.3,
        6, darkGrey);
    char scoreText[10];
    sprintf(scoreText, "%d", score);
    Vector2 textSize = MeasureTextEx(font, scoreText, 38, 2);
    DrawTextEx(font, scoreText, {(float)offsetX + 320 + (170 - textSize.x) / 2, (
        float)offsetY + 65}, 38, 2, WHITE);
    DrawRectangleRounded({(float)320 + offsetX, (float)215 + offsetY, 170, 180}, 0.3,
         6, darkGrey);

    // Preview the next block
    int previewX, previewY;
    switch (nextBlock.id) {
        case 3:
            previewX = offsetX + 255;
            previewY = offsetY + 290;
            break;
        case 4:
            previewX = offsetX + 255;
            previewY = offsetY + 280;
            break;
        default:
            previewX = offsetX + 270;
            previewY = offsetY + 270;
            break;
    }
    nextBlock.Draw(previewX, previewY);
}
```

Explanation:

- The `grid.Draw(offsetX, offsetY)` call renders the game grid at the specified offset.
- The `currentBlock.Draw(offsetX + 11, offsetY + 11)` call draws the current block with a slight offset to align it within the grid.

- UI elements, such as the score, player number, and next block preview, are drawn using the `DrawTextEx` and `DrawRectangleRounded` functions, positioned based on the provided offsets.
- The next block preview is drawn at a specific position based on the block's ID, ensuring it is displayed correctly.

This function encapsulates the drawing logic for the game, making it easy to render the game state at different positions on the screen, which is essential for supporting multiplayer modes.

5.6 Drawing the Game Over Screen

The `Game::DrawGameOverScreen` function is responsible for rendering the game over screen, displaying the winner and their score.

```cpp
// src/game.cpp

void Game::DrawGameOverScreen(Game &winner) {
    BeginDrawing();
    ClearBackground(BLACK);
    DrawText("Game Over!", 100, 150, 40, RED);
    DrawText(TextFormat("Winner: Player %d", winner.playerNumber), 100, 220, 30,
        WHITE);
    DrawText(TextFormat("Score: %d", winner.score), 100, 260, 30, WHITE);
    DrawText("Press ENTER to restart", 100, 320, 20, LIGHTGRAY);
    EndDrawing();
}
```

Explanation:

- The `BeginDrawing` and `EndDrawing` functions are used to start and end the drawing process.
- The background is cleared with a black color using `ClearBackground(BLACK)`.
- The game over message, winner information, and restart instructions are drawn using the `DrawText` and `TextFormat` functions.

Shows how the game over screen is drawn, displaying the winner and their score.

5.7 Clearing Full Rows in the Grid

The `Grid::ClearFullRows` function checks for and clears full rows in the game grid, moving the rows above down as needed.

```cpp
// src/grid.cpp

int Grid::ClearFullRows() {
    int completed = 0;
    for (int row = numRows - 1; row >= 0; row--) {
        if (IsRowFull(row)) {
            ClearRow(row);
            completed++;
        } else if (completed > 0) {
            MoveRowDown(row, completed);
        }
    }
    return completed;
}
```

Explanation:

- The `IsRowFull` function checks if a row is completely filled with blocks.
- The `ClearRow` function clears a full row.
- The `MoveRowDown` function moves rows down to fill the cleared space.
- The function returns the number of rows cleared.

Shows how the grid checks for and clears full rows, moving the rows above down as needed.

5.8 Updating the Score

The `Game::UpdateScore` function updates the player's score based on the number of lines cleared and additional points for moving blocks down.

```cpp
// src/game.cpp

void Game::UpdateScore(int linesCleared, int moveDownPoints){
    if (linesCleared == 1) score += 100;
    else if (linesCleared == 2) score += 200;
    else if (linesCleared == 3) score += 400;
    else if (linesCleared == 4) score += 800;
    // No need for else case as we can't break more than 4 lines at once
    score += moveDownPoints;
}
```

Explanation:

- The score is updated based on the number of lines cleared.
- Additional points are added for moving blocks down.
- The function ensures that the score is updated correctly based on the game events.

Shows how the score is updated based on the number of lines cleared and additional points for moving blocks down.

6 Screenshots

Below are some screenshots demonstrating various aspects of the Tetris project.

6.1 Title Screen

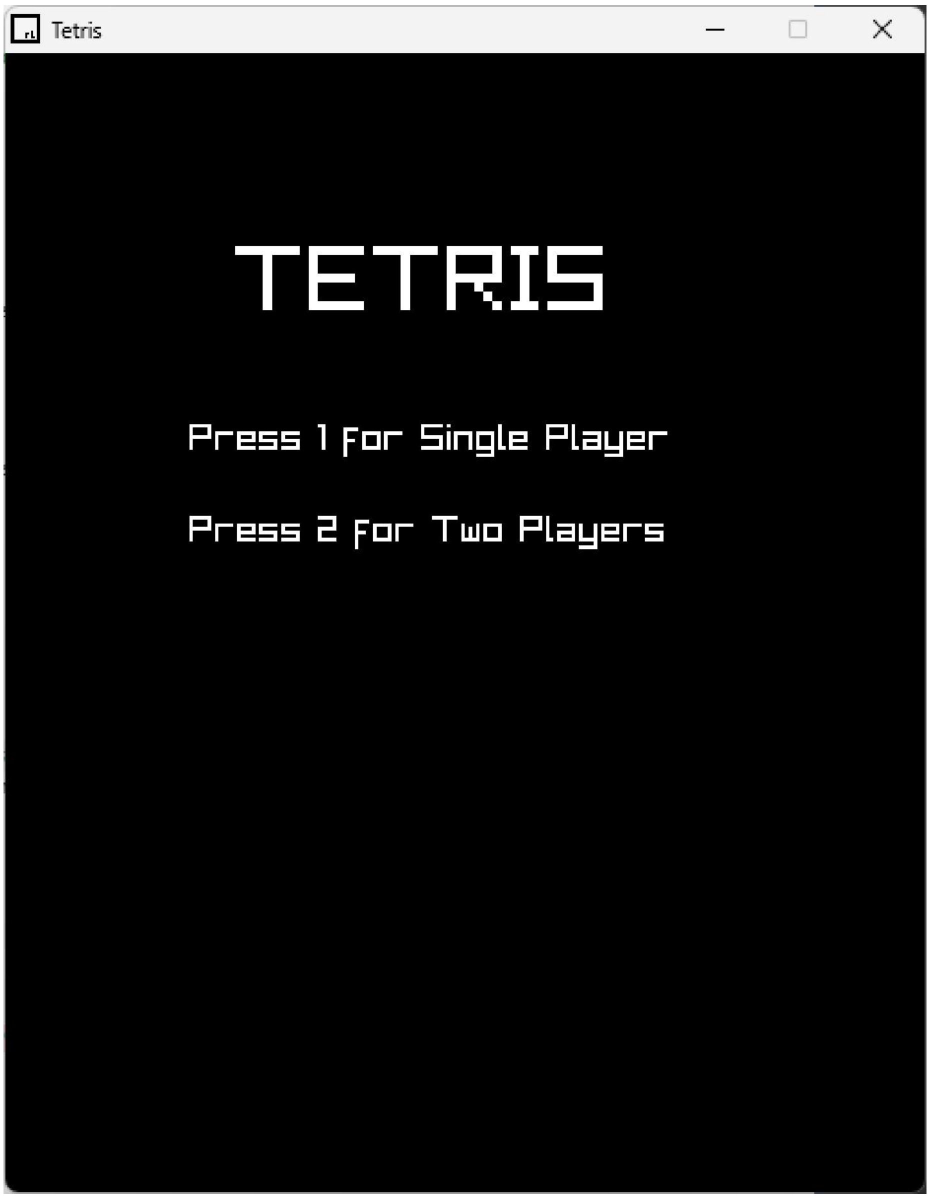


Figure 1: Title screen where the player can choose between single-player and two-player modes.

6.2 Single-Player Mode

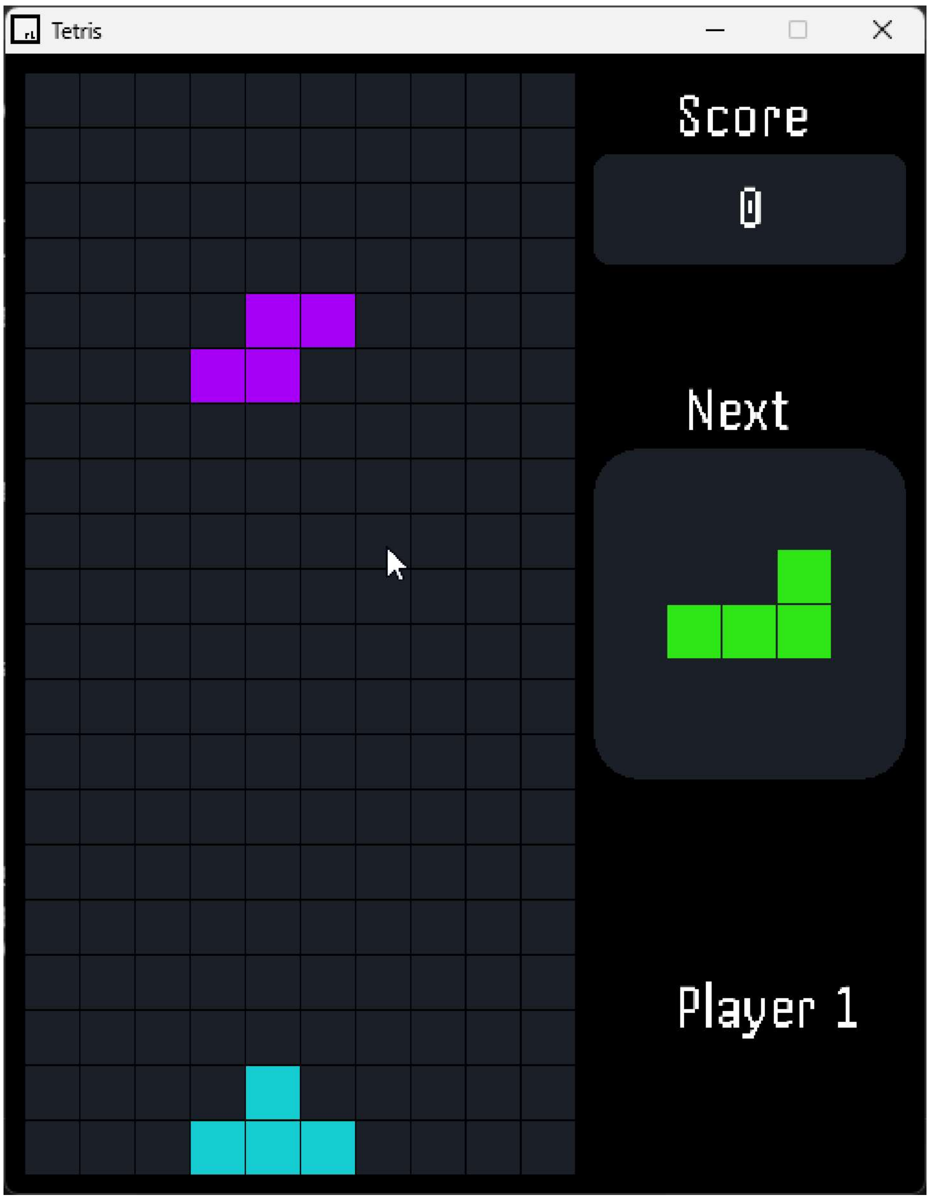


Figure 2: Single-player mode showing the game grid, falling block, and UI elements.

6.3 Two-Player Mode

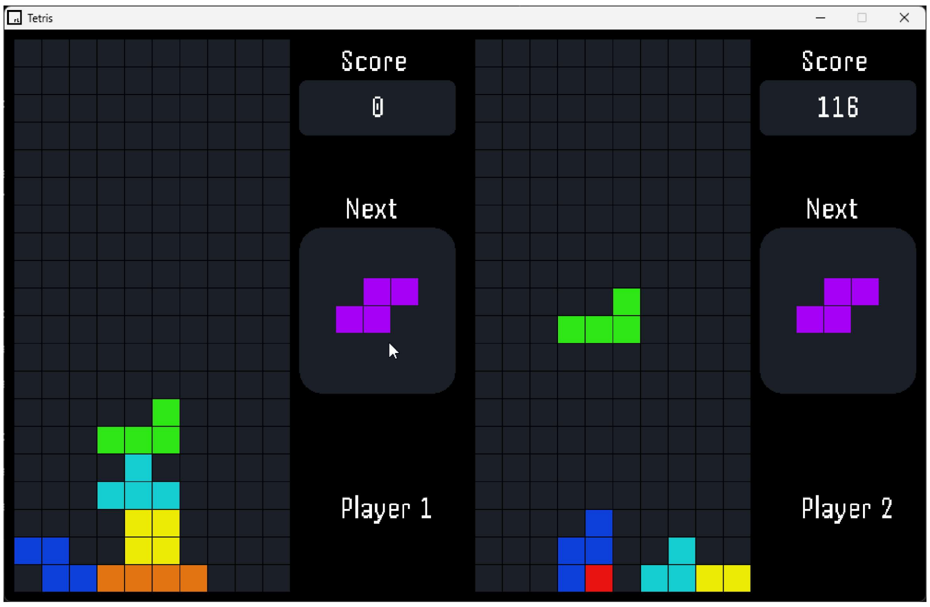
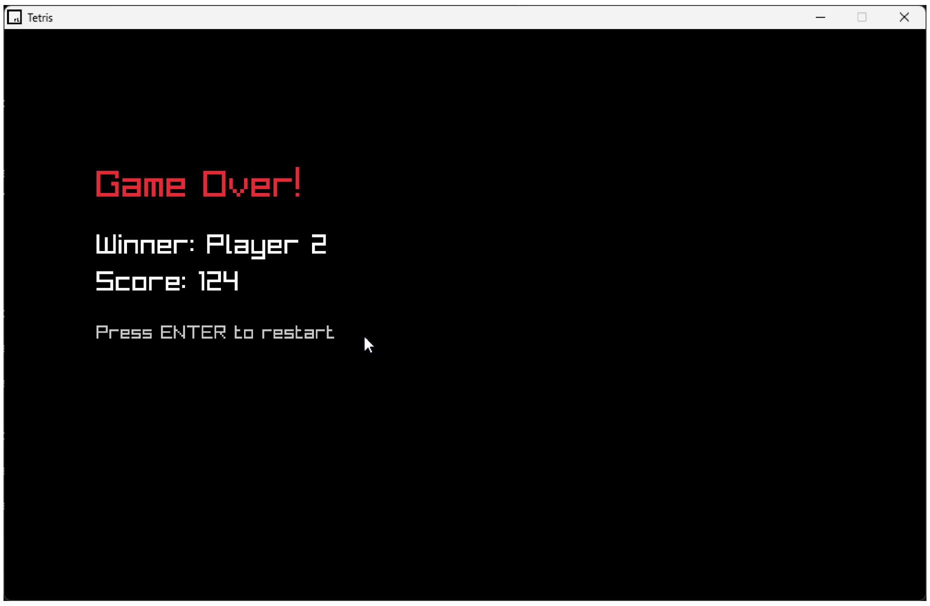


Figure 3: Two-player mode showing both game grids side by side with UI elements for both players.

6.4 Game Over Screen



Game Over!

Winner: Player 2
Score: 124

Press ENTER to restart

Figure 4: Game over screen displaying the winner's information and score.

7 Building the Project

The project is developed in C++ using the Raylib graphics library, and the instructions are tailored for Windows users. This project was developed on Windows 10 using MinGW-w64 as the compiler. The instructions assume a similar setup, but the project can be adapted for other platforms with minor modifications. The project can also be downloaded from my GitHub `https://github.com/Athidathan/OOPtetris/`.

7.1 Requirements

- **Operating System**: Windows 10 or higher
- **Compiler**: MinGW-w64 (GCC for Windows) or Microsoft Visual Studio
- **Library**: Raylib (version 4.0 or higher)
- **Make Build System**: MinGW-w64 includes `mingw32-make` for building the project using the provided `Makefile`.

7.2 Instructions

1. Install the required compiler and Raylib library.
2. Open the Command Prompt and navigate to the project directory:
```
    cd path\to\your\project
```
3. Compile the project using the provided `Makefile`:
```
    mingw32-make
```
4. Run the game executable:
```
    game.exe
```
5. Alternatively, you can use Visual Studio Code to compile and run the project:
   - Open the project folder in Visual Studio Code.
   - Press `F5` to compile and run the project.

8 Conclusion

Working on this Tetris project has been a valuable experience that helped me deepen my understanding of Object-Oriented Programming (OOP) in C++. Through implementing key concepts like encapsulation, abstraction, inheritance, polymorphism, and operator overloading, I've developed a better grasp of how OOP principles apply in real-world software development.

This project taught me how to structure a C++ application with classes and inheritance, handle game logic and state management, and use the Raylib graphics library for rendering and input. Using polymorphism with the `GameMode` abstract class and its derived classes (`SinglePlayer` and

`TwoPlayer`) highlighted how flexible and scalable this design is, especially when it comes to adding new features like additional game modes.

Implementing exception handling showed me the importance of managing errors properly to make the program more stable and reliable. Raylib's built-in error handling also made resource management simpler and kept the code cleaner.

Looking ahead, there's plenty of room for improvement and extensibility in this project. I'm particularly interested in exploring features that could help me learn new concepts, like networking, AI, or creating a more polished user interface. These additions would not only make the game more engaging but also give me the opportunity to take on more advanced programming challenges.

Overall, this project has been a solid introduction to OOP in C++ and has given me a better understanding of how to apply these concepts in game development. It's been a rewarding experience and a great starting point for more complex programming projects.

References

[1] Raylib Official Website. https://www.raylib.com/

[2] Raylib Learning Resources. https://github.com/raysan5/raylib/wiki

[3] Tetris Game. https://en.wikipedia.org/wiki/Tetris