[TETRIS]

INTRODUCTION

The aim of this project is to develop a Tetris game using object-oriented programming principles in C++. Tetris, a classic puzzle game, has remained popular for decades due to its simple yet addictive gameplay. This project seeks to recreate the essence of Tetris while also providing a learning opportunity for implementing OOP concepts in a practical scenario.

BACKGROUND

The selection of the Tetris game as the project stems from its widespread recognition and appeal. Tetris is not only an entertaining game but also serves as an excellent platform for

learning and applying fundamental programming concepts. By undertaking this project, we aim to enhance our understanding of OOP principles and strengthen our programming skills.

PROJECT SPECIFICATION

The Tetris game will be developed in C++ using object-oriented programming paradigms. The following features are expected to be included in the final implementation:

1. GAME BOARD: A grid-based game board where Tetriminos (the geometric shapes) will fall.
2. TETRIMINOS: Seven different geometric shapes consisting of four blocks each, which can be rotated and moved horizontally as they fall.
3. USER INPUT: Ability for the player to move and rotate Tetriminos using keyboard controls.
4. COLLISION DETECTION: Detection of collisions between Tetriminos and the game board or other Tetriminos.
5. LINE CLEARING: Removal of completed lines from the game board when a horizontal line is completely filled.
6. SCORE TRACKING: Keeping track of the player's score based on the number of lines cleared.
7. LEVELS AND SPEED ADJUSTMENT: Increasing game speed and difficulty as the player progresses through levels.

PROBLEM ANALYSIS

Several challenges may arise during the development of the Tetris game, including:

Implementing efficient algorithms for collision detection and line clearing to ensure smooth gameplay. Designing a user-friendly interface for controlling Tetriminos using keyboard inputs. Managing memory efficiently to handle dynamic allocation and deallocation of Tetriminos and game board elements. Balancing game difficulty to provide an enjoyable experience for players of varying skill levels. Testing the game thoroughly to identify and fix bugs or glitches that may affect gamepl

PROJECT BREAKDOWN

LIBRARIES USED

1. iostream: Provides input/output streams for C++.
2. cstdlib: Contains standard C library functions, such as srand and rand.
3. raylib.h: A simple and easy-to-use library for creating games and visual applications.
4. string: Provides string manipulation functions for C++.
5. vector: A dynamic array provided by the Standard Template Library (STL) in C++.
6. map: An associative container provided by the STL in C++.
7. math.h: Contains mathematical functions for C++.
8. ctime: Contains functions for manipulating the system time in C++.
9. fstream: Provides file stream classes for C++.

FUNCTIONS

1. getCellColors(): A private function of the Colors class that returns a vector of Color objects representing the colors used in the game.
2. Colors::Colors(): The constructor for the Colors class that initializes the gameColors vector with the colors returned by getCellColors().
3. Colors::getBlockColor(int blockId): A public function of the Colors class that returns the color of a block with the given blockId.
4. Grid::Grid(): The constructor for the Grid class that initializes the game grid with a size of 20x10 and sets all cells to 0.
5. Grid::setGrid(): A function that sets all cells in the grid to 0.
6. Grid::setGridCell(int Xindex, int Yindex, int cellValue): A function that sets the value of a cell in the grid at the given Xindex and Yindex.
7. Grid::getGrid(): A function that prints the contents of the grid to the console.
8. Grid::Draw(): A function that draws the grid on the screen using the raylib library.
9. Grid::isCellInsideGrid(int row, int column): A function that checks if a given row and column are within the bounds of the grid.
10. Grid::IsCellEmpty(int row, int column): A function that checks if a cell in the grid at the given row and column is empty.
11. Grid::ClearFullRows(): A function that clears any full rows in the grid and returns the number of rows cleared.
12. Position::Position(): The default constructor for the Position class that initializes the row and column variables to 0.
13. Position::Position(int row, int column): A constructor for the Position class that initializes the row and column variables to the given values.
14. Block::Block(): The constructor for the Block class that initializes
15. the uniqueId, cellSize, rotationState, moveX, and moveY variables to 0.
16. Block::setUniqueId(int uniqueId): A function that sets the uniqueId variable of the Block class to the given value.
17. Block::getUniqueId(): A function that returns the uniqueId variable of the Block class.
18. Block::Draw(int offsetX, int offsetY): A function that draws the block on the screen using the raylib library.
19. Block::getCellsPosition(): A function that returns a vector of Position objects representing the positions of the cells in the block.
20. Block::move(int rows, int columns): A function that moves the block by the given number of rows and columns.
21. Block::initialOffset(): A function that sets the initial offset of the block based on its type.
22. Block::Rotate(): A function that rotates the block clockwise.
23. Block::UndoRotation(): A function that rotates the block counter-clockwise.
24. LBlock::LBlock(): The constructor for the LBlock class that initializes the cells map with the positions of the cells in the L-shaped block.
25. LBlock::initialOffset(): A function that sets the initial offset of the L-shaped block.
26. JBlock::JBlock(): The constructor for the JBlock class that initializes the cells map with the positions of the cells in the J-shaped block.
27. JBlock::initialOffset(): A function that sets the initial offset of the J-shaped block.
28. IBlock::IBlock(): The constructor for the IBlock class that initializes the cells map with the positions of the cells in the I-shaped block.
29. IBlock::initialOffset(): A function that sets the initial offset of the I-shaped block.
30. OBlock::OBlock(): The constructor for the OBlock class that initializes the cells map with the positions of the cells in the O-shaped block.
31. OBlock::initialOffset(): A function that sets the initial offset of the O-shaped block.
32. SBlock::SBlock(): The constructor for the SBlock class that initializes the cells map with the positions of the cells in the S-shaped block.
33. SBlock::initialOffset(): A function that sets the initial offset of the S-shaped block.
34. TBlock::TBlock(): The constructor for the TBlock class that initializes the cells map with the positions of the cells in the T-shaped block.
35. TBlock::initialOffset(): A function that sets the initial offset of the T-shaped block.
36. ZBlock::ZBlock(): The constructor for the ZBlock class that initializes the cells map with the positions of the cells in the Z-shaped block.
37. ZBlock::initialOffset(): A function that sets the initial offset of the Z-shaped block.
38. Game::Game(): The constructor for the Game class that initializes the game grid, blocks, current block, next block, game state, score, level, and fall speed.
39. Game::saveState(): A function that saves the game state to a binary file.
40. Game::loadState(): A function that loads the game state from a binary file.
41. Game::getRandomBlock(): A function that returns a random block from the blocks vector.
42. Game::getAllBlocks(): A function that initializes and returns a vector of all the block types.
43. Game::Draw(): A function that draws the game on the screen using the raylib library.
44. Game::handleInput(): A function that handles user input for the game.
45. Game::isBlockOutside(): A function that checks if the current block is outside the grid.
46. Game::RotateBlock(): A function that rotates the current block.
47. Game::LockBlock(): A function that locks the current block into the grid and generates a new block.
48. Game::BlockFits(): A function that checks if the current block fits into the grid.
49. Game::Reset(): A function that resets the game state.
50. Game::UpdateScore(int linesClear, int moveDownPoints): A function that updates the score based on the number of lines cleared and the number of moves down.
51. Game::moveBlockLeft(): A function that moves the current block one cell to the left.
52. Game::moveBlockRight(): A function that moves the current block one cell to the right.
53. Game::moveBlockDown(): A function that moves the current block one cell down.

CLASSES

1. Colors: A class that represents the colors used in the game. Grid: A class that represents the game grid.
2. Position: A class that represents a position on the grid.
3. Block: An abstract base class that represents a block in the game.
4. LBlock, JBlock, IBlock, OBlock, SBlock, TBlock, ZBlock: Classes that inherit from Block and represent the different block shapes.
5. Game: A class that represents the game state and logic.

INTEGRATED OOP PRINCIPLES

ENCAPSULATION

In the Colors class, the gameColors vector is declared as private, encapsulating the color data within the class. Access to this data is controlled through getter methods (getBlockColor) which ensure proper encapsulation of the color information.

INSTANTIATION:

Objects of various classes such as Colors, Grid, and Block are instantiated throughout the code.

Colors colors; Grid gameGrid; Block currentBlock;

GETTER-SETTER (ACCESSOR-MUTATOR):

Getter and setter methods are used for accessing and modifying private member variables. For instance:

int getUniqueId(); void setUniqueId(int uniqueId);

PRIVATE-PUBLIC ATTRIBUTES:

Member variables like uniqueId, cellSize, rotationState, moveX, and moveY in the Block class are declared as private, encapsulating their access and allowing controlled modification through public methods.

CONSTRUCTORS:

Constructors are used for initializing objects. For example:

Colors::Colors() { ... } Grid::Grid() { ... } Block::Block() { ... }

MEMBER INITIALIZERS:

Member initializers are used in the constructors to initialize member variables. For instance:

Game::Game()

: gameGrid(Grid()), blocks(getAllBlocks()), currentBlock(getRandomBlock()), nextBlock(getRand omBlock()), gameOver(false), score(0), level(1), fallSpeed(0.3), linesToNextLevel(2), linesCleare d(0) { ... }

INHERITANCE:

Inheritance is utilized for creating specialized block classes (LBlock, JBlock, etc.) inheriting common properties and methods from the Block class.

POLYMORPHISM (RUNTIME AND COMPILE-TIME):

Runtime polymorphism is achieved through virtual functions like DifficultyLevel() in the gameLevel hierarchy.

Compile-time polymorphism is demonstrated through function overloading, such as different constructors for the Position class.

FUNCTION OVERRIDING AND OVERLOADING:

Function overriding is employed in subclasses like LBlock, where initialOffset() is overridden to provide specific behavior.

Function overloading is utilized in constructors of various classes, providing different initialization options.

FRIEND CLASSES AND FUNCTIONS:

Friend classes and functions are used sparingly to allow access to private members of certain classes. For example:

friend class Grid; friend class Block;

COMPOSITION:

Composition is employed in the Game class, which contains instances of the Grid and Block classes as member variables.

ARRAY OF OBJECTS:

Arrays of objects are created dynamically, such as in the Grid constructor where a 2D array is allocated.

LATE BINDING (VIRTUAL FUNCTIONS):

Late binding is utilized through virtual functions like DifficultyLevel() in the gameLevel hierarchy.

ABSTRACT CLASSES AND FUNCTIONS:

Abstract classes are employed in the gameLevel hierarchy, where gameLevel is an abstract class with a pure virtual function DifficultyLevel().

FILE HANDLING:

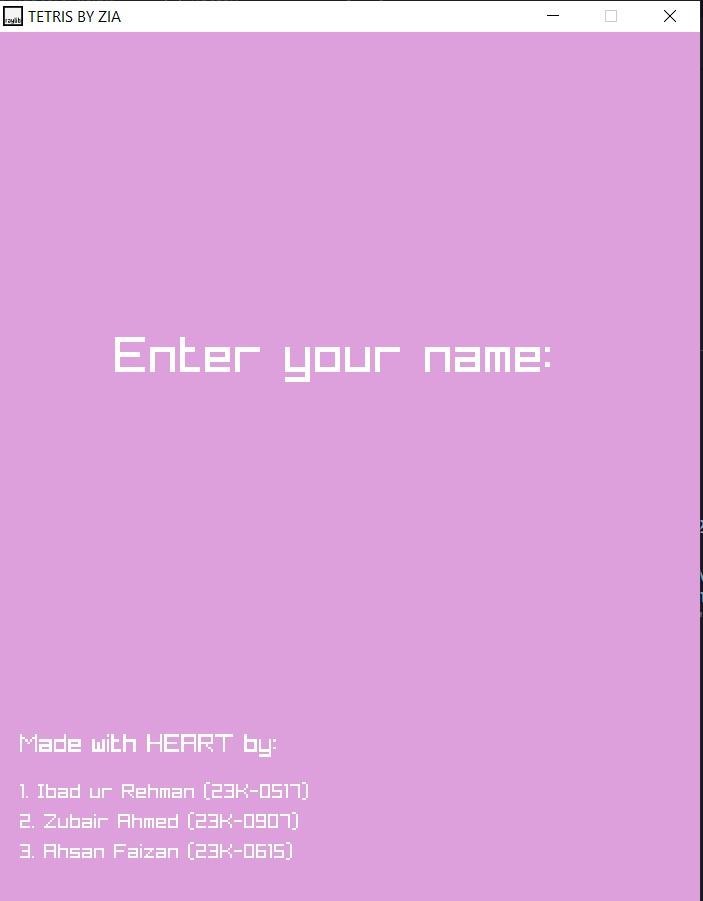
File handling are used for loading and saving game states and high scores int loadState(); void saveState();

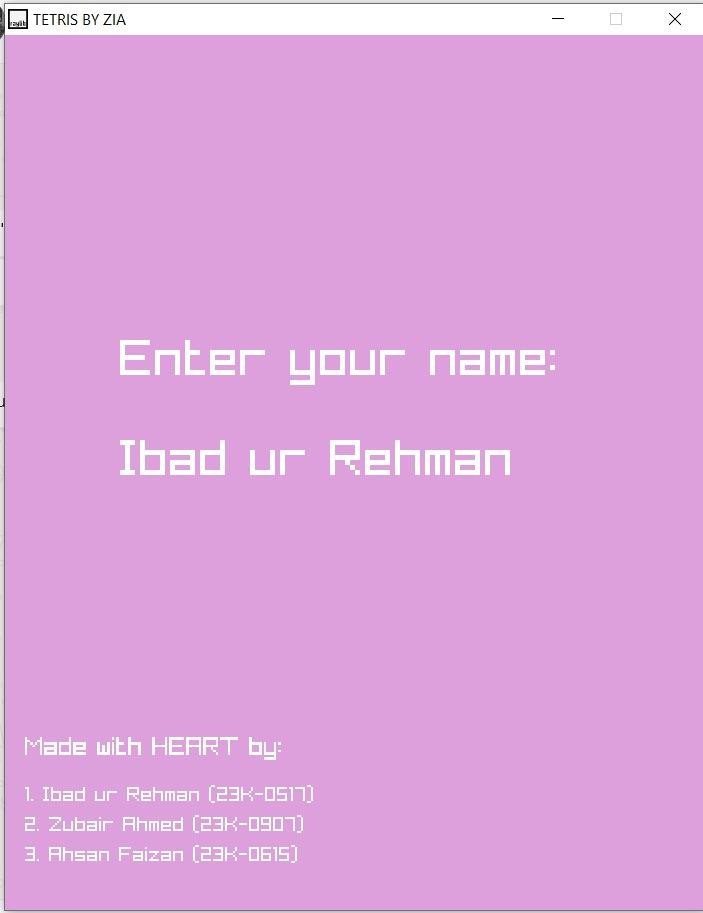
IMPLEMENTATION & TESTING:

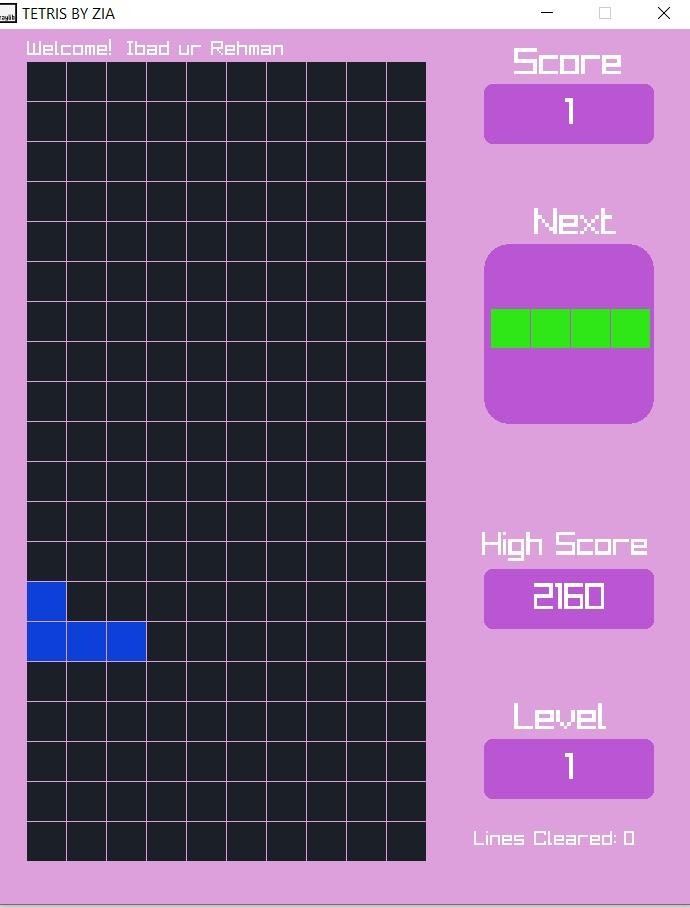
The implementation leverages object-oriented programming principles such as encapsulation, inheritance, and polymorphism. Each class has a clear responsibility, promoting modularity and code organization. The game logic is structured and well-defined, making it easier to maintain and extend.

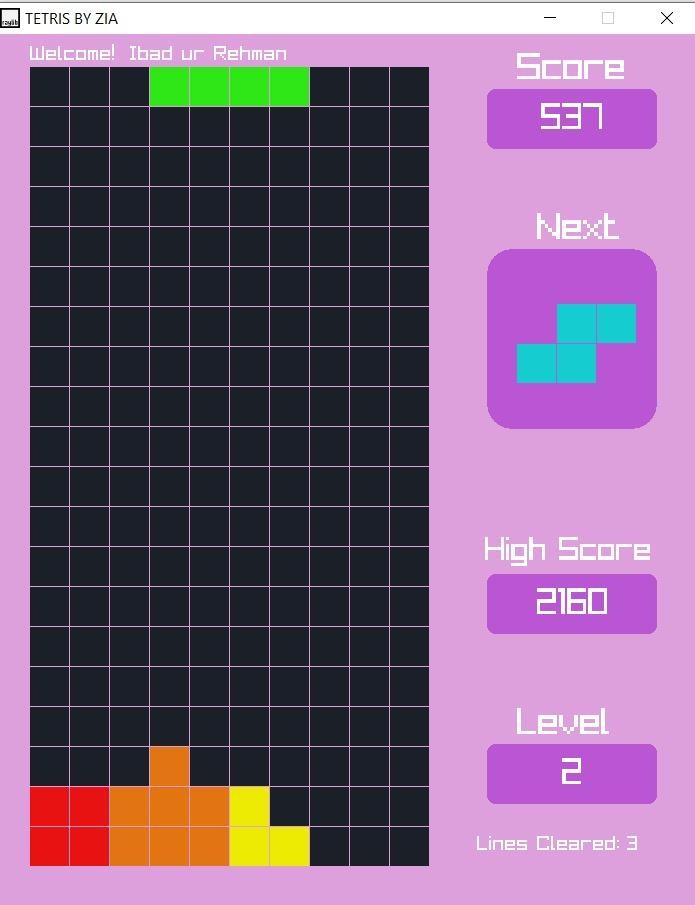
Testing involves verifying that each component functions correctly in isolation and integrates seamlessly with others. This includes unit tests for individual classes/methods and integration tests to ensure they work together as expected. Additionally, gameplay testing ensures the game behaves as intended, responding correctly to player input and maintaining consistent game state.

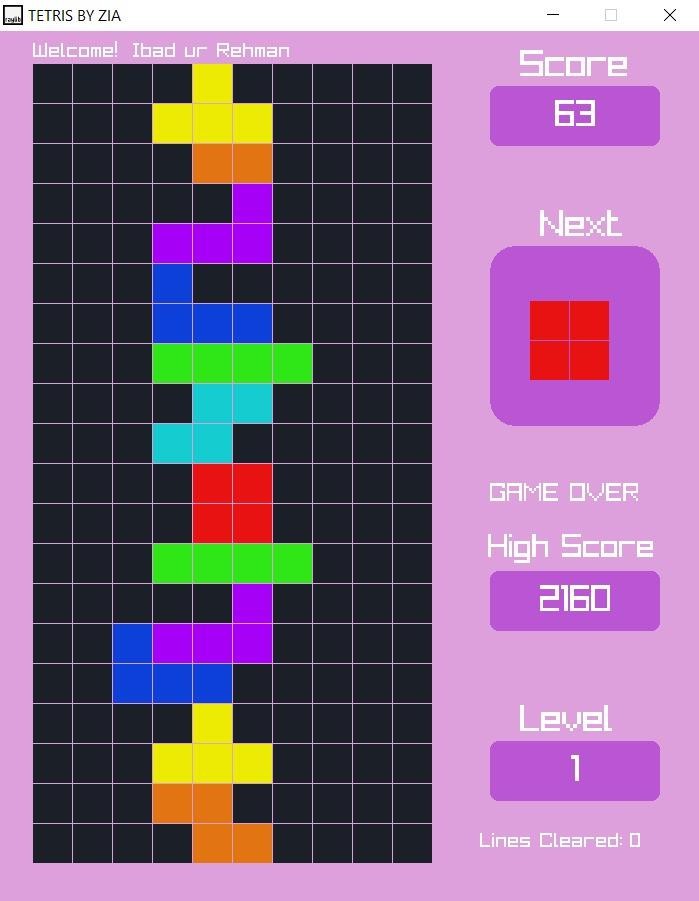
SCREENSHOTS/ OUTPUTS:











CONCLUSION

In summary, this project aimed to develop a Tetris game using object-oriented programming

principles in C++, leveraging the game's timeless appeal to reinforce our understanding of OOP concepts. Throughout the development process, we successfully implemented key features such as the game board, Tetriminos, user input controls, collision detection, line clearing, and score tracking. Additionally, we incorporated advanced functionalities such as levels and speed adjustment to enhance the gameplay experience.

DISCUSSION

The development of the Tetris game presented several challenges, each requiring careful consideration and problem-solving. Efficient algorithms were crucial for collision detection and line clearing to ensure smooth gameplay, especially as the game speed increased with progressing levels. Designing a user-friendly interface for controlling Tetriminos using keyboard inputs demanded meticulous attention to detail to provide a seamless gaming experience. Memory management was another critical aspect, necessitating the efficient handling of dynamic memory allocation and deallocation to prevent memory leaks and optimize performance. Balancing game difficulty proved to be a delicate task, as we strived to cater to players of varying skill levels while maintaining an enjoyable and challenging experience for all.