Biquadris Final Report

CS 246 Spring 2023

Jiaxuan Ren (jren)

Introduction

This report discusses the technical aspects of the two-player game Biquadris, a strategic block-placement game with both text and graphical interfaces. The game implemented in C++ offers five distinct levels with increasing complexity and allows players to eliminate rows for survival and accumulating scores in the process. In addition, it adds an additional layer of complexity with special actions that allow players to disrupt their opponents.

Overview

The core logic and control flow of Biquadris are managed within the main.cc file, which is responsible for iteratively processing user inputs and controlling the various stages of the game. A key class in the codebase, 'Gameboard', manages the specific game settings and governs the display of gameboard. Player, Pixel, and Xwindow class takes responsibility for managing player statistics, gameboard pixel status, and display setting respectively. These three classes are governed by Gameboard.

Two abstract classes, 'Block' and 'Level', each have multiple subclasses which encapsulate the behavior of the various types of blocks and game levels. This design showcases the power of object-oriented programming, allowing for modular code and enhancing the flexibility of the game design.

Updated UML Class Model

The UML graph is attached on the last page.

Changes to the Original UML:

There are no significant structural changes to the original UML except for removing unnecessary and redundant classes, which are GeneralBoard and GameboardDecorator. Redundant functions are modified as well (e.g. up/down/left/right functions are combined into move function). Fields are modified to accommodate specific implementation requirements, especially adding and deleting fields in the classes Block, Level, and Pixel.

Design

Note: Only important fields and functions are listed and discussed below.

Block

Block is an abstract class with concrete subclasses IBlock, SBlock, JBlock, LBlock, OBlock, ZBlock and TBlock. Each concrete Block class stores the x coordinate and y coordinate of each point of the Block in a vector<int> and supports clockwise rotation by calculating the coordinate of new coordinate with formulas. This class supports the change and get of coordinate, rotation, deep copy and several judge functions. This design ensures flexible block dynamics in alignment with the game rules.

- ***Rotate* :** The integer rotate represents how the state of rotation (0 stands for original location, 1 stands for location after 1 clockwise rotations… 3 stand for location after 3 clockwise rotations).
- ***name:*** The character name stores the type of the Block, such as 'I' (IBlock), 'J' (JBlock) and so on.
- ***Xcoordinate & Ycoordinate:*** Xcoordinate and Ycoordinate are two vector<int> to store the location of each point of the block currently on board.
- ***heavyBlock:*** The bool heavy is assigned to true if the level is bigger than or equal to 3 when the block is created .
- ***set:*** The function set sets the value index of either Xcoordinate or Ycoordinate to be value num. If judge is "X", the function modifies Xcoordinate. If judge is "Y", the function modifies Ycoordinate.
- ***CW():*** The function CW rotates current block clockwise. The core logic is to calculate the x and y of each new coordinate based on different rotate value. We will first find the lowest x and y in current 2 vector<int> and use set to assign every new coordinate by pattern. For example, if the block is IBlock and it never rotates before (rotate == 0) and we use x to denote smallest x coordinate and y to denote smallest y coordinate. Then we can write a for loop with variable I from 0 to 4, and in each loop, program will execute set("X", i, x – i) and set("Y", i, y). After the for loop we will increase rotate by 1. If it hits 4, we reset it to be 0.

Level

Our Level class, following the Factory design pattern, is solely dedicated to generating Block objects, exemplifying the Single Responsibility Principle. The concrete Level class generates the type of block and returns it to Createblock function in base Level class to constructs diverse Block type. In Level0 class, we use a vector<char>(only exists in Level0) to stores the block types in given file with a integer index to track. If we hit EOF, we then reset index back to 0 to continuously generate block if the game hasn't ended yet. In other concrete subclasses, we use the remainder of certain number to achieve expected possibility for each block and we set the heavyBlock state in generated new Block to be true when level is bigger or equal than 4. We also introduced a new Level subclass, Level5, for player who wins in one round with level 4 only.

- ***createBlock:*** The createBlock function exists in base Level class. It takes in character such as 'I', 'J' and returns respective Block pointer.
- ***generateBlock:*** The generateBlock function gets the char type of block by either

reading the file or designed mathematical probability function and pass the character to createBlock and return the Block pointer back to the program.

Gameboard

Our Gameboard class is the key class for managing and modifying the board state while interpreting and processing player actions. It contains two pointer of player and supports clearing line, initializing of current and next block, rotation, shortcut generate, player move and the change of the map, updating the game display. During the initialization and movement of the block, we use an integer total in each player to track how many blocks they have dropped, and a vector<int> in each player to record the current state of each block. Every time we try moving a block, it's necessary for us to swap the number(we assign the current total to each point of the block when it's created), color and level(we assign the current level to each point of the block when it's created) between two points properly by using GetColor, SetColor, GetLevel, SetLevel, GetNumber and SetNumber.

- ***initPlayer:*** The initPlayer initializes the board. It assigns the level, hiscore, command(whether to read input from file) and file(string name) value and initialize other fields of player properly. It resizes vector such as PixelList, and uses a for loop to resize and initialize vector<vector<Pixel>> such as Range and nextArea in each player.

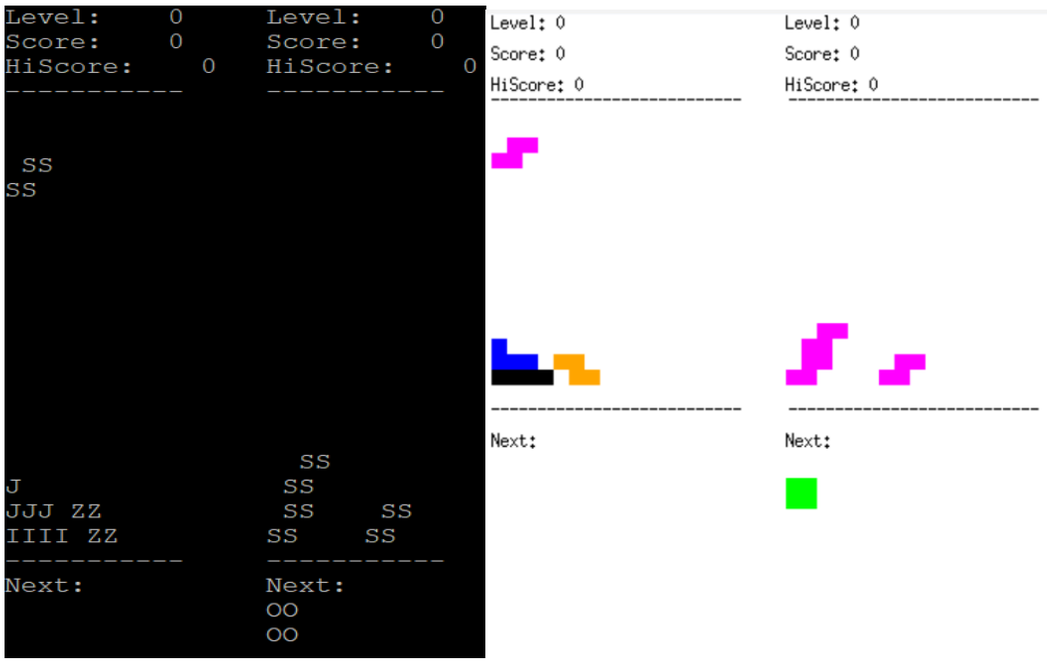


Figure 1: sample text display and graphical display respectively

- ***generate:*** We printed the board in text by function generate in Gameboard. In Range and nexrArea section, we can directly use cout << Range[i][i]; or cout << nextArea[i][i] in two for loops; to output the content in each Pixel by overloading << operator in Pixel class.
- ***graphicDisplay:*** We printed the board graphically by using Xwindow class provided in a3. The graphicDisplay function in Gameboard draws the text appropriate location by drawString. Similarly, we have 2 for loops for Range and nextArea and draw each block

with size 10 by fillRectangle.

- ***clearLine:*** The function clearLine detects whether it's possible to clear line after the command of player and modifies the board, the integer reference add to be score that can be earned by letting a block completely disappear, and the integer reference num to be total number of lines that can be cleared at this moment.
- ***endCheck:*** The endCheck function checks whether the block can move down any more.
- ***starblockAdd:*** When level is bigger than or equal to 4, starblockAdd is triggered to locate a starBlock in the central of the board.
- ***move:*** Move function can modify the board by different string input such as "left", "right", "down".
- ***Shortcut:*** The shortcut function returns the real command string analyzed from the shortcut input, and modifies the int reference times to be the times this command should work(0,1,2…).
- ***CW:*** The CW function first gets a deep copy of current block and calls the CW function of the current block to get the set of new x and y coordinate and modifies the board by swapping the number, level, color of old and new coordinate. If it is CCW, then calls the CW function for 3 times.

Player

Player is a structure that includes all fields needed by other classes.

Main

Our main program acts as the controller in this project, we use an integer turn to decide whether it's turn of player one or player two. We construct game loop for each plater, consistently interpreting user input commands, and subsequently performing corresponding actions such as moving and rotating game blocks or changing game levels.
After each command in the while loop, the program calls endCheck function to detect whether the block cannot go down anymore. When endCheck returns true or player chooses to drop the block, program breaks.
When program breaksthe command processing loop, it will first calculate the score earned in this round and check the possibility of special action and validity of the block. If the player clears 2 or more rows, program will prompt player to choose from blind, heavy, force to add on opponent. After special action, program will check the validity of the block and if it is invalid, gameover value will be assigned true and program will output end message and ask whether player wants to play again to decide whether quits or restarts the game.
Generally speaking, main program updates the state of the blocks, levels, scores and highest scores while handling end-game scenarios and transitions between game levels. It also manages the game display updates to accurately reflect the current game state. Moreover, the controller oversees the correct instantiation and disposal of game entities, ensuring game flow integrity. It clearly separates the user input handling and game logic from the model, embodying the principles of the Model-View-Controller (MVC) design pattern.

- ***Board:*** It is a Gameboard and we modifies all the changes on it.
- ***index1 & index2:*** It tracks the current location of the command generate from a file(If encounters sequence file before) in vector<string>cm1 or vector<string>cm2. Every times when player takes a command string from the vector<string>, index increases by one, and if it hits EOF, program will assign the Boolean terminate of player to be true.
- ***LevelRestrict1 & LevelRestrict2:*** It is the bool value that is assigned to true only when the player has won at least once in a Level4 game.
- ***neverAdd1 & neverAdd2:*** It is the bool value that is assigned to true only when the level is 5 and player placed 10 blocks without clearing at least one row. If it is assigned to true, we will never add the value 'add'(extra scores you can earn by eliminating a whole block) when calculating score.

Pixel

Pixel class is the component of Range and nextArea in player. You can consider it as every point that made up of the board, which can be empty or specific letter. Please notice that block is not made up of Pixel, block only contains the 2 vector<int> that stores the x and y coordinate of each point of the block displayed on board and pixel only works for Range and nextArea.

- ***currentLevel:*** the number of level when the pixel is created.(When we initialize a block on board, we take the number in its Xcoordinate and Ycoordinate and construct pixel on that location, but please notice that pixel is not part of the block!)
- ***blockNumber:*** the number of total when the pixel is created.
- ***blind:*** a Boolean value that is only assigned to true when the player has blind effect on it.
- ***color:*** the string of color of a pixel, such as "I", "J" and so on.
- ***TurnBlind:*** The TurnBlind function modifies the value of blind of the pixel to be true.
- ***NotBlind:*** The NotBlind function modifies the value of blind of the pixel to be false.
- ***<<operator:*** It is an overloading operator and it outputs the color of the pixel when the blind state is false, and outputs "?" when the blind state of the pixel is true.

Resilience to Change

Factory Design Patterns

Our program is flexible to changes by adopting Factory Design Patterns. For example, an abstract Level class is implemented and branches into multiple level0…leveln concrete classes. New levels are introduced by simply defining a new concrete subclass and update the factory Level class. Level deletion or modification also only requires modification to the concrete class and update the factory. Similar idea applies to factory Block class and concrete subclass e.g. IBlock, TBlock, SBlock.

Graphical Features

The design of the Window class has been done thoughtfully to accommodate future changes and updates. It includes features to handle a variety of dimensions and supports an extendable color map, providing adaptability for additional graphical requirements.

New Commands

New commands like different input shortcuts in the main.cc file are accommodated by the shortcut function in Gameboard. If the user wants to modify and add different shortcuts or commands, they can access it via shortcut function instead of modifying the whole program.

Answers to Questions

Question 1:

How could you design your system (or modify your existing design) to allow for some generated blocks to disappear from the screen if not cleared before 10 more blocks have fallen? Could the generation of such blocks be easily confined to more advanced levels?

Answer: we modified the player class to include a variable int total, representing the number of blocks dropped since last row elimination. Then, for each pixel in our grid, we check if (total – pixel.getnumber() == 11). If true, we reset that pixel to blank, and shift all above blocks down by one by swapping that blank pixel with all pixels above it. This mechanism can easily be confined to advanced levels by setting conditional logic checking the player's current level.

Question 2:

How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?

Answer: Similar to Resilience to Change subpart Factory Design Pattern, introduction of new levels can simply be achieved by defining a new subclass, updating the factory to include this new level in its creation method, and then linking the new subclass to the base Level class. Hence, when a new level is added to the system, we only need to recompile the Level class and the new level subclass.

Question 3:

How could you design your program to allow for multiple effects to applied simultaneously? What if we invented more kinds of effects? Can you prevent your program from having one else-branch for every possible combination?

Answer: To allow for multiple effects to be applied simultaneously, we designed the program with independent conditional checks for each special effect. This allows us to activate any combination of effects simultaneously without any conflicts.

Our system does not rely on a single "else-if" chain to manage effects. Instead, the effects are managed independently with separate "if" conditions (e.g., "if(blind)", "if(force)", "if(heavy)"). This design allows easy inclusion of new effects, as we simply need to add a new independent "if" condition for each new effect. The effects can then operate in parallel without interfering with one another, enabling the system to handle any combination of effects.

Question 4:

How could you design your system to accommodate the addition of new command names, or changes to existing command names, with minimal changes to source and minimal recompilation? (We acknowledge, of course, that adding a new command probably means adding a new feature, which can mean adding a non-trivial amount of code.) How difficult would it be to adapt your system to support a command whereby a user could rename existing commands (e.g. something like rename counterclockwise cc)? How might you support a “macro” language, which would allow you to give a name to a sequence of commands? Keep in mind the effect that all of these features would have on the available shortcuts for existing command names.

Answer: Adding new commands or changing existing ones would involve minimal changes to our source code. The changes would only be made in the main.cc file, which handles user inputs. This is where we use if-else statements to check if the user input matches a particular command, where we easily change command line.

To support macros within the current command structure in main.cc, we need to add a command for defining a macro. Specifically, outside the command processing loop, we need a vector<vector<string>> macros to store all macro names. Inside the command processing loop, we need to handle the ‘define_macro’ command, storing macro name and the following command strings in the vector<vector<string>> macros.

Then, when the user inputs a macro name as a command, the command processing loop should recognize it, retrieve the commands from vector<vector<string>> macros, and execute them in order. Hence, supporting macro requires introducing a new variable vector<vector<string>> macros and adding an additional if conditional recognizing ‘define_macro’ command.

Extra Credit Features

Keyboard Support:

We implemented additional keyboard support for better user game experience. Instead of typing the full “down”, “left”, “right”, the program can simply scan for keyboard arrow input ←, ↓, →. Keys a, s, d, q or ccw, e or cw, dd also serves similar purpose representing left, down, right, counterclockwise rotation, clockwise rotation, and drop commands respectively.

New Level 5:

We implemented new level5 that adds new features in addition to level4 features. If a player failed to cancel any rows in 10 turns, all blocks existing on that player’s gameboard do not count for score when eliminated and the effect for affected blocks are permanent. Level 5 is also restricted to access until the player has won at least one game in level4. Invalid access will pop warning message and return to the original level.

Final Questions

Question 1:

What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

Answer: Working on the Biquadris project as a team emphasized the importance of clear communication and division of responsibilities. For example, one significant challenge during teamwork that we faced was with synchronization. Given the project's complexity, we divided tasks among team members based on the functionalities we were to develop. For instance, one member was working on the Player class, while another was focused on the Gameboard class. This approach led to issues in syncing our progress, as some functionalities were dependent on the completion of others.

We overcame this by implementing agile development methodologies. We organized our tasks into smaller, manageable sprints, which were updated daily in our group chat. This practice improved our coordination, allowing us to anticipate potential dependencies and plan accordingly. As a result, the development process became smoother and less prone to blockers.

If I am working alone, although I don’t need to share codes with others, I would still like to regularly update my code onto Github for version control purpose. This would help track changes andrevert to a previous state if needed. It's like a 'save point' in a video game, allowing experiment without worrying about irreversibly breaking the system.

Question2:

What would you have done differently if you had the chance to start over?

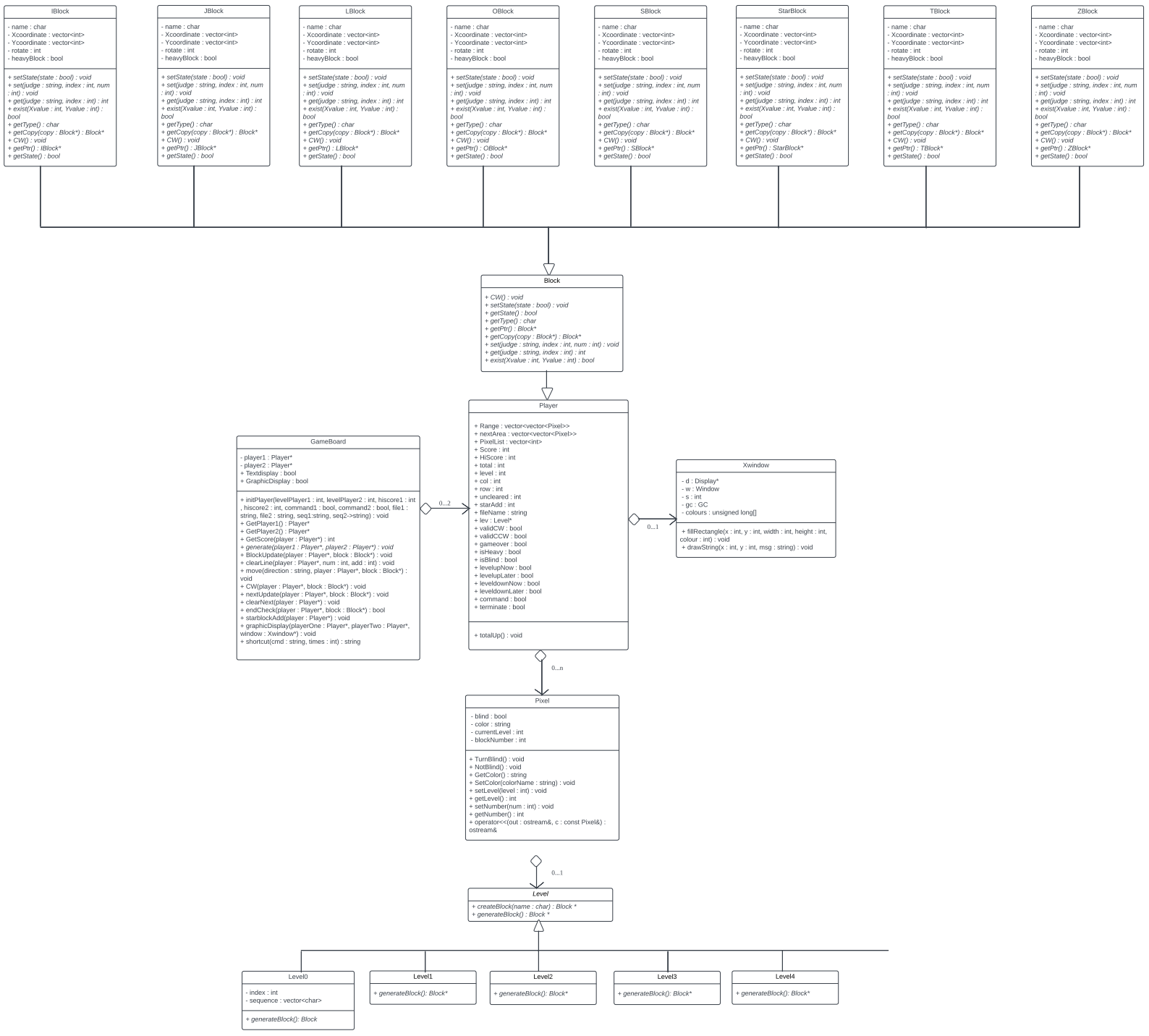
Answer: During the development, we decided to store x, y values of a block using a vector<vector<int>> structure. It seemed intuitive, but soon complicated our game mechanics, leading to unnecessary mathematical complexity and debugging time. In hindsight, a 4x4 pixel-based structure for all block types would have been better. This uniform grid would simplify operations across different blocks, save development time, and reduce the chance for errors.

Learning from this, in future developments, I'll consider the downstream implications of my design choices more thoroughly. Even if an option seems simple initially, I'll ensure it remains beneficial when handling complex mechanics, ultimately improving software quality.

Conclusion

The design and implementation of Biquadris significantly enhances our understanding to object-oriented programming principles. The use of abstract classes Block and Level, and their respective subclasses, underlines the principle of inheritance and polymorphism, which allows for enhanced flexibility of the code. Single responsibility principles are demonstrated in the implementation of Player, Pixel, Gameboard classes where we divide a complicated game structure into several classes each taking certain responsibility.

In addition, this project offered valuable experience in collaborative software development. The clear division of responsibility among the Player, Pixel, Block, Level classes facilitate efficient parallel development and encourages writing maintainable modular code.


IBlock
- name : char
- Xcoordinate : vector<int>
- Ycoordinate : vector<int>
- rotate : int
- heavyBlock : bool
+ setState(state : bool) : void
+ set(judge : string, index : int, num : int) : void
+ get(judge : string, index : int) : int
+ exist(Xvalue : int, Yvalue : int) : bool
+ getType() : char
+ getCopy(copy : Block*) : Block*
+ CW() : void
+ getPtr() : IBlock*
+ getState() : bool
JBlock
- name : char
- Xcoordinate : vector<int>
- Ycoordinate : vector<int>
- rotate : int
- heavyBlock : bool
+ setState(state : bool) : void
+ set(judge : string, index : int, num : int) : void
+ get(judge : string, index : int) : int
+ exist(Xvalue : int, Yvalue : int) : bool
+ getType() : char
+ getCopy(copy : Block*) : Block*
+ CW() : void
+ getPtr() : JBlock*
+ getState() : bool
LBlock
- name : char
- Xcoordinate : vector<int>
- Ycoordinate : vector<int>
- rotate : int
- heavyBlock : bool
+ setState(state : bool) : void
+ set(judge : string, index : int, num : int) : void
+ get(judge : string, index : int) : int
+ exist(Xvalue : int, Yvalue : int) : bool
+ getType() : char
+ getCopy(copy : Block*) : Block*
+ CW() : void
+ getPtr() : LBlock*
+ getState() : bool
OBlock
- name : char
- Xcoordinate : vector<int>
- Ycoordinate : vector<int>
- rotate : int
- heavyBlock : bool
+ setState(state : bool) : void
+ set(judge : string, index : int, num : int) : void
+ get(judge : string, index : int) : int
+ exist(Xvalue : int, Yvalue : int) : bool
+ getType() : char
+ getCopy(copy : Block*) : Block*
+ CW() : void
+ getPtr() : OBlock*
+ getState() : bool
SBlock
- name : char
- Xcoordinate : vector<int>
- Ycoordinate : vector<int>
- rotate : int
- heavyBlock : bool
+ setState(state : bool) : void
+ set(judge : string, index : int, num : int) : void
+ get(judge : string, index : int) : int
+ exist(Xvalue : int, Yvalue : int) : bool
+ getType() : char
+ getCopy(copy : Block*) : Block*
+ CW() : void
+ getPtr() : SBlock*
+ getState() : bool
StarBlock
- name : char
- Xcoordinate : vector<int>
- Ycoordinate : vector<int>
- rotate : int
- heavyBlock : bool
+ setState(state : bool) : void
+ set(judge : string, index : int, num : int) : void
+ get(judge : string, index : int) : int
+ exist(Xvalue : int, Yvalue : int) : bool
+ getType() : char
+ getCopy(copy : Block*) : Block*
+ CW() : void
+ getPtr() : StarBlock*
+ getState() : bool
TBlock
- name : char
- Xcoordinate : vector<int>
- Ycoordinate : vector<int>
- rotate : int
- heavyBlock : bool
+ setState(state : bool) : void
+ set(judge : string, index : int, num : int) : void
+ get(judge : string, index : int) : int
+ exist(Xvalue : int, Yvalue : int) : bool
+ getType() : char
+ getCopy(copy : Block*) : Block*
+ CW() : void
+ getPtr() : TBlock*
+ getState() : bool
ZBlock
- name : char
- Xcoordinate : vector<int>
- Ycoordinate : vector<int>
- rotate : int
- heavyBlock : bool
+ setState(state : bool) : void
+ set(judge : string, index : int, num : int) : void
+ get(judge : string, index : int) : int
+ exist(Xvalue : int, Yvalue : int) : bool
+ getType() : char
+ getCopy(copy : Block*) : Block*
+ CW() : void
+ getPtr() : ZBlock*
+ getState() : bool
Block
+ CW() : void
+ setState(state : bool) : void
+ getState() : bool
+ getType() : char
+ getPtr() : Block*
+ getCopy(copy : Block*) : Block*
+ set(judge : string, index : int, num : int) : void
+ get(judge : string, index : int) : int
+ exist(Xvalue : int, Yvalue : int) : bool
Player
+ Range : vector<vector<Pixel>>
+ nextArea : vector<vector<Pixel>>
+ PixelList : vector<int>
+ Score : int
+ HiScore : int
+ total : int
+ level : int
+ col : int
+ row : int
+ uncleared : int
+ starAdd : int
+ fileName : string
+ lev : Level*
+ validCW : bool
+ validCCW : bool
+ gameover : bool
+ isHeavy : bool
+ isBlind : bool
+ levelupNow : bool
+ levelupLater : bool
+ leveldownNow : bool
+ leveldownLater : bool
+ command : bool
+ terminate : bool
+ totalUp() : void
GameBoard
- player1 : Player*
- player2 : Player*
+ Textdisplay : bool
+ GraphicDisplay : bool
+ initPlayer(levelPlayer1 : int, levelPlayer2 : int, hiscore1 : int , hiscore2 : int, command1 : bool, command2 : bool, file1 : string, file2 : string, seq1:string, seq2->string) : void
+ GetPlayer1() : Player*
+ GetPlayer2() : Player*
+ GetScore(player : Player*) : int
+ generate(player1 : Player*, player2 : Player*) : void
+ BlockUpdate(player : Player*, block : Block*) : void
+ clearLine(player : Player*, num : int, add : int) : void
+ move(direction : string, player : Player*, block : Block*) : void
+ CW(player : Player*, block : Block*) : void
+ nextUpdate(player : Player*, block : Block*) : void
+ clearNext(player : Player*) : void
+ endCheck(player : Player*, block : Block*) : bool
+ starblockAdd(player : Player*) : void
+ graphicDisplay(playerOne : Player*, playerTwo : Player*, window : Xwindow*) : void
+ shortcut(cmd : string, times : int) : string
0...2
Xwindow
- d : Display*
- w : Window
- s : int
- gc : GC
- colours : unsigned long[]
+ fillRectangle(x : int, y : int, width : int, height : int, colour : int) : void
+ drawString(x : int, y : int, msg : string) : void
0...1
0...n
Pixel
- blind : bool
- color : string
- currentLevel : int
- blockNumber : int
+ TurnBlind() : void
+ NotBlind() : void
+ GetColor() : string
+ SetColor(colorName : string) : void
+ setLevel(level : int) : void
+ getLevel() : int
+ setNumber(num : int) : void
+ getNumber() : int
+ operator<<(out : ostream&, c : const Pixel&) : ostream&
0...1
Level
+ createBlock(name : char) : Block *
+ generateBlock() : Block *
Level0
- index : int
- sequence : vector<char>
+ generateBlock(): Block
Level1
+ generateBlock(): Block*
Level2
+ generateBlock(): Block*
Level3
+ generateBlock(): Block*
Level4
+ generateBlock(): Block*