The Game of Biquadris

CS246 Assignment5 group project

Shaojun Chen

Wilson Wan

Yaqi Zha

Introduction:

Our group will design the project Biquadris. This game requires two players, two boards and several seven different types of blocks. Players place blocks on the board and when a row of the board is filled, the row is cleared, and the player gets the corresponding scores. If a player is unable to place a block on the board, the game ends.

Overview:

In our design, we build out a Board class through three classes, Block, Level, and Scoring. We can control two different boards through the Boardctrl class and implement different commands which we read from main. At the same time, the Boardctrl can also show the graphic display and text display for the two boards after each command.

Design:

- **Class – Block:**

In this part, we use the inheritance relationship. The Block class is an abstract class inherited by 8 concrete derived classes, I, J, L, O, S, Z, T and a block star. Block star is a block dropped in the middle of board in level 4 when player did not clear at least one row in every 5 blocks.

The block stores its current position, block type, level when a block is created, heavy value, etc. Block has methods that implement move left, right, down, drop, rotate clockwise and rotate counterclockwise for different blocks. The board will clear the cells in the old position and update the cells in the new position according to the different motions.

- **struct – Pos:**

Every cell in the 2D array of a board is a struct Pos. The fields x and y indicate the current position of the cell. The field c and cover indicate the content in each cell. If blind effect is triggered, the cover will become '?' and show as '?' in the display of

the board. If blind effect is disabled, there is no cover and thus the content c will be print.

In addition, the blockSqNum indicate which block that the content c is belong to.

- **Level:**

In this part, we use the inheritance relationship. The abstract Level class inherited by 5 concrete level classes**.** The Level class records the level in the field **curlevel** and create the next block according to the current level using **makeBlock**.

The below is a brief of the functions in class Level:

1. **+ readFile:** read the sequence of block from input and store them into an array, that is the field **blocks**.
2. **+ setFilename:** set the field **filename** as the the name input.
3. **+ setrandom:** change the state of random (the field **israndom**) to be true or false.
4. **+ getFilename** and **getcurrlevel**: getter functions for filename and currlevel.

There are three virtual methods:

+ levelUp levelDown: to increase or decrease the level in range from 0 to 4

+ makeBlock: the core function in this class. It creates a new block according to the current level, the array of block and the state of randomness.

- **Scoring:**

The class Scoring has three fields. The field **curr** represent the current score that the player has. The field **hi** represent the highest score that the player gets. If **curr** is bigger than **hi**, then hi will be update to be the value of **curr**. The field level represent the current level of the player.

The below is a brief of the functions in class Scoring:

1. **+ clearRow(int num):** consume a parameter num, which represent the number of rows reduced, to update the current score.

2. **+ clearBlock:** consume a parameter blockLevel, which represent the level of the block reduced, to update the current score.
3. **+ restart:** set the current score into zero.
4. **+ getcurrscore and gethisscore:** getter function for the fields curr and hi.
5. **+ sethiscore(int i):** set the field **hi** to be equal to **i**.
6. **+ addscore(int i):** add **i** on the current score (**curr**)

- **Boardview:**

The class Boardview takes responsibility of displaying text and graphic to clients. The class BoardCtrl is composited of one BoardView so that we can control the display of our game in BoardCtrl. In this class, we have two fields. One of them is a Boolean value called **textOnly** which means that if we display only the text to clients. Another field is a unique pointer called **xw** points to a Xwindow object which we will use it to support our graph display. Thus, this class is like a viewer or observer so that we can look into our game.

We will briefly introduce the methods we have in this class as follows.

1. **+ displayText:** This is a method to display the text proportion of our game. It takes a shared pointer which points to BoardCtrl as a parameter so that we can get the information we want, like score, board and so on. The main reason I didn't implement this with the method which displays graphic together is that sometimes we need to display text only. So I wrote this independently.

2. **+ initialGp:** This is a method to initial the graphic of the game. When we start the game, we will have two empty boards with scoring and level. So in this function, we just draw a general board with two boards and draw some strings with important information.

3. **+ isTextOnly:** This is a trivial method which does exactly like its name. That is, it determines if the current mode is textonly. If true, we display the text part only. We display both graphic and text parts otherwise.

4. **+ fillCell:** This is a core function of this class because we fill each cell for the block we have by using this method. This function takes some parameters,

including which board we currently at, the row and column, the width and height and the colour we intend to assign to the cell.

5. **+ fillGp:** This is another core function which we can fill our graphic with some important information like score and level and update them. Moreover, this method calls fillCell we mentioned above. So it is the general function which we can control our graphic display.

6. **+ displayNext:** This is a specific method to display next block in our graphic. It takes two parameters whichBoard and type. The whichBoard is a parameter which we know the board which the next block belongs to. And it's clear that type is the type of next block.

7. **+ display:** This is a method to control both text and graphic display in this class. So it simply call the methods which display text and graphic.

8. **+ end:** This is a method to end the game. And we display the winner in this function. When this method is called, it normally calculates the highest scores of both players and the person who has a higher score wins the game. However, it takes a special int parameter called whoLose, if whoLose is 1 or 2, then player 1 or player 2 lose the game by some reasons. Otherwise, the function will do what I've mentioned above.

- **Class – Board and BoardCtrl:**

The relation between the class Board and class BoardCtrl is composition. There are two boards in BoardCtrl indicated that there are two players. The fields contain in the board are the **2D array** of cells with size of 18*11, the **current block** the player is controlling, the **next block**, the **current level**, the **score**, and the **array of BlockInfo** that record all the block in information of all blocks. The are some getter functions for the BoardCtrl to get these values. When BoardCtrl get these values from each Board, it can control the board. Also, there are some helper functions in class Board to help BoardCtrl better control the board.

The below is a brief of the helper command in class Board:

1. **+ clearRow and findBlock:** clearRow find out how many rows was clear when ends turns and findBlock check whether a block is completely cleared. Both functions may update the score.
2. **+ placeIn, placeOut, checkPlace:** these are help function for moving or placing a block in the board. For each time we move a block, we first place out the block, and then move it to the direction we want. Then, we need to check whether it is valid to place in the new position. If not, turn back and place back. If yes, place the block in the new position.
3. **+ blind, unblind, heavy, force:** the implement of the special action. When special action is triggered in class BoardCtrl, these functions will help update the special effect of the board.

The below is a brief of the control command in class BoardCtrl:

1. **+ left, right, down, drop:** these functions are to move the current block in different directions. When a player drops the block or reach end of the current in some case, it will turn to the next player (the other board).
2. **+ rotateCW** and **rotateAW:** there functions are to rotate the current block into different directions clockwise or anticlockwise.
3. **+ levelUp** and **levelDown**: to increase or decrease the level of the board in range from 0 to 4.
4. **+ setInPut:** read a sequence of command from input and then execute them.
5. **+ changerandom:** command to set the set or unset random property of the board relative to level 3 and level 4.
6. **+ force:** force to replace the current block by other block.
7. **+ Clear_and_calScore:** clean the score when end turn, and then update the score.
8. **+ whoWin:** Decide who win the game

Resilience to Change:

In order to support the possibility of various changes to the program specification, we implemented the base classes as abstract classes. We also use the composition relationship to associate several important classes together. For example, Block class and Level class are both abstract classes, if we need to increase or decrease the number of blocks and levels, we just need to add a derived class and write the override function in a derived class. Besides, we can add the board's smart pointer in Boardctrl to add more players, so that more people can play the game at the same

time and increase the competitiveness and fun of the game. So, we can achieve high cohesion.

Almost all of our commands are implemented in the Boardctrl class, and the remaining classes have very little effect on each other. Assume, we need to add a new command, we only need to write the method function of the command in Boardctrl class and add the string of the command in main function. Other classes will not be affected. Thus, we achieve low coupling.

Answers to Questions:

Question1: 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?

In the block class, we can add an integer field and a Boolean field. The integer field counts how many rounds this cell has appeared on the board. When the count is greater than 10, the Boolean field is true, otherwise it is false. After dropping a new block, we will check the boolean values of all cells and clear the cells with a Boolean value of true. This generation of such blocks be easily confined to more advanced levels, as this cannot limited by the certain level and we can it will be easily connected to similar rules in different level.

Question2: How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?

To accommodate the possibility of introducing additional levels into the system with minimum recompilation, we can use the factory method pattern. We can create an abstract class called Level and add each level as a subclass which is concrete. For example, if we want to add another level 5, we can add a concrete subclass, level 5, and we only need to compile class level 5.

Question3: 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?

One method is to use the decorator design pattern to connect multiple effects. First we create an abstract and general class of all effects. Then we create classes for each effect under the abstract class so that they won't affect each other if they are applied at the same time. If we invented more kinds of effects, then we simply create more subclasses under the general class. This exactly prevents the program from having one else-branch for every possible combination.

Question4: 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.

We will have a specific class called BoardCtrl to manage and own all command methods in Board. If we add and change some of methods, we will have minimal changes to source and minimal recompilation. It can be annoying, but we can adapt our system to support a command whereby a user could rename existing commands by adding some if statements. We can set some specific methods to call the combination of some commands and name the methods.

Extra Credit Features:

- Memory management:

  In all classes, there is no explicit memory management. All memory management is done through shared pointers and vectors and there is no memory leak.

- Player's Handbook:

The player's manual can be obtained before the game starts by using the "-help" command. The manual includes an introduction to the different commands and different levels and the usage of special actions.

Final Questions:

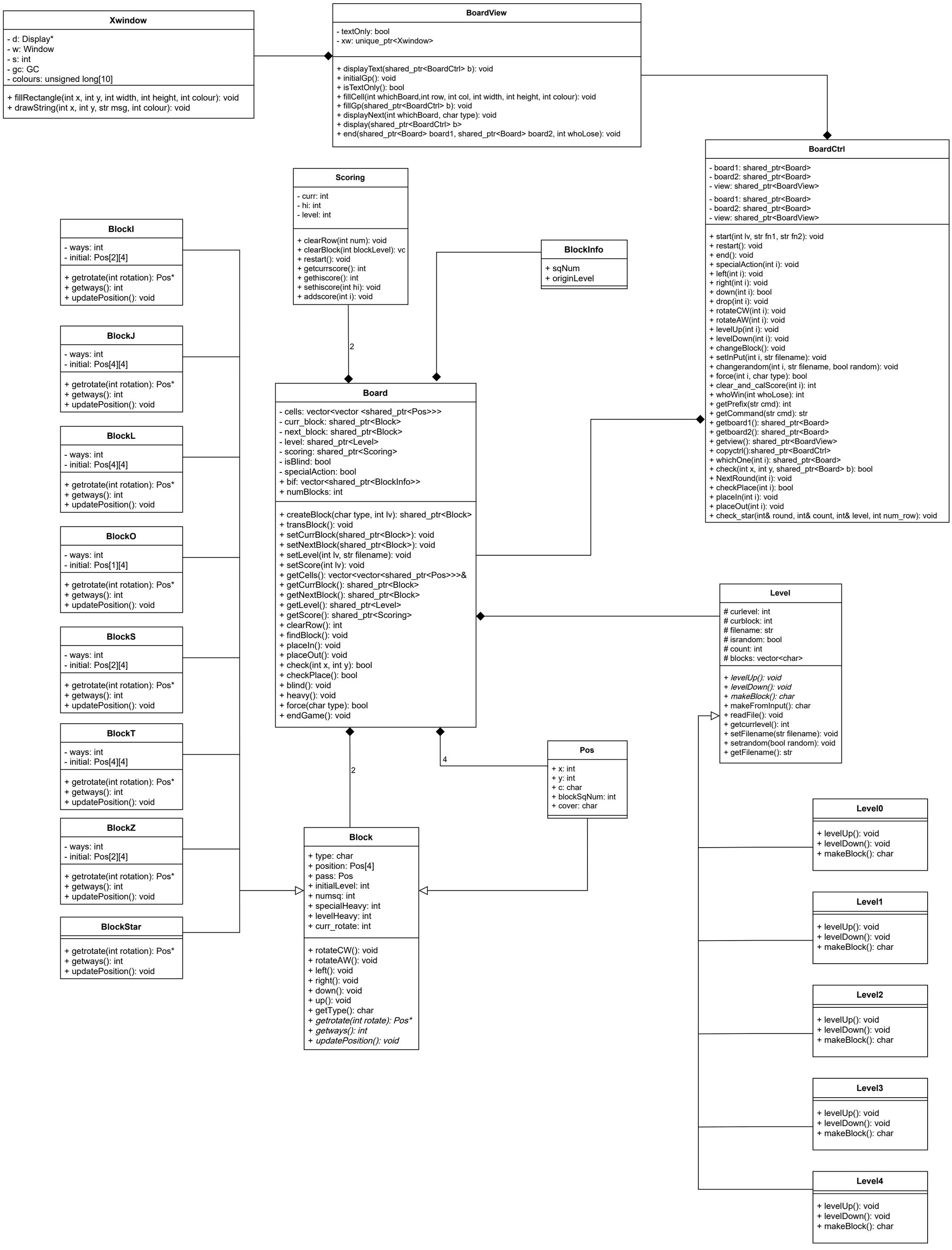
Question1: What lessons did this project teach you about developing software in teams?

This teamwork made us realize the importance of planning and uml diagram. The advantage of teamwork is that we can discuss our ideas with others and get an uml diagram quickly during the discussion. At the same time, planning can help us manage our time better and set aside time for debug and testing.

In addition, the name of each function and the brief documentation are very important. Because we were working as a team, there were bound to be times when we needed to use others' functions. Without a clear documentation representation, it would take some time to read the whole code and understand the usage, which usually takes time. On the other hand, a clear and simple documentation can avoid this situation and quickly get to compiling other functions or classes. The exception is when debugging, where we must be familiar with the usage of each function.

Question2: What would you have done differently if you had the chance to start over?

We will definitely spend more time on the program. If there was more time spent on improving a more perfect uml diagram, there would be much less time spent on writing code.


Xwindow
- d: Display*
- w: Window
- s: int
- gc: GC
- colours: unsigned long[10]
+ fillRectangle(int x, int y, int width, int height, int colour): void
+ drawString(int x, int y, str msg, int colour): void
BoardView
- textOnly: bool
- xw: unique_ptr<Xwindow>
+ displayText(shared_ptr<BoardCtrl> b): void
+ initialGp(): void
+ isTextOnly(): bool
+ fillCell(int whichBoard,int row, int col, int width, int height, int colour): void
+ fillGp(shared_ptr<BoardCtrl> b): void
+ displayNext(int whichBoard, char type): void
+ display(shared_ptr<BoardCtrl> b>
+ end(shared_ptr<Board> board1, shared_ptr<Board> board2, int whoLose): void
BoardCtrl
- board1: shared_ptr<Board>
- board2: shared_ptr<Board>
- view: shared_ptr<BoardView>
- board1: shared_ptr<Board>
- board2: shared_ptr<Board>
- view: shared_ptr<BoardView>
+ start(int lv, str fn1, str fn2): void
+ restart(): void
+ end(): void
+ specialAction(int i): void
+ left(int i): void
+ right(int i): void
+ down(int i): bool
+ drop(int i): void
+ rotateCW(int i): void
+ rotateAW(int i): void
+ levelUp(int i): void
+ levelDown(int i): void
+ changeBlock(): void
+ setInPut(int i, str filename): void
+ changerandom(int i, str filename, bool random): void
+ force(int i, char type): bool
+ clear_and_calScore(int i): int
+ whoWin(int whoLose): int
+ getPrefix(str cmd): int
+ getCommand(str cmd): str
+ getboard1(): shared_ptr<Board>
+ getboard2(): shared_ptr<Board>
+ getview(): shared_ptr<BoardView>
+ copyctrl():shared_ptr<BoardCtrl>
+ whichOne(int i): shared_ptr<Board>
+ check(int x, int y, shared_ptr<Board> b): bool
+ NextRound(int i): void
+ checkPlace(int i): bool
+ placeIn(int i): void
+ placeOut(int i): void
+ check_star(int& round, int& count, int& level, int num_row): void
Scoring
- curr: int
- hi: int
- level: int
+ clearRow(int num): void
+ clearBlock(int blockLevel): vc
+ restart(): void
+ getcurrscore(): int
+ gethiscore(): int
+ sethiscore(int hi): void
+ addscore(int i): void
2
BlockInfo
+ sqNum
+ originLevel
BlockI
- ways: int
- initial: Pos[2][4]
+ getrotate(int rotation): Pos*
+ getways(): int
+ updatePosition(): void
BlockJ
- ways: int
- initial: Pos[4][4]
+ getrotate(int rotation): Pos*
+ getways(): int
+ updatePosition(): void
BlockL
- ways: int
- initial: Pos[4][4]
+ getrotate(int rotation): Pos*
+ getways(): int
+ updatePosition(): void
BlockO
- ways: int
- initial: Pos[1][4]
+ getrotate(int rotation): Pos*
+ getways(): int
+ updatePosition(): void
BlockS
- ways: int
- initial: Pos[2][4]
+ getrotate(int rotation): Pos*
+ getways(): int
+ updatePosition(): void
BlockT
- ways: int
- initial: Pos[4][4]
+ getrotate(int rotation): Pos*
+ getways(): int
+ updatePosition(): void
BlockZ
- ways: int
- initial: Pos[2][4]
+ getrotate(int rotation): Pos*
+ getways(): int
+ updatePosition(): void
BlockStar
+ getrotate(int rotation): Pos*
+ getways(): int
+ updatePosition(): void
Board
- cells: vector<vector <shared_ptr<Pos>>>
- curr_block: shared_ptr<Block>
- next_block: shared_ptr<Block>
- level: shared_ptr<Level>
- scoring: shared_ptr<Scoring>
- isBlind: bool
- specialAction: bool
+ bif: vector<shared_ptr<BlockInfo>>
+ numBlocks: int
+ createBlock(char type, int lv): shared_ptr<Block>
+ transBlock(): void
+ setCurrBlock(shared_ptr<Block>): void
+ setNextBlock(shared_ptr<Block>): void
+ setLevel(int lv, str filename): void
+ setScore(int lv): void
+ getCells(): vector<vector<shared_ptr<Pos>>>&
+ getCurrBlock(): shared_ptr<Block>
+ getNextBlock(): shared_ptr<Block>
+ getLevel(): shared_ptr<Level>
+ getScore(): shared_ptr<Scoring>
+ clearRow(): int
+ findBlock(): void
+ placeIn(): void
+ placeOut(): void
+ check(int x, int y): bool
+ checkPlace(): bool
+ blind(): void
+ heavy(): void
+ force(char type): bool
+ endGame(): void
2
4
Pos
+ x: int
+ y: int
+ c: char
+ blockSqNum: int
+ cover: char
Block
+ type: char
+ position: Pos[4]
+ pass: Pos
+ initialLevel: int
+ numsq: int
+ specialHeavy: int
+ levelHeavy: int
+ curr_rotate: int
+ rotateCW(): void
+ rotateAW(): void
+ left(): void
+ right(): void
+ down(): void
+ up(): void
+ getType(): char
+ getrotate(int rotate): Pos*
+ getways(): int
+ updatePosition(): void
Level
curlevel: int
curblock: int
filename: str
israndom: bool
count: int
blocks: vector<char>
+ levelUp(): void
+ levelDown(): void
+ makeBlock(): char
+ makeFromInput(): char
+ readFile(): void
+ getcurrlevel(): int
+ setFilename(str filename): void
+ setrandom(bool random): void
+ getFilename(): str
Level0
+ levelUp(): void
+ levelDown(): void
+ makeBlock(): char
Level1
+ levelUp(): void
+ levelDown(): void
+ makeBlock(): char
Level2
+ levelUp(): void
+ levelDown(): void
+ makeBlock(): char
Level3
+ levelUp(): void
+ levelDown(): void
+ makeBlock(): char
Level4
+ levelUp(): void
+ levelDown(): void
+ makeBlock(): char