The Game of Biquadris

11.30.2018

—

Jiadong Mai - j4mai
Yijian Wang - y2549wan
Yinyao Zhu - y375zhu

CS246 - Fall 2018
Assignment 5

Introduction

The game of Biquadris consists of two players and seven different types of blocks (plus two additional bonus blocks in level 5). Whenever a line of cells disappear, a player earns scores. Whenever a new block cannot be properly initialized, the player's game is over. The player will have the choice to either start a new game, or quit the game when a player's game is over. Details of the game are outlined in the assignment.

Overview

This game has five different classes: Player, Cell, Board, Level and Block. Note that there is no explicit memory management in any class. All memory management is done through shared pointers, unique pointers and vectors. This report will explain the functionalities and dependencies of each class and their subclasses. A detailed structure of the game is outlined below.

Design

Class: Player

This class is designed as a structure (or a class with only public fields). The purpose of this class is to provide the necessary information of the current player that the Block, Board, Level and Cell need when making a change in the game. This class owns a 2-D vector of Cells, two Blocks, and a Level. Note that most of the following fields are initialized through the BasicBoard method called **SetPlayer**(This will be explained later in the Board class). This class has no methods.

- **PlayField** is the gameboard of the current player. It is two-dimension vector of Cells.
- **name** is the current name of the player.
- **id** is used to determine the correct level of the current player.
- **Score** records the score of the corresponding player.
- **level** is the current level of the player.
- **col** and **row** determines a correct spot which the block will be printed on the board.
- **LeftPossible, RightPossible, DownPossible,cwPossible, ccwPossible** are boolean values determining whether the action left, *right, down, clockwise or counterclockwise* are valid moves.
- **cant_down** is a boolean value for the action *drop*. It determines whether a *down* movement is possible.
- **levptr** is a pointer to a level that generates the correct blocks.

- **SpeAttack** counts the number of blocks that has been dropped on the board for a player. This only counts in level 4 and level 5.
- **IBlock, JBlock, LBlock, OBlock, TBlock, SBlock,ZBlock** are integer values used to calculate scores when a entire block disappear. However, at the end, we found it infeasible to accomplish this goal through the initial design of our game. These values are used and updated when a line of Cells disappear. We did not include the XBlock and CBlock in this section to increase the difficult of the game in level 5.
- **heavy** is the boolean value for whether there needs to be a heavy special action.
-
- **reverse** is the boolean value for whether there needs to be a reverse special action. Note this is a bonus feature and need to be turned on mannually.
- **staron** determines whether there needs to be a StarBlock appearing on the board.
- **high** keeps track of the high score of the current player.
- **CurBlock** is the current block of the player, a shared pointer.
- **NextBlock** is the next block of the player, a shared pointer.

Class: Cell

This class represents each Cell in the PlayField (mentioned above). It has several characteristics that are represented as boolean and strings, which are used for determining whether some of the actions are possible (right, left, etc.). Details are outlined below.

- **isAlive** is a bool to determine whether a Cell is live or dead.
- **blind** is a bool to determine whether a Cell need to change their color into "?" color
- **color** is a string to show the Cell's color
- **x and y** is to determine the position of the Cell in Xwindow function
- **width and height** is to determine the size of the Cell in Xwindow function

For the function sectors:

+ **TurnOn()** is to active the Cell and change the state of Cell into live, which is true.
+ **TurnOff()** is to close the Cell and change the state of Cell into dead, which is false.
+ **TurnBlind()** is to turn the blind filed of Cell into true.
+ **NotBlind()** is to turn the blind filed of Cell into false.
+ **GetColor()** is to get the color of the Cell
+ **IsAlive()** is going to return whether a Cell is live or dead.
+ **draw()** is to draw the Cell on the Xwindow board
+ **Operator<<** is to output the cell's color if the cell's blind is not true, otherwise, output "?" to instead the cell's color

Class: Block

Block class is an abstract class. It has ten different blocks that are used to be reflected on the PlayField. Whenever a block is rotated, the Block in this class will be modified first, and then get updated on the board.

- **v** is the main block of its type. It is a 2-D vector of strings of the size four by five. Most of the strings in this vector are space, except the corresponding positions where a Cell (this will be more comprehensible when the DrawBoard is discussed later) should be alive. For example, if the Block is a T Block, the vector will look like this:

```
v = {
  {" ", " ", " ", " "},
  {" ", " ", " ", " "},
  {" ", " ", " ", " "},
  {"T", "T", "T", " "},
  {" ", "T", " ", " "},
};
```

- **CW** is the virtual method that is used to make a clockwise change for the 2-D vector. It keeps track of the *rotate_state(included in each individual blocks)* of the Block and modify the vector accordingly. For example, after the CW change, the TBlock above would look like this:

```
v = {
  {" ", " ", " ", " "},
  {" ", " ", " ", " "},
  {" ", "T", " ", " "},
  {"T", "T", " ", " "},
  {" ", "T", " ", " "},
};
```

- **CCW** is the virtual method that is used to make a counterclockwise change for the 2-D vector. It can be achived by calling CW three time. Note for some Blocks such as the OBlock or the XBlock, it is not necessary to make any changes, thus the CW and CCW methods for certain blocks are correspondingly simplified. A TBlock, after a CCW change from the initial position, will be modified this way:

```
v = {
  {" ", " ", " ", " "},
  {" ", " ", " ", " "},
  {"T", " ", " ", " "},
  {"T", "T", " ", " "},
  {"T", " ", " ", " "},
};
```

- **print** is a virtual method that prints the block.
- **getBlock** is a virtual method that returns the current block as a 2-D vector.

- clone makes a deep copy of the current block, and return the copied object as a new Block. This function essentially serves as a Factory. Note this function is for the solely use of Aclone. Note the copy assignment operator and the copy constructor are modified in order to make a deep copy.
- **Aclone** is a wrapper for clone. This method return a unique pointer that contains the block returned from clone. This way, the newly generated Block from clone will be handled through smart pointers instead of separate memory management.
- **GetType** is a virtual method that returns the BlockType of the current Block. This is used to determined the colours of each block when printing.

The subclasses of Block are**: IBlock, JBlock, OBlock, TBlock, SBlock, ZBlock, LBlock, StarBlock, XBlock and CBlock.** Rather than the required 7 blocks and the StarBlock that is used in level 4 and 5, two additional blocks, XBlock and CBlock are added as extra features in level 5. All subclasses override each pure virtual functions in Block, and has three private fields: BlockType, v, and rotate_state. The uses of these private fields have been discussed above.

Class: Board

The Board class has two pure virtual functions: *DrawBoard* and *Recalculate.* This class serves as a general idea of what a board mainly does, which is printing based on different situations. The subclass, Baseboard, is the class that contains the main game logic. This class will discuss in depth in this section.

Subclass: Baseboard

Private fields:

- **Player1** and **Player2** are two shared pointers to two players, each contains distinct information regarding these two players, including the player's individual PlayFields. Note the two PlayFields of the players are the main game display of the Baseboard. The board owns these two players.

Public methods and fields:

+ **IsText** is a boolean value determining whether the game should start with text mode or a graphic mode. This value can be publicly accessed and modified based on the command line options.

+ **SetPlayer**, as discussed in the player class, sets the two players to their initial state.

+ **GetPlayer1** and **GetPlayer2** return the corresponding player shared_pointer. This is used to determine when they player should take turns.

+ **GetCurScore** returns the current score of the current player. It consume a parameter which is a share_pointer of a player.

+ **DrawBoard** is the main game display. For text-only mode, it prints the level, score, high score, board and the next block of each individual player, it has three parts. First part correctly

formats where the level, score, and high score should appear on the BasicBoard. The second part parallel prints the two PlayFields. The content of each individual cells determines what should be printed. Initially, if no other methods are called in main, and the players have no current or next block, the method will only prints two boards of "_" (Note this will never happen as we set the players before the client can ever start the game). Once a cell's content is modified, the accurate content will be printed. If the cell has a color "T", the letter "T" will be printed instead of "_". If the cell is blind, a "?" will be printed. If the cell has a color of "*", a "*" will be printed. This method also integrates the graphic component of the game by using the IsText field. If it is true, then only the text version of the game will be displayed, otherwise the graphic version will take place. The last part of the game will display the NextBlock of the player. This is achieved by calling DrawNextBlock, which will be discussed shortly.

+ **DrawBlock** is the main signal of when a cell is turned on (or alive), or turned off (or dead). Every time a block is at the bottom or when a down move is not possible, the corresponding cells are turned off, and the color of that cell is set to the color of the block. Initially, the block will be reflected on the upper left of the board (as explained in the assignment) if that spot is available. Until a move is not possible anymore, the cell that the block translate on the board is alive and colored as the BlockType of the block.

+ **DrawNextBlock** parallel prints two blocks simultaneously. These two blocks are the NextBlock of the players. This method also integrates the graphic parts by explicitly calling fillRectangles from the Xwindow library. In text mode, blocks are printed in a preformatted way.

+ **CellsAva** determines whether a cell on the board is available. If it is not, then the method DrawBlock will not turn those cells to alive or color them differently. It will simply do nothing. And if that happens for the first appearance of the new block, the game is over. This will be more explained in GameOver.

+ **turn** is a boolean used to determine which player should take the turn. If it is true, then it should be player1's turn, vice versa.

+ **NowPlayer** returns a shared pointer that points to the current player of the game who is taking the turn to play.

+ **ChangeTurn** changes the turn from one player to another by changing the turn boolean.

+ **OtherPlayer** returns a shared pointer that points to the player who is not currently playing.

+ **left, right, down, drop** are methods that changes the initial position of where the block should be printed on the board. For example, if a block can move to the left, the player's col value would be decremented by one so that when the board is printed, it will print starting from the current col-1 position. Whether these movements are possible are determined through the player's fields - leftPossible, rightPossible etc. These fields are updated in the update method.

+ **cwpossible** and **ccwpossible** are treated separately from left, right and down because there are more tricky edge cases to consider. Every time the client calls for cw rotation or ccw rotation, a copy of the current block is generated with Aclone, a method in Block (for more details in Block). The copy is subsequently rotated and the board tries to print the copy on the PlayField.

However, if after rotation, the copy cannot be properly printed, the corresponding boolean value of the player (i.e. cwPossible and ccwPossible) are turned to false, and a cw or ccw instruction will have no effect.

+ **SetPossibles** sets both players boolean indicators to true, except the cw and ccw booleans which are treated separately as discussed above. This is necessary as if not done so, a block may have incorrect indicators that prevent the player from making a legal move.

+ **update** method makes the update to all move indicators of the current player, i.e. LeftPossible, RightPossible, DownPossible, cwPossible and ccwPossible. Again, cwPossbile and ccwPossible are treated separately by calling cwpossible and ccwpossible. A call to update is necessary every time a move is made to make sure that the possibility of player's next move.

+ **Recalculate** calculate how many lines will disappear every time a player's turn is over, and tells ClearLine that information (vector of integer).

+ **ClearLine** clears lines when all cells in a row are dead and none of their color is space. It updates the score and the high score of the current player based on the rules explained in the assignment. As mentioned above, we were unable to update the score when a block completely disappears (i.e. if a OBlock that was generated before is now gone), but the scores works properly for the first part of the scoring rules. A new board will be generated to replace the previous one with the updated cells' states.

+ **LevelUp** and **LevelDown** increase or decrease the current player's current level by the amount specified by the client.

+ **KillLiveCell** is called every time after the current player makes a move. This ensures that the board is displayed correctly after the update.

+ **LevelHeavy** implements the heavy special action for level as described in assignment.

+ **BlockHeavy** implements the heavy special action every time a player decides to use heavy on the other player when they clear two or more lines at a time.

+ **Blind** is called when blind special action is triggered by a player. It turns an area of the board's cells to Blind, and update the cell's state.

+ **GameOver** decides when a game is over and who has lost the game. A game is over when the NextBlock of that player can not be properly displayed on the board.

+ **Restart** restarts the game without changing the high score of both players. This is done through resetting both players.

Class: Level

This class is designed using the Factory Method design pattern, which is used to create and store the sequence of the Blocks that should be presented in the required Level.

<u>Abstract Level Class:</u>

Methods in Public fields:

+ **GetLevel** GetLevel() function is to return level_num, which will identify the current level and determine whether a special action will be triggered.

+ **GetBlock** GetBlock will be overridden by other subclass and return the required Block share pointer. Every Level will have two vector section to holds a random sequence and a file input sequence of the block sequences. It depends on which level it is.

+ **addBlockSeq** *AddBlockSeq* and *randomBack* are the functions that would be useful for Level 3 and Level 4. In those levels, command *norandom file* and *random* will pass a sequence of Block names to the level. Is order to store that, Level 3 and Level 4 each has a vector of strings in their private field.

+ **randomBack** randomBack has the opposite functionality of addBlockSeq, it will changes Level3 and Level4 backs to random block sequences.

Subclass:

Private fields:

- **std::vector<std::string> seqList**
- **int seqCount**
- **std::vector<int> seqList_int**
- **int seqCount_int**

Each private level object will contains two sets of vectors that represent block sequences. Each set is for one single players.Since rand function is pseudo-random, so if we generate a random block in the ***GetBlock*** method, every time will pass the same block. In order to make this game more reasonable, when the level initialized, a sequence of five random number will be generate by the global seed number. the seed number can be given by -seed + num.

Every vector array comes with a ***seqCount*** integer, which holds which Block number was counting at, it reset back to zero when one around run over.
In our program, since random gives a pre random sequence, we have an extra feature that makes the ***GetBlock*** method completely random everytime. This feature can be turned on by command line -seedfree.

- **bool random** Random is the boolean to identify which vector set will be pass to the ***GetBlock*** method.

Main function

In our main function, Command-line Interface initialized the status that this game will be present:

- whether or not graphic will be turn on(-text)
- passing the seed number(-seed num)
- initial block sequence of level0(-scriptfile1,-scriptfile2)
- starting the game in different level(-startlevel n)

- additional commands:

  - turning on the sound feature(-sound)
  - Easy mode (-easy)

after setting things up, main function will generate a BasicBoard, two players , and the relating blocks (based on the selected level) to the start the game.

Command Interpreter:

There are two kinds of command that could be input, one is single command keyword, another is multiple command keyword.
We check whether this command has number in front of it, if so, we will store the number and a multiple of same commands will be executed.
We place the command interpreter in the alphabetical order to that it is easier to read and change.
The logical behind the command interpreter is to compare the existing command until the char what distinguish them. Also, in a case of a typo, the program will keep running, and waiting for the next valid command.
Easy commands(shortkeys) are separated from the alphabetical order list and be wrapped by a boolean condition.
At the end of every turn, the special punishment in level 4 will be checked. After that, board will update and wait for the player's next move.

An easy mode of the game is added as a bonus feature to the game, which significantly shortened the amount of letters a player must enter. This is explained in more detail in the Bonus Feature section below.

Updated UML

Updated UML is attached as an appendix at the back.

Resilience to Change

As discussed in the design of the game, each of the five classes serves different functionalities. A change to one of the classes will have minimal changes to the others. For instance, we added additional blocks to the game: XBlock and CBlock. Since the block is an abstract class, and each subclass has its own overridden functions, the process of adding those blocks become fairly simple. We added a corresponding .h and .cc file for each of the two blocks, which are 90% similar to what other blocks look like. The way the board handles the blocks in our design significantly decreased the effort we need to put in if we want to make a change to happen. In this case, each of the block is treated separately, with low to zero coupling.
In addition to that, the way we handled the players are fairly efficient as well. For instance, if we were ever asked to add a third to a fourth player, we can simply modify the BasicBoard class only to achieve what we want. A minimal effort will be distributed to the display of the game as well, but it would have been a disaster if we did not have player as a separate class, or in this case, a structure.
Overall, the design of this game achieved low coupling and high cohesion.

Assignment Questions

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?

If we want to make some blocks disappear from the screen they are not cleared under some conditions, we can add a Boolean field and a counter to the Cell class, and modify the counter and the Boolean each time a block is dropped. If some group of blocks are not cleared before the condition, and the levels are not advance enough, Turnoff() will be called to those cells, and the update of the board will make them disappear from the PlayField. In our design, level is treated as a separate class, each is capable of accomplishing different tasks. If the level is high enough, Trunoff() will not be called. Therefore they can be easily confined if we want to.

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

In fact, we already did this by introducing level 5. That was an effortless task as we treated levels as separate objects. They essentially serve as block factories, and we designed our blocks in a very low-coupling way. Each block is separate from another, therefore they are all easily modifiable and changeable. Levels used the factory method design pattern. They are only related to the players. The recompilation time is therefore minimized.

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?

We can design our system using decorator pattern to achieve this. If new effects are introduced to the system, adding another decorator will easily solve the problem with minimal compilation time. If we have a decorator class that serves as an abstract class, and then have sub-decorators under it, we can add multiple effect on the target object simultaneously.

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? 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?

In order to make command names more flexible, for example, "lef" will have the same effect as "left", we will sort our command line in alphabetical order and decide which command will be run. We will store the commands using string, and compare each character with the commands we wish to have. For example, we will have "left" and "levelup" command all under (string[i] == "l") catalog, then keep comparing until we can distinguish which command the client wish to run. Whenever we need to add or change command names to the program, we will go through our existing alphabetical order list and find the place wherever suits the new command.

To go through a sequence of commands (macro), we could generate an array of shortcuts of command to go through the list of command names. For example, we can run a double loop. The outer loop is used to get the first key work, the other one is used to execute on or more commands.

```
String command;
Vector <string> string_of_commands;
While (Cin >> command){
If (command != one_of_our_commands){
String_of_command.push_back(…);
} // and is a sequence command key word
Else{
String_of_command[0] = command; //e.g. ["drop"]
}
} } }
// we generate our multiple commands array here
For (int i = 0; i < string_of_commands.size(); ++i){
If (...){// we find our single command key word here
// and takes action }}}
```

Extra Credit Features

In addition to the requirements outlined in the assignment, the following extra features are included in the game.

- **Reverse punishment**

    - Additional punishment. This punishment includes a reverse effect on the control of the game. Once a player cleared more than two lines at a time, they have a choice to make a reverse punishment on the opponent player. The opponent player's left and right commands are reversed until the game is over.
- **Sound Effect**
    - This is especially challenging as we needed to learn the SFML library from scratch.
    - By adding the SFML library, we were able to add sound effect to the game.
        - Cover page - exciting music for the start of the game
        - Right, left drop and down - sound effect when making a movement
        - Clearing a line - different but fitting sound effect than movement
- **Extra Blocks and extra level**
    - In addition to the 7 already existing blocks in the game of quadris, we added two additional blocks.
    - XBlock: a block of a cross-shape, aimed to add more difficulty to the game
    - CBlock: a block of C-shape, has four completely different shapes when rotating clockwise and counterclockwise.
    - These blocks are available in level 5, which is an extract level in the game. This level inherits every feature from level 4, with 2 additional blocks aiming to increase the uncertainty and difficulty of the game.
- **Simple Mode**
    - Too many letters to type? No worries. A simple mode of the game is available by adding "-easy" on the command line. The new input of the game will be more familiar with what we would expect: "A" for left, "S" for down, "SS" for drop and "D" for right.
- **Cover page and manual page**
    - By adding a cover page and a rule explanation page for the game, we made the game more friendly and easier to understand. Note this is available for both text mode and graphic mode.

Final Questions

1. What lessons did this project teach you about developing software in teams?

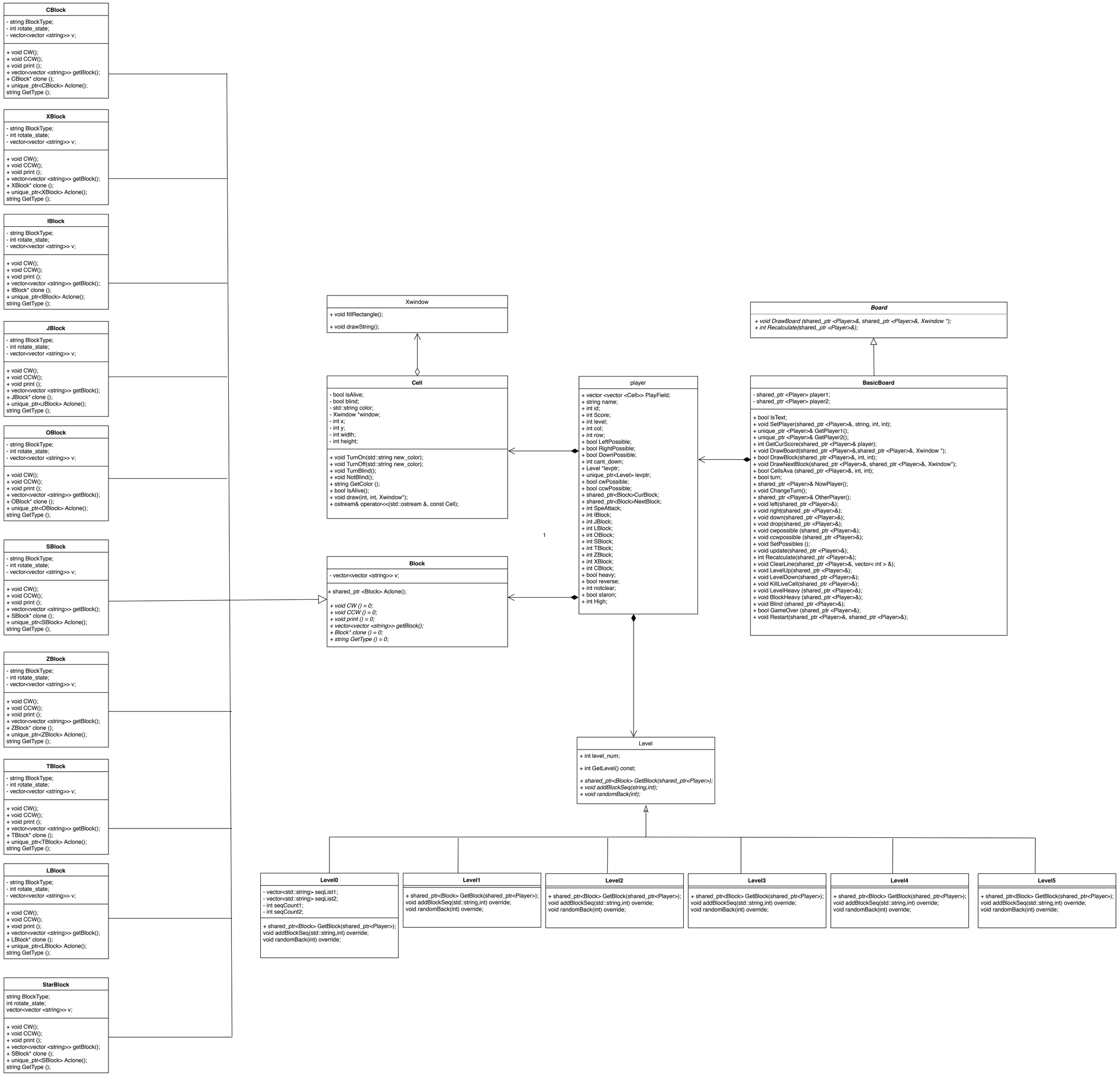
Brainstorming together is better and quicker than thinking alone. Although at the beginning of developing this game, we experienced a hard time, but we still finished it on time with decent quality. The advantage of coding together is the excellent efficiency of debugging. None of us is an excellent tester. Especially, for this long coding program, it is fairly a challenge that one person can catch every bug out of couple of thousands of lines of codes. As a team, we code and debug every day, discuss the obstacles that we encountered, solve the problems through debating, and celebrate the joy together. This is a very enjoying experience for all three of us.

Moreover, this project also taught us about the importance of sharing knowledge. Amy is so much familiar with design patterns. Harry is not an expert on catching up the changes after codes have been modified by others. John tends to overcomplicate simple codes. However, when we work together, we share our advantages to each other and offset our disadvantages. All three of us learnt the importance of team working. None of us could have finished this project without another.

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

If we have a chance to start over, we would use a better way to calculate the score and clear lines. We would place pointers to each cell that points to their related blocks, and deactivate those pointers once the corresponding blocks disappear to add the extra points. Due to the limitation of the ClearLine method that we used, which is print and count, we were unable to calculate the additional score that is awarded to players once a block is completely gone.

The Updated UML is attached on the following page


CBlock
- string BlockType;
- int rotate_state;
- vector<vector <string>> v;
+ void CW();
+ void CCW();
+ void print ();
+ vector<vector <string>> getBlock();
+ CBlock* clone ();
+ unique_ptr<CBlock> Aclone();
string GetType ();
XBlock
- string BlockType;
- int rotate_state;
- vector<vector <string>> v;
+ void CW();
+ void CCW();
+ void print ();
+ vector<vector <string>> getBlock();
+ XBlock* clone ();
+ unique_ptr<XBlock> Aclone();
string GetType ();
IBlock
- string BlockType;
- int rotate_state;
- vector<vector <string>> v;
+ void CW();
+ void CCW();
+ void print ();
+ vector<vector <string>> getBlock();
+ IBlock* clone ();
+ unique_ptr<IBlock> Aclone();
string GetType ();
JBlock
- string BlockType;
- int rotate_state;
- vector<vector <string>> v;
+ void CW();
+ void CCW();
+ void print ();
+ vector<vector <string>> getBlock();
+ JBlock* clone ();
+ unique_ptr<JBlock> Aclone();
string GetType ();
OBlock
- string BlockType;
- int rotate_state;
- vector<vector <string>> v;
+ void CW();
+ void CCW();
+ void print ();
+ vector<vector <string>> getBlock();
+ OBlock* clone ();
+ unique_ptr<OBlock> Aclone();
string GetType ();
SBlock
- string BlockType;
- int rotate_state;
- vector<vector <string>> v;
+ void CW();
+ void CCW();
+ void print ();
+ vector<vector <string>> getBlock();
+ SBlock* clone ();
+ unique_ptr<SBlock> Aclone();
string GetType ();
ZBlock
- string BlockType;
- int rotate_state;
- vector<vector <string>> v;
+ void CW();
+ void CCW();
+ void print ();
+ vector<vector <string>> getBlock();
+ ZBlock* clone ();
+ unique_ptr<ZBlock> Aclone();
string GetType ();
TBlock
- string BlockType;
- int rotate_state;
- vector<vector <string>> v;
+ void CW();
+ void CCW();
+ void print ();
+ vector<vector <string>> getBlock();
+ TBlock* clone ();
+ unique_ptr<TBlock> Aclone();
string GetType ();
LBlock
- string BlockType;
- int rotate_state;
- vector<vector <string>> v;
+ void CW();
+ void CCW();
+ void print ();
+ vector<vector <string>> getBlock();
+ LBlock* clone ();
+ unique_ptr<LBlock> Aclone();
string GetType ();
StarBlock
string BlockType;
int rotate_state;
vector<vector <string>> v;
+ void CW();
+ void CCW();
+ void print ();
+ vector<vector <string>> getBlock();
+ SBlock* clone ();
+ unique_ptr<SBlock> Aclone();
string GetType ();
Xwindow
+ void fillRectangle();
+ void drawString();
Cell
- bool isAlive;
- bool blind;
- std::string color;
- Xwindow *window;
- int x;
- int y;
- int width;
- int height;
+ void TurnOn(std::string new_color);
+ void TurnOff(std::string new_color);
+ void TurnBlind();
+ void NotBlind();
+ string GetColor ();
+ bool IsAlive();
+ void draw(int, int, Xwindow*);
+ ostream& operator<<(std::ostream &, const Cell);
1
Block
- vector<vector <string>> v;
+ shared_ptr <Block> Aclone();
+ void CW () = 0;
+ void CCW () = 0;
+ void print () = 0;
+ vector<vector <string>> getBlock();
+ Block* clone () = 0;
+ string GetType () = 0;
player
+ vector <vector <Cell>> PlayField;
+ string name;
+ int id;
+ int Score;
+ int level;
+ int col;
+ int row;
+ bool LeftPossible;
+ bool RightPossible;
+ bool DownPossible;
+ int cant_down;
+ Level *levptr;
+ unique_ptr<Level> levptr;
+ bool cwPossible;
+ bool ccwPossible;
+ shared_ptr<Block>CurBlock;
+ shared_ptr<Block>NextBlock;
+ int SpeAttack;
+ int IBlock;
+ int JBlock;
+ int LBlock;
+ int OBlock;
+ int SBlock;
+ int TBlock;
+ int ZBlock;
+ int XBlock;
+ int CBlock;
+ bool heavy;
+ bool reverse;
+ int notclear;
+ bool staron;
+ int High;
Board
+ void DrawBoard (shared_ptr <Player>&, shared_ptr <Player>&, Xwindow *);
+ int Recalculate(shared_ptr <Player>&);
BasicBoard
- shared_ptr <Player> player1;
- shared_ptr <Player> player2;
+ bool IsText;
+ void SetPlayer(shared_ptr <Player>&, string, int, int);
+ unique_ptr <Player>& GetPlayer1();
+ unique_ptr <Player>& GetPlayer2();
+ int GetCurScore(shared_ptr <Player>& player);
+ void DrawBoard(shared_ptr <Player>&,shared_ptr <Player>&, Xwindow *);
+ bool DrawBlock(shared_ptr <Player>&, int, int);
+ void DrawNextBlock(shared_ptr <Player>&, shared_ptr <Player>&, Xwindow*);
+ bool CellsAva (shared_ptr <Player>&, int, int);
+ bool turn;
+ shared_ptr <Player>& NowPlayer();
+ void ChangeTurn();
+ shared_ptr <Player>& OtherPlayer();
+ void left(shared_ptr <Player>&);
+ void right(shared_ptr <Player>&);
+ void down(shared_ptr <Player>&);
+ void drop(shared_ptr <Player>&);
+ void cwpossible (shared_ptr <Player>&);
+ void ccwpossible (shared_ptr <Player>&);
+ void SetPossibles ();
+ void update(shared_ptr <Player>&);
+ int Recalculate(shared_ptr <Player>&);
+ void ClearLine(shared_ptr <Player>&, vector< int > &);
+ void LevelUp(shared_ptr <Player>&);
+ void LevelDown(shared_ptr <Player>&);
+ void KillLiveCell(shared_ptr <Player>&);
+ void LevelHeavy (shared_ptr <Player>&);
+ void BlockHeavy (shared_ptr <Player>&);
+ void Blind (shared_ptr <Player>&);
+ bool GameOver (shared_ptr <Player>&);
+ void Restart(shared_ptr <Player>&, shared_ptr <Player>&);
Level
+ int level_num;
+ int GetLevel() const;
+ shared_ptr<Block> GetBlock(shared_ptr<Player>);
+ void addBlockSeq(string,int);
+ void randomBack(int);
Level0
- vector<std::string> seqList1;
- vector<std::string> seqList2;
- int seqCount1;
- int seqCount2;
+ shared_ptr<Block> GetBlock(shared_ptr<Player>);
void addBlockSeq(std::string,int) override;
void randomBack(int) override;
Level1
+ shared_ptr<Block> GetBlock(shared_ptr<Player>);
void addBlockSeq(std::string,int) override;
void randomBack(int) override;
Level2
+ shared_ptr<Block> GetBlock(shared_ptr<Player>);
void addBlockSeq(std::string,int) override;
void randomBack(int) override;
Level3
+ shared_ptr<Block> GetBlock(shared_ptr<Player>);
void addBlockSeq(std::string,int) override;
void randomBack(int) override;
Level4
+ shared_ptr<Block> GetBlock(shared_ptr<Player>);
void addBlockSeq(std::string,int) override;
void randomBack(int) override;
Level5
+ shared_ptr<Block> GetBlock(shared_ptr<Player>);
void addBlockSeq(std::string,int) override;
void randomBack(int) override;