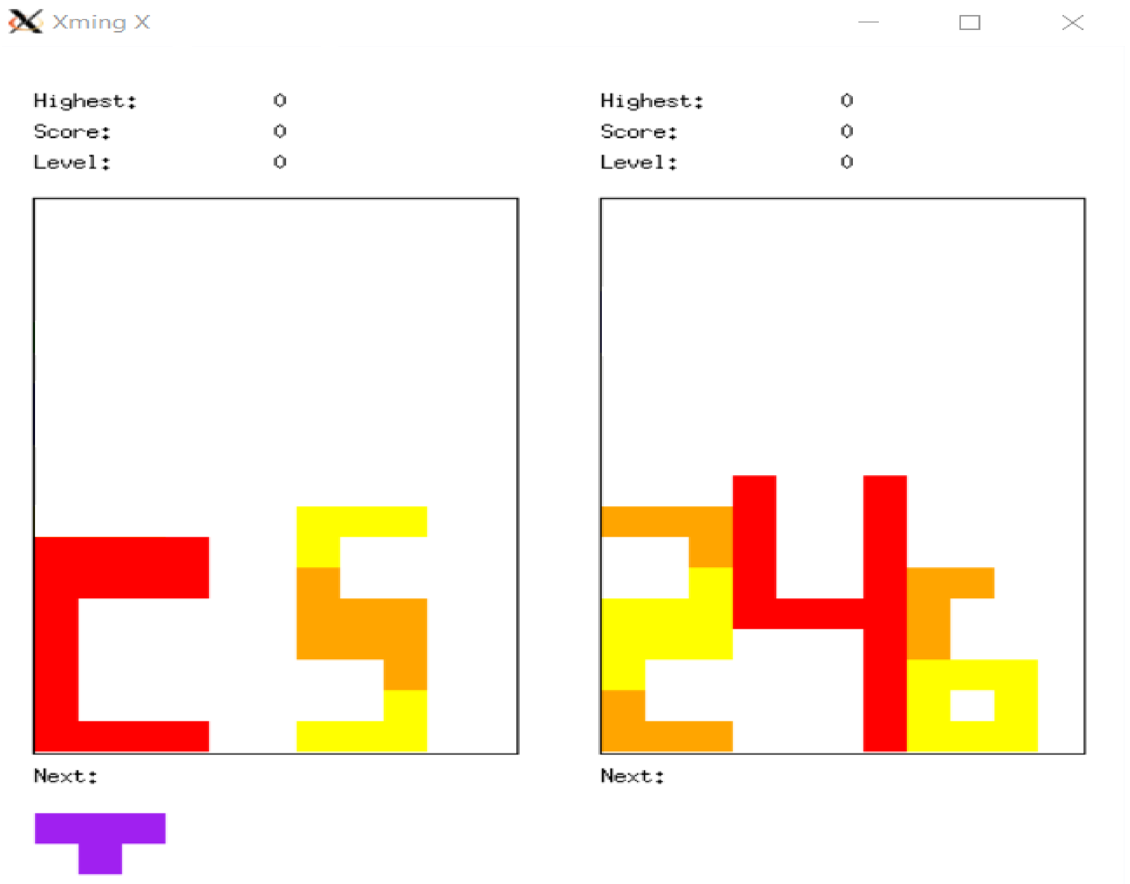


```
Highest:        0        Highest:        0
Score:          0        Score:          0
Level:          0        Level:          0

Next:                    Next:
```

Biquadris

Aug 13, 2021

Chengyue(Melissa) Wu - c334wu
Yicheng(Helen) Zhang - y3453zha
Wenjing(Jane) Lu - w82lu

CS246 - Spring 2021
Final Project

Introduction

Biquadris is a game for two players. Users take turns to control their level and the movement of the block and drop them in the gameboard (11 X 15). The block has seven types and each has a different appearance. When an entire row is filled, this row will disappear and this turn's player earns a score. Special actions will be called when players cancel more than two lines by dropping one block. Players can also choose the level they want to challenge. When a new block cannot be generated, the game is over. Players can choose whether to start a new game or to quit. More details can be consulted by looking at the project's outline.

Overview

The implementation of the game consists of six classes: Board, Player, Cell, Level, Xwindow, and Block. There is no explicit memory management. The usage of smart pointers and vectors is throughout the implementation. Exception handler exists to prevent any error that happens when reading the command line in the main function. The details of each class and their dependencies are delivered in the section below.

Design

Class Player

The class contains both fields and methods and all fields and methods are public. The purpose of this class is to organize all information about one user and manage individually. (Since there are two players in the game.) Changes will happen when each player drops his block. The class has four int, a 2-D vector of Cell pointers, four blocks, five boolean, and one int method. All fields are initialized by the class's constructor. Details will be explained below:

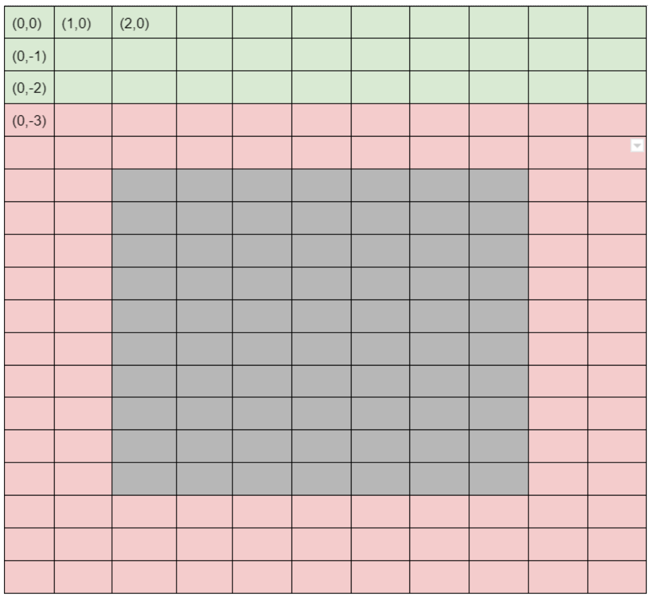
+ **int highestScore**: is the highest score that the player achieves until the program terminates
+ **int Score**: is the score that the player achieves until the game is restarted. When the game is restarted, the current score reverts to 0.
+ **shared_ptr<Block> currentBlock**: is a pointer to the block that is generated and controlled currently
+ **shared_ptr<Block> nextBlock**: is a pointer to the block that is generated and will be controlled in the player's next turn
+ **shared_ptr<Block> currentBlockCpy**: is a pointer to the block that all of its fields have the same value as that of the block which is pointed by the currentBlock.
+ **shared_ptr<Block> nextBlockCpy**: is a pointer to the block that all of its fields have the same value as that of the block which is pointed to by the nextBlock.
+ **bool blind**: is a bool. If it is false, then there is no special action, blind, inserted on this player. If it is true, then there is a special action, blind, inserted on this player.
+ **bool heavy**: is a bool. If it is false, then there is no special action, heavy, inserted on this player. If it is true, then there is a special action, heavy, inserted on this player.
+ **bool force**: is a bool. If it is false, then there is no special action, force, inserted on this player. If it is true, then there is a special action, force, inserted on this player.
+ **int blockNum**: is an int that counts the total number of blocks that are dropped in this round of game.
+ **bool blockClear**: is a bool. If it is false, then there is no line that has been cancelled during the process of dropping five blocks. If it is true, then there is at least one line that has been cancelled during the process of dropping five blocks.
+ **int dropcount**: is an int that determines how many blocks should be dropped directly.
+ **vector<vector<shared_ptr<Cell>>> gameBoard**: is a 2-D vector of Cell pointers. This represents the game board of the player.
+ **bool starTerminate**: is a bool. If it is false, then dropping a star at level 4 won't terminate the game. If it is true, then dropping a star at level 4 will terminate the game.

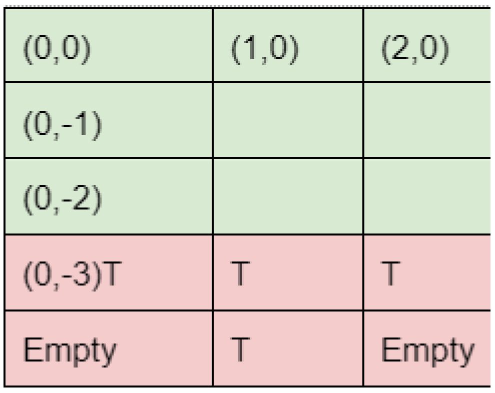
- **int setStar()**:
  is a function that sets a star at the lowest possible row in the gameboard. It returns an integer to indicate which row the star drops at.

Class Block
This class uses the Factory Method design pattern to control each type of block's movement and attribute.

Here, there are seven subclasses in the class Block and each subclass has the same fields and methods. All subclass' methods are served to override virtual methods declared in the parent class. To help understand the logic and the implementation. I will use the subclass T-Block as an example.

Here, according to Biquadris' requirement, the gameboard is (11 x 15). However, there are three additional rows above which are designed to be used when blocks are counterclockwise or clockwise. Therefore, we assume that the gameboard in the player's class is actually 11 X 18. When the block dropped and one of its coordinates is located in the green section, then the we consider the drop command fails and the game ends. Notice here that the coordinate for the block matches the cartesian coordinate system. (An example for the coordinate system is provided below on the left.)





(0,0)	(1,0)	(2,0)
(0,-1)		
(0,-2)		
(0,-3)T	T	T
Empty	T	Empty

<u>Subclass:</u>
Class T-Block
- **int count:** is an int that counts the total number of cells in the player's gameboard field that stores a generated T-block. When each T-block is declared, the count should always be initialized as 4.
- **int deLevel:** is an int that stores the level when a T-block is generated
- string blockType: is a string that stores the type of the generated T-block. It should be always initialized as "T".
- **vector<bool> coordFill:** is a bool vector that stores whether the coordinate is filled by the T-block. It should be always initialized as {true, true, true, false, true, false}. (Please see the right picture above). True means the coordinate is filled and false means that coordinate is empty.
- **vector<vector<int>> coords:** is a 2-D vector that stores the coordinate for each T-Block's initial position in a cartesian coordinate system. It should always be initialized as

{vector<int>{0, -3}, vector<int>{1, -3}, vector<int>{2, -3}, vector<int>{0, -4}, vector<int>{1, -4}, vector<int>{2, -4}}.
- **vector<vector<int>> coordsCpy:** is a 2-D vector that copies the vector<vector<int>> coords.

Notice: All other subclasses in the Block class have exactly the same field and methods. The difference only happens in the initialization of coordFill, coords, and coordsCpy. This is due to each type of block's appearance and different position that each type of block is initially placed on a coordinate system.

Then we move on talking about the methods.

Abstract Level Class:
Methods in Public Fields: (All the methods in Public Fields are virtual and public)

+ **String getType()** will be overridden by other subclasses and returns the block's Type. For example, if this is an I-Block, then the function returns "I"
+ **void left()** will be overridden by other subclasses. It changes the field called coordsCpy in each subclass by moving the coordinate stored in the vector one unit left: --coordsCpy[i][0];
+ **void right()** will be overridden by other subclasses. It changes the field called coordsCpy in each subclass by moving the coordinate stored in the vector one unit right: ++coordsCpy[i][0];
+ **void down()** will be overridden by other subclasses. It changes the field called coordsCpy in each subclass by moving the coordinate stored in the vector one unit down: --coordsCpy[i][1];
+ **int getLevel()** will be overridden by other subclasses. It returns the level that generated this type of block.
+ **bool modifyCount()** will be overridden by other subclasses. Every time it is called, it minus the field called count in each subclass by one. When the count's value is reduced to 0, it returns false. Otherwise, it returns true. False means that the block has been totally removed from the gameboard field in the player's class.
+ **void clockwise()** will be overridden by other subclasses. It changes the field called coordsCpy in each subclass by rotating clockwisely the coordinate stored in the vector.
+ **void counterClockwise()** will be overridden by other subclasses. It changes the field called coordsCpy in each subclass by rotating counterclockwise the coordinate stored in the vector.
+ **void changeContent(bool check)** will be overridden by other subclasses. It consumes a bool. If check is true, then the field called coords in each subclass will copy the value of the field called coordsCpy in each subclass. Otherwise, coordsCpy copies the value of coords.
+ **vector<bool> &getCoordFill()** will be overridden by other subclasses. It returns the reference of coordFill in each subclass.
+ **vector<vector<int>> &getCoords()** will be overridden by other subclasses. It returns the reference of coords in each subclass.
+ **vector<vector<int>> &getCoordsCpy()** will be overridden by other subclasses. It returns the reference of coordsCpy in each subclass.

Class Level
This class, together with the Board class, uses the Factory Method design pattern to create the blocks for players in different levels.

Abstract Level Class:

Methods in Public Fields:

+ **string getBlock()** will be overridden by other subclasses and returns a block type for a level. For example, if an I-block is generated, it returns “I”.
+ **int getLevel()** will be overridden by other subclasses and returns the corresponding level. For example, in `level0.cc`, it returns 0.
+ **void changeState(string file)** will only be used in level 3 and level 4. It changes the randomness in the level. The parameter `file` indicates the file name for the sequence of blocks if the game changes to non-random.

Subclass:

Class Level0
Private Fields:

- **int index**
- **string filename**
- **vector<string> store**

The index field specifies the current index in the block sequence file.
The filename field specifies the name of the block sequence file.
The store field stores the block sequence.

Class Level1 **and** ***Class Level2***
No private fields

Class Level3, Class Level4, and Level5
Private Fields:

- **bool checkRandom**
- **int randomIndex**
- **string filename**
- **vector<string> store**

CheckRandom specifies the randomness of the level.
If it is not random, then we use randomIndex to track the current index of the block in the block sequence file.
The last two fields are the same as level0.

Class Cell
This class represents each cell in the gameboard. It uses the Observer Design Pattern, with the xwindow being its observer. Everytime the cell content changes, it notifies xwindow to make corresponding changes to the graphical display

Private Fields:

- **int x** represents the x coordinate of the cell
- **int y** represents the y coordinate of the cell
- **int width** represents the width of the cell in xwindow
- **int height** represents the height of the cell in xwindow
- **shared_ptr<Xwindow> w** This is the xwindow observer of the cell. Color of the cell changes in the display as the content in the cell changes
- **bool showWindow** specifies whether window needs to be shown
- **bool player** specifies which player the cell belongs to, true if player1, false if player2

Public Fields:

- **shared_ptr<Block> storedBlock** This is the block in the cell. If the cell is empty, then it is a nullptr.
- **bool blind** Specifies whether the cell is blind or not
- **bool star** Specifies whether there is a star in the cell

Methods:

+ **bool filled()** returns whether the cell is filled (has a block or a star in it)
+ **void draw()** draws the cell in xwindow
+ **void undraw()** undraws the cell in xwindow
+ **void clear()** clears the cell content and undraw in xwindow
+ **void update(shared_ptr<Cell> other)** changes the content of the cell to another cell other
+ **void setBlind()** sets the cell to blind and notify xwindow
+ **void unsetBlind()** unblinds the cell and notify xwindow

Class Xwindow

The Xwindow class is responsible for the graphical display of the game. It is an observer of the cells.

Private Fields:

- **Display *d** specifies the connection to the X server
- **Window w** is the window
- **int s** is the screen number
- **GC gc** is the graphical context
- **unsigned long colours[10]** contains the ten colours being used in the game
- **XRectangle recs[2]** are the two frames for the two players

Public Fields:

● All of the bool check, bool player parameters serve as the same purpose: true means player1, false means player2

+ **void fillRectangle(int x, int y, int width, int height, int colour=Black)** fills the rectangle with top left coordinates being (x, y), with the specified width, height and colour
+ **void drawString(int x, int y, std::string msg)** prints the message whose bottom left coordinates are (x, y)
+ **void undrawString(int x, int y, int width, int height)** unprints a message
+ **void drawFrame()** draws the two frames for the two players
+ **void setUp(int l1, int l2)** sets up the default display of the board
+ **void setHighest(bool check, int highest)** refreshes the highest score
+ **void setScore(bool check, int score)** refreshes the score
+ **void setLevel(bool check, int level)** refreshes the level
+ **void clearNextBlock(bool player)** unshows player's next block from the window
+ **void setNextBlock(bool check, std::string bType)** shows player's next block

Class Board

The Board class is the most important and the largest class in our program. It is the only connection to main and it connects to the Player class, Level class, and Xwindow class. It contains all the functions we need for a gameBoard to operate properly. It uses Factory Method to create level objects and blocks.

Private Fields:

- **shared_ptr<Player> player1** represents player1
- **shared_ptr<Player> player2** represents player2
- **shared_ptr<Level> player1Lvl** represents the current level of player1

- **shared_ptr<Level> player2Lvl** represents the current level of player2
- **shared_ptr<Xwindow> w** represents the window
- **bool showWindow** determines whether we need to show the window display

<u>Public Methods:</u>

- All of the bool check, bool p, bool player parameters serve as the same purpose: true means player1, false means player2

+ **void setUp(string filename1, string filename2, int level1, int level2)**
  Initialize the board
+ **void setLevel(int n, bool b, string file)** sets the level for a player. If it is level0, give the corresponding filename file.
+ **void init()** give players their blocks
+ **void getCurBlock(bool check)** gets the current block for a player.
+ **void getNextBlock(bool p)** gets the next block for a player
+ **void show(bool showWindow)** displays the current view of the gameboard (show the window if showWindow is true)
+ **void putBlock()** put two current blocks onto the gameBoard at the start of the game
+ **bool putCurBlock(bool p)** puts the current block of a player (player 1 if p is true, player2 otherwise) onto the game board. If there is a place to put the block, put it on and return true. If there is no place to put it on, return false.
+ **bool checkFilled(shared_ptr<Player> p)** checks whether it is valid to put the current block onto p's gameBoard.
+ **bool validCW(bool check)** checks whether it is valid to rotate a block clockwise for a player (player1 if check is true, player2 otherwise)

The following have the same logic as above:

+ **bool validCCW(bool check)**
+ **bool validLeft(bool check)**
+ **bool validRight(bool check)**
+ **bool validDown(bool check)**
+ **bool drop(bool check)**

+ **bool checkGrid(bool check)** returns true if special actions are triggered, false otherwise. Meanwhile, cancels lines for a player and updates his/her score (player1 if check is true, player2 otherwise)
+ **bool isHeavy(bool check)** returns if a player is in heavy mode
+ **void restart()** restarts the game
+ **bool changeBlock(string bType, bool player)** changes the current block of player to type bType
+ **string win()** returns a string indicating which player wins. If it is a tie, return "Tie"
+ **void changeToHeavy(bool check)** changes the state of a player to heavy
+ **void changeToBlind(bool check)** changes the state of a player to blind
+ **void changeRandom(bool check, string filename)** changes the randomness of a player
+ **void modifyDrop(bool check, int count)** change the dropCount of a player to count
+ **int getDroptime(bool check)** get the dropCount for a player
+ **void removeBlind(bool check)** removes the blind state of a player

The following methods all contribute to hint

+ **void hint(bool check)** produces a hint for a player and prints it out
+ **bool checkFilledHint(shared_ptr<Player> p, vector<vector<int>> coordinates)** checks whether the block for player p with the specified coordinates is filled with other blocks. If it is valid to put in a new block, returns true. Otherwise, returns false.

+ **bool operate(shared_ptr<Player> p, vector<vector<int>> coordinates, int &leftcount, int &rightcount, int &downdeep)** the main operation of hint. The idea of our hint implementation is to put the block in a position as low as possible. Therefore, we copy four blocks, each with a different rotation direction. Then, we move each copy to every column and check how many times it can go down. Combinations that can go down the most become our final choice.
+ **void leftHint(vector<vector<int>> &coords)** moves coords one cell left
+ **void rightHint(vector<vector<int>> &coords)** moves coords one cell right
+ **void downHint(vector<vector<int>> &coords)** moves coords one cell down
+ **bool dropHint(shared_ptr<Player> p, vector<vector<int>> &coords)** determines whether the block with coordinates coords can be dropped for a player p

+ **int minY(vector<vector<int>> coords)** returns the minimum y-coordinate in coords
+ **int getLevel(bool check)** returns the current level for a player
+ **string finalWin()** returns the final winner of the game by comparing the highest score of two players
+ **bool StarTerminate(bool check)** returns whether a star can be put in for a player
+ **void scoreDeduction(bool check)** Used for level 5 only. Deduct 5 points from a player. (The function is called when a player changes block in level 5).

Changes in Uml:
There are no obvious changes in the relationships among six classes between the old and the new uml. Since we consider the implementation of all commands, we add more functions in each class.

1. In the six subclasses of the Level class, we add a new function called changedState. This function is trivial for the level0, level1, and level2 classes and will only be used in level3, level4, and level5. This function helps to tell the level whether to randomly generate a block or to generate a block according to a file.
2. In the seven subclasses of the Block class, we add three new functions called getCoordscpy, getCoords, and getCoordsFill. Since there are three new fields called coordFill, Coords, and Coordscpy, and we need to know the position of each cell in the block, then we need to access these private fields. These functions help by returning references. Also, there is a new function called changeContent which helps Coords and Coordscpy update.
3. There is an extra field in the player class named starterminate. This function helps us to determine whether a star cannot fall onto the game board. If it cannot fail, then we need to terminate the game. We didn't implement this field before because we didn't consider this edge case.
4. We add more functions in the board class, such as checkFilledHint, operate, leftHint,rightHint,downHint, dropHint, minY, and StarTerminate. All the new functions help to realize the requirements listed in the instruction sheet. We have already described their usage in the section above. Again, we didn't include these functions in the old UML due to the fact that we didn't realize we needed to use some of these functions as help functions or as functions to realize commands.
5. We add more methods in the XWindow class for displaying the strings and frames for the two players, and put and clear the next block. We also add to the Cell class a few more fields (width, height, player, storedBlock) and some methods (clear, update, setBlind, unsetBlind) to assist graphical display. We didn't have them in the old UML since at that time, we haven't gained a full understanding of the Window class, and so haven't fully figured out how the graphical display works.
6. To earn the bonus credit, we change all the pointers into smart pointers.

Resilience to Change

Since in the previous section, we discussed the design of the game, then we will prove how our project is low coupling and high cohesion. Our project has five classes and each has unique functionalities. However, all serve for the main purpose: handling all the requirements in the project. Player class organizes all the information that each player should possess (score, highest score, current block, etc). The cell class organizes all the information that each cell in (11 X 15) gameboard should possess (the block, whether this cell has a star exists, etc), the block class (with its subclasses) organizes all the information that each block should possess (the block type, the level it has been generated, how does it look like in a game board, etc). The level class (with its subclasses) organizes the information that each level should possess (the block that should be generated in each level, the level type, etc). Finally, the board organizes all the four classes above and manipulates them according to the commands passed in the main function. Hence, our design recipe fulfills the requirement for high cohesion. Then we turn our attention to low coupling. Since we have five classes and each class has its unique responsibility and functionality. Therefore, a change in a class will have minimal changes to others. For instance, now, there are two player pointers in the board class. If the game wants to increase the player to three or to four players, we can change the implementation by merely adding two more player type pointers in the board class. This avoids overall changes in the program since the player class organizes everything that each player should possess. The board will manipulate each player's field separately. Additionally, if we want to add a new block type, we can merely create another subclass under the block class. Since all the functions in the block class are pure virtual and the way of overriding them in a subclass is similar, creating a new subclass becomes simple. By adding new .cc and .h files and initializing the fields according to the new block's appearance in the game board, the introduction of a new block is completed. The step is almost the same if we want to introduce a new level with different attributes. Last but not least, the board class controls the player pointers separately and each player class controls its 2-D vector of cell pointers and its current and next block pointer separately. This design pattern allows each level, each player, and each block to function separately. Thus, if a new feature is added, the pattern prevents this new feature from affecting other members significantly and reduces its influence to a minimum (ideally to zero). Therefore, we believe that this project's design pattern achieves the goal of low cohesion and high coupling.

Answers to Questions:

Question1:
How could you design your system (or modify your existing design) to allow for some generated blocks to disap- pear 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?

There are differences between these two answers in plan.pdf and design.pdf. Our team can add two extra fields in the Block. One is to count the number of rounds that the generated block survives. Another is to check whether the counter increases by one when another block is dropped each time. Then, if the counter reaches 10, then we need to remove the block from the player's game board. We can call the clear() method in the cell class to achieve the goal. In our design of the project, each level is implemented in a separate class by the factory method. Therefore, each level can execute different requirements. If the requirement asks for generation of blocks confined to more advanced levels, we can use the getLevel() method in the Level class to obtain the level. If the level is high, then we will not execute the clear() method even if the counter in the block reaches 10. Therefore, the second goal in this question is easy to complete.

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

Since we used the factory method to implement levels for the project, we are more confident about the answer. We still provided the exact same answer that we responded to in plan.pdf. When we introduce additional levels into the system, we create additional level classes (level i) that implement the Level interface. Each level i class has methods to generate blocks based on the level. Each level will also contain a method that returns a string to indicate its type. Therefore, when introducing new levels, we only need to recompile the level class. Additionally, the level is only related to the board class. The low coupling enables minimum recompilation.

Question3:
How could you design your program to allow for multiple effects to be applied simultaneously? What if we invented more kinds of effects? Can you prevent your program from having one else-branch for every possible combination?

This time in our project's implementation, we didn't create a class called SuperEffect and use a decorator design pattern to allow multiple effects to be applied simultaneously. However, we still provided the same answer here because we regard it as realizable. To allow for multiple effects to be applied simultaneously, we create a superclass called SpecialEffect, and make a public field that holds a SpecialEffect pointer in the Class Player. In the SpecialEffect superclass, it has specific effects as its subclasses. Then we apply the decorator design pattern. We decorate the Player with a special effect whenever it is introduced, which can prevent our program from having one else-branch for every possible combination. If more kinds of effects are invented, we just create more subclasses under SpecialEffect.

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 can create a map to store all the commands at the beginning of the main function. The keys are the name of the commands and the values represent the functionality. When we add new commands, we add a new key value pair. When we rename a command, we remove the old name from the map and put in the new name. In this way, we only need to modify the key of the map. Therefore, it would be easy for the system to support a rename command. So when reading from the user, we first check if the input string is a valid key in the map, if so, execute the corresponding command; otherwise, we follow the original shortcuts of recognizing commands.

Extra Credit Features

- **Smart Pointers:** All the pointers in this program are smart pointers. For example, we use shared pointers for Player, Level, Block, Cell and Xwindow. We use a unique pointer for Board.
- **Level 5:** I-Block and O-Block will not appear in this level. The probability of generating other blocks is the same. The rules of dropping stars in Level4 remain the same in Level 5. Every time the player chooses to change a block, there is a 5-mark deduction on his/her score.

Final Questions

1. What lessons did this project teach you about developing software in teams?
Through working with team members to develop software, we all felt the power of collaboration. When we first read the outline for the project, we all thought that it was difficult and would take a lot of time. However, when we had a meeting to brainstorm ideas, we found out that the plan came out quickly. Each group member contributed to the development of software by their strong points. Jane always provided a blueprint of implementation to solve questions quickly. However, sometimes she could

miss some requirements in the question and make implementation complicated. Helen and Melissa listened to her ideas and provided suggestions and comments. For instance, Melissa is so good at matrices that she used matrix transformation to manipulate blocks' coordinates. Helen could quickly understand Jane's ideas, simply and turn them into accurate codes. These saved time in implementation. Another efficiency of working together appeared when we debugged the code. Helen is an excellent tester and she took the responsibility of writing edge test cases. Under her assistance, we always found bugs. Melissa is good at debugging code, especially handling segmentation fault errors and memory leaks. Jane was good at remembering each variables' value and notice their changes among lines and functions. Therefore, even none of us could read thousands of lines and debug, when we collaborated, things turned out to be fun and easy. We celebrated and were surprised when Valgrind showed that there was no memory leak.
Therefore, through collaborating, we all shared our advantages to conquer difficulties in coding. There was always a sense of accomplishment when we handled each day's tasks. Working together always brings a miracle.

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 make the while loop in the main function more clear and neat. This time, we notice that the command could be inputted with a prefix index or as long as it could be distinguished from other commands. However, we didn't consider the allowance of any capital letters or case sensitives. Additionally, when users provided inaccurate commands, we didn't provide any suggestions to remind the user. This could make them confused. Last but not least, if we had the chance to start over, we could enable users to personalize their name instead of calling them player1 and player2.

Conclusion

Biquadris is an excellent and fun project to be introduced at the final stage of this course. This project enables us to apply core concepts learned in CS246, such as oop and smart pointer. Through coding, we always found the unlimited possibility to create nice solutions to handle various tasks. We also learned from coding: we were confronted with segmentation fault errors twice in debugging. At first, we were puzzled and spent nearly three hours tracing between lines. However, after noticing that the error happened when the compiler tries to read an illegal memory location, things became easy. We learned to check when a new memory was allocated or when it was freed. During the second time, we discovered the error quickly in 15 minutes. Through coding a game and recognizing the capability of C++, all of us became more interested in computer science. We will bring this passion into further study.