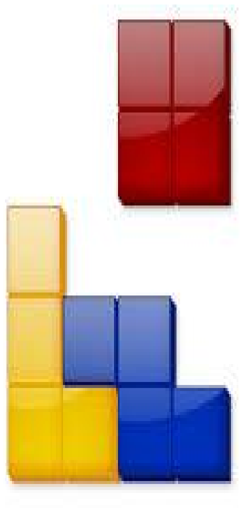


# Biquadris

5th April 2022

Group Members:
Bosen Zhang - b393zhan
Xiancheng(Andrew) Zang - x5zang
Mingyuan Ren - m28ren

CS 246—Assignment 5, Group Project (Winter 2022)

## Introduction

The video game Biquadris, which is a Latinization of the game Tetris, expanded for two player competition.

A game of Biquadris consists of two boards, each 11 columns wide and 15 rows high. Blocks consisting of four cells (tetrominoes) appear at the top of each board, and you must drop them onto their respective boards so as not to leave any gaps. Once an entire row has been filled, it disappears, and the blocks above move down by one unit. Biquadris differs from Tetris in one significant way: it is not real-time. You have as much time as you want to decide where to place a block.
Players will take turns dropping blocks, one at a time. A player’s turn ends when he/she has dropped a block onto the board (unless this triggers a special action; see below).
During a player’s turn, the block that the opponent will have to play next is already at the top of the opponent’s board (and if it doesn’t fit, the opponent has lost).

## Overview

The major components of the system are: Cell, Block, NextBlock, Board, Level and Display(GameDisplay and PlayerDisplay).

### Cell:

The starting point would be implementing the Cell class. This class is easy but crucial to our project as the Block class and Board class are composed of cells, and we have other classes composed of these two classes.

**Value** records whether this cell is occupied by a block. For example, if the cell is occupied as a part of the T block, the value is set as “T”.  If the cell is not set, the value will be “ “.

**X, Y** record the position(X, Y) of the cell in the board.

**Left, Right, Bot** are the shared_ptr to record cells on the left, right and bottom of the cell.

**ValueSet** records whether the cell is set to a value.

**Cell()** is the constructor of the cell object.

**getValue()** gets the value of the cell.

**getLeft(), getRight(), getBot()** get the information of cells on the left, right and bottom of the cell.

**setLeft(), setRight(), setBot()** set the shared_ptr to record cells on the left, right and bottom of the cell.

**setValue()** sets the value of the cell.

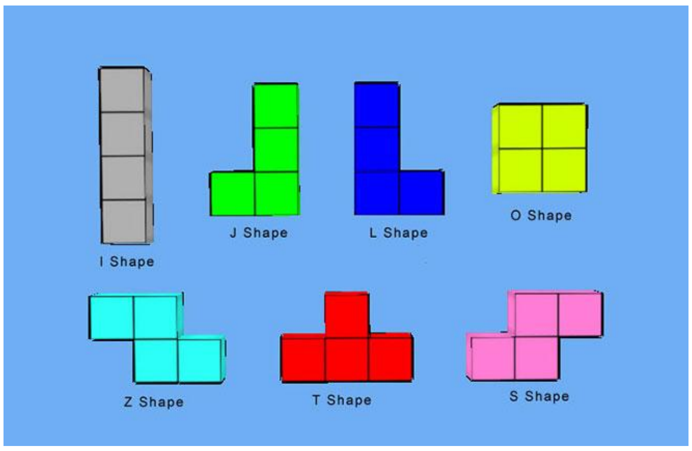
**setX(), setY()** set the position (X, Y) on the play board of the cell.

**setLabel()** sets the label of the cell.

**cellMove()** moves the cell.
**unsetValue()** sets the value of the cell to " ".
**setbd()** sets the ** of the cell.

## Block:

A Block class is an abstract class. And for different shapes of blocks, there are several derived classes(Lblock, Jblock etc.) from the base class. The private field value is a string variable representing the name of the block(Lblock, Jblock etc.). And We use a 2-D vector of cells to represent a block object. For rotating the block during the game process, the virtual methods consisting of clockwiseRotate, counterClockwiseRotate will be used to modify the vectors accordingly.



```
class Block {
  protected:
  std::string value;
  int lab;
  vector<vector<shared_ptr<Cell>>> grid;
  int rotateState = 0;

  public:
  Block( char c, int lab );
  virtual ~Block();
  void virtual clockwiseRotate() = 0;
  void virtual counterClockwiseRotate() = 0;
  std::vector<std::vector<std::shared_ptr<Cell>>>& getGrid();
  std::vector<std::vector<std::shared_ptr<Cell>>>* getGridptr();
  int getLab(){return lab;}
};
```

**Value** of a block is set as the type of the block(I, J, Z etc.)
lab
**Grid** is a 4*4 2-D vector of cell which is used to contain a type of block.
**RotateState** used for recording the rotated state of a block.
**Block()** is the constructor of the block object.
**~Block()** is the destructor of the block object.

**clockwiseRotate()** clockwise rotates the block.
**counterClockwiseRotate()** counter clockwise rotates the block.

## Level:

The Level class is used as part of the factory design pattern with the NextBlock class. It has one pure virtual method that is implemented in its concrete derived class ConcreteLevel.
**createNextBlock()** is used to generate NextBlock objects based on level specified and text file (for level zero only).

## NextBlock:

The NextBlock class is used to generate the incoming block to be placed into the board based on specified level. For this class we used a factory design pattern where we created an object of class ConcreteLevel.
We have derived classes for each level that all inherited from the NextBlock class. We have made the constructors within each derived class to be private. All of the derived classes are friends with the class ConcreteLevel so that the ConcreteLevel can call the constructors.
There are two private fields within the NextBlock class. An integer field level to store the level and a vector of char to store blocks for level 0.
**getLevel()** is a getter method to get the level of the current NextBlock object and t
**blockSelector()** is a pure virtual method. It is used to select the incoming block for a player according to the level. For the probability specified for each level, we used the rand() function to randomly generate a number and obtain the remainder of it based on the total possibilities. And we divide each chunk to specify which block the blockSelector function should return back.

## Board:

The Board class also contains a 2-D vector of 18 x 11 cells since we need to reserve 3 extra rows for rotations. And for 2 users, we have two separate boards, one for each user. The board is stored in the PlayDisplay class and the Display class has two PlayerDisplay objects, one for displaying each player's gaming information.

```
class Board {
  vector < vector<shared_ptr< Cell > > >content;
  public:
  Board();
  vector < vector<shared_ptr< Cell > > > getContent();
  void insertBlock(Block *);
  void pushBack(Block *); // throws an exception pushdErr if cannot push
  void pushLeft(Block *); // throws an exception pushlErr if cannot push
  void pushRight(Block *); // throws an exception pushrErr if cannot push
  int clearLine();
};
```

**content** is a 2-D vector of 18 x 11 cells.

**Board()** is the constructor of the Board class.

**InsertBlock()**, **pushBack()**, **pushLeft()** and **pushRight()** will insert, push back, push left and push right a block inside the Board.

**clearLine()** clears the lines that are filled with set cells and returns the number of lines that are cleared.

## Display:

The display object is the major section of the program and which is also the interface that the user can see and play with.

GameDisplay

```
class GameDisplay {
  Xwindow *pxw; // 600*500, each cell takes 15, margin = 50, each string takes 30, distance
      between two boards is 70
  PlayerDisplay *p1;
  PlayerDisplay *p2;
  int currentSeed; // used for setting seed by user.

  public:
  GameDisplay(Xwindow *, PlayerDisplay *, PlayerDisplay *);
  void boardSetUp();
  void play();
};
```

The **GameDisplay** is the object containing the pointers to Xwindow, Two PlayerDisplay(p1, p2) and the currentSeed used for setting seed by user.

**pxw** is the Xwindow that displays the main information of the Game Board. It is two-dimensional vectors of Cells. pxw is 600*500 pixels, each cell takes 15 pixels, margin is 50, each string takes 30, distance between two boards is 70.

Two PlayerDisplay **p1, p2** are the gameboards of different players. The details will be provided in the next section about PlayerDisplay.

**GameDisplay()** is the constructor of the GameDisplay object which takes Xwindow, and two PlayerDisplay objects.

**BoardSetUp()** will use fillRectangle and drawString to display the game boards on the interface.

**Play()** sets up the basic information of the playboard,the currentseed and  initializes the information of players(level, score etc.). It also controls the rotation of two players and whether the game will end.

PlayerDisplay

The **PlayerDisplay** is the object containing the pointers to Xwindow, the player's board and also the Block information of the current player.

**BoardContent** is the content of the Player's Board.

**Level** is the Game Level that the player chooses.

**Playerpos** records the position of the player's information displays on the interface.

**CurrBlock** and **NextBlock** keep track of the current block information received and the next block information the player received.

**PlayerDisplay()** is the constructor of the **PlayerDisplay** object.

**~PlayerDisplay()** is the destructor of the **PlayerDisplay** object.

**NewNext()** creates the next block.

**BlockUpdate()** and **BlockClear()** are responsible for updating the block information and also clearing a block.

**PrtLevel()**, **PrtScore()** and **PrtNext()** print the Level, Score and Next block information on the Playerboard.

**MoveLeft()**, **MoveRight()** and **MoveDown()** move the current Block of the PlayerDisplay.

**RotateClock()** and **RotateCounterClock()** are responsible for rotating the current Block.

**CheckEndGame()** and **CheckEndRound()** will check whether the game ends or the round for the player ends.

**ClrLine()** clears the line that is filled with set cells.

## Design

**Vector, shared_ptr, unique_ptr and exception safety**
We implemented the vector that is a sequence container to represent our 2-D Board which is easier to handle future growth since the storage of the vector is handled automatically, being expanded and contracted as needed. We tried our best to avoid memory issues. To avoid unnecessary procedures to call the destructors, we used smart pointers that will automatically do it for us.

**Abstract Class and Inheritance**
**Factory Design Pattern**
Proceed to the resilience to change section

## Resilience to Change

Abstract Class and Inheritance:

The degree to which the elements of a module/class belong together is very high, the related codes are binded as closely as possible and should be close to each other. But we also achieved a low degree to which the different modules/classes depend on each other as independently as possible among different modules/classes. For example, we kept these functionalities in the Block class. A Block class is an abstract class. And for different shapes of blocks, there are several derived classes(Lblock, Jblock etc.) from the base class which are easy for us to adjust one type of block with the minimal changes to the others.

And we applied the same strategy to Level class. We implemented the abstract class Level, and also its concrete subclasses “Level 0”, “Level 1” etc.   For different levels of games, we have achieved maximum independence for different levels to facilitate Resilience to Change.

Factory Method:

The factory method is applied when generating the next block. In the Factory pattern, we create NextBlock objects without exposing the creation logic to the client and the client uses the same common interface to create a new type of object. Concrete Creators override the base factory method so it returns a different type of NextBlock.

## Answers to Questions

Question: 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?
We can add a parameter "count "into the Cell class to keep track of for every single cell how many rounds a unit of block occupies the cell. After each round of falling a block, if the cell is in an occupied state, its count will be incremented by one. And if the cell is cleared, the count of the cell class will be reset as 0. And since the level of the game only changes the way of falling blocks which works the same way on cell class.

Question: 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 separate levels of the game as a different Level class, and we use Block class to make it easy to modify a block. And we will use pimple strategy, so that when we accommodate the possibility of introducing additional levels into the system, we only need to re-compile Level, block and main with minimum recompilation.

Question: How could you design your program to allow for multiple effects to applied simul-taneously? What if we invented more kinds of effects? Can you prevent your program from having one else-branch for every possible combination?
To allow for multiple effects to be applied simul- taneously, we will use the Decorator design pattern. For more kinds of effects, we will allow effects to be added to an individual object, dynamically, without affecting the effects of other objects from the same class which prevent our program from having one else-branch for every possible combination.

Question: 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.

To accommodate the addition of new command names, we will add separate methods(if-else) for every single command. And while adding or changing to the new command names, we ignore the existing command name

## Extra Credit Features

This program was written without memory leaks since we used smart pointers to ensure the memory management. We used shared_ptr while constructing the cells in the block and board that retains shared ownership of a cell through a pointer. Several shared_ptr objects may own the same block or board object. Similarly, we applied the same strategy to the shared_ptr for blocks in the display.

## Final Questions

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?

It is totally different compared working alone to working in a team environment. We all have definitely learned a lot through writing this assignment. We will explain things we learned in detail in the following paragraphs.

First off, it is important to write human readable code. When working alone, it is easy to follow one's thoughts and write a large program with all the features that he/she wants. Working in a team involves reading the code other people write and understanding how their implemented feature works. Thus, avoiding trivial naming for class fields and methods could save the whole team a lot of time understanding the codebase. In addition, having codes separated by modulation and writing documentations on the complex methods and classes could definitely help understanding code easier.

Secondly, we found having a place to keep track of everyone's thoughts is really useful. We have a google docs file that everyone can write their thoughts on how to implement each class. By doing so, we can easily compare everyone's ideas on how we can implement features in the easiest way and this saves us a lot of time.

Git is a very powerful and useful version control tool when working on a team project. We learned how to check out everyone's work into different branches. This allowed us to implement each part of the code separately and merge them back with a code review session. We also learned the difference between forking a repository and working under the same repository.

Overall, the collaboration and version control skills are pretty valuable when working in a large program and we learned skills that we believe that are also useful when working in the industry.

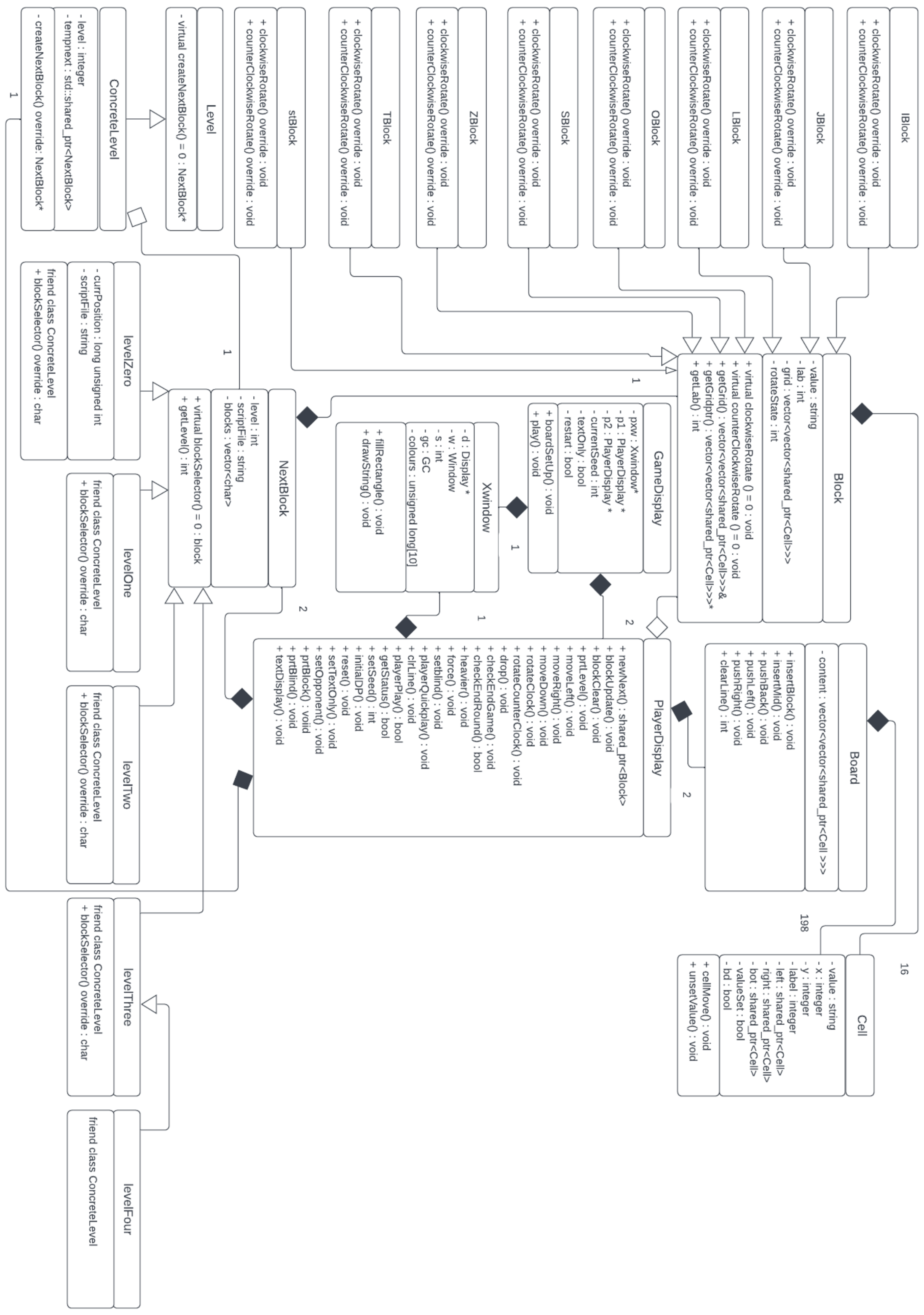
2. What would you have done differently if you had the chance to start over?

We are pretty proud of our achievement but there are definitely many things we can improve if we can start over.

First off, we would have a more comprehensive plan of attack and stick more strictly to each of the deadlines. We wasted some time waiting for some features to be implemented and with setting up the environment. Everyone has slightly different expectations for each feature to be implemented but we did not discuss it in advance. Therefore, we would spend more time doing a plan of attack if we can start over.

Secondly, we would make sure that each part is bug free and test the functionality before we merge them into the main branch. We believe this practice would save us more time compiling and debugging the whole project. We spent unnecessary time debugging and finding mistakes that other people have made which should be avoided.

# Updated UML


IBlock
+ clockwiseRotate() override : void
+ counterClockwiseRotate() override : void
JBlock
+ clockwiseRotate() override : void
+ counterClockwiseRotate() override : void
LBlock
+ clockwiseRotate() override : void
+ counterClockwiseRotate() override : void
OBlock
+ clockwiseRotate() override : void
+ counterClockwiseRotate() override : void
SBlock
+ clockwiseRotate() override : void
+ counterClockwiseRotate() override : void
ZBlock
+ clockwiseRotate() override : void
+ counterClockwiseRotate() override : void
TBlock
+ clockwiseRotate() override : void
+ counterClockwiseRotate() override : void
stBlock
+ clockwiseRotate() override : void
+ counterClockwiseRotate() override : void
Level
- virtual createNextBlock() = 0 : NextBlock*
ConcreteLevel
- level : integer
- tempnext : std::shared_ptr<NextBlock>
- createNextBlock() override: NextBlock*
1
levelZero
- currPosition : long unsigned int
- scriptFile : string
friend class ConcreteLevel
+ blockSelector() override : char
levelOne
friend class ConcreteLevel
+ blockSelector() override : char
levelTwo
friend class ConcreteLevel
+ blockSelector() override : char
levelThree
friend class ConcreteLevel
+ blockSelector() override : char
levelFour
friend class ConcreteLevel
NextBlock
- level : int
- scriptFile : string
- blocks : vector<char>
+ virtual blockSelector() = 0 : block
+ getLevel() : int
1
2
Block
- value : string
- lab : int
- grid : vector<vector<shared_ptr<Cell>>>
- rotateState : int
+ virtual clockwiseRotate () = 0 : void
+ virtual counterClockwiseRotate () = 0 : void
+ getGrid() : vector<vector<shared_ptr<Cell>>>&
+ getGridptr() : vector<vector<shared_ptr<Cell>>>*
+ getLab() : int
1
GameDisplay
- pxw : Xwindow*
- p1 : PlayerDisplay *
- p2 : PlayerDisplay *
- currentSeed : int
- textOnly : bool
- restart : bool
+ boardSetUp() : void
+ play() : void
1
Xwindow
- d : Display *
- w : Window
- s : int
- gc : GC
- colours : unsigned long[10]
+ fillRectangle() : void
+ drawString() : void
1
2
PlayerDisplay
+ newNext() : shared_ptr<Block>
+ blockUpdate() : void
+ blockClear() : void
+ prtLevel() : void
+ moveLeft() : void
+ moveRight() : void
+ moveDown() : void
+ rotateClock() : void
+ rotateCounterClock() : void
+ drop() : void
+ checkEndGame() : void
+ checkEndRound() : bool
+ heavier() : void
+ force() : void
+ setblind() : void
+ playerQuickplay() : void
+ clrLine() : void
+ playerPlay() : bool
+ getStatus() : bool
+ setSeed() : int
+ initialDP() : void
+ reset() : void
+ setTextOnly() : void
+ setOpponent() : void
+ prtBlock() : void
+ prtBlind() : void
+ textDisplay() : void
2
Board
- content : vector<vector<shared_ptr<Cell >>>
+ insertBlock() : void
+ insertMid() : void
+ pushBack() : void
+ pushLeft() : void
+ pushRight() : void
+ clearLine() : int
198
16
Cell
- value : string
- x : integer
- y : integer
- label : integer
- left : shared_ptr<Cell>
- right : shared_ptr<Cell>
- bot : shared_ptr<Cell>
- valueSet : bool
- bd : bool
+ cellMove() : void
+ unsetValue() : void