## Introduction

The game of Biquadris consists of two players and seven different types of blocks Whenever a line of cells disappears, a player earns scores. Whenever a new block cannot be properly initialized, the player's game is over, and he has the choice to either start a new game or quit the game. Unlike a usual *Tetris* game, players have as much time to

## Overview

This game has five different classes: Biquadris, *Cell*, *Board*, *Level* and *Block*. 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.

## Design

### *Class: Biquadris*

This class is to be included in the main file to read in the commands and start the game. Play() is the only public method here in Biquadris, while others are all helper functions.

**play()** is to initialize the game based on the inputs.

**init()** is to initialize the board we need for the game.

**restart()** is to clear the previous board, reset previous score and status, and start a new game.

**setSeed()** sets the seed to the input *seed,* **setStartLevel()** sets the start_level to the input *start_level*, **setScript1(), setScript2()** sets the scriptfilen[0] and [1] to input *fname.*

**readArgs()** is to read in arguments and act based on different command line arguments.

**readCmd()** is to read in commands and run different methods based on the input.

**exCmd()** rename an existing command if the input is "rename', otherwise, it executes the move command ("left", "right", "counterclockwise", "levelup", etc.) If the input is invalid, output "invalid" and provide a list of valid commands.

**allGamesOver()** returns true if both player finish their game. **printGame()** prints "all games over" if **allGamesOver()** is true, otherwise it prints the updated board after the previous move.

**~Biquadris()** is the destructor for our boards.

## *Class: Cell*

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

- **level** is an integer to determine what the current level is.
- **r** and **c** are to determine the position of the Cell in Xwindow function (r: row, c: column).
- **type** is a char that represents the shape of the cell. It's empty if the cell does not belong to any blocks.
- **isSet** is a Boolean value to determine whether the cell is currently occupied by one of the existing blocks.
- **detachObs** is a Boolean value that determines whether this cell needs to be detached.
- **isBoard** is a Boolean value that determines whether this cell needs to be included in our Board.
- **shared_ptr<Block> block** is a smart pointer that points at a block

For the function sectors:

**Cell**() is to initialize a cell at the given row **r** and column **c**.

**notify**() is to tell the text and graphical displays that this cell has been changed.

**getInfo()** is to return the information of the previous cell.

**setCell**() is to set the current cell using the pass-in information.

## *Class: Block*

Block class is an abstract class.

**getMaxY(), getMaxX(), getMinX(), getMinY()** return the dimension of a block, so when we need to rotate our block, we can use these data to calculate if the move is valid. If yes, then what is should look like after the rotation.

**moveRight(), moveLeft(), moveDown()** move the current block to right, left or down by one unit respectively.

**rotateCW():** first we make a copy of the current block and execute the rotation command. If failure, then there is no effect, but if success, we print the new block.

**getCoords(), get_theCells(), getType(), getLevel()** return the coordinates, current cells, type of block, and current level respectively.

**setCoords()** set the block to the new location (x, y), **set_theCells()** set the cell to the position *pos*.

**removeCoords()** removes all the cells that match the given location, i.e. cells at the provided location.

**printRow()** prints out the block on our text display based on their shapes by searching through our Cells vector. For example, if there is an I-block, we print IIII at its corresponding location. Otherwise, we leave the place empty.

The subclasses of Block are: **IBlock, JBlock, LBlock, OBlock, SBlock, TBlock, XBlock,** and **ZBlock.** Rather than the required 7 blocks, an additional blocks, **XBlock** is added as extra features in level 5. All subclasses override each pure virtual functions in Block.

## *Class: Board*

Public methods and fields:

**getInfo()** is to return the information of the current board.

**Board**() is just a constructor. **~Board()** is just a destructor.

**init()** is to initialize the text only board.

**inRange()** returns a Boolean value that returns true if **c** is between 0 and maximal width and **r** is between 0 and maximal height, otherwise it returns false.

**isEmptyCell()** returns true if the cell at the given location is empty, otherwise false.

**Movable()** returns false if any of the cell will either be outside of our board, or overlap with another existing cell after we execute a single move, otherwise it returns true.

**isRowFull()** determines if every cell in a particular row is not empty. If yes, then return true; otherwise returns false.

**changeDrops()** set the number of remaining blocks to the input value, and **getDrops()** returns the number of remaining blocks. **getRowsCleared()** returns the number of rows we already cleared, while **setRowsCleared()** reset the number of rows we cleared to 0.

**printRow()** prints ? if our isBlind value is set to true, otherwise it prints every row.

**moveRight()** is to move the current block to the right by one unit. If **isHeavy** is set to true, then the block moves down by 2 units at the same time when moving to the right.

**moveLeft()** is similar to **moveRight()**, except it moves the current block to the left by one unit.

**moveDown()** is similar to **moveRight()**, except it moves the current block to the bottom of the board by one unit.

**rotateCw()** is treated separately from left, right and down because there are more tricky edge cases to consider. Every time the client calls for cw rotation rotation, a copy of the current block is generated with Aclone, a method in Block (for more details in Block). The copy is rotated and the board tries to print the copy on the PlayField. However, if after rotation, the copy cannot be properly printed, it will have no effect. When we want to execute **rotateCcw(),** we just need to execute **rotateCw()** by 3 times.

**dropBlock()** is to drop the current block to the bottom of the Board as far as possible.

**clearRow()** is activated when a row is filled with blocks, and then generates a new one to replace this row. If multiple lines are cleared at the same time, **multiRows()** comes in handy and activates the bonus effect.

**checkFullRows()** checks for all rows on the game board, clears them, updates the score, and adjusts the falling count based on the game's level.

**genNewBlock()** is activated after the drop command, which replaces the current block with our next block and test if the game is over. If yes, outputs "Game Over" and freeze. If not, keep the game running and generates a new block for our next move.

**setBlind()** and **unsetBlind()** are to switch the statue of isBlind. Same as **setRandom** and **unsetRandom.**

**setCurBlock**() reset the current block to one of the 7 types of blocks we have.

**setLevelPtr()** uses smart pointer to points at different levels.

**levelup()** and **leveldown()** allow us to switch between different levels if our change if valid, i.e. 0 <= level <= 5.

**getLevel(), getScore(), getNextBlock(), gameOver(), isHeavy(), getDropped()** return the corresponding statues of our game.

**toggleHeavy()** and **toggleDropped** act like a switch of the Boolean value isHeavy and isDropped respectively, while **setHeavy()** strictly sets isHeavy to true.

**setSeq()** changes the current block sequence to the input sequence, while **resetSeq** does the opposite of **setSeq()**.

**print()** prints out the block at its current location. If we need to move the block to a new location, we first undraw the block at its current position on the board, determine if the move is valid, and then after all, we use **print()** to show its new position. We use the observer method to draw/undraw the blocks at the same time.

## ***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. If we need to introduce additional level, all we need is simply write a new subclass of *level* and recompile this subclass and the level.

### <u>Abstract Level Class:</u>

Methods in Public fields:

**~Level():** virtual ~Level() function is the destructor for level, and it's virtual so that every subclasses of level can have different implementation for their own destructors.

**Level():** Level() will be overridden by other subclass and return the required Block share pointer. Every Level has vector section to hold a random sequence and a file input sequence of the block sequences. It depends on which level it is. As the difficulty increases, we need to introduce more parameters to Level(), including *seed, sequence, isRandom*, etc.

**genBlock():** genBlock() function generates new blocks based on the level of difficulties. This function is overridden by other subclasses since there are different rules of generating blocks for different levels.

**getLevel():** getLevel() simply return the integer representing the current level.

### *Main function*

Our main function is a wrapper function of our Biquadris file, with an additional feature that let the game keep going.

### *Command Interpreter:*

**parseTimes()** returns the number of times we want to execute the given command, 1 as the default value if it is not given.

**parseCommands()** is to accept partial command, for example, if the input is "le", the method searches for matched command name in our memory, and then executes the command "left".

**rename()** is to rename an existing command name with a new string input.

**removeCase()** takes a command and return it all in lower case letters.

**printCommands()** prints out a list of valid commands that we can input.

## Resilience to Change

First, we use a lot of inheritance in our implementation, e.g. block.cc and board.cc. If any of our program specification changes, for example, we want a different block type, or a slightly different rule in our game, we need minimal recompilations to achieve our goals. Second, we introduce different design patterns in our implementation. For our level implementation, we have an abstract data class *level* and a bunch of virtual methods, and all we need to do is when we create a new level, we inherit from *level* class, and override the virtual methods for different rules. Thus, we do not need to change our base class and other subclasses once we want to introduce new levels into the game. Finally, we use observers when implementing our *cells* and *board*. Cells inherit from *subject* and *observer*, so it can both be treated as a subject, and we can use it to make many of our methods easier to be implemented.

## Previous 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?***

By introducing a counter to our *Block* class, we can keep track of how many rounds each block survives. Whenever the counter of a particular block (let's call it B) exceeds 10, we know that 10 blocks have fallen after B is generated and thus B needs to disappear from the screen. Since our game has different levels of difficulties, we can easily confine the generation of such blocks to advanced levels by adding the counter only when there exists a rule about block disappearing after certain amounts of blocks fallen. For example, we can add a Boolean value in our class *Level(harder)*, enable the feature of auto disappearance after 10 fallen blocks only for higher difficulties.

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

Let's consider the Factory Method design pattern. We can first implement a *Level* class, and for each level of difficulties, we create a subclass which inherits from the public *Level* class (e.g., class Level0: public Level, class Level1: public Level, etc.) and allows us to set up different rules for different levels. Hence if we want additional levels into our system, we simply need one more subclass of our *Level* class (could be Level5, Level6 or even more). When recompiling, only our parent class (*Level)* and the new children *Level* class (the one we just introduced to our system) need to be recompiled.

***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 could have three Boolean values in our *Board* class, call them *isBlind, isHeavy, isForce.* When we need to apply any of these effects, we just need to set the corresponding Boolean value to true, and then activate the corresponding function as well. (*blind, heavy, force*) Once we invent a new kind of effect, we apply the same procedure: adding a new Boolean value to our *Board* class and write a function that takes the Boolean value as the input. (e.g. *makeHeavy(bool*) {...}) By doing so, we don't need to list every possible combination of effects, but simply change the state of our Boolean values to achieve our goals of applying multiple effects 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?***

We could have a string library that indicates a specific command, such as *left* or *right.* For example, if we want alias for command *left*, we could have vector< "l", "le", "lef", "left" >. When we receive one of these inputs, we run the command *left*. If we want to add an alias, simply use *emplace_back*. If we want to instead rename the existing command name, simply pop_back and then add the new name to left[1]([0]position is the const command string name for this specific command, which introduced later). This is for one command, and if we want to apply this pattern to all the commands, we just need a new vector of all the vector of command strings. If we want to support a "macro" language, we can define a constant command string name that cannot be changed for each command. For example, if want "lr" as a name of left and right, simply create a vector<const command name of left, const … of right>. When we call the macro name, the equivalent vector will go into the executing while loop and execute each specific command.

## Final Questions

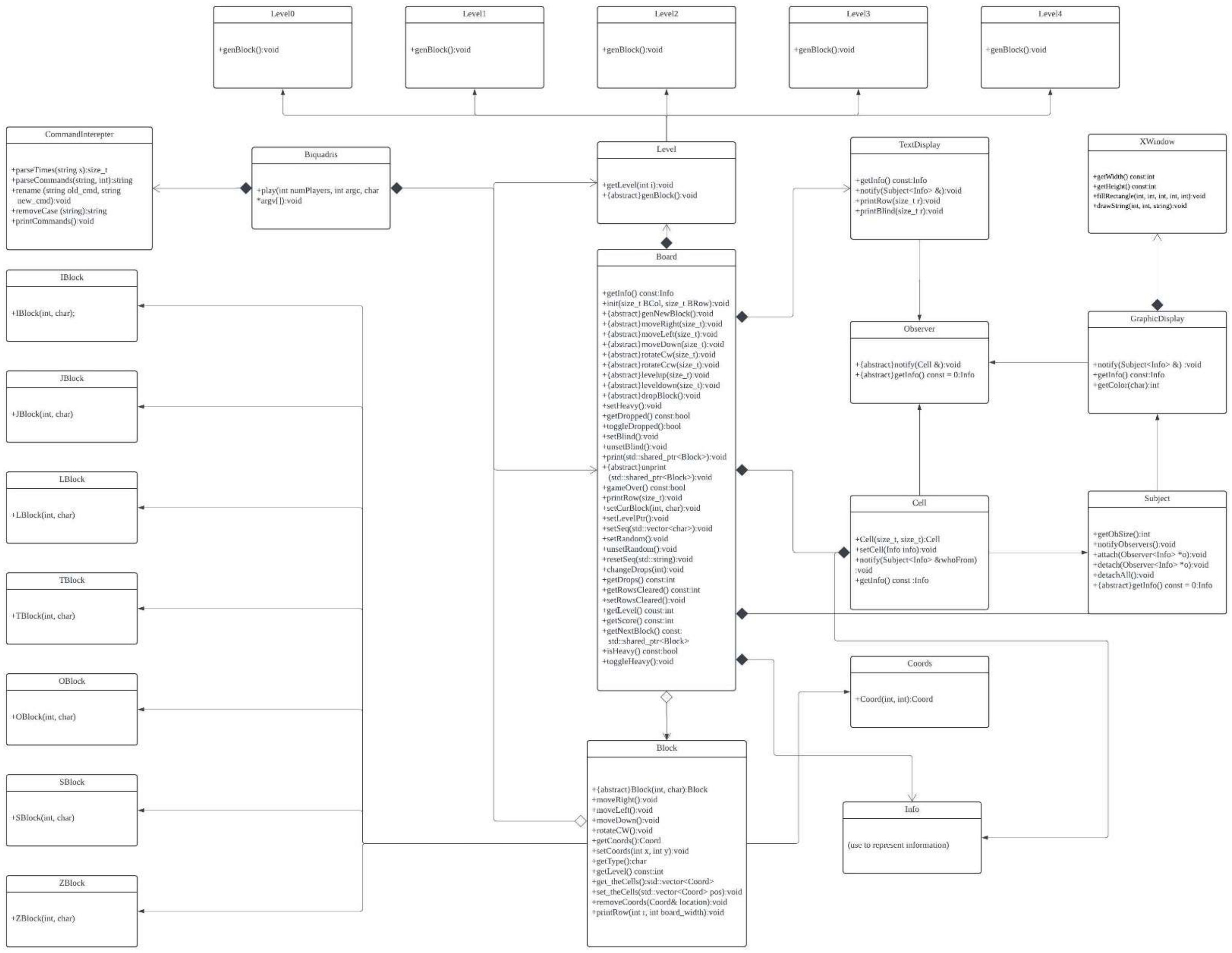
***1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?***

We learned a lot during this project. First, we understand the importance of teamwork. This project is challenging, whether it's the initial design, implementation, or late-stage debugging, it's impossible for any one of us to independently complete it. Second, we realize that it's important to allocate tasks reasonably based on the abilities of team members and their available time. If our timetable is not initially reasonable, we might encounter situations such as inability to complete all the code as the deadline approaches, insufficient time for debugging, or the program can't run, which would prevent us from achieving our final goal. Finally, communication among team members is vital. Without communication, we can't tell everyone’s progress. Once there is a conflict in progress between two people, we will also be unable to complete this project.

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

If we have a chance to start over, we can focus on one of our suggestions in the first week, so we won’t waste so much time on deciding which approaches we should use to finish this project. We could’ve also made our planning better, because we wasted a lot of time reorganizing our plan thus we missed a lot of time for implementation.

## Updated UML



Level0
+genBlock():void
Level1
+genBlock():void
Level2
+genBlock():void
Level3
+genBlock():void
Level4
+genBlock():void
CommandInterepter
+parseTimes(string s):size_t
+parseCommands(string, int):string
+rename (string old_cmd, string new_cmd):void
+removeCase (string):string
+printCommands():void
Biquadris
+play(int numPlayers, int argc, char *argv[]):void
Level
+getLevel(int i):void
+{abstract}genBlock():void
TextDisplay
+getInfo() const:Info
+notify(Subject<Info> &):void
+printRow(size_t r):void
+printBlind(size_t r):void
XWindow
+getWidth() const:int
+getHeight() const:int
+fillRectangle(int, int, int, int, int):void
+drawString(int, int, string):void
Board
+getInfo() const:Info
+init(size_t BCol, size_t BRow):void
+{abstract}genNewBlock():void
+{abstract}moveRight(size_t):void
+{abstract}moveLeft(size_t):void
+{abstract}moveDown(size_t):void
+{abstract}rotateCw(size_t):void
+{abstract}rotateCcw(size_t):void
+{abstract}levelup(size_t):void
+{abstract}leveldown(size_t):void
+{abstract}dropBlock():void
+setHeavy():void
+getDropped() const:bool
+toggleDropped():bool
+setBlind():void
+unsetBlind():void
+print(std::shared_ptr<Block>):void
+{abstract}unprint (std::shared_ptr<Block>):void
+gameOver() const:bool
+printRow(size_t):void
+setCurBlock(int, char):void
+setLevelPtr():void
+setSeq(std::vector<char>):void
+setRandom():void
+unsetRandom():void
+resetSeq(std::string):void
+changeDrops(int):void
+getDrops() const:int
+getRowsCleared() const:int
+setRowsCleared():void
+getLevel() const:int
+getScore() const:int
+getNextBlock() const: std::shared_ptr<Block>
+isHeavy() const:bool
+toggleHeavy():void
Observer
+{abstract}notify(Cell &):void
+{abstract}getInfo() const = 0:Info
GraphicDisplay
+notify(Subject<Info> &) :void
+getInfo() const:Info
+getColor(char):int
Cell
+Cell(size_t, size_t):Cell
+setCell(Info info):void
+notify(Subject<Info> &whoFrom) :void
+getInfo() const :Info
Subject
+getObSize():int
+notifyObservers():void
+attach(Observer<Info> *o):void
+detach(Observer<Info> *o):void
+detachAll():void
+{abstract}getInfo() const = 0:Info
IBlock
+IBlock(int, char);
JBlock
+JBlock(int, char)
LBlock
+LBlock(int, char)
TBlock
+TBlock(int, char)
OBlock
+OBlock(int, char)
SBlock
+SBlock(int, char)
ZBlock
+ZBlock(int, char)
Coords
+Coord(int, int):Coord
Block
+{abstract}Block(int, char):Block
+moveRight():void
+moveLeft():void
+moveDown():void
+rotateCW():void
+getCoords():Coord
+setCoords(int x, int y):void
+getType():char
+getLevel() const:int
+get_theCells():std::vector<Coord>
+set_theCells(std::vector<Coord> pos):void
+removeCoords(Coord& location):void
+printRow(int r, int board_width):void
Info
(use to represent information)