Biquadris Design Documentation

CS 246 Final Project

Yizhen Chen

Emilia Hou

Yi Wang

Table of Contents

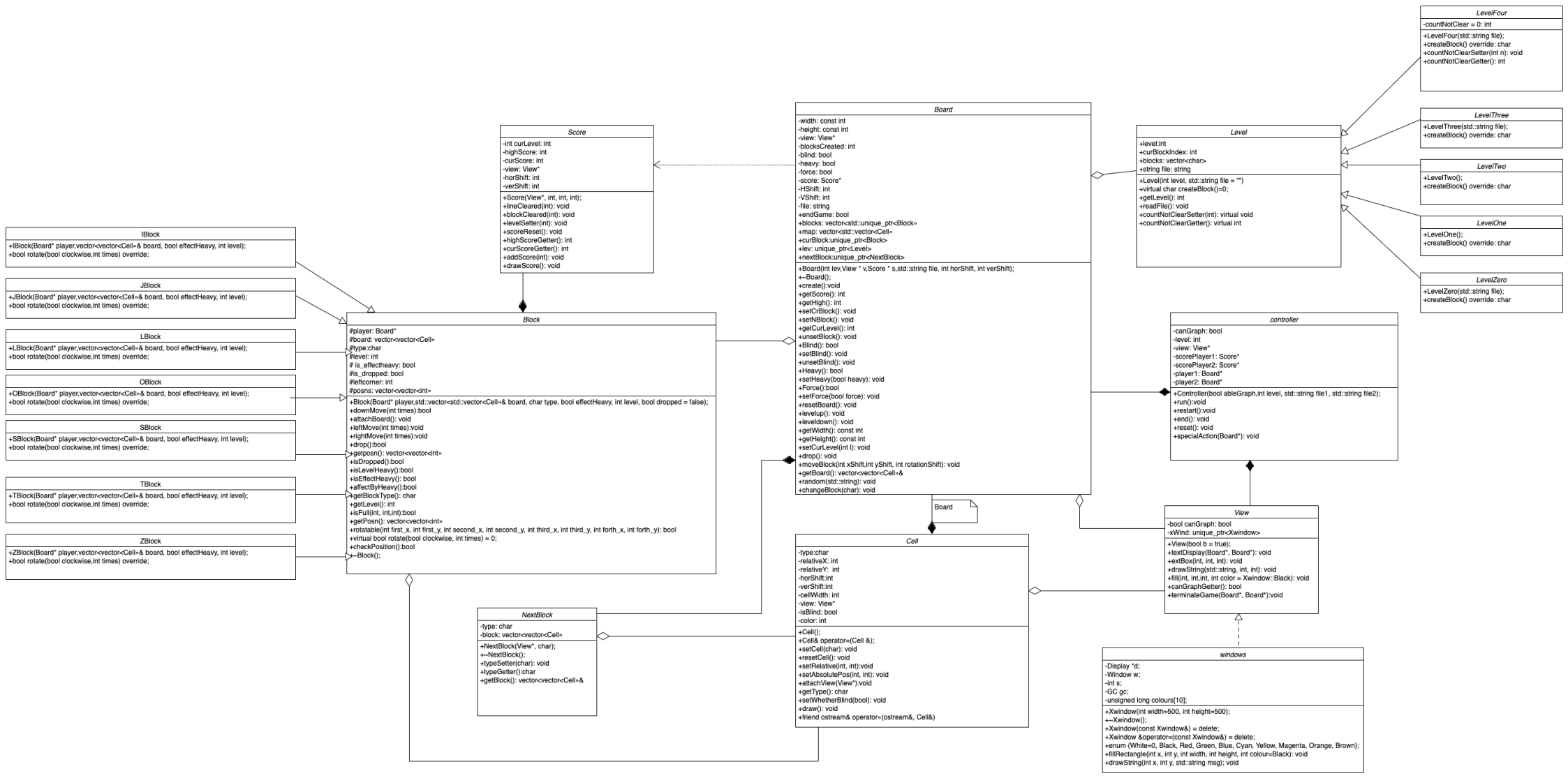
Introduction:

Biquadris is a Tetris-style game for two players. Each player takes turns manipulating a block on one of two boards. A row can be deleted if all of its conditions are met. A block can be deleted if all of its cells are cleared. When a player's block on one of the boards cannot be filled, he or she loses. The number of lines and blocks deleted will be used to calculate your score. Moreover, players can take some special actions during the game to interfere with their opponents' turns.

Overview:

Our implementation of Biquadris follows the Model-View-Controller Architecture

- **Model**
  - Board: Model for players. Contain board, cells, and are manipulated by the Controller class.
  - Cell: Model for a single unit on the board, using the Observer pattern to connect with view, if the status of a cell is changed, it will notify the View.
  - Block: Model for a block of cells, including methods that can move, drop or rotate the entire block.
  - Level: Model for the different levels according to the specifications.
  - Score: Model for calculating player's score depends on how many rows are eliminated successfully.
  - Model classes only contain data, and it has no idea of how to present data to the user.
- **View:**
  - This class displays in both text-format and graphical-format of biquadris game. Some functions inside this class are wrapper methods for the Xwindow class; textBox, as an example, is a wrapper function for window::fillRectangle.
  - This shows the Model-View-Controller architecture where View class handles types of visual presentations to the user; moreover, View is able to access data but it does not know what the data represent and how to use it.
- **Controller:**
  - We use the Controller Class to manipulate the commands coming from either the input or file to the Board class, which is the player in our program. The method run explicitly controls the whole game as it asks for input and then executes the command to call methods as needed. The method specialAction is used to manipulate special actions that may be taken by a player to another player.
  - This Controller shows the Model-View-Controller pattern where there is a component to accept input and translate it for the Model and View classes.


LevelFour
-countNotClear = 0: int
+LevelFour(std::string file);
+createBlock() override: char
+countNotClearSetter(int n): void
+countNotClearGetter(): int
LevelThree
+LevelThree(std::string file);
+createBlock() override: char
LevelTwo
+LevelTwo();
+createBlock() override: char
LevelOne
+LevelOne();
+createBlock() override: char
LevelZero
+LevelZero(std::string file);
+createBlock() override: char
Level
+level:int
+curBlockIndex: int
+blocks: vector<char>
+string file: string
+Level(int level, std::string file = "")
+virtual char createBlock()=0;
+getLevel(): int
+readFile(): void
+countNotClearSetter(int): virtual void
+countNotClearGetter(): virtual int
Board
-width: const int
-height: const int
-view: View*
-blocksCreated: int
-blind: bool
-heavy: bool
-force: bool
-score: Score*
-HShift: int
-VShift: int
-file: string
+endGame: bool
+blocks: vector<std::unique_ptr<Block>
+map: vector<std::vector<Cell>
+curBlock:unique_ptr<Block>
+lev: unique_ptr<Level>
+nextBlock:unique_ptr<NextBlock>
+Board(int lev,View * v,Score * s,std::string file, int horShift, int verShift);
+~Board();
+create():void
+getScore(): int
+getHigh(): int
+setCrBlock(): void
+setNBlock(): void
+getCurLevel(): int
+unsetBlock(): void
+Blind(): bool
+setBlind(): void
+unsetBlind(): void
+Heavy(): bool
+setHeavy(bool heavy): void
+Force():bool
+setForce(bool force): void
+resetBoard(): void
+levelup(): void
+leveldown(): void
+getWidth(): const int
+getHeight(): const int
+setCurLevel(int l): void
+drop(): void
+moveBlock(int xShift,int yShift, int rotationShift): void
+getBoard(): vector<vector<Cell»&
+random(std::string): void
+changeBlock(char): void
Score
-int curLevel: int
-highScore: int
-curScore: int
-view: View*
-horShift: int
-verShift: int
+Score(View*, int, int, int);
+lineCleared(int): void
+blockCleared(int): void
+levelSetter(int): void
+scoreReset(): void
+highScoreGetter(): int
+curScoreGetter(): int
+addScore(int): void
+drawScore(): void
Block
#player: Board*
#board: vector<vector<Cell»
#type:char
#level: int
is_effectheavy: bool
#is_dropped: bool
#leftcorner: int
#posns: vector<vector<int»
+Block(Board* player,std::vector<std::vector<Cell»& board, char type, bool effectHeavy, int level, bool dropped = false);
+downMove(int times):bool
+attachBoard(): void
+leftMove(int times):void
+rightMove(int times):void
+drop():bool
+getposn(): vector<vector<int»
+isDropped():bool
+isLevelHeavy():bool
+isEffectHeavy(): bool
+affectByHeavy():bool
+getBlockType(): char
+getLevel(): int
+isFull(int, int,int):bool
+getPosn(): vector<vector<int»
+rotatable(int first_x, int first_y, int second_x, int second_y, int third_x, int third_y, int forth_x, int forth_y): bool
+virtual bool rotate(bool clockwise, int times) = 0;
+checkPosition():bool
+~Block();
IBlock
+IBlock(Board* player,vector<vector<Cell»& board, bool effectHeavy, int level);
+bool rotate(bool clockwise,int times) override;
JBlock
+JBlock(Board* player,vector<vector<Cell»& board, bool effectHeavy, int level);
+bool rotate(bool clockwise,int times) override;
LBlock
+LBlock(Board* player,vector<vector<Cell»& board, bool effectHeavy, int level);
+bool rotate(bool clockwise,int times) override;
OBlock
+OBlock(Board* player,vector<vector<Cell»& board, bool effectHeavy, int level);
+bool rotate(bool clockwise,int times) override;
SBlock
+SBlock(Board* player,vector<vector<Cell»& board, bool effectHeavy, int level);
+bool rotate(bool clockwise,int times) override;
TBlock
+TBlock(Board* player,vector<vector<Cell»& board, bool effectHeavy, int level);
+bool rotate(bool clockwise,int times) override;
ZBlock
+ZBlock(Board* player,vector<vector<Cell»& board, bool effectHeavy, int level);
+bool rotate(bool clockwise,int times) override;
controller
-canGraph: bool
-level: int
-view: View*
-scorePlayer1: Score*
-scorePlayer2: Score*
-player1: Board*
-player2: Board*
+Controller(bool ableGraph,int level, std::string file1, std::string file2);
+run():void
+restart():void
+end(): void
+reset(): void
+specialAction(Board*): void
View
-bool canGraph: bool
-xWind: unique_ptr<Xwindow>
+View(bool b = true);
+textDisplay(Board*, Board*): void
+extBox(int, int, int): void
+drawString(std::string, int, int): void
+fill(int, int,int, int color = Xwindow::Black): void
+canGraphGetter(): bool
+terminateGame(Board*, Board*):void
Board
Cell
-type:char
-relativeX: int
-relativeY: int
-horShift:int
-verShift:int
-cellWidth: int
-view: View*
-isBlind: bool
-color: int
+Cell();
+Cell& operator=(Cell &);
+setCell(char): void
+resetCell(): void
+setRelative(int, int):void
+setAbsolutePos(int, int): void
+attachView(View*):void
+getType(): char
+setWhetherBlind(bool): void
+draw(): void
+friend ostream& operator=(ostream&, Cell&)
NextBlock
-type: char
-block: vector<vector<Cell»
+NextBlock(View*, char);
+~NextBlock();
+typeSetter(char): void
+typeGetter():char
+getBlock(): vector<vector<Cell»&
windows
-Display *d;
-Window w;
-int s;
-GC gc;
-unsigned long colours[10];
+Xwindow(int width=500, int height=500);
+~Xwindow();
+Xwindow(const Xwindow&) = delete;
+Xwindow &operator=(const Xwindow&) = delete;
+enum {White=0, Black, Red, Green, Blue, Cyan, Yellow, Magenta, Orange, Brown};
+fillRectangle(int x, int y, int width, int height, int colour=Black): void
+drawString(int x, int y, std::string msg); void

Updated UML

During our process of actually implementing the method, we made some changes regarding the UML on due date 1. Here below are the main differences:

1. On due date 1, we did not consider how we are going to grid the blocks, and so we do not have a class for it. On due date 2, we added a Cell class to solve the issue and in this case we can create vector<vector<Cell>> to manipulate the grid as we want.
2. Our Block classes contain only simple move and getter methods on due date 1.When we are progressing, we find that it is needed to add some fields in the superclass, which are thus held by different types of blocks. In order to easily manipulate the various types of blocks, we thus add more methods to the superclass as necessary on due date 2. We also add a rotation method for each type of Block, where we did not consider the rotation on due date 1.
3. We did not include a View class on due date 1, which violates the Model-View-Controller Pattern. So we add a View class on due date 2. Furthermore, we add methods in order to handle both text-output and graphical-output. Including a View pointer in Controller then allows us to display the output as we need.
4. On due date 1, we did not have a NextBlock class. We add it on due date 2 since it allows us to generate the next block type while not changing our current clock and allows the View to know it as it is available.
5. Moreover, we did not have a Board class on the due date 1. However, we find that it is necessary to set a class, which wrapps up all demanded fields for a player and behaves as a model in the MVC pattern. So we add it on the due date 1.
6. We did not consider how the main function is going to manipulate a whole game and we thought of asking commands in main on due date 1. This then becomes inefficient as we want to hold multiple commands.Thus, we then more thoroughly implement the Control class and manipulate the game in Controller class.
7. Score class on due date 1 was not considered to be complex, just containing two fields and the corresponding getters. As we were coding, we found that since the score is related to the level, the board, and the blocks for each player, it is necessary to do the calculations and set current and highest scores in this class for future maintenance. So we added some methods on due date 2 to handle such demands.

Design:

- **Factory Pattern**

We use the Factory Method design pattern in our Level class. We have an abstract class Level and there are Level 0, Level 1, Level 2, Level 3, and Level 4 derived classes which have override method createBlock. Attaching a Level class pointer to the Board class, we can easily create blocks according to the specification of the current level. This provides high flexibility so that we can add many more derived classes for different levels.

- **Model-View-Controller Architectural Pattern**:

Our implementation of Biquadris follows the Model-View-Controller Architecture, we have Board, Block, Cell, Level, and Score as our Model. The controller can control the entire game Flow. And when there are some changes to the status of the board, we use View to present it to the player.

- **RAII Idiom**

Our implementation of Biquadris uses vector, unique_ptr so that our object lifetimes are bound to other objects with dynamic memory, where the destructor will be called and this object will be cleaned automatically. Using unique_ptr prevents us from copying ptrs while we are not aware of that.

- **Polymorphism**

For Block and Level, we define superclasses Block and Level. Each superclass has many derived classes that are inherited from the superclass. In this case we can access objects from different derived classes through the same interface. The superclass has the ability to accommodate multiple specific types under one abstraction.

- **Reduced Compilation Dependencies**

When the inclusion is not absolutely necessary in our program files, we try to not include the header file. Instead we use a forward declaration, which helps reduce our Compilation Dependencies.

Resilience to Change:

Block: Inheritance from the base class

Each type of block is inherited from the base class block, and the method rotate is overridden in each subclass, if we want to add a new type of block, we could add another derived class inherited from the base class.

Level: Inherited from the base class

Each level is inherited from the base class level, and the method createBlock() is overridden according to the specification for each level. If we want to add a new level, we could add another derived class inherited from the base class level and override the createBlock() according to the specification.

Controller:

The controller class is our command-line interpreter, it generally controls the flow of the biquadris game, if there are some changes to the existing command, or if we want to add some new commands, we just need to modify our controller class. By doing so we can also minimize the recompilation.

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?

At the base level, our system will support assigning positions on the board to the generated blocks. In this case, we can track each block's position at any time. In order to allow some generated blocks to disappear from the screen if not cleared before 10 more blocks have fallen, we need to use inheritance since this allows us to have normal blocks and special blocks that will disappear after 10 more blocks fall.

In the derived classes for the basic blocks, we will need to add a count field for each object, which counts how many more blocks have fallen since this block was generated. Each time when a new block has fallen, these derived blocks will have their count field increased by 1. Thus, we randomly generated blocks, in which some will disappear when it has a count field of 10.
The generation of such blocks can be confined to more advanced levels where such blocks disappear if not cleared following different numbers of blocks. For example, some will disappear if not cleared after 8 more blocks have fallen and some will disappear if not cleared after 11 more blocks have fallen. In order to implement this, we will think of adding another field for the randomly generated number where such a generated block will clear itself after this number's more blocks have fallen.

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

Our program will have an level as a basic abstract class, and many derived classes corresponding to the specific level. Our base class will have a virtual method called createBlock(), it can return a block according to the rule of that level, and we override it in each subclass.

When a new level is introduced, we could add a new derived class which contain the method we needed into our program, and when we recompile our program, only the additional levels will be compiled. That is how we minimize the recompilation when introducing additional levels into the program.

- Question: How could you design your program to allow for multiple effects to be 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?

In order to allow for multiple effects to happen simultaneously, we will think of applying a decorator pattern. Since the effects are going to be applied simultaneously, the order of effects does not necessarily matter. Using a decorator pattern to the special actions allows us to add effects to a player dynamically and thus we could achieve applying multiple effects at any time.

When there are more kinds of effects, we just have to add more derived classes for each effect and then the decorator pattern is expected to handle multiple effects simultaneously. In this case, the decorator pattern is also expected to prevent our program from having one else-branch for different combinations.

- **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.**

It will not be hard for us to add a new feature to support a command whereby a user could rename existing commands, We have rename method in controller class, it can bind the new command to a existing command.

To support a "macro" language, we could modify our command-line interface, such that when a name of a sequence of commands is in the input, our program will execute these new commands

Final Conclusions:

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

One of the most important lessons we learned from this project was about time allocation. Having a detailed and well-thought-out plan is more important than we think. This will greatly save us the time and effort of putting together all the code at the end. Everyone has their own exam time, which results in very few opportunities for us to sit together and discuss and study our projects. So most of the time we have to do it alone. Therefore, the division of labor before the project starts, and the complete and detailed plan can play an important role when working alone. For example, a complete uml will help everyone to clarify their ideas.

And, we also learned that git is a very important skill. Because everyone's programs are basically updated at any time, if other members can't get the updated code in time, it will affect his next work. But since none of our members were familiar with Git and working with Git repositories, we gave up on Git after trying to learn it. We share our code in onedrive, so each time we can only download the code of the team members and save it locally, which is a very troublesome thing. Next time we need to learn to use git, so that we can update our code at any time and get updates from team members at any time.

Additionally, we learned that collaborative debugging is more efficient and productive than individual debugging. Everyone has different habits in the coding process. For example, in the eyes of different people, the same method has different names. So it is very important to communicate well before starting. And since everyone will encounter different errors throughout the coding process, collaborating can help us find ways to identify errors, pinpoint them, and fix them in a more timely manner.

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

If there is an opportunity to start over, we will first find an opportunity to sit together and discuss our project. First read carefully and ensure that everyone fully understands the requirements, and then we will make a detailed plan, including the classes we want and the parts that everyone takes responsibility for. Secondly, everyone will arrange their time reasonably and try their best to make sure that everyone is on the right track. When we write our own section, we communicate with other students in the relevant sections to make sure everyone's section is clear and reasonable. Also, we will learn the use of the git, saving us time and effort.