University of Waterloo
Faculty of Mathematics
Cheriton School of Computer Science

Final Design

CS246 Spring 2021 Project – Biquadris

CS 246 - Object-Oriented Software Development

Submitted by

Yi Cai (y237cai)
Eric Li (z589li)
Duoduo Liu (d262liu)

(Academic Year: 2020-2021)

1 Introduction

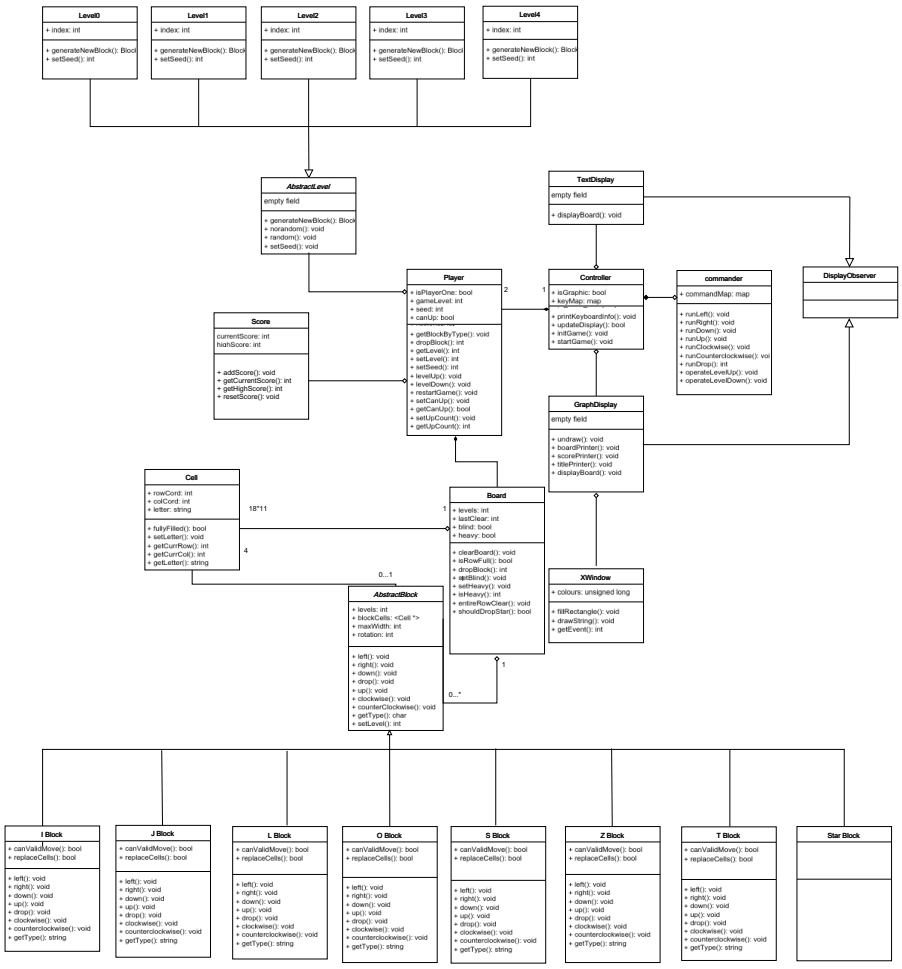
The game of Biquadris is a Latinization of the well-known game Tetris, consisting of two players competition and seven different types of blocks (with one additional star block). A player obtain scores for filling all the cells of one line, as well as deleting all the cells of one block, where different levels significantly impact the amount of score. One player's game is over when a new block cannot be properly fitted on the game board. 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 does not fit, the opponent has lost. Player can choose to start a new game or quit when a game is over.

2 Overview

The overall design philosophy employs the Model View Controller design pattern.

- **Model**:
  The class `Player` is an abstraction of all the models include the `Board`, the `Level`, the `Cell`, and the `Score`. It notifies the View to update the game display.
- **View**:
  The class `DisplayObserver` is an abstract observer class. It is inherited by `TextDisplay` and `GraphicsDisplay` to show the UI of the game.
- **Controller**:
  The `Controller` class is in charge of starting/restarting the game, controlling the game states, and communicating between the **Model** and the **View**. Apart from a default command controller, we also implemented a keyboard controller, which is described in detail later.

The relationship of all classes can be directly shown by the UML and the detailed description of important classes is included below.


Level0
+ index: int
+ generateNewBlock(): Block
+ setSeed(): int
Level1
+ index: int
+ generateNewBlock(): Block
+ setSeed(): int
Level2
+ index: int
+ generateNewBlock(): Block
+ setSeed(): int
Level3
+ index: int
+ generateNewBlock(): Block
+ setSeed(): int
Level4
+ index: int
+ generateNewBlock(): Block
+ setSeed(): int
AbstractLevel
empty field
+ generateNewBlock(): Block
+ norandom(): void
+ random(): void
+ setSeed(): void
TextDisplay
empty field
+ displayBoard(): void
Player
+ isPlayerOne: bool
+ gameLevel: int
+ seed: int
+ canUp: bool
+ getBlockByType(): void
+ dropBlock(): int
+ getLevel(): int
+ setLevel(): int
+ setSeed(): int
+ levelUp(): void
+ levelDown(): void
+ restartGame(): void
+ setCanUp(): void
+ getCanUp(): bool
+ setUpCount(): void
+ getUpCount(): int
Controller
+ isGraphic: bool
+ keyMap: map
+ printKeyboardInfo(): void
+ updateDisplay(): bool
+ initGame(): void
+ startGame(): void
commander
+ commandMap: map
+ runLeft(): void
+ runRight(): void
+ runDown(): void
+ runUp(): void
+ runClockwise(): void
+ runCounterclockwise(): voi
+ runDrop(): int
+ operateLevelUp(): void
+ operateLevelDown(): void
DisplayObserver
Score
currentScore: int
hishScore: int
+ addScore(): void
+ getCurrentScore(): int
+ getHighScore(): int
+ resetScore(): void
GraphDisplay
empty field
+ undraw(): void
+ boardPrinter(): void
+ scorePrinter(): void
+ titlePrinter(): void
+ displayBoard(): void
Cell
+ rowCord: int
+ colCord: int
+ letter: string
+ fullyFilled(): bool
+ setLetter(): void
+ getCurrRow(): int
+ getCurrCol(): int
+ getLetter(): string
18*11
4
Board
+ levels: int
+ lastClear: int
+ blind: bool
+ heavy: bool
+ clearBoard(): void
+ isRowFull(): bool
+ dropBlock(): int
+ setBlind(): void
+ setHeavy(): void
+ isHeavy(): int
+ entireRowClear(): void
+ shouldDropStar(): bool
XWindow
+ colours: unsigned long
+ fillRectangle(): void
+ drawString(): void
+ getEvent(): int
0...1
AbstractBlock
+ levels: int
+ blockCells: <Cell *>
+ maxWidth: int
+ rotation: int
+ left(): void
+ right(): void
+ down(): void
+ drop(): void
+ up(): void
+ clockwise(): void
+ counterClockwise(): void
+ getType(): char
+ setLevel(): int
0...*
I Block
J Block
L Block
O Block
S Block
Z Block
T Block
Star Block
+ canValidMove(): bool
+ replaceCells(): bool
+ left(): void
+ right(): void
+ down(): void
+ up(): void
+ drop(): void
+ clockwise(): void
+ counterclockwise(): void
+ getType(): string

3 Design

Following are the object oriented components deployed in this project:

- **Observer Pattern:**
  There are basically two (subject, observer) pairs in this program. The first is the aforementioned MVC design pattern, where the view is the observer of the models. The other one is that each cell in the board is an observer to the block, because as the block's state changes, the cell update its states accordingly.
- **Strategy pattern:**
  The different levels can be treated as a family of algorithms to generate the next blocks. The strategy pattern encapsulates each one of them, and makes them interchangeable at run time.
- **Polymorphism**:
  Both the `AbstracBlock` along with concrete blocks and the `AbstractLevel` along with concrete levels utilize the inheritance relationship, and thus making it easy to add additional features and minimizing the re-compliance.
- **Single Responsibility Principle**:
  The overall design of our program follows the Single Responsibility Principle, where each of the class serves its own set of functionality. For example, the `Score` class can be easily incorporated into the player class, but we encapsulate it to be a separated class because we probably want to add more scoring rules to the game.

Following are some important classes and their corresponding descriptions.

1. class `Score`:
   This class is used to represent the score of each player, each player has their own score, we the blocks are eliminated, score need to be added to that player.
   Important fields:
   currentScore is used to store the current score that a player currently get.
   highScore is used to count the highest score each player own, if the score updated, the highest score need to documented that score.
   Important Method:
   resetScore(): when the game is over, score should be reset to zero.
   getHighScore(): get the score that a player ever gain for the whole game, no matter how many times game restarted.
2. class `Controller`:
   The class Controller is like a main button that used to control the what the basic game should preform. For example, for a regular game we should initialize it, start playing it, check on time which player are playing now, is the game over or not, do we want to restart the game and extra.

Important fields:
useGraphic: check the input to see whether we need to use graphic display for playing the game or not.

isFirstPlayer: check the player that currently play, if it is the first player(the player that first play when we start the game), then return true, otherwise return false.

isKeyBoard: it is one of our extra feature that we implemented, if the user hope to use keyboard when running the game, it should be true.

isGameOver: check if one of the player have finished the game after the player drop block, if this field is true, then we should finish this game and display which player are the winner.

Important methods:

initGame(): generate the blocks for players to play.

setGraphic() and *getGraphic()*: if player hope to play with graphic display, *setGraphic()* will set *useGraphic* to be true, and we should use *getGraphic()* whenever we hope to check.

changePlayer(): if one player drop the block, use this function to change the player to the next one.

3. class `Commander`:
It is a main class to implement command input. For example, use can move the block to left, right, down, clockwise, counterclockwise if it is within the board.

Important methods:

runLeft(), *runRight()*, *rundown()*, *runClockwise()*, *runCounterclockwise()*:
basic operation for the block to move left, move right, move down, move clockwise or counterclockwise.

runUp(): it is a extra feature we added for the game, if we triggered special action moveUp, then that specific play can run its block totally five times for no matter how much round it have to prevent other actions such as heavy applied to it.

operateLevelUp(), *operateLevelDown()*: up and down the game level.

4. class `Cell`:
This class is represent the cell on the board for player to play. It contains some fields and methods for one cell, for example, which letter is contains, which coordinate it is located and so on. **Important fields:** *rowCord*, *colCord*: contain the cell's coordinate that should be operate. Letter: contain which type of block that should be operate with.

Important methods:

copyData(): copy one row to the following row.

getCurrRow(), *getCurrCol()*: get one cell's coordinate for operation.

setLetter(): set the letter for one specific cell.

5. class `Player`:
   The class's purpose is to provide necessary information for the player. For example, player can setLevel up and down, and also can restart the game and getScore() or getBoard() by calling this function.

   Important fields:

   gameLevel: contain the play's current level of game.

   canUp: extra feature use only, if it is true, then that player can up the blocks.

   upCount: extra feature use only, if the count is equal to zero, that player can not move up.

   Important method: *levelUp()*, *levelDown()*: player can choose the level of hardness of the game to play, and up and down the level for playing by using it. *random()*, *norandom()*: random choose which block to be generated for the player to play.

   restartGame(): player can choose to restart the game by calling this function.

   setLevel(), *getLevel()*: set and get gameLevel for one player.

6. class `abstractLevel`:
   The class abstractLevel o not contain any fields since it is the abstract class, and class level0 to level5 extend that class to generate blocks by obeying the rule of that specific level.

   Important method:

   generateNewBlock(): it is a virtual method and will be implemented and generated new block.

   norandom(), *random()*: both virtual methods are used to generate new blocks randomly by the level rules.

4 Resilience to Change

The implementation of this game is designed to achieved low coupling and hence highly adaptable to changes. The description of core five different classes shows they have difference main functionalities. Hence, changes to any of them will have no significance impact on the others. Consider the additional star block as an example, this additional block is a subclass, and thus it has the individual override function inherited from the same class. Although we add complexity to the block family, we do not need to change other core classes. Also, the implementation of the star block is almost the same as other blocks. Therefore, we can achieve low coupling with each of the block as derived abstract subclass is treated separately. Similarly, we can add additional players with another playing board.

5 Answer to Questions

1. 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?

   Answer:

   We can add a private field `count` in each of the Block class to keep track of the number of rounds a cell is alive, and a public method `isAlive` to check whether it should disappear or not. Yes, the generation of such blocks will be easily confined to more advanced levels.

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

   Answer:

   We can utilize the inheritance relationship between an abstract `Level` class and multiple concrete level sub-classes. So with the abstract class `Level`, which has several sub-classes, namely `Level1`, `Level2`, etc., to indicate different difficulty levels, we only need minimum recompilation (the newly added sub-classes) for introducing additional difficulty levels.

3. Question: 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?

   Answer:

   We can take advantage of Decorator design pattern. The Decorator design pattern is suited to scenarios when we want to add features or effects to an object at run-time rather than to the class as a whole. These features or effects might also be withdrawn. New effects are invented by adding new sub-classes to the decorator abstract base class, not modifying existing code, which reduces the chance of introducing errors. Multiple effects are applied simultaneously by the internal linked list of decoration objects, and thus it prevents our program from having one else-branch for every possible combination.

4. 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.

   Answer:

We can encapsulate our command line interpreter into a `Controller` class just like what we did in assignment 3 question 3. If new command names are introduced, we just add another public method onto the controller class and make corresponding adjustments in our main function, thus allowing minimal changes to source and minimal recompilation (assuming the new command does not require additional implementation on other classes).

To support command renaming, we need to store the default command string in a string buffer, and overwrite the string buffer when the user change the default command to another name.

To support a "macro" language, which would allow us to give a name to a sequence of commands, we can create a `std::map` from the name for a sequence of commands (`std::string`, the key) to the actual sequence of commands (`std::stringstream`, the value) at run-time. If the user input a self-defined sequence of commands, our program look for that command in our `std::map` and execute the sequence of commands by reading from the `std::stringstream`.

6 Extra Credit Features

I. Keyboard Controller:
Since real players are likely to use the graphical interface rather than the terminal text display, we add an additional support for players to use keyboard to control their moves and thus making the game experience better. Table 1 in the following shows the (key, action) mappings.
The `Xlib` has direct support to event listening, where we can check specificly on

Key	Action
←	move left
→	move right
↓	move down
↑	rotate clockwise
c	rotate counterclockwise
[space]	drop
=	level up
-	level down
I	mutate to I block
J	mutate to J block
L	mutate to L block
O	mutate to O block
S	mutate to S block
T	mutate to T block
Z	mutate to Z block
R	restart the game
0	enable/disable multiple special actions

Table 1: The keyboard to action mapping

`KeyPress` events and get the `keycode` of the `KeyPress` event. Then we can refer to the keyboard⟺keycode conversion table (available at `https://chromium.googlesource.com/chromium/src/+/d87e618b08ffd716085acb367c3837cb1ac90ef5/ui/events/keycodes/keyboard_code_conversion_x.cc`) to translate the keyboard inputs into text commands, hence controlling the game.

II. New special action — Move Up:
In order to allow the players to gain more flexibility when they successfully deleted more than two line of cells, we add a new special action, which assign the player 5 opportunities to move the current block up thereafter. This helps players to revert back to some previous position if they mistakenly made certain actions.

III. Informational display:
We add useful printed information both in the terminal and the graphic display. For example, when the game starts, the terminal will tell the players the keyboard instructions; when the player types an invalid command, the game prompts an alert saying "bad command"; when the player is asked to type special actions and an unknown action is received, the game asks for the player to type again; when one round of the game ends, it shows whether there is a draw, or a win/loss. In conclusion, there is no invalid input (e.g., misspelled commands,) that can cause our program to crash.

7 Final Questions

1. What lessons did this project teach you about developing software in teams?
   At the beginning of the project, we developed the project plan as a whole and split tasks for different time periods. When close to deadline one, we met together to discuss the approach to complete the project. Since we have finals at different times, the previously made plan ensures working productively to finish the project in time. Furthermore, all team members must be able to communicate effectively, exchange important information with others, and hold conversations about specific program difficulties. Unlike our personal assignments, we must adapt to our teammates' codes, ask for explanations, make recommendations, and have total faith in our teammates' ability to accomplish their jobs on time. Furthermore, brainstorming with others is more effective and efficient than thinking alone. The benefit of coding together is that debugging is much more efficient, and we recognize the necessity of sharing information when completing this project.

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

   We would have organized our time more carefully during the last weeks and begun the project as soon as feasible if we hadn't been so burdened with assignments and finals. It would also allow us to spend more time debugging, improving the design, adding new features, and writing report documentation. Time management is always a crucial component of a successful project, especially during such a stressful time. Also, we may want to improve the structure with more design pattern and add more features.

8 Conclusion

To summarize, this initiative has benefited every member of our team tremendously. It was the majority of our first experience working on a large-scale project that necessitated tight coordination with the whole team. We not only learned to communicate well, manage time efficiently, and work successfully, but we also learned to put all the knowledge we had gained over the semester into actual coding. As a result, with the knowledge from CS 246, we are better equipped, or at the very least more seasoned, to tackle any future tough tasks. We are glad for the chance to collaborate on this important project.