CS246 Final Project Biquadris

Kefan Chen, Ruoxuan Han, Jingfei Peng

Index Page

Introduction:

Biquadris is a two player competition version of the game Tetris. It consists of a two boards, with each player takes turns to manipulate a block. If a row is fulfilled, the row can be eliminated. When the block of one the of board is unable to fill, the corresponding player lose. Score will be calculated according to the number of lines and blocks eliminated.

Overview:

The structure of our Biquadris follows the Model View Controller Architectural Pattern.

- ### Model:

  Defines data structures. Models update changes to View.

  - ❖ BoardManager(Player):

    Model for the state of players. Manipulated by the Controller classes, it controls the movement of Block class and the corresponding result including elimination of the rows of Cells, the Score class update, and the triggered special actions

  - ❖ Board:

    Model for vector of vector of Cell.

  - ❖ Cell:

    Model for a single square on the Board, using Observer pattern to update changes to the View class

  - ❖ Block:

    Model for a four-cell object that implements a transformation matrix and a rotation matrix to process Block movements

  - ❖ Score:

    Model for the scoreboard that calculates the score of each player, updates the high score, and determine the winner of each round.

  - ❖ Level:

    Model(superclass) for the different levels.

- ### View:

  Defines display(UI) using View class observed from Model classes.
  View produce the display and interact with the user. It serves two purposes: using text to display or using an Xwindow to do graphical display. The function fillcell,

fillstring, and unfillstring is a wrapper for the function in Xwindow, through making it a wrapper, we can have a safeguard inside the function, to make sure the Xwindow is defined before we draw to the screen. It is also easier for other class to call the function. The View class is an observer of the Cell class. After the Cell has been set or unset, it calls a function inside view which modifies the Xwindow. This way, each class has achieved single responsibility.

- Controller:

  We used a Controller and a KeyController class (extra feature) to manipulate the model class BoardManager according to user input. The method runGame is responsible for start running the game. Then it interprets user input commands such as “3lef”, and “0ri” and calls the appropriate methods in boardManager class. Other methods are private. The resetGame() method is used to reset the Model classes after each game ends. The askSpecialAction() method is used to handle user inputs for what special action to use when it occurs. This also demonstrates MVC principal as all parts dealing with user inputs are dealt with in the Controller class. The restartGame() and endGame() are responsible for applying the respectively changes to Model classes during restart and quit process.

Design:

In the design of the project, we used several techniques to solve design challenges:

- Observer Pattern:

  The Observer pattern plays a key role in the Model-View-Controller architectural pattern. The Cell and Board classes are Subjects, and the View class is the Observer. When a subject changes state, View class is notified and updated make make changes to the screen.Each cell inside the board is also an observer to the block class. After the block has shifted, it will notify the corresponding cell to set and unset to store the right state of the block.

- Factory Pattern:

  We defined a separate Level Class which is an abstract class to create the Block object. In our case, we create a char that represents the Block type in order to avoid the return of pointers. There are different concrete subclasses such as Level0, Level1… Those classes implemented their own logic or randomness of creating new Blocks so that it is flexible if new levels need to be added.

- Polymorphism:

  The Block class and the Level class use a single interface to entitle to different types. Apply dynamic dispatch that determines the type of the object at runtime with the keywords virtual and override on these two classes.

- Single Responsibility Principle:

  Our codes are well designed so that every class has responsibility over a single part of the functionality in the project. Block class only handle its own transformation without knowing how how and when it is generated or how view is displayed.

- RAII Idiom:

  We used unique_pointer, vector, and map in the BoardManager(Player) class so that even if something throw, the program won't crash because of memory leak. There is no delete in our code.

- Reducing compilation dependencies:

  We forward declared the View class to the Cell, NextBlock, Board classes to void compilation dependencies(cycles).

- Low Coupling & High Cohesion:

  The degree of module dependencies is low. No friend class and no public fields are used. In addition, with Block responsible for Block movements, Board and Cell(as containers) responsible for notifying the view to update, BoardManager responsible for manipulating the Blocks, Score responsible for calculating scores, Level responsible for creating different Blocks... Each class is responsible for one task to realize high cohesion.

- MVC Architectural Pattern:

  The users will use the Controller class to input commands that manipulate the Model classes, including the BoardManager(Player), Block, Board, Cell, Level, and Score. The Model classes handle the data and update the View class, which presents the Models' data to the user.

Resilience to Change:

- Abstract classes: Level

  The level class is designed as an abstract virtual class. There are currently 5 level subclasses, level 0 to 4 in the game. This allows the possibility to add more potential level subclasses which demonstrates scalability.

- Inheritance hierarchy: Block

  The block class is designed as an abstract virtual class. Each type of block is defined in its own subclass. If we want to add another type of block, we can create another block subclass for this type of block.

- Separate Controller class

  The controller class is responsible for handling user input and set up the game such as initializing the BoardManager class and the Score class. The runGame method in the Controller is responsible for interpreting user inputs and call methods in the BoardManager class to apply changes. If we need to add a new command, we would add one more condition to the runGame method to interpret that command. In addition, if we want to add a brand new input system such as listening to Keyboard directly instead of interpreting text commands, we can add a new method in the controller class to handle it.

- Separate View class

  The main responsibility of the view class is to provide text display and graphical display that interact with the user. Through separating the view class, we can easily add another display if we wish to. Since the View class's only responsibility is to produce output, it only has two fields, a boolean to indicate whether we use graphical display or not, and a pointer to an Xwindow if we wish to have a graphical display. In other class, we have a function that notifies the view class if something has changed, therefore we can minimize the number of times we redraw to the screen.
  In this case, the View class is purely an observer to other class.

Answers to Questions:

1. 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?
Ans: Set a counter in each block class to count the number of blocks generated. If one of the blocks is cleared, reset the counter to 0. If 10 blocks already exist without clearing any row, find the block using cell blockId. Set these cells contained by the block to default(i.e. Make them disappear from the board).
In order to be confined to more advanced levels, we will generate blocks in each level and give each block an indicator from the level.(eg. Every block has an integer field indicating its level) The counter will only increase if it is created by a certain level.

2. How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?
Ans: We can add one more subclasses under our abstract Level class in addition to the level 0-4 subclasses that are already here. The subclass will contain the necessary information for the new level. This achieves minimum recompilation as only the new levelx.cc will be recompiled.

3. 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?
Ans: We can use a decorator design pattern to create subclasses under an abstract SpecialEffect class using decorator design pattern. Therefore, if more than one effects are applied simultaneously, it wraps up the effects to the opponent's Board independently without affecting others. If more effects are invented, we can write more subclasses under the abstract SpecialEffect class. These subclasses will do the respective special effect accordingly to the BoardManager class.

4. 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.
Ans: Use a CommandController class which each new command has its corresponding methods. If one would like to add a new command, just add a new method. Thus, only the CommandController class will be recompiled without affecting other classes. To rename an existing command, we just need to change the if statement. For example, change

```
    if (command == "counterclockwise")
        Into
    if (command == "cc")
```

For example, in our code, we added an extra KeyController class. Since we have a separate Controller class, we only need to recompile the main class. If new condition is added, we only need to modify the if statement.

Extra Credit Features:

- ### Keyboard Listener

  We implemented a KeyController class which extends the functionality of the Controller class with a minimum recompilation on the existing classes(it only needs to recompile the main class). The keyboard mapping will be shown below:

  - "J" for left
  - "K" for down
  - "L" for right
  - "I" for Clockwise rotation
  - "U" for CounterClockwise rotation

- "Space" for drop
- "R" for reset
- "E" for exit
- "S" for skip
- "+" for levelup
- "-" for leveldown
- "H" for heavy
- "B" for blind

To implement the KeyController functionality, we modify the XWindow class by adding a pollEvent function which takes in the KeyPress event from the user and returns the keycode. The KeyController Class listens to events triggered by the XWindow and executes the appropriate reaction to these events.

- Skip a block feature

  A new command "skip" is added to skip a block. The user can enter as minimal as "sk" to issue this command. The command skips the current block and pops up the next block. However, score points are deducted by the (current level add 5).

- ScoreBoard/highScore

  The graphical display and the text display of the game will show who won the game and each player's respective high score when the game finishes and the player chooses to exit.

Changes to UML on Due Date 2:

In the process of coding,, we found some parts of our UML on due date 1 needs to be changed. Here are the main differences between our UML on due date 1 and due date 2:

1. Added observer design pattern for View Class.
   If we use our original design for View/display class on due date 1, we might need to redraw our board whenever there is a move, which makes the UI not user-friendly and slow. Therefore, we implemented the Observer Design pattern for the View class and the Cell, Board, Score and Nextblock classes.
2. Deleted the rotation function in every subclass of the Block class.
   By applying the rotation matrix algorithm, we are able to use one super method for all different Block types. Therefore, the only difference in the subclasses is the constructor, which defines the coordinates of the four cells.
3. Added horiShift and vertiShift field inside Board, Cell, Score, and nextBlock classes to indicate the positions of the board and cell to be displayed inside Xwindow. The relativeX and relativeY fields indicate the relative position of each cell before the horizontal and vertical shifts. We have also added a View class pointer in those classes to allow them to notify the view class directly. We achieved Observer pattern through notifying the View class which position needs to be drawn as soon as the state is modified.

4. We have added a separate nextBlock class, which enables us to notify the View class as soon as we know what next Block is. We have minimized number of times we draw to screen.
5. We have added an extra field called isBlind in the Cell class. This achieved the Single Responsibility Principle since each cell is responsible for knowing its current state and output its correct state.
6. Instead of a fstream pointer, we stored the file name as string in the level. This allows us to switch mode and switch level as many times as we want.
7. Controller now has 1 Score object for each player. We figured out that since the the high score for each player has to be stored every time after a game ends, and the current score for player should be reseted, it makes more sense to have controller rather than the BoardManager class to own 2 score objects. Otherwise, the controller must have 2 integer fields to save the highest score for each player. This still demonstrates The Single Responsibility Principle as the Controller is only responsible for the setup and reset of the Score objects, and all the manipulations of the Score objects are done in the boardManager class.

Final Questions & Conclusion:

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?**
   From this group project, we distributed our responsibilities based on the Model View Controller architecture. We discussed the overall structure together but implemented the classes independently. This strategy is close to the real world programming style which demonstrates the encapsulation principle. Therefore, documentation is really essential in our coding process. Our codes are well documented so that we don't have to understands each other's implementation. This helps us to work effectively and efficiently.
   Besides, our team put the collaborative effort to achieve a common goal with good communications. Even though sometimes we might hold different views on a problem but we figured out a way to solve our conflict: consulting one of the ISAs for suggestions. Over the last two weeks, we have built a trusty team relationship through daily interactions.
   However, we also realized that keeping each team member's program updated is quite difficult. Most of our group members are unfamiliar with Git and using a Git repository so we gave up using Git after some attempts to learn it. To keep each member's file updated, we transfer our files to each other through chat groups on social media whenever there is a change. Although we are able to do this through good communication, I could see how difficult this is despite we are only a 3 members group. So I learned that being able to use Git is an important skill when developing software in teams.

2. **What would you have done differently if you had the chance to start over?**
   Overall, we did a great job on this project. We have worked effectively as a team. However, we can make our work even more efficient by learning git and using it in

our project. This will not only help us to have better management over our files, but also serve us in future as Git is a very necessary skill in the workplace.