Final Project

CS246

Anjali Gupta

Isshana Mohanakumar

Jathurshan Theivikaran

Table of Contents

Introduction

Biquadris is an interactive game that is quite similar to the standard Tetris game. Specifically, it is a version that incorporates multiplayer functionality, where it consists of two boards for each of the two players. Within those boards, players will see blocks appear from the top, which they can strategically drop on their turns to not leave any gaps. Once a row has been filled, the horizontal line disappears. The score in this game is calculated based on the number of lines erased. However, in Biquadris, there is a catch! There is no time-limit for each player to settle each block and the player turns alternate after a player drops a block.

Overview

This is the structure we have decided to implement for Biquadris (refer to the UML diagram for more details on methods):

- **Classes**
  - **Board:** The board will be 11 columns wide and 18 rows high and that keeps track of the score and high score. The board will be made as a vector of a vector of cells.
  - **Cell:** The cell will represent a square on the board. We will be using the observer design pattern to keep updating the display of the board based on the cells.
  - **Blocks:** We will have a block decorator that will extend to the seven different types of blocks. Each block will also have its own rotation and transformation changes.
  - **Levels:** We will have a base class of Level which will extend to the five different levels. Each level will then contain the properties of each level (e.g. probabilities).
  - **Command Interpreter:** This class will focus on handling all the commands and reading in input.

- **Display**
  - **Text Display:** This will be what the user sees. This display involves text. It is an observer of cell class. Once the cell changes, it will call a function which modifies the display.
  - **Graphic Display:** This will be what the user sees. This display will involve the Window class (graphical display). It is an observer of cell class. Once the cell changes, it will call a function which modifies the display.

- **Design Patterns**
  - **Observer Pattern:** Cells, Text Display, Graphical Display
  - **Decorator Pattern**: Boards
  - **Factory Method Pattern:** Levels

Design

As a group, we ensured to design our game strategically and implement design patterns, so that we could identify issues and conveniently construct algorithms in the development of the program. By this approach, we ensure that the reusability, readability, and maintainability of our code is effective for us when developing the program.

- **Low Coupling**
  - Low module dependencies
    - Unrelated modules are not included in each other
    - Only a few friend classes are used
  - Used **encapsulation** by making access specifiers of all fields private or protected only, not public
    - Prevented unrelated classes from easily accessing fields from another class
    - For some classes with private fields, we added accessors and mutators, for other classes to access the values responsibly, not accidentally

- **High Cohesion**
  - Used **Single Responsibility principle**. Every class has a specific functionality and unique purpose for the program
    - Board notifies and updates the game board
    - Cell notifies board when the cell is filled or not
    - Level creates different blocks for each level
    - Block moves the block
    - Text display outputs the game to player

- **Polymorphism**
  - Share common fields or methods among various objects. Used dynamic dispatch by using virtual keyword in abstract class and override keyword in inheriting classes
    - Blocks inherit from abstract block class to create multiple types of blocks. Dynamic dispatch establishes the type of the block object, at runtime
    - Levels inherit from abstract level class to create multiple blocks for the game. Dynamic dispatch establishes the type of the block at each level, at runtime

- **Observer Pattern**
  - Observers resemble the View in the **Model View Controller pattern**
  - Cell class is observer to block class
    - When the block moves and is placed on the board, it will notify which cells it fills to update all cells in the board
  - Display classes like Text Display and Graphics Display are observers to the board and cell classes

        - When the board is updated and the filled cells of the board is changed, it will notify and output the game board, appropriately
- **Factory Pattern**
    - Concrete subclasses inherit from abstract classes to add more functionality for that subclass in addition to abstract class features
        - The abstract virtual level class and block class creates the block object for the level and moves the block object on the board, respectively
        - LevelZero, LevelOne, LevelTwo, LevelThree and LevelFour are concrete subclasses of the abstract level class to create a block with a specific type for that particular level
        - IBlock, OBlock, SBlock, TBlock, JBlock, ZBlock and LBlock are concrete subclasses of the abstract block class to move that specific block with a designated type in a particular manner

- **Decorator Pattern**
    - BoardDecorator is a decorator class for the board class
        - When the cells on the board change states and the board updates, it is decorated by another board with the updated cells
        - Allows us repeatedly adjust extra features like blind, heavy and force on the current board
- **Resource acquisition is initialization Idiom**
    - Ensured that the program would not crash and terminate due to a memory leak on the heap or failure to delete allocated memory
        - Used smart pointers primarily in main and boardDecorator implementation files
        - Used vectors In various classes, but especially the board files

Resilience to Change

- **Scalability by Abstract Virtual Classes**
    - The abstract level class and block class uses factory method and can be inherited by concrete subclasses which will have specific added features and the primary function of the abstract class
        - Presently, there are 5 concrete level subclasses from level 0 to level 4, but we could easily implement more levels like level 5
        - Presently, there are 7 concrete block subclasses, but we could implement another type of block by adding another block subclass

- **Scalability for User Interface**
    - The commands that are inputted by the user are read and interpreted in the command handler where it uses conditions to decide what the program should execute

        - By adding more of these conditions and implementing new methods in the board, cell, and block classes, we could add additional valid commands for user and functionality for program
        - By changing existing conditions in the command handler class, we could change the behaviour of the program and this is class is convenient for us to alter the logic of the code

- **Change Display Strategy**
    - The text display class is designed to carry out all functions to output the game to the user through the console
        - Since the output of the program is focused in this one class, we could change the logic in the class to change how the output is formatted
        - Similar to the text display class, the graphics display class has the functionality to output the game, but through a window, instead
        - Other display classes could be created to change the manner in which the game is outputted and shown to user (e.g. we could create a class that describes what is happening on the board for each turn, instead of showing the actual board)

- **Change Inputting Strategy**
    - The command handler class is developed to perform all operations to consume and undergo complicated analysis of the user input
    - Typing the commands and entering could become tedious to move the block a long distance, so now, by using the logic of moving the block in the particular direction, we could easily add an implementation to change how input is interpreted
        - This class already has the logic for the right, left, down, and drop commands, that is, after acknowledging which movement to perform, it manipulates the blocks, cells and board accordingly to execute it
        - We could duplicate the logic to implement a keystroke input listener where when a key is pressed, it performs those commands. Specifically, when ‘A’ is pressed, the block moves left, when ‘S’ is pressed, the block moves down, when ‘D’ is pressed, the block moves right, and when the spacebar is pressed, the block drops
        - By a similar justification, we could replace the stopgame command with the backspace key, replace the restart command with the escape key, and so on since we already have the logic to clear the board

Answers to 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?

In the block classes, a variable can be instantiated to keep track of this. This counter variable can be initialized at zero when the block is first generated and incremented by one each time a new block is added to the board. If the block has not been cleared by the time the counter has the value of 10, it will be cleared off the board. Every time a block is added to the board, the board will be checked by every cell individually to either update the counters, remove completed rows or remove the block if the counter reaches 10. If a block (or parts of a block) is removed from either completing a row or reaching the count of 10, the cells containing these would have their value set to empty (i.e. reset to the default value). To confine the generation of blocks to more advanced levels we can implement the counter variable to be set inside the board class when generated in the corresponding levels.

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

We can design our program to include subclasses that extend from an overall base class. As shown in our UML diagram, this has already been implemented with levels zero to four. The additional levels can be implemented in the same way and contain the necessary information needed for the specific level. Since we have separated into its own class it will result in minimum compilation as only the .cc file associated with the newly added level(s) would have to be recompiled.

How could you design your program to allow for multiple effects to be applied simultaneously? What if we invented more kinds of effects? Can you prevent your program from having one else-branch for every possible combination?

The decorator pattern is a great way to add on extra effects to components at runtime. An example of this can be seen in our UML diagram for the special actions. Each action (blind, force, and heavy) has its own subclass which is derived from the AbstractBoard class. As each special action is created, the board is wrapped with these additional subclasses, with each one applying its corresponding effects. If we invented more kinds of effects, they will be added as subclasses that extend from the Abstract class (i.e. special actions would be created in the same way that heavy, force and blind are made). The described implementation removes the need to have if-else branches for every combination as the effects operate individually of each other.

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.

Our system uses a command interpreter class (as shown in our UML diagram) which will take care of all the commands a player can enter. The main handler will contain either a switch statement or if-else statement which will call specific methods depending on which commands the player enters. This promotes minimal recompilation because only the command handler class and classes associated directly with the newly edited/added command would be recompiled. It will not be difficult to rename existing commands (internally) as only the if-else or switch statement would have to be changed.

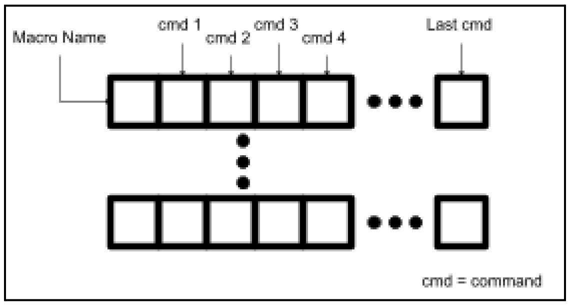
An example of renaming commands internally is shown below:

```
if (cmd == "counterclockwise") {
    ...
}
```

```
if (cmd == "cc") {
    ...
}
```

Let's say the player wanted to add a custom command. We would implement this feature by creating a vector of strings which contain the new command that the user would like to add. It would be a two dimensional vector that contains the new command and the original command it refers to. When they are specifying this command, it is required for them to also specify the original command it is supposed to refer to. For example, let's say that a user wants to add a command called "cc" to reference the "counterclockwise" command in the game. The user would have to enter the command as `addcmd <original command name> <new command name>`. For example: `addcmd counterclockwise cc` would be valid (as long as "cc" was not specified as another custom command). Additionally, it promotes recompilation as only the command handler class would have to be recompiled.

We can make our project to support a macro language by also using a two dimensional vector as specified in the last paragraph where the first index contains the macro name followed by the sequence of commands that macro should execute. Thus, after the user calls a macro, all the commands associated with that macro will be executed in the order specified. A diagram is shown on the right.



Extra Credit Features

- **Adding Command References**
    - We have implemented one extra feature, which allows the addition of new command names, or changes to existing command names, with minimal changes to source and minimal recompilation.
    - For example, after the implementation, the user can type “lef”, for the left command or “do” for the down command
        a) We achieved this by creating a vector of main commands, and one for all commands.
        b) This allows the user to simply enter a command by typing “rename <old command> <new command>”
        c) The allCommands vector stores the new command with the index of the command it refers to (index is based on the main commands vector)
        d) After doing this, a filter function verifies that this new command does not conflict with any other commands, and if it does not, the new command is added

- **Repeat Command (Proposed implementation)**
    - The command, rep or repeat, is a user input without any flags and magnifiers that executes the previous entered command
    - For example, if the user types “rep” at the start of the program, nothing occurs since no valid previous commands have been entered, but when they type “levelup” then “rep”, it levels up one more time. Typing rep again would level up again.
        a) In the board class, we added a field named “previousCommand” with an initial value of an empty string
        b) Prior to executing any command, prevCommand is set to that command. An exception is that when the user had entered “dro” to specify the drop command for instance, prevCommand would store the primary reference to the command, “drop”
        c) Let’s consider that the “repeat” command or another reference to this command, created by the rename command, is entered. Now, if previousCommand is an empty string, then nothing occurs and the user is prompted to enter another command. Other, the previousCommand has the value of a valid command, and in this case, it executes that command.
    - Essentially, this allows the user to repeat processes multiple times instead of having to give a magnifier as for example, instead of entering “3drop”, the player could also enter “drop”, then “rep”, then “rep”, which results in the same board at the end of it. This applies to various commands, not just “drop”, and more significantly, this allows the player to see the board after each drop. Moreover, if the player wanted to execute the command of their previous turn, simply typing “rep” would achieve this.

Final Questions

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

This project was an insightful opportunity and demonstrated how we could effectively develop large scale software in the real world, collaboratively. To begin, like we would in an actual software development environment, we designated specific tasks and classes for each member to emphasize on, and since we were all contributing different aspects of the program, simultaneously, it was essential that we consistently tested the program to behave as intended, throughout the development. We created functional tests to ensure that the program is running appropriately and after adding our updates, we tested the program with regression testing to ensure that the update does not impact other parts of the program. Moreover, we determined the need to constantly communicate with each other and request clarifications with our collaboratively decided implementations and inform each other on our progress, which helps us understand our responsibilities more thoroughly, acknowledge where we are in the development, and diligently work at a faster rate. It was crucial for us to be organized and plan thoroughly because otherwise, a member could get lost quickly or miscommunication could lead to a member working on the wrong part of the project. The Unified Modeling Language Diagram helped outline what we must do for the program and how to connect our program. In fact, we arranged meetings every day for the last few weeks to plan and collaboratively work on the project. Also, each of the team members had different thought processes and approaches to creating the board and moving the blocks, and by building on each other's ideas, we were able to derive the most optimally efficient solutions by combining our ideas. For example, one member suggested using methods in the block class to notify the board class, and another determined that we could essentially use the observer design pattern.

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

Each of us had sufficient knowledge of Github and Visual Studio Live Share, and effectively applied it to share code and collaboratively work on the same file, simultaneously. Instead, we could have put more emphasis prior to the actual development of the code to learn more about these tools and other commands at a deeper level. The platforms allow us to conveniently update code so that we do not have to manually send every updated program continuously, so a better understanding of git would definitely assist with our time management. Furthermore, we were generally satisfied with the implementation of the program, but one way we could have approached it is by inefficient or redundant code. There were some instances in the development where we found repeating code that was not necessary since two members may write the same code. Hence, instead of looking for these instances after coding, it would be more convenient for us to acknowledge the newly implemented updates first. Moreover, although we did add documentation, we should have put more effort into summarizing documentation for every block of code. This informs the other members what that block accomplishes and that it has been tested thoroughly, so that we could spend less time trying to interpret what the code does. In addition, we should have first implemented the smaller classes like the cell class, that other classes depend on and then work our way towards creating the more important, bigger classes like the board class. For example, instead of trying to create all classes at once, randomly, we should have followed a process where we developed the cell class, then block class, then board class. This would be much easier to implement the bigger classes at the end, since the smaller classes would already have been created, which could be referred to, and we could inherit from the working smaller classes as well. Randomly working on different class functionalities clearly was not the most time efficient approach, but we did manage to successfully implement all classes, as desired.

Conclusion

Overall, this was a great learning opportunity, to apply the knowledge that we acquired in this course, and work with others with different techniques and understanding. In fact, it was a pleasurable experience to learn from each other and build a very interactive game, which is a fulfilling accomplishment. In addition to the techniques, patterns, structures, and algorithms already incorporated, we hope to add more advancements that we will be learning. In fact, we hope to deploy this C++ program as an iOS or a native application, one day in the near future.