1. Introduction

Biquadris is a spin off from the ever-popular game Tetris, using text input instead of a graphical user interface. Most of the functionality remains the same, as two players compete to strategically drop shaped blocks onto their board in order to gain points and put obstacles on their opponent's board. This version was built for the University of Waterloo's CS246 course, by Ethan O'Farrell, Botoo Wei, and Andy Yang.

2. Overview of the code

Our game was built centered around the Model-View-Controller design. When a user enters a command, it is processed by the controller which then calls the correct method in the model (what we designated the Game class), before telling the View (Drawers) to update the text or graphical output.

The Game is comprised of two boards, one for each player, and each board contains a list of all the Blocks (tetrominoes) dropped onto it, as well as the current level assigned to the gameplay. When a method is called in Game, it processes which player's turn it is, then calls that command on the correct board, such as, "move the current block to the right," or, "drop the block."

To abstract the functionality of the blocks themselves away from the board, a Block is simply a class containing the locations of four pixels, 1x1 blocks, in the xy-plane of the board. Specific types of blocks inherit from this Block class, and more specifically define the locations of these pixels in relation to each other. Calling a method on a block generally only mutates the locations of these pixels. Similarly, a level has one purpose, to tell the Board which type of block to spawn, as block generation can be pseudo-random or follow a list predefined by the client.

3. UML

This would create a bit of a mess to put the UML here, so please see the Appendix.

The largest change to our UML came once we started designing the Decorator pattern for the special action boards, as we realized our original plan would not work. The solution was to rearrange which Board methods ended up in which class, and incorporate the abstract Decorated Board class.

We also decided to use the pImpl idiom for the implementation of the View (Drawer) classes, in order to hide the private implementation of the methods from the displays themselves.

4. Design

Throughout the project, we stuck to well-versed practices of object-oriented programming while also utilizing several design patterns.

The Single Responsibility Principle states that each class should have exactly one task, and the vast majority of classes in our design follow this standard. For example, the Controller's only job is to process a user's command and invoke the appropriate function. No additional bells and whistles are included. Similarly, the only task of a Block is to keep track of exactly where on the board each of its four pixels should go. For our implementation, Blocks are passive. That is, when called upon, they only change state, and never call further functions. Levels follow a similar pattern, as their sole purpose is to generate a block type based on the current level of difficulty. It is important to note the distinction between "generating" and "spawning" a block. A Level will generate the block; it will tell the board which block to put in the pile, but it won't actually do the spawning itself.

The most prevalent design pattern we used was the MVC (Model-View-Controller) as discussed in section 2. This allowed us to very easily separate the back-end functionality from the user input and output interfaces. Along with the MVC came a modified version of the Observer pattern. The View was never able to change the state of the Model, but it did could call accessor functions to read the game state. To tell the View to update, whenever a function that changed the state of the board was called, the Controller would tell the View to update. We left the updating to Controller because the displays must update after each function, so there would be much repeated code if that responsibility was left to the Model.

To spawn a block, the Levels used the Factory Method pattern. Every type of block inherits from an abstract Block class. The board could not know ahead of time which type of block was about to be generated, so to get around this, it would call the spawnBlock() method and expect a Block in return. Of course, the spawnBlock() method uses functionality that is overridden in each Level, and would return a specific block, such as the O or the Z block, to the board.

We utilized the pImpl idiom in order to provide a compilation firewall and have a cleaner design within our Drawer class (the View in the MVC). By making the player structure a pointer implementation, we've made it such that if the private implementation changes, the client code doesn't have to be re-compiled. This saves time and effort in the long-run. Additionally, making the player struct a unique-pointer in the Drawer class allowed for a cleaner design in our MVC and Drawer class method implementations.

We used a slightly modified version of the Decorator pattern to apply special effects to the boards. A Normal Board represents the basic functionality, while decorated boards include Blind, Forced, and Heavy boards. Applying the decorator pattern allows us firstly to add more special boards to the program fairly easily, but secondly and more importantly, we can layer special boards. If in higher levels, clearing two or more rows allows the user to select two special actions to apply, we can easily implement this using the decorator. Just like decorating a basic pizza with cheese first and pepperoni second, we can decorate a normal board with a blind board first and extra heaviness second, or any combination in any order. Where we deviate from the standard Decorator pattern was that we want to be able to remove a special board without losing the underlying normal board, so our implementation uses aggregation relationships instead of composition.

We did surprisingly well to stick to our plan of attack, and much of our implementation reflects the original plan. The one notable exception was that we originally planned for Levels to take care of scoring; in the end this fell to the boards. This was a better decision because it maintained our strong single responsibility in the Level classes. In terms of the timeline, we met all of our ambitious deadlines, and as planned, we finished the code itself two days early.

5. **Resilience to change**

By separating the classes doing the heavy lifting, we were able to maintain good examples of high cohesion and low coupling. For coupling, we only had one friend method in the entire system, used to print the displays. Instead of storing the state of the boards in public fields, we kept the fields private and instead used public accessors so that the display could see what the boards looked like, and we carried this strategy throughout all of our classes. That way, the only coupling between classes comes from composition, aggregation, and accessing data through public methods. Never does a function explicitly edit the private fields of another class.

We also maintained strong cohesion throughout the program, as each class was designed such that its fields and methods together served a single purpose. As we have discussed, the Level class is a fantastic example of this. It's sole public method was to return the type of the next block to be generated, and the 3 important functions this task is built around, spawnBlock, spawnRandom, and spawnNorandom, only play one role- to decide which block to generate.

Many of our core classes were abstracted to make way for future development, including the View, Boards, Blocks, and Levels. Because we also stuck to the single responsibility principle very well for these classes, building new ones, such as adding a new level or a new type of block, is trivial. Even better for us, we implemented a function to rotate our blocks using linear algebra, instead of hardcoding the rotation mappings into each specific block class. Similarly, the only variance between levels is the probability tables, so building a new level takes no more work than a copy-paste and changing a few numbers. As a demonstration of this, we added half a dozen special blocks to the game, including a donut shape, a diagonal line, and a random scatter block, along with a 5th Level where these funky blocks can be spawned. If we wanted to introduce these blocks to the standard levels 0-4, it would be as simple as editing the probability tables for block generation.

We abstracted the View (drawer) class to make it easier for us to use both a text display and a graphical display, but this design allows for future expansion. If we wanted the game to be deployed on a server and the graphics would be sent to different clients on different operating systems, we would only have to add a few classes such as ViewWindows, ViewIOS, etc, to support compatibility across many operating systems, instead of having to process what type of system the display was for every time it updated.

As mentioned previously, we used the decorator pattern for special boards to allow us to easily add even more special boards down the line, or even layer them on top of one another. For example, we very easily implemented an Upside Down special action to disorient the player, as well as a special action that doesn't allow block rotation.

6. **Answers to DD1 questions and account for differences from DD1**
    a. **How would your design allow for some blocks to disappear from the screen if not cleared before 10 or more blocks have fallen? Can this be confined to certain levels?**

Because our blocks types all inherit from the abstract Block class, we can add another abstract block type that inherits from Block, called DisappearingBlock. New blocks that inherit from DisappearingBlock can have a timer or lifetime field that tracks how many turns it has left before it disappears, and during every pass of the list of blocks in a Board, this field can be decremented by one. To confine this behavior to higher levels, we simply need to change the spawn rate of disappearing blocks in levels 3 and 4, or whichever levels are desired.

    b. **How could you introduce additional levels?**

All of our levels inherit from the abstract Level class, so adding new levels, such as Level69, is as simple as inheriting from Level and overriding one virtual method that we specified. This method acts when the level is in random generation mode, and randomly selects a block type for the board to spawn. Therefore we can add levels without changing the base functionality of the preexisting ones. This implementation is an example of the factory method pattern, because the board doesn't know what type of block spawnBlock() will return, so it polymorphs a concrete block into the abstract Block type.
This slightly deviates from our plan of attack because we ditched the scoring system in Levels and left if for another class to take care of, maintaining our single responsibility principle.

c. Can you allow for more than one special effect to be applied simultaneously or introduce a new special effect entirely?

Special effects are treated as decorators to boards. Because of the versatility of the decorator pattern, it is very easy to apply multiple effects simultaneously. A decorated board points back to the abstract top level Board class, instances of which can either be another decorated board or the Normal Board. For example, if both Heavy board and a Blind board is applied at the same time, calling a method on the Blind board will execute the necessary code, then call the same method on the Heavy board, which will execute the necessary code before calling the baseline method on the Normal board. Following our common theme, creating a new special effect is no more complicated than inheriting from the abstract Decorated Board class and overriding the relevant methods. Adding such an effect requires next to no recompilation of the existing program.

d. Can you accommodate new or changed command names? How could you allow the user to rename commands during runtime? How could a set of macros be implemented?

Our Controller class handles the user commands. The input string is checked against a constant list of strings to see that it represents a valid command, and then is executed if that command passes. Adding or changing a command simply requires adding to or changing the list of valid commands, then editing the correct code block in the execution phase to match the new name. This could also be done during runtime, if the user wants to change the name of a command mid-game. A rename() command can be implemented to change the name of any valid command that the player desires. If the player wants to add macros, these can be stored in a separate data container from the list of valid commands. If a defined macro is called, its command list can easily be run through the standard execution method in Controller. Although many of these actions

could change the behavior of command shortcuts, the user would have to be aware of these and rename their commands accordingly.

7. Extra features

We've implemented a few extra features into our rendition of Biquadris in order to improve user experience and program efficiency and design. Firstly, we've omitted the use of keywords new and delete throughout our entire program. This means we did not explicitly heap allocate any arrays or pointers. By doing this, and using unique-pointers as a substitute, we've nullified our margin of error for memory leaks. If we didn't explicitly allocate memory, we also aren't responsible for freeing it. This reduces human error and allows for cleaner code overall.

To help the player get accustomed to the game, we've also added a help menu that displays the possible commands and rules of the game. This screen is accessible at any time during the game to the player should they require a reminder.

We've also introduced extra block types on top of the originals in Biquadris. We believed that given a greater amount of blocks, we would have more to work with in terms of ramping up levels and creating a better experience for the player as they would have to learn to deal with a greater variance of blocks (from sample playthroughs, we've realized that the original game is a bit too easy). This leads directly into our implementation of an additional level 5 which utilizes the aforementioned extra blocks. The addition of a harder level also adds to user enjoyment and serves as a rewarding challenge for committed players.

To make life easier for players, we implemented an additional input feature which allows for players to type in a portion of the command, and our controller interpreter will process it into the intended command. Shortcuts such as typing 'r' for right, 'l' for left, and 'd' for down (instead of drop) will increase quality of life for users and made testing easier for us as well.

Lastly, we implemented a 'store run' feature that allows for any game to be recorded and saved beyond the scope of the run. This allows for players to optimize their playstyles and reflect on past games. Additionally, this feature also enabled us to test the game with greater efficiency as we could save runs that broke the game.

8. Final questions

This project taught us about the importance of communication in development in a team setting. In any group project, it is integral for individuals to work in conjunction with one

another in order to achieve a common goal. In our case, as we were all roommates, face to face communication was easy. However, we soon learned that version control was a necessity in group software development. Using git and GitHub was a massive improvement, and allowed us to save time in understanding others' code, and combining it.

Additionally, development in a team setting allows for a substantial improvement when it comes to debugging. Having extra sets of eyes proved to be so invaluable that we started having group debugging sessions to speed up the process.

If given the opportunity again, we would have increased program cohesion and put more emphasis on planning. In terms of cohesion, we specifically should have seperated scoring from blocks. This is due to the fact that each class should have a specific purpose, and thus scoring shouldn't have been contained within the Blocks module as its purpose was to keep track of each player's score- something unrelated to Blocks. Greater cohesion in our modules overall would have allowed for better readability and an easier time implementing additional features.

In terms of planning, given the chance, we would have made a more concrete schedule while accounting for external factors. Due to midterms and assignments surmounting, each group member had different amounts of time they allocated to the project. As a result of loose planning, certain group members were on different pages at multiple points throughout the project. This caused confusion amongst us and led to some idle time for members that were ahead as they were unable to continue without the work of others. A stricter schedule would have pretended this.

I. Appendix

A. Section 1- UML

This image is hazy, please reference the uml.pdf file submitted to Marmoset for a clarity version.