CS 246 Biquadris: Plan of Attack

Daniel Zhao, Edward Wang, Yiyan Huang

Schedule

Date	Description	Delegations
11/20/2024	**1. Discuss project details:** ● Read over biquadris.pdf ● Read over project_guidelines.pdf **2. Brainstorm ideas of implementations/functionality** ● Decide classes to implement ● Discuss class relationships ● Add public class methods and members **3. Start drafting UML diagram**	All group members
11/21/2024	**1. Revise brainstorm (to match UML)** **2. Fill in function information** ● Decide any algorithms to use (what functions to use) ● Decide any virtual/pure virtual functions to use ● Decide access modifiers for each field in the classes **3. SUBMIT: Finalize UML diagram to reflect above changes** **4. SUBMIT: Complete plan of attack document** **5. Delegate classes to implement within group members**	All group members **Current planned delegated tasks:** ● **Daniel** - Player ● **Yiyan** - Shape and Level ● **Eddy** - Game and Studio
11/22/2024	Group coding session #1 (ONLINE) ● 2-3 hrs ● Meeting with Discord Special Notes: ● Create Github repository for collaborating working remotely ● Work on each part (making sure to avoid editing the same file) - create modules	Each group member works on their dedicated section

11/23/2024	Group coding session #2 (ONLINE) ● 2-3 hrs ● Meeting with Discord Special Notes: ● Work on each part (making sure to avoid editing the same file)	Each group member works on their dedicated section
11/24/2024	Group Check-in #1 ● Each group member summarizes their progress (what they have achieved, what they still need to implement, any blockers/questions) ● Potentially, if implementations are complete, can start connecting modules together to create a basic running program ● Example of basic program: ○ Able to rotate/move all 7 blocks ○ Able to render blank Studio ○ Able to output sample of text interface ○ Able to interpret input commands and command-line arguments	All group members
11/25/2024	Group coding session #3 (ONLINE) ● 2-3 hrs ● Meeting with Discord Special Notes: ● Work on each part (making sure to avoid editing the same file) ● Work to finalize delegated tasks	Each group member works on their dedicated section
11/26/2024	Group coding session #4 (ONLINE) ● 2-3 hrs ● Meeting with Discord Special Notes: ● Work on connecting each individual part (solve any dependency errors/import errors) ● First draft of final product	All group members
11/27/2024	**1. Adding Special Actions:** ● Add functionality for blind, heavy, force (dedicated each group member for one special action)	All group members **Dedicated special action**:

	<ul><li>Add checkers for special actions (checks if you have completed at least two or more rows simultaneously)</li></ul>**2. Adding Graphical Interface:**<ul><li>Create graphics class (figure out how we output different coloured shapes)</li><li>Add interface for user to select game level or game reset (potentially a bonus feature)</li></ul>	<ul><li>**Daniel** - Force</li><li>**Yiyan** - Blind</li><li>**Eddy** - Heavy</li></ul>
11/28/2024	1. Focus on adding bonus features: 2. Finalize updated UML diagram 3. Start working on final report	All group members
11/29/2024	1. Finalize final report 2. SUBMIT: Executable 3. SUBMIT: Final report 4. SUBMIT: Final UML diagram	All group members

---

Project Specific 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 studio class, we could add an **int time** variable that keeps track of the current time (the number of blocks placed). In addition, we could also add a vector that keeps track of the shapes that have been played. After every turn, we can iterate through the vector to determine whether we should remove a shape or not.

Within the Shape class, we could also add another time variable that keeps track of when we need to remove it (the lifetime of that block). This variable will be checked after every turn. To remove the shape, since we have access to its coordinates on the gameboard (2D matrix), we could iterate through it and set every element equal to a whitespace. After, we would call the gravity function to let other shapes fall into the proper place if needed.

This feature could also be adapted into more advanced levels. For instance, instead of blocks disappearing in 10 block places, we could increase the time threshold to 20 or even 50 to increase the difficulty of the levels. We could also add a feature where the type of blocks that have disappeared are more likely to appear in the user's next block (changes up probability of blocks and limits variety).

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

To minimize recompilation when adding new levels, we could use the Factory Method Pattern, where we would have a base class called "Level" that acts as an interface to how every level would look/behave. We would create Level subclasses to add behaviours unique to each level. In Biquadris, the subclasses would specify the probabilities of the shape sequence and would add the "heavy" and random 1x1 block enhancements. Therefore, if we were to add additional levels to our game, we could create more subclasses and implement the specific level details while only having to recompile the Level module.

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?

To design our program to allow for multiple side effects and simultaneous side effects, we could use the Decorator Design Pattern. Our base class would consist of a blank board class, where we would add our concrete Decorator classes on top of the current board to add effects. This way, multiple effects may be added simultaneously (so opponents may be under two or more different effects in one turn).

This approach also avoids having an if-else branch for every possible combination of effects. For instance, we would only add the blind board Decorator if the user chooses blind, the heavy board Decorator if the user chooses heavy, the force board Decorator if the user chooses force, etc.

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.

In our current design, we already support a "macro" language through the GameProperties class. The idea behind this is to use a hashmap to store the new macro command as a ("new macro command", "original command") key-value pair. For example, if the user wants to create a shortcut for left, our map could contain the key-value pair ("l", "left"). To prevent the player from making one command do two things at once, we can prompt the user that they will be overriding the original command and make them rebind the original command. Thus, if we have a (“l”, “left”) and the player attempts to create another macro (“l”, “right”), we plan to rebind “l” to the "right" command and then prompt the user to set a new macro for the left command.

To accommodate new commands, we have to add a new key-value pair for the command and create another else-if branch where we will implement the feature.

Project Breakdown:

Shape Class:

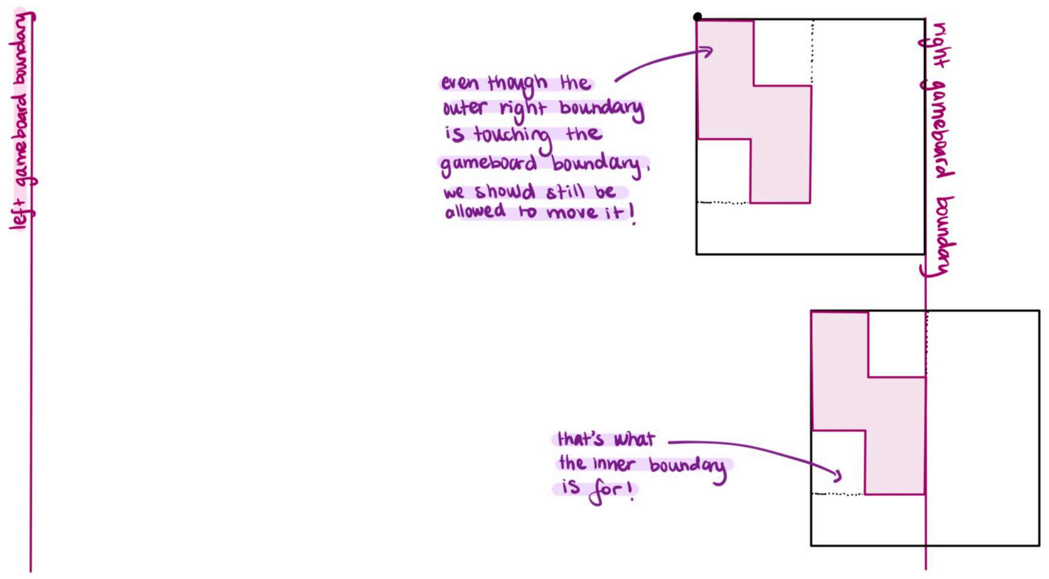
We designed the shape class to have an “outer” 4x4 grid that simplifies the rotating process for any shape. This is accomplished using outer top, left, height and width parameters (int). This is especially helpful since rotating square matrices allow us to change the element values rather than completely reconstructing the matrix. Specifically, the math to rotate every element clockwise within the original grid is:

$$(x,\ y) \rightarrow (y,\ 3\ - x)$$

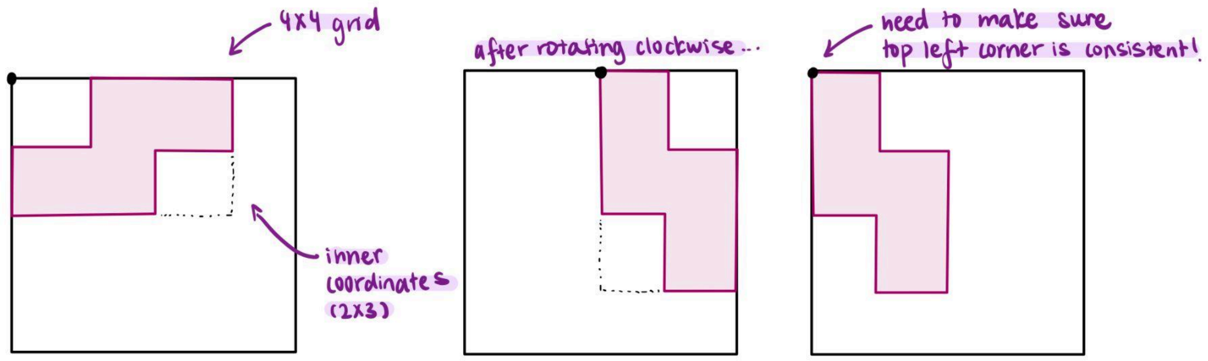
The y-value becomes 3-x because the grid will always be 4x4. The math for counterclockwise is similar but in the reverse direction.

However, another consideration for our shapes is the ability to move left and right before the player drops their pieces. That’s why we are also incorporating inner top, left, height and width (int) parameters to indicate where the actual shape is located within the grid. This allows the user to move the shape from one boundary to the other (when shifting their blocks left and right) and ensures that the top-left corner is kept static after rotating. Reference the following diagrams:

Translation


left gameboard boundary
right gameboard boundary
even though the outer right boundary is touching the gameboard boundary, we should still be allowed to move it!
that's what the inner boundary is for!

Rotation:


4x4 grid
inner coordinates (2x3)
after rotating clockwise...
need to make sure top left corner is consistent!

The Shape class will also hold a vector of vectors, which represents a 2D array that determines where the shape is located within the grid (mix of whitespaces and characters). We have listed the methods for the Shape class below.

Changes made since the initial design document:

- **createEmptyShape():** creates an empty grid, which is called every time a modification is made to the grid or shape (i.e. rotateCW, rotate CCW). This is accomplished by creating rows (vectors) of whitespaces and adding these to a final vector.
- **createShape(<vector<vector<char>> newShape):** creates a new shape and changes the inner top, left, height and width values. This function is used to make sure the top left coordinate stays consistent after rotations in either direction.

- **addShape():** adds the shape to the grid by iterating through particular rows and columns in the 4x4 grid and changing their values.
- **getT():** gets the top coordinate for the grid. This function was useful for determining whether the player can move the current shape left or right.
- **getL():** gets the left coordinate for the grid. This function was useful for determining whether the player can move the current shape left or right.
- **getWidth():** gets the width of the shape. This function was useful for determining whether the player can move the current shape left or right.
- **getHeight():** gets the height of the shape. This function was useful for determining whether the player can move the current shape down.

Functions we declared in the initial design document:

- **move(int x, int y):** if canMove returns true, then we will move the shape by x coordinates horizontally and y coordinates vertically.
- **rotateCC():** will rotate the current shape clockwise 90 degrees (see math above).
- **rotateCCW():** will rotate the current shape counterclockwise 90 degrees. The math for rotating the block counterclockwise is the inverse of rotating the block clockwise.
- **charAt(int x, int y):** will return the character at a specific location in the grid (either whitespace or character)

We constructed various subclasses of Shape to match the blocks listed in the project outline. While there are no unique data members/methods for these subclasses, their constructors make it easier to determine inner/outer coordinates and the 2D dynamic array (since they vary between each block type). This includes the L-block, Z-block, the S-block, etc.

Studio Class:

The Studio class represents the Biquadris game board, which stores an 11 x 18 game board (3 extra rows on top) that is represented by a vector of vectors (2D dynamic array). The following methods will be available to the studio class:

Changes made to final design document:

- **getBoard():** returns the game board, which is especially useful for rendering.
- **setBoard(vector<vector<char>> newBoard):** resets the board to **newBoard**, which is useful when changes are made by either player (i.e. left, right, down, drop).
- We removed the **gravity() function** listed in our initial design document since it makes it easier for players to get a higher score.

Functions we declared in the initial design document:

- **canRemove():** determines whether a player can remove a row from the game board after their turn by iterating through each row. It will return the row index if possible, otherwise, it will return -1
- **remove():** removes a row from the dynamic 2D array and adds an empty row at the start, which was accomplished using vector methods like **.erase()** and **.insert()**.
- **charAt(int x, int y):** returns the character at coordinates (x, y) on the gameboard.

Player Class:

The player class replicates a one-player game, which has access to the next shape that will be dropped onto the Biquadris board, the Studio class (board), the time (turn counter), the level, and the total number of rows removed. The player class will also contain booleans that determine whether the player has lost, is blinded or is heavy if the opposing player has applied a power-up. We also added a file stream that reads a text file containing a sequence of blocks. As an extra feature, we created a “shadow” piece that allows players to see where the block would be placed if it were dropped. Also, we added **dropNum** to keep track of the number of blocks that must be automatically dropped (from calling [multiplier]drop). The player class will have access to the following methods:

Changes made since the initial design document:

- **handleMovement(int moveCol, int moveRow):** condenses our code and encapsulates our movement logic into one function to improve readability and modularity. We used this function to check and implement block movement and shadow block generation.
- **resetBoard():** complements the **restart()** function in the Game class by creating a new empty Studio game board and resetting all the booleans/integers to their original values.
- **generateShadow():** an extra feature we implemented that creates a shadow shape for the current shape. This is accomplished by:
    - Using **calculateDropDistance()** to find the final position of the shadow
    - Creating a copy of the current shape
    - Moving the copy into position
- **calculateDropDistance():** Calculates the distance the block would drop. It does this by creating a copy of the current shape and moving it as low as possible
- **setShape(char c):** based on character input, setShape will set the current shape to a new block (i.e. ‘I’ input will generate I-Shape).
- **setNextShape(char c):** based on character input, setShape will set the next shape to a new block (i.e. ‘I’ input will generate I-Shape).
- **setForce():** since we handled the force mechanics within the game class, we did not need to use this function anymore, and this was removed after coding our project.
- **setDownLevel():** similar to **setNextLevel(),** this allows players to decrement their level by 1.

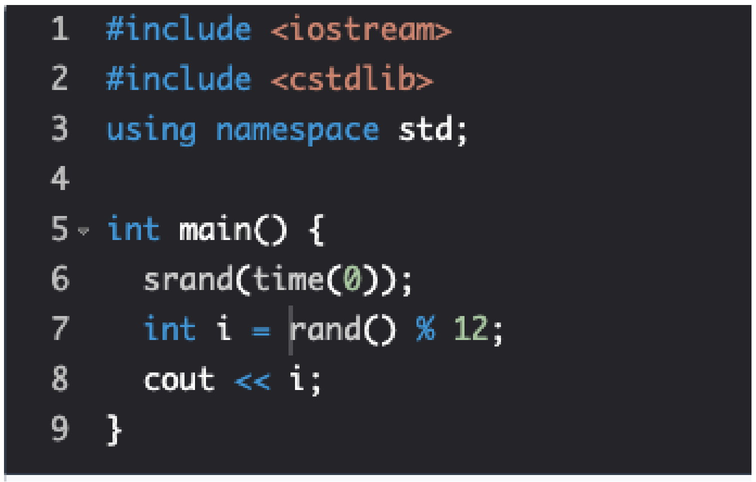
- **updateTurn(string cmd, int multiplier):** represents one turn, which returns the number of rows cleared (to determine whether the player can apply powerups). With each turn, the player will retrieve the next block and allow players to move or drop it. When the player has dropped their block, this signals the end of their turn. At the end, Studio's removeRows() function will be called to apply special effects if applicable.
    - cmd is a string that represents the command (without the multiplier) that is to be applied
    - multiplier represents the multiplicity of the commands (e.g calling 3drop will have a multiplier of 3)
- **getDropNum():** will return dropNum
- **decrementDropNum():** will decrement dropNum

Functions we declared in the initial design document:

- **canMove(int x, int y):** determines whether the next block can move to a particular location based on the game board (the block cannot go out of bounds or overlap with other shapes). It returns a boolean.
- **getScore():** will return the score based on the level and number of rows cleared.
- **renderRow(int i):** this will return the row at index i as a string. We decided to render one row at a time so that the gameboards of both players could appear side-by-side instead of one after another on the terminal.
- **getLevel():** will return the current level
- **getRowsCleared():** will return the number of rows the player has cleared
- **getHighScore():** will return the all-time high score of the player.
- **setBlind():** blinds the player (boolean is changed to true). The conditionals will be determined by the Game class, or specifically, if the other player cleared 2 or more rows during their list turn.
- **setHeavy():** sets player as heavy (boolean is changed to true). The conditionals are determined by the Game class, or specifically, if the other player cleared 2 or more rows during their list turn.
- **dropBlock():** allows the player to drop the current block on their board. It continuously calls the Shape's move method until **canMove()** returns false. In other words, the block has collided with other shapes already placed on the game board.
- **getLost():** returns true/false depending on whether the player has lost
- **setLost():** changes the value of the **lost boolean**, which will signal when a player has lost.
- **setNextLevel():** sets the level pointer in the player to the next level based on the current level

Level Class:

The Level class determines game properties depending on the level selected. The Player class will have a pointer to a Level class, which can be reassigned to change game difficulty. Every Level class will have a vector of characters, which will represent the weighted probabilities listed in each level. For example, if the S block appears 2/12 times, then the vector will be size 12 and S will appear twice. Then, we will use the **<cstdlib>** library's **rand** and **srand** functions to pick a random number between 0 and 11, which represents the index of the dynamic array. Depending on which letter is picked, a certain block type is returned, and the Player's next block is set to this return value. Finding a random index will look like such:



```
#include <iostream>
#include <cstdlib>
using namespace std;

int main() {
  srand(time(0));
  int i = rand() % 12;
  cout << i;
}
```

Each Level will have the following virtual methods:

Changes we made since the initial design document:

- **dropRandBlock():** a virtual function that returns an arbitrary index value between 0 and 10 (representing a random column). This applies to level 4, where a random 1x1 block is dropped after 5 moves have been completed without dropping a block. For other levels, the **dropRandBlock()** function does nothing.
- **getBlock(char):** used to generate a certain block-type depending on a character input (i.e. 'I' will generate an I-Block). This is a useful helper function for getRand(), which returns a pointer to a random block.
- **getRand():** returns a pointer to a random block according to the probabilities listed for each level. After implementing this project, we decided not to apply "heavy" and random 1x1 block drops in this function. The "heavy" effect is automatically applied in the Player class if **getLevel()** returns 3 or 4, and the random 1x1 block automatically drops in the Player class if **getLevel()** returns 4 and **getTime()** returns 5.

Functions we declared in the initial design document:

- **getLevel():** will return the player's current level as an integer
- **getTime():** will return the current time
- **setTime(int time):** sets time equal to the argument
- **incrementTime():** increments the time by one

The following subclasses will be implemented as follows:

LevelZero:

- **Final changes:** LevelZero will not implement any of the text file commands since everything will be handled in the Player class. The **setShape(char c)** method is used to generate a certain block based on user commands.

LevelOne:

- Create a vector of 12 characters where 'S' and 'Z' appear once and every other block appears twice.
- When **getRand()** is called, a random number between 0 and 11 will be generated using **rand** and **srand** to determine which block will be next for the current player.
- No changes were made to LevelOne

LevelTwo:

- Create a vector of 7 characters where each block appears once - same logic as LevelOne
- No changes were made to LevelTwo

LevelThree:

- Create a vector of 9 characters where 'S' and 'Z' appear twice and every other block appears once.
- The same logic to find a random character is implemented for this level.
- Every time the player moves a block left or right (before drop), call **shape->move()** and **canMove()** twice - once to move the block horizontally, and the second time to move the block down.
- **Changes:** the text inputs for the "norandom" command are handled in Player since Shape does not have access to enough information. The **getNotRand()** function was eliminated.

LevelFour:

- Implement the same features as LevelThree in LevelFour.
- If a player successfully removes at least one row, reset the Level timer to 0. If the timer ever reaches 5, choose a random column to drop a 1x1 block
    - The logic to randomly choose a column is the same as described in previous levels
    - This returns an integer, not a shape
- No changes were made to LevelFour

Game Class:

The game converts Biquadris from a one-player game to a two-player game by storing two Player components as private fields. It will also have a turn counter, which will increment after each player goes - if the turn counter is odd, then Player One goes. Otherwise, it is Player Two's turn. The following methods are available to the Game class:

Changes we made since the initial design document:

- **initializeGraphics()**: initializes graphics on xWindow

Functions we declared in the initial design document:

- **update():** determines which player goes depending on the turn counter. There will be a while loop inside that implies continuous gameplay until either EOF is called or a player is declared the winner. **Player->updateTurn()** will be called inside this function.
- **render():** renders the game, including both player game boards, their scores, and their next block. We want to do this side-by-side by rendering each row one at a time.
- **getWinner():** this is called at the end of **update()** when the game has ended. It will return the name of the player who has won (where Player->getLost() is still false).
  - If neither player loses, it will compare their high scores to determine the winner.
- **restart():** restarts the game by setting all players' scores to zero and resetting their game boards (and other necessary data fields).

To start and run the Game class, we can define a Game object in our **main.cc file**. The players will be directed to choose their level before starting, and during each player's turn, they can input other commands that are specific to their board (i.e. left, right). These will not be present in the **main.cc** file.

Game Properties Class:

This class allows the players to define shortcuts and rebind commands to different keywords using a hashmap. In addition, this class can load properties that were previously defined by allowing players to save new shortcuts and rebind keywords. The following methods are available to the Game Properties Class:

Changes we made since the initial design document:

- **saveFile():** will output all the key-pair values in the hashmap into a text file.

Functions we declared in the initial design document:

- **getProp(string prop):** acts like a getter for the map **properties** and retrieves the value associated with the key prop**.**

- **saveProperty(string prop, string value):** acts like a mutator for the map **properties** and will simply add/redefine the key value pair for (**prop, value**).
- **loadFile(string file):** will attempt to load and parse the file passed in.

If provided, this class will store a file stream and can be constructed from a text file. Likewise, when the destructor is called, the class will save to an existing or new text file.

Bonus Features/Optimizations:

- **Game properties:** allows users to create custom commands by importing a text file
- **Shadow block:** allows players to see where the block would be located if the player dropped their current block
- **Graphics rendering:** improved graphics rendering speed by filtering for necessity (i.e. only re-rendering if there are changes).
- **Shape rotations:** instead of keeping the left-bottom consistent, we decided to keep the left-top consistent after each rotation to make our game mimic Biquadris more accurately.

Design Patterns:

- **Factory Design Pattern: For Level and Shape classes (base classes that act like interfaces for every Level and Shape), each subclass specifies different behaviour.**
  - Each Level subclass specifies the probability of the sequence of shapes.
  - Each Shape subclass specifies the Grid property (every Shape has a different structure)

Coupling & Cohesion:

Coupling:

Our project has low-moderate coupling because our program elements (classes, functions, etc) are mostly independent, meaning that they don’t rely on each other. One instance of moderate coupling is our Player class, which is tightly coupled with other classes (Level, Shape, Studio, etc). This means that if changes/errors were to appear in Player, it may ripple and affect the classes that it is connected to. As a result, it may be harder to reuse the Player class without implementing all members again. In our other classes, they are more independent so they don’t rely on other classes to function.

Cohesion:

Our project is high cohesion because every class has a specific purpose that layers on top of each other to build Biquadris. Specifically, the Shape class handles each block the player drops and moves in the game by handling movement and rotation. The Level class handles particular game properties, including probabilities of choosing a random block and “heavy” blocks. Its primary purpose is to return a pointer to a random shape so the player can retrieve the next block they will drop. The Player class handles particular elements unique to each player, including their game board, left/right commands, score and whether they are affected by special effects. The GameProperties class handles key-pair values that allow users to input unique commands to call specific functions. Finally, the Game class oversees both players and keeps track of the total game state (i.e. winner/loser, which player's turn). Specifically, we noticed that our project had high cohesion when we had to add additional features we missed on the first read (see below).

Accommodating changes:

The biggest way we accommodated changes was by separating different features into functions. For example, creating a setShape() function in the Player class helped add sequence files and block commands during a player’s turn. We forgot to add the block commands initially (i.e. inputting ‘I’ to change the current player’s block to an I-Shape), but this additional feature took minimal time to implement since setShape() supported this feature. Similarly, for sequence files (to read blocks), having setShape() helped streamline this process, especially since we added this feature at the end. Additionally, new shapes and players could be easily added by creating new subclasses of Shape (like we did for IShape, SShape, etc) and by adding more players into the Game constructor to enable multiple opponents.

Additionally, in GameProperties, since we organized a hash map that binds commands to specific functionalities, it was easy to change our bindings or add new bindings (editing/adding to the map). Specifically, we also forgot to implement the "auto-detect" feature for player commands (i.e. lef is the same as calling left), but since we had a hashmap that ties user commands to specific functionalities, all we needed to do was add another key-value pair for this to work. Therefore, if a user would like to change bindings, it accommodates this change effectively. In addition, since our program is loosely coupled, it is simpler to add changes without rippling or causing errors in other program elements.

Final Questions:

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?

The biggest lesson we learned was the importance of extensive planning and testing, especially if everyone is working on different classes that are dependent on each other. Additionally, most of us approach projects by coding and debugging on the go. By thinking of design patterns and implementation ideas, this approach of drawing a UML and thinking of potential approaches helped streamline the implementation process for us.

In addition, we learned that it is vital that we delegate our project into subtasks to maximize productivity. On the other hand, while it is important to complete our delegations, it is also equally important to make time for connecting all individual parts to form a game like Biquadris. This portion of the project took up a significant part of our time which we underestimated in our original plan of attack.

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

From a graphical interface standpoint, we implemented it without using the Observer Pattern, which resulted in cleaner code but at the cost of rendering time. As a result, if we were given the chance to start over, we would implement the Observer Pattern so that each observer renders part of the board (similar to the assignment question about the boxes).

From a memory management standpoint, it was often very difficult to debug our code due to memory errors (invalid free, uninitialized values, memory leaks, etc). If we had the chance to start over, we would use smart/unique pointers to reduce the need and stress of managing our own memory. In addition, we would get additional bonus marks for implementing the project without explicitly managing memory.