CS246 Final Project Design Documentation

BIQUADRIS

University of Waterloo

Justin Jonany, Linjia Tang, Brandon Zhou

August 1, 2023

Contents

1 Introduction

Biquadris is a captivating two-player, turn-based game inspired by Tetris, where each player has their own board to control. During their respective turns, players maneuver blocks to adjust their positions and angles, strategically aiming to complete as many rows as possible. When a row is filled, it is cleared, and players earn points based on the number of rows and blocks they have cleared. The game builds in intensity until one player's board becomes unable to accommodate new blocks, resulting in a loss for that player. Biquadris offers a distinctive and pleasant take on a traditional game with its captivating blend of strategy, rivalry, and spatial awareness.

2 Overview

Our game has Seven Main classes:

- CommandInterpreter
- Board
- Subject
- ShowDisplay
- Level
- Block
- XWindow

Each main class has wrapper-classes and sub-classes.
This report will outline the detailed structure of the game and explain each class and its sub classes.

3 Preliminary UML class model

Please see Figure 1 below.

4 Updated UML class model

Please see Figure 2 below.

5 Design

1. **main**
   Our program starts with the main file `main.cc` where we process the command line arguments, create the boards for player 1 and 2, create the game (which has the two players), and process the commands.

   (a) `-seed xxx, -startlevel n, -scritfile1/2 xxx` These are simply taken from the command line and included when calling the constructors for the board. However, note that first, providing a start level below 0 to the program will set the initial level to 0. Giving a start level higher than 4 (or any future higher levels) will set it to the highest level that exists in the design of the program. Second, when the file of a scriptfile is given, it saves its contents in a vector and passes it to the `board` constructor. If the file is empty, it will keep the default files.

   (b) `-text` If this command is given, it simply states that it is text-based only for display when the constructor for the command interpreter is created.

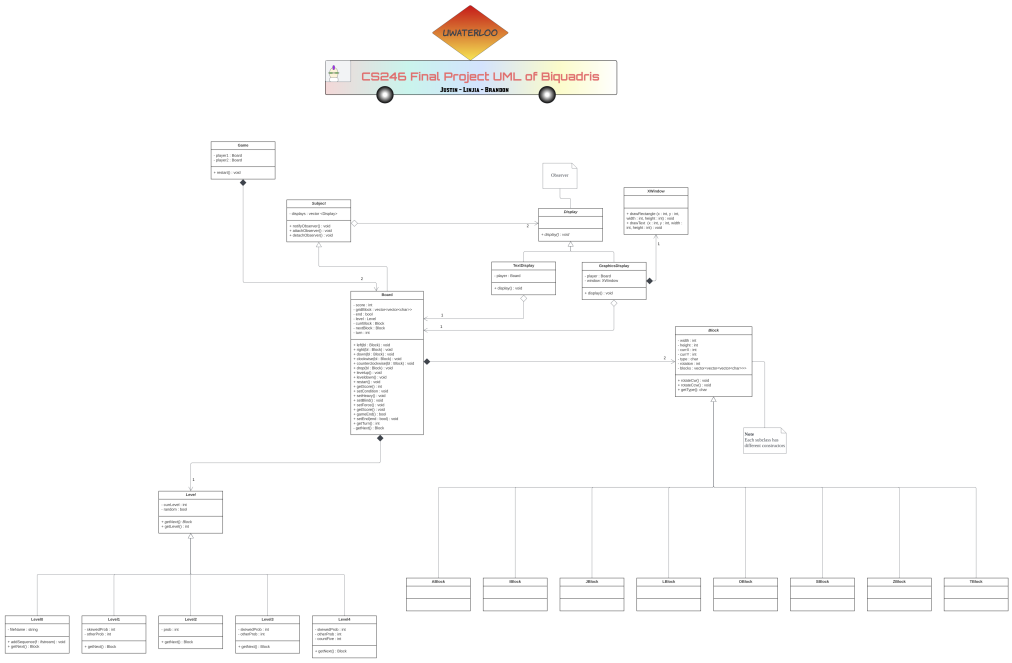

Figure 1: Preliminary UML Diagram

2. **Class : CommandInterpreter**
   Different to DD1's version, we decided to add `CommandInterpreter` because it's better for modularization and encapsulation. The class's goal is to run all commands and prompt users.

   The command interpreter has two while loops: for ncurses commands and for text commands. **Details about ncurses/keyboards commands will be covered in the extra credit section.**

   In making the general text-based commands, the notable design challenge we faced was alternating player turns. However, we were able to solve it by using a boolean value that is set to true if it is player1's turn, and false if it's player2's. The loop will keep going until either player's board indicates the game has ended and the users want to quit the game. We provide the option for the users to restart the game if they want after a round of the game ends.

   Invalid commands have been considered and will prompt the user to input the correct command. It won't continue until the correct command is given.

   Details about the multiplier prefix and auto-complete will be discussed in Section 7 number 4.

3. **Class : Game**
   The `Game` class acts as a wrapper class for the two players we have. It also has two accessors that return a pointer to player1 and player2, allowing the command interpreter to run each player's commands.

4. **Observer Pattern**

   (a) **Class : Subject**
   The class `Subject` is part of the observer pattern, which includes functions `notifyObservers` that is used to notify the all the observers in the vector that stores instances of the observers `TextDisplay` and `GraphicsDisplay` to print the game board, `notifyCheckout` that prints the checkout page when the game ends, and `notifyClear` that clears the screen or window if the users want to restart the game. Additionally, the functions `attach` and `detach` in the class are used to attach and detach `TextDisplay` or `GraphicsDisplay` when we create any instance of them.

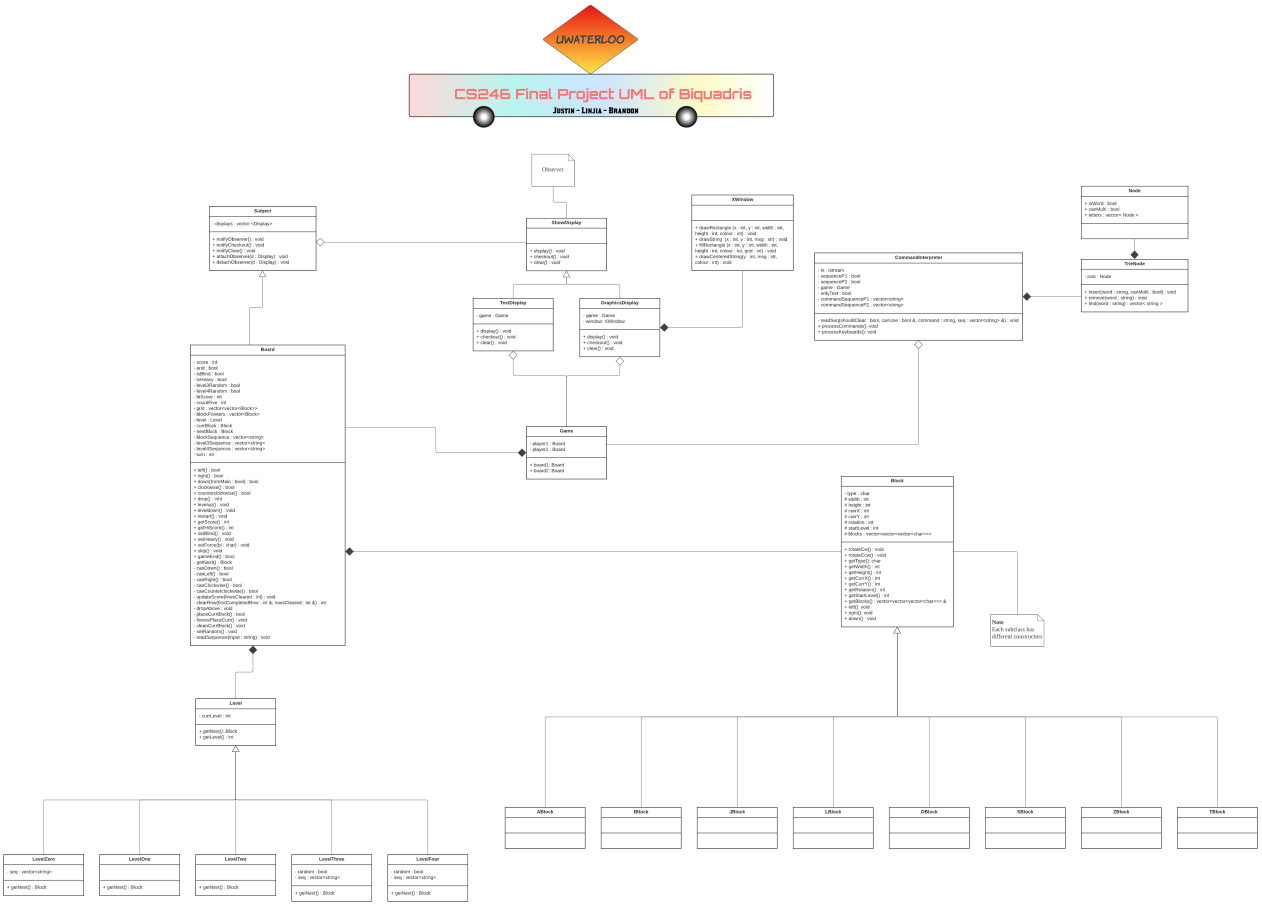

Figure 2: Updated UML Diagram

(b) **Class : ShowDisplay**
`ShowDisplay` is an abstract class, for which the classes `TextDisplay` and `GraphicsDisplay` are inherited from. They are the observers of `subject`. The virtual function `display`, `checkout` and `clear` will be overridden in each derived class, for which `display` will print each player's current game level, current score, highest score gained so far, board and next block, `checkout` will display the winner of the game when the game ends, and `clear` will clear the window for the graphic display if the users want to restart the game.

5. **Class : Level**
Our Level's main task is to give the next block.

The `Level` class has five different sub classes, where each subclass represents a different game difficulty. This class is designed using the *Factory Method Design* pattern with the **main task of generating blocks** according to the game's level and game difficulty. Each level, has the method `getNext`, which returns the next block.

There is a notable challenge for the `norandom` setting for levels Zero, Three, and Four we tackled this by first saving the commands from the file in the command interpreter in a vector and saving it as a field of levels Zero, Three, and Four. For level Zero, we saved it `main`, while levels Three and Four's are in the command interpreter.

Our level is slightly different from the first version because we didn't consider the command `norandom`. We also decided to change `LevelZero` to accept a sequence of blocks (which is saved to `Board`) instead

of the filename. This allows us to save the sequence of commands even if we `levelup` or `leveldown`.

Our level displays low coupling and high cohesion. It only needs the block modules for the `getNext` method, which determines the next block, indicating high cohesion and low coupling.

6. **Class : Block**
   The blocks all have two main tasks: give the appropriate shape and the specifications in accordance to the rotation and the coordinates in the grid.

   `Block` class is an abstract class. It has eight different blocks. Each block has a parameter that holds the width, height, current position, and rotation. The shape of the blocks is stored in a vector of vectors of the vector of chars (3D Vector). The main feature of our blocks is that we hold all possible forms of rotations and increment or decrement the field rotation to indicate which rotation a block is in to prevent changing the array of characters each time with rotate. The block has general mutators that allow clients to change the rotation and position easily.

   Our `Block` class displays low coupling and high cohesion as it doesn't depend on any other module and adding any new block can be done simply by adding a subclass for the `Block` parent class.

7. **Class : Board**
   The Board classes are player1 and player2. Board goals to save everything we need to know about a player. Let's discuss each feature:

   (a) **Grid**
   The grid, or the location of where we store each cell, is implemented using a vector of vectors of shared pointers to a block. When a board is created, we immediately use the `getNext` method to generate a block based on the starting level and place the block (according to the position) into the grid. On DD1, we stated that the grid will be storing in a vector of vectors of chars. However, we change it to adjust for the scoring rules and the movement-related method, allowing them to work for all types of blocks, even if we add new types of blocks.

   (b) **Drop**
   The drop is done in the following steps:
   (1) calling the function down until it can't go down any further.
   (2) starting from the bottom of the grid, checks for full rows, and remove them. Then, emplace corresponding number of empty rows in front of the grid.
   (3) If any rows are removed, we need to check if there are floating cells that should fall. This is done by checking each row above the empty cells if it's a dangling cell or not. Every time we find a dangling cell, we make it fall down.
   (4) We repeat (2) and (3) if making a cell fall results in more rows being cleared
   (5) Update score
   (6) We then set the current block to the next block. Put a shared pointer of the current block to the vector of the existing block, and placed the current block on the grid. We then call method `getNext` belonging to the current Level we're pointing at for the next block.

   Shared pointers allow (3) to be done. Using a shared pointer allows us to check if the **whole entity** of a block can fall or not.

   (c) **Special Actions Force**
   When force is called upon a player (by another player), we call `setForce(char bl)` on that player and we will forcefully change the current block. If the block force can't be placed, the game ends.

   (d) **Scoring**
   The score is updated every time a drop is called. A problem in updating the score is how to keep track of when an entire entity of a block is completely gone from the screen. We tackled this problem by using shared pointers (hence why the grid's cells are shared pointers).

   Every time we place a whole entity of a block, say a block with x squares, we also save a shared pointer for that block in another vector(a field of the board), making the count pointing to that object entity $x + 1$. Therefore, when we update scores, we just need to check the counts of each

pointer in the vector of shared pointers. If any of the count is one, it means that when we dropped, we removed rows that causes an entity of a block to be removed completely. Therefore, we can update the score.

To keep track of the level in which the blocks were generated, we added a field in `Blocks` and initialize it to the level when the block was generated. Therefore, if we change levels, the current block and next block will still stay as a block generated in the previous level.

(e) **Placing Current Block**
Using a double for loop we place the current block into the 2D Vector grid. It returns false, if it can't place and will indicate the game ends.

(f) **Level**
We have a field that is a pointer to a level object. Each time we level up/down, we change what it's pointing at. This field is used to get what the **next block** will be and allows the specific requirements of each level like the heavy settings of level 3 and level 4, to make the current block go down when we move to level 3 and 4, and also to implement the star blocks of level 4.

To implement level four's star block, we added a field specifically to count the number of turns. Every time the number of turns is divisible by five, we drop a star block from the top of the grid (not the third row).

(g) **Special action blind and heavy**
When a special action is called upon, a boolean field is set in the class. For heavy, when the left and right function is called, we call heavy. For the blind, the observers will print out a blinded version.

(h) **Left, right, down, counterclockwise, clockwise**
These functions are referring to the movement of the current block.

Special for the left right and down functions, when the board is set to Heavy, these functions will drop immediately and return false if it can't go down anymore. Allowing the command interpreter to change the turn.

For all the movement-related commands, it follows a general pattern of

(1) Checks if the current block can go left/right/down/counter-clockwise/clockwise using a double for loop.
It checks if the current block does any movement, whether it will go out of bounds, or overwrite an existing pixel. If it does, no changes occur, and the function returns.

(2) We remove the current block from the grid. Apply the movement to the current block (therefore changing the position or rotation fields).

(3) Then, place the current block.

(4) We then support the special actions and level requirements.

(i) **Restart**
The restart method is done by clearing the grid and resetting all the fields, except the high score.

8. **Ending the game**
The game ends when we can't place the current block anymore. A check-out screen will appear and prompt both players to restart or end the game.

6 Resilience to Change

1. **Adding new Blocks**
The addition of new blocks can be done simply by adding a new subclass for that type specified with the correct width and height. We must also define all possible rotations of the shape too. We should also make it so that the height should be at most 4 because the game is set that the current block is on the 4th-pixel row from the top.

2. **Defining new levels**
   The addition of new levels can be done easily by adding a new subclass of `Level` and defining how the next block should appear. The module only depends on the `Block` module to get the next block. No matter what, it will always depend on the `Block` module because it needs to know the types of blocks that exist.

3. **Changing size of grid**
   Changing the size of the grid can be done by just changing the static constants of the Row and Grid in the board. It will work because:

   (a) All the movement actions, including placing/dropping blocks in the grid, in `Board` is implemented based on the width and height of the block. The edges are also checked with respect to the static constants from the `Board`.

   (b) The observers print the display for text and graphics based on this constant, and not based on a set value in the for loops.

7 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?*

   As we use a 2D vector to store the blocks, we can use another 2D vector to store the integers, that is, the number of rounds that the generated blocks have existed and not cleared yet. If the pixel is not occupied, the value is 0. Each round we increase every non-zero value by 1, and when it reaches 10, we can clear the corresponding block in the 2D vector for the block and reset the value to 0 unless there are blocks above that can fall. In more advanced levels, such as 1*1 squares (which can also be counted) and different probabilities would not affect this 2D vector, therefore the generation of such blocks could be easily confined to more advanced levels. This method can be easily used on any level as long as the general fundamental of the game stays the same.

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

   First, note that in our original plan of Biquadris, we created different derived concrete classes for each level 0 to 4 for abstract base class Level in different interfaces and implementation files. These derived classes implement the different block generation logic and other features unique to each level. To design the program to accommodate the possibility of introducing additional levels into the system with minimum recompilation, we can use Factory Method Pattern to implement this requirement. In the class `Level`, we need to ensure that the other core game logics are decoupled from the level implementations such that we can easily create derived classes called Level5, Level6 without change and recompile the entire program.

   Therefore, we can just add other levels as a subclass of `Level` for the game to implement the specific behavior and features for that level that only need recompile `Level` class for it.

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?*

   In our design, we have three methods in the `Board` class, `setHeavy()`, `setBlind()` and `setForce()`, respectively. If we want to apply multiple effects simultaneously, we can just call the corresponding methods in a call so that the desired effects can be applied on the screen. If we invented more kinds of effects, we would also have extra methods for them, so that we can just call the methods to apply the effects. We can just set/indicate it in `Board` when it has multiple effects going on, and `Board` will operate in accordance to the effect. In the while loop, we have many if statements for each method like:

```
if (command == "heavy") setHeavy();
if (command == "blind") setBlind();
if (command == "force") setForce();
...
```

With this, we do not need to have all the combinations to call the corresponding methods.

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.*

   Our implementation for rename is use vector to store each action with different name in a specific location of the vector and we can execute corresponding command by calling the element in vector in a fixed position which store present name of command. That is, the commands have a default name that is stored in the vector. Each time the client changes the name, the name is updated in the vector. Therefore we can change of command names can be done easily by simply changing the if-else conditions of the command interpreter when we called command rename. This way the user can keep updating the name for an unlimited number of times.

   Additionally, adding new commands can be done also just by adding a new condition in the command interpreter, insert new element in vector, and editing the classes for new methods.

   For rename conmmand, we have a more detailed introduction in Extra Credit Features, here we introduce some codes for clarity:

   e.g.

   ```
   // Creater vector and insert commands
   std::vector<std::string> name;
   name.emplace_back("left");
   name.emplace_back("right");
   name.emplace_back("down");
   ...
   // Run commands
   if (command == name[0]) left()...;
   else if (command == name[1]) right()...;
   else if (command == name[2]) down()...;
   ...
   ```

   Instead of

   ```
   // Run commands
   if (command == "left") left()...;
   else if (command == "right") right()...;
   else if (command == "down") down()...;
   ...
   ```

   This way the user can update the names in the vector of commands by having another set of commands `rename`.

   ```
   // Command for rename
   if (command == name[12]) {
   ```

```
    std::string firstCommand;
    std::string secondCommand;
    std::in >> firstCommand >> secondCommand;
    auto it1 = std::find(name.begin(), name.end(), firstCommand);
    auto it2 = std::find(name.begin(), name.end(), secondCommand);
    // If the first command is found and the second command do not exist before,
    // replace it with the second command
    if (it1 != name.end() && it2 == name.end()) {
        *it1 = secondCommand;
    }
}
```

For "macro" languages we can save the sequence of commands in an array of vector of strings (2D Vector). When we want to save a sequence of commands, we put the name of the sequence in the first item of an vector then put the commands after. Then, if the command is called, we do a while loop on the vector to find the matching names, then use another while loop to run the commands (double while loop). If we can use HashMap, it is similar but easier, that way we can search up the name faster.

In our project, we use and implement the Trie tree data structure. This allows us to insert, store, search, and delete different commands name. After each renaming process, we ensure that different auto-completions are provided for various new names, avoiding confusion caused by shortcuts not matching the actual command names.

In the command line interpreter, we first check if there is a possible numeric prefix to repeat the execution of a command. In the code, we start the search from the root node, traversing the entire existing Trie tree. During each auto-completion, we return a vector of possible complete command names to check for potential ambiguities, and ultimately determine the complete command name.

By using the Trie tree structure, our system efficiently handles command management and auto-completion, providing a smooth and user-friendly experience while avoiding naming conflicts and confusions.

8 Extra Credit Features

Smart Pointers In the design of the program, we use smart pointers (unique pointers and shared pointers) instead of raw pointers for memory management, which will avoid the need to call `delete` for heap memory and the risk of memory leak.

Keyboard Command Instead of inputting text-based commands each time, our program allows the use of keyboard to manipulate the game, which is user-friendly and convenient. The corresponding commands are in Game Command Table below. However, to enable keyboard manipulation, users have to provide command line arguments `-keyboard` when running the program, and if keyboard is enabled, only grpahics display is available.
Table for keyboard command provided in Table 1.

Skip The users can choose to skip the current block provided, but there will be some penalties to the user.

Rename The users can change the name of command to what ever they like, even `rename` command itself. This command receives two inputs where first input is the name of command that the user wants to replace, and second input is the new name of that command. Please note the first input must be the full name of command to avoid confusion and the second input should not be the same as the existing command name. The way we implement the `rename` function is we have a vector that stores each command with a specific location in that vector such that we can easily replace the name of command whenever we want.

Text Command	Key Command
left	KEY_LEFT
right	KEY_RIGHT
down	KEY_DOWN
clockwise	>
counterclockwise	<
drop	Space
levelup	=
leveldown	-
L, J, etc	l, j, etc
restart	r
blind	b
heavy	h
force	f

Table 1: Game Command Table

9 Final Questions

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 project, we realize the importance of UML diagram, which is a crucial part of software development as it drives us to think of the blueprint of the program before programming and provides us with the overall structure of the program. It allows communication in a group to be more coherent and structured as we have something to base upon when coding, especially for large programs. As a team member, not making a module that's low coupling and high cohesion makes combining work extremely hard. However, if we're clear on the tasks of each class.

   If we worked alone, we learned that modularization is a key in writing large programs as it is easier to manage and debug, which can be more flexible if anything changes.

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

   If we had the chance to start over, there are a couple of things we want to change:

   (a) Use a decorator pattern to store blocks in `Block` as this would allow a more efficient way of updating the grid every time we drop a block. Some of our methods of changing the grid (down, right, left, etc) could've been done in a more efficient way.

   (b) Improve graphics to make it look like the Tetris game, more animations

   (c) If we want to add a new level, it's easy to do if it's only how you generate the next block, like the probabilities or the types of blocks. However, if we want to add special actions, maybe like modifying how it moves or unique scoring rules, we need to modify the board. Therefore, we would have made the levels to be easier to modify for all kind of features.

10 Conclusion

Overall, from the beginning of concept to the finish of programming, this project took approximately two weeks to complete. We appreciate every one's efforts as a team and those of the instructors. When developing the program, we put our all into making sure every element worked and adding extra features that could add to the program's appeal. The project helped us understand the fundamentals of C++ further and showed us the capabilities of C++.