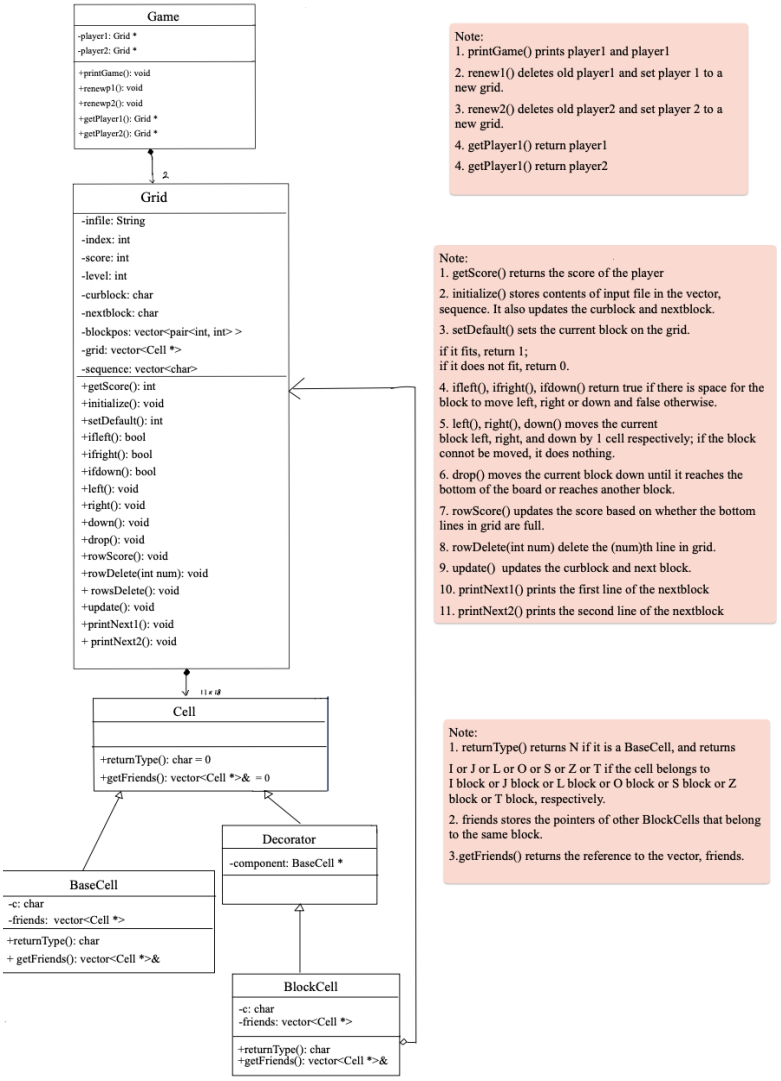
Biquadris Project Report
Yixing Zhang & Yilin Mo
CS246
August 13, 2021

Overview

The following is our revised UML diagram which describes the overall structure of our final project.



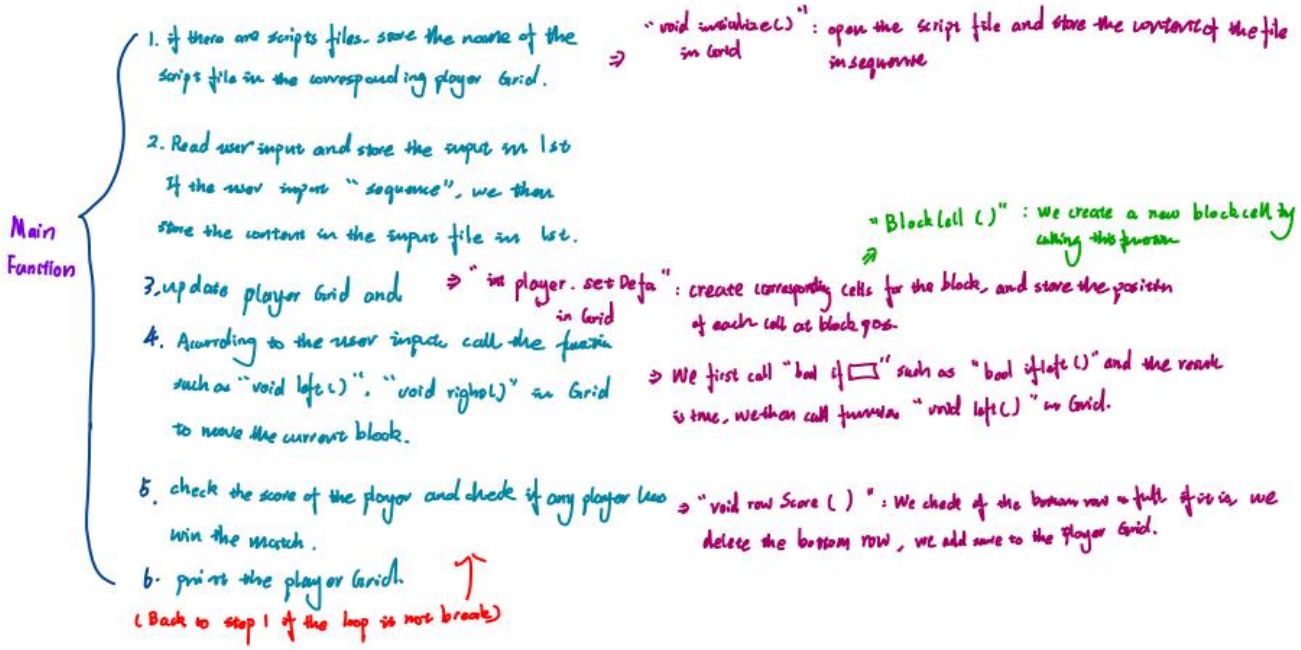
As shown in the UML diagram, we have six different classes/structures in our implementation. We have our Game struct which stores all the information we need about this Biquadri game. This includes the pointers to two Grid items, player1 and player2; each player is stored as a Grid. The Game is a struct and we can get its stored information directly without calling additional functions; however, we still have functions, “Game* get Player1()” and “Game* getPlayer2()” which return the pointers to player1 Grid and player2 Grid, when the users call these functions. The reason for this is that if the game developer wants to hide specific information of the

game from others users by only giving them access to the information of player1 and player2, they can simply change the type of Game to be a class; there is no further modification required. We also have functions “void renewp1()” and “void renewp2()” which help to delete the information stored in player1 and player2 and reset them to a new grid; these functions are called when the player wants to restart the game. The function “printGame()” is called when a player’s turn ends; it prints out the blocks that are currently in each player’s grid as well as the type of the upcoming block.

Each player is stored as a Grid and it contains the information required for each player playing the Biquadris game. This includes the name of the input file (string infile), the index (int index), player’s current score (int score), the current level of the game (int level), type of the current block (char curblock) and the type of the upcoming block (char nextblock). There are also three vectors, blockpos, sequence, and grid, stored in the Grid for each player. The vector, blockpos is for storing the coordinate in grid of each cell in the current block. The vector, grid is for storing the pointers to cells from the added blocks. The vector, sequence is for storing the information given in the input file. In Gird, we also have functions “void ifleft()”, “void ifright()”, “void ifdown()” which return true is there is space for the block to move left, right or down; return false otherwise. Then, when we are trying to move the current block, we first call one of these three functions; if the function returns true, we can then call “ “void left()”, “void right()”, “void down()”, “void drop” to move the current block. In the main function, at the end of each player’s turn, we call the function, “void rowScore()” and “void rowDelete(int num)” to update the player’s score and deleting the fulled row in player’s grid.

We have four classes, Decorator, Cell, BlockCell and BaseCell to implement the cell using a decorator. Each cell in the grid is initialized to be a BaseCell. When a cell is updated, both the char c and the vector friends are modified. The char c is the type of the cell and the vector friends stores the pointers to the cells that are in the block with the current cell.

The following diagram shows the design of this project visually.



Design

Our final design is slightly different from our original UML in many ways. When we are implementing the program, we found some non-considerations in our original designing plan; we use our programming techniques to solve these problems.

Firstly, in the beginning, we programed the Cell class to only contain a character c, which is the type of block that the cell is in. We knew that there were two ways for the player to earn scores, having a full row in the grid or getting all the cells in a block deleted. However, we found that, when we are counting the score for the player, we were not able to know whether all the cells in a certain block were all deleted since we did not have a variable storing the information of all the blocks that were created and the cells in each block. We solve this problem by adding a vector, friends, in the Cell class and this vector stores the pointers to all the cells that are in the same block with the current Cell. Then, when we are counting the score of a player, if the cells in a row are deleted, we call the function "void getFriends()" to determine whether other Cells in the same block have already been deleted. Then, when the current cell is deleted and the size of the vector, friends is 0, the player scores additional points and we can update the player's score.

We have a vector called, grid in each player Grid and it stores board for the game using 11*18 cells and each Cell in the grid is initialized to be type "N". At the start of the game, each cell in the grid is independent and they are not in a block with other cells; therefore, it is unnecessary for the cell initially contain the vector, friends. However, each time when there is a new block entered, we need to update the player grid and let the newly added block be at the initial position. Each Cell in the added block now needs to have a vector, friends since they are not independent and they are affected by the Cells that are also in the same block. Then, to achieve this, we use a decorator which allows us to attach new behaviour to the Cell class by placing the Cell class inside special wrapper objects, Based Cell and BlockCell. Then, when the cell is first initialized, it is set to be a Based Cell. When Grid is updated, the corresponding Cell is now set to be a BlockCell such that it contains the pointers to its relative Cells in the vector friends. Therefore, we solve the problems, and these challenges let us practice the programming techniques that we learn in class.

Resilience to Change

Our design of the program supports the following possible changes:

a. It allows for additional levels into the system. In our Grid structure, we have a variable, "level" and it stores the level of the game for players. In the main function, we call the function "player1->update()" or "player2->update()" so that the corresponding player Grid would be update. When the player is updated, its curblock and nextblock would be changed according its level. If there is an additional level being introduced to the system, we could add another "else-if + condition" statement to the code for the function, "void update()" in Grid. For example, if we want to insert Level X, where X is a natural number, we could add the following in the implementation for "void update()".

```
else if (level == X) {
        (Set the curblock and nextblock based on the rule for level X)
    }
```

Then, when we satisfy the condition for "level == X", we run the program within the bracket and update the the character stored by curblock and nextblcok.

b. It allows for additional types of blocks into the system. Each type of block is stored as several Cells; for example, each I block is stored using four Cells. Each Cell contain a

character c, which represents the type of block that it is in. If we want to added a new type of block into the system, we could add another else-if + condition to the code for the function, “void update()” in Grid. For example, if we want to insert a new type of block, X, we could add the following in the implementation for “void update()”. Assume each X block has Y Cells.

```
else if (curblock == X) {
        (create Y numbers of Cells and complete the vector, friends for each cell)
        }
```

Then, we satisfy the condition for “curblock == X”, we run the program within the bracket and a new X block is then added to grid.

c. The program allows for additional effects on the blocks to be added to the system. We can invent more kinds of effects by simply adding corresponding boolean variate to the Grid structure. For example, to add a special effect, super_heavy such that whenever a player moves a block left or right, the block automatically falls by three rows after the horizontal movement, we add the boolean variate if if_super_heavy to the Grid structure. Then, when we find that two or more rows are deleted at the time when a new block is being added to board A, we use the while-loop to check any special action inputted by player A. If player A inputs super_heavy, the if_super_heavy is set to be true for grid B. Then, when a block is added to broad B and the if_super_heavy for grid B is equal to true, we will let the block fall by three rows after every horizontal move.
d. I design the system to accommodate the addition of new command names. When a new command name is added, I will add a method function to the Grid structure to implement this command. Since the method function is in the Grid structure and the vector of the pointer of the cell is also in the Grid structure, we can modify the grid, which is a vector of the pointers to all the cells, by directly calling the method function. For example, we want to add the command, most_left that moves the current block to the left as far as possible until it comes into contact with either the left of the board or a block. In this case, we could add the function “void most_left()” into the Grid structure. When the player inputs the command “most_left”, we then call this function and we modify the grid vector in this function. Also, since all the command names are implemented as method functions in the Grid structure. we could simply go to its corresponding method function and make changes to the functions when we want to change an existing command name.
e. It allows for additional players into the system. The information of each player is stored in a Grid. We have a variate, turn, in the main function that stores the information of which player is currently playing the game. Now, we have two pointers of Grid, player1 and player 2, in the Game class as there are two players in the game. If we want to add player3 to the game, we could add “Grid* player3;” in the game class. Then, the variate, turn, change the value between 1, 2 and 3. When “turn == 3” is true, we call “player3->update()” and move the curblock in player3 as what we did for player1 and player3. At the end of the main function, we also need to delete all the player Grid that we create at the beginning.

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 displayed in the UML diagram, each BlockCell object has a field of type vector<BlockCell *> called friends. We create every block at the default position in the grid by function setDefault(). To achieve this, we simply turn every BaseCell object at the block's default position into BlockCell objects of a specific type using the Decorator design pattern. Then, for every newly created BlockCell object, we store the pointers of the other 3 BlockCell objects which belong to the same block in the variable friends. When we may need to delete a row in function compScore(), the BlockCell objects are deleted. Before they are deleted, we access their "friends" by the vector friends, and for every "friend" we change the pointer of the object that is going to be deleted to nullptr in the vector friends. For example, we created an O-block by setDefault(). Suppose the block consists of 4 BlockCell objects called A, B, C, and D. Then after setDefault(), the fields of A, B, C, and D will be like:

A: type: O friends: pointer of B, pointer of C, pointer of D
B: type: O friends: pointer of A, pointer of C, pointer of D
C: type: O friends: pointer of A, pointer of B, pointer of D
D: type: O friends: pointer of A, pointer of B, pointer of C

And suppose that the block is like:

A B
C D

Then if the row that C and D are in has no gap anymore, and we need to delete it by function compScore(), we first delete C and then D. Before deleting C, we access A, B, and D by friends, and change every pointer to C in A's friends, B's friends, and D's friends to nullptr. The same operation happens before deleting D. After C is deleted, the fields are like:

A: type: O friends: pointer of B, ~~(pointer of C)~~->nullptr, pointer of D
B: type: O friends: pointer of A, ~~(pointer of C)~~->nullptr, pointer of D
D: type: O friends: pointer of A, pointer of B, ~~(pointer of C)~~->nullptr

After D is deleted, the fields are like:

A: type: O friends: pointer of B, nullptr, ~~(pointer of D)~~->nullptr
B: type: O friends: pointer of A, nullptr, ~~(pointer of D)~~->nullptr

In addition, every BlockCell object has a function called checkDelete(). This function checks if all pointers in friends have been deleted. If yes, it returns true; otherwise, it returns false.

To add the feature described in the question, we can add an int data field called round num and set it to -1 by default. In every setDefault() call, after we create the BlockCell objects, we add 1 to the round num of every BlockCell object in the grid, including the newly created ones. Then we loop through the entire grid to find the BlockCell objects with a round num equal to 10. Then we delete them one by one and fill their positions with new BaseCell objects.

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

In our Grid structure, we have a variable, "level". Then, any method function in the grid structure can change the "level" when necessary. Also, when we determine the block coming next, we use an if-else if statement based on the "level". We would have a code like below.

```
if (level == 1) {
        ……
        next = …..;
        …..
} else if (level == 2) {
        …….
} else if (level == 3) {
        …….
}
……
```

If there is an additional level being introduced to the system, we could add another else-if + condition to the code as shown above. For example, if we want to insert Level 5, we could add the following code.

```
else if (level == 5) {
        …….
}
```

We find the next coming block at the end of each turn. When we satisfy the condition for "level == x" where x is a natural number, we run the program within the bracket to find the type of the next block.

3. 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 implement a program such that every time when the player clears two or more rows simultaneously, one or more actions could be called, we could use the while-loop. We implement the Grid structure like the following, which include the boolean variate if_blind, if_heavy, and if_force.

```
struct Grid{
        vector<Cell*>;
        bool  if_heavy;
        bool  if_blind;
        bool  if_force;
        ……
        };
```

The initial value of if_heavy, if_blind and if_force are set to be false for both grid A and grid B. When a new block is being added to board A, we will check if any two or more rows are deleted. If there is, we will have a while-loop to check any special action inputted by player A. For example, if player A calls blind and heavy, the if_blind and if_heavy will be

set to false for board B. Then, when it is Player B's turn, we will first check if any of if_heavy, if_blind, and if_force is equal to true. We will apply the corresponding action while displaying board B and moving the block. For example, whenever we are moving the block left or right, we will check if the if_heavy is true. Also, whenever we are displaying the board, we will check if the if_blind is true. At the end of player B's term, we will reset all these three boolean variates to be false. In this case, our program supports us to record the multiple actions that should be applied to the board and apply them simultaneously.

We can invent more kinds of effects by simply adding corresponding boolean variate to the Grid structure. For example, to add a special effect, super_heavy such that whenever a player moves a block left or right, the block automatically falls by three rows after the horizontal movement, we add the boolean variate if if_super_heavy to the Grid structure. Then, when we find that two or more rows are deleted while a new block is being added to board A, we use the while-loop to check any special action inputted by player A. If player A inputs super_heavy, the if_super_heavy is set to be true for grid B. Then, when a block is added to broad B and the if_super_heavy for grid B is equal to true, we will let the block fall by three rows after every horizontal move.

Since I am not placing different special actions into combinations, there will not be any else-branch for combinations. We record any special actions that one player wants to apply to another player. We know that each action is applied at different points in the process. For example, the special effect, blind, is applied while displaying the board; and the special effect, heavy, is applied while moving the block horizontally. Then, we are not analyzing the special action in combination but individually; therefore, we will not have an else-branch.

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 complication? How difficult would it be to adapt your system to support a command whereby a user could rename existing commands? How might you support a "macro" language, which would allow you to give a name to a sequence of commands?

When a new command name is added, I will add a method function to the Grid structure to implement this command. Since the method function is in the Grid structure and the vector of the pointer of the cell is also in the Grid structure, we can modify the board, which is the vector of the pointer of the cell by directly calling the method function. For example, we want to add the command, most_left which moves the current block to the left as far as possible until it comes into contact with either the left of the board or a block. In this case, we add the function "void most_left()" into the Grid structure. When the player input the command "most_left", we then call this function and we modify the grid vector in this function. All the command names are implemented as method functions in the Grid structure. Then, when we want to change an existing command name, we could simply go to its corresponding method function and make changes to the functions.

It would not be difficult for the user to rename an existing command in the system. All the strings in this program should be stored using string variates. Then, when we want to rename a certain existing command, we could simply change the value of the string variable

and all appearance of this string in the program would be changed. For example, we would implement the existing command, “left” as follows.

```
string c_left = “left”;
……
// location (1)
if (s == c_left) {
……
}
……
// location (2)
if (s1 != c_left) {
……
}
……
```

In this case, if we want to rename the existing command, “left” to “move_left”, we could do so by signing c_left to “move_left”, string c_left = “move_left”. Then, the name of the commands is changed at both location(1) and location(2).

If I want to give a name to a sequence of commands, I can implement a new method function in the Grid structure and call the corresponding function for each command in the sequence in this new function. For example, if I want to use the name “left_right” to represent the sequence of commands, “left” and “right”. Then, we add the function “void left_right()” in the Grid structure and we program this function as below.

```
void left_right() {
        void left();
        void right();
}
```

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 program?

   This is our first-time developing software in groups; it teaches us with how to use different software, such as GitHub, for developing computing projects in team and it also helps us develop skills for collaboration.

   By completing this project, we explored the tools and features on GitHub as well as got comfortable with constantly uploading the changes in code from our local programming software to GitHub. We discovered some new features on GitHub, such as codespace, which is a place for building, testing and debugging code on Github with an instant cloud development environment. We also used the feature, pull requests, on GitHub; that is, when a group member made a change to the code, the contributor could notify every member in the group with what they had recently pushed to the repository using this feature. Our group members used to implement the program using the online programming tool, such as Online GDB. However, since the project that we worked on this time is large, we learnt to use C++ IDE, such as visual studio and Eclipse to program. Therefore, this project let us explores

new programming software and C++ IDE and teaches us with required techniques for developing software in teams.

Our planning skills, communication skills and problem-solving skills also get improved from the experience of working on this programming project in a team. Firstly, we made a really specific plan for the project before we started the actual programming; the plans included the estimated completion date, the division of work and a UML class diagram that described how we would anticipate the system to be structured. Even though we did not follow exactly what we wrote on our plan, it helped us not to rush before the project deadline. Working in a group with others requires us to have strong communication skills. When we had different ideas for the project, we tried to support our own ideas with sufficient evidences as well as understand others' perspectives with an open mind. To have efficient communication, we learnt to deliver a clear message in a short amount of time. Also, developing software in teams gives us many chances to solve problems in groups. As discuss in the design section of this report, we face a lot of challenges while programming the project; for example, we fount it difficult to prevent memory leaks and to check if other cells in the same block with the current cell had already been deleted when we were trying to delete a cell. To solve these problems, we reviewed the programming techniques that we learnt in class, such as decorator. Therefore, this project teaches us with required skills for working in groups with other software developers.

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

There are still a lot of improvements for this project and if we could start over, we would do the following.

a. We should manage our time better with more considerations on what might come up during the examining week. We were planning to complete most of the coding the week before the exam. However, in the first two days of the examining week, we were busy working on the exam for other courses and it did not leave us with much time working on this project. Therefore, if we could start this over, we should start this project earlier so that we won't leave all the programming tasks to the last minutes.
b. We should make sure the basic functionalities work before implementing advanced features. At the beginning, we were trying to implement all the features, including the extra credit features in our program; however, it ends up having too many bugs in our code and we were only able to fix some of them. In the end, we had to clear our code and programed again from the beginning to make sure our program at least supported the basic functionalities.
c. We should make an agreement on what software development platform we are going to use before starting the project and we should have quick learning on the features of the platform ahead of time. We spent too much time on switching between different software development platforms; for example, we had tried to use Online GDB, Eclipse, Visual Studio for programming. Therefore, we wasted a lot of time on uploading changes, downloading codes, and sending codes to the group members. Also, because it was our first time using GitHub, we were not familiar with many

functions that GitHub supports and this led to the result where we did not have sufficient collaboration at the beginning.

d. We would debug while are implementing the program instead of leaving all the debugging to the end. We were planning to spend about two days on debugging; however, we found that when the code was really long, it became much more difficult for debugging. Therefore, if we could start over, we should constantly check if we have a memory leak and memory double-free in our program to reduce the time spending on debugging at the end.