CS246 Final Project Report, Biquadris

Written by Haoqi Zhang
Shengkang Zhang

Index

Introduction

The project is to implement Biquadris using C++ with the knowledge of class, inheritance, design pattern and RAII, etc. The rule is to eliminate the blocks with 7 different shapes which are randomly generated until there is no room for the board to add a block. This is a competition between two players according to their final score. In this report, a description of the project breakdown is given, which indicates how the project is arranged. Also, the questions in the project are answered in the report, supported by the design of the program represented by Unified Modeling language.

Overview

The data structures of Biquadris are separated into several different classes and there exists some relationship between the classes, so that all of them can contribute to the whole program together.

The structures are mainly divided into 5 classes. Each one of the classes' job in the program are given below:

Game class:

Game class makes connections to two different boards (also known as two different players), it owns two board classes, so that some of the boards' content may be changed through Game class.

Board class:

Board class is the one which stores all the data of the process of the game for each player, two board classes are created independently and they have the same working principle. The information of the current block and the next block can be accessed through it, also it has the information of the game (is there space remaining, which spaces are filled, etc.)

Cell class:

Cell class is used for given information of each individual unit of space in the board. It is an observer used to notify the TextDisplay class to update.

Block class:

Block class is mainly responsible for the change of position (move in different direction, but not available to go up, and rotation) and the initial position of generated blocks. It is an abstract class which has 7 different concrete classes based on the type/shape of blocks generated, which are "I", "J", "L", "O", "S", "T", "Z".

Level class:

Level class is mainly responsible for the generation of blocks and some additional conditions of the game. In different levels, the rules to generate a new block are different. It is also an abstract class that has 5 different concrete classes, Level0, Level1, Level2, Level3, Level4, they all have some same method but with different effects.

To let game players see the game, TextDisplay class and Graph class are created.

TextDisplay class:

The TextDisplay class shows the process of the game by characters.

Graph class:

The Graph class shows the process of the game by images, which are contributed by rectangles with different colors and sizes. Some information about game processes like score, level, highest score are also shown as strings in the images. The way of drawing those images is provided by Xwindow.

Design

Factory Design Pattern:

We use this to generate different types of blocks. We define an abstract class which is Level class, to create Blocks, with different concrete subclasses of Level class (Level0 to Level4). These concrete subclasses do not influence each other and have their own algorithm/idea/way to generate new blocks. Thus, such a Design Pattern makes it convenient if we want to add new different levels of difficulty to the game.

Decorator Design Pattern:

We use this to change the level of the game, we can wrap the level of the game over and over, and only look at the outermost shell of the object. Also, we use this to do the special action, and we process all of the shell of the object simultaneously to make sure there can be more than one special action applied.

Observer Design Pattern:

We use this to make changes to the Graph class and TextDisplay class, we treat Cell class and Board class as Subject, and we treat GraphicalDisplay and TextDisplay as Observers. When the state of the Subject changes, observers are notified to update. Also, Blocks can be the Subject and Cells can be the Observer. When a block is moved or being removed, the corresponding cells will be notified and updated.

Polymorphism:

We design both Level class and Block class as abstract classes with pure virtual methods, then implement those pure virtual methods in the corresponding concrete classes. Dynamic dispatch is applied to choose the most appropriate method while the program runs to this part. Since all blocks with specific types and different levels are written in different classes, it is easy to add a new type of block or level and do not cause any effects to other already existing block or level types.

RAII:

We do not use delete in our program, since we use unique pointers in a vector of Board class, which can help us to delete the heap memory automatically when the corresponding heap memories are no longer needed.

Compilation Dependency:

By using forward declaration of required classes in the appropriate, we reduce the recompilation's process, because the dependency in each class is reduced.

Resilience to Change

When we design the program, we think there are several things to consider that potentially need to be changed. The first thing is that what if a new type of block is going to be added? The second thing is that what if a new level is going to be added? The third thing is that what if people do not want to see the graphical display of biquadris (since it is slow or something) or text display of biquadris (since it is not vivid or something)? The fourth thing is that what if people want to see a graphical display with different sizes (maybe the initial one is too small or too big)? Here are the solutions we have.

For the first thing and second thing, all blocks with specific types and different levels are written in different classes, which are the subclasses of the abstract class, which are Block class and Level class, respectively. It is easy to add a new type of block or level and do not cause any effects to other already existing block or level types.

For the third thing, we wrote text display and graphical display into two different classes, which are TextDisplay class and Graph class. By this way, we can just set two boolean values related to them, then decide if one or both or none of them are shown to people.

For the fourth thing, we did not use magic numbers in Graph class which is used for graphing the image and showing the image to people. The only thing that needs to be changed for adjusting the size of the image is to modify the integer field in Graph class, there is no need to change the content in the definition of the method.

Answers to Questions

Question: 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?

Answer: Since our blocks have a field called num, which is an integer used to record the order of blocks that have been generated (num in the existed blocks which is not "empty cell" will increases 1 while a new block is generated), we just need to add a field which is called count, which is an integer, to record the number of the blocks that have been generated but no clear happenings. When count achieves 10, this means no blocks were cleared in the past 10 blocks were generated. Then, we use the field num in each cell, to find the smallest num in the board (the oldest block), clear it, and reset count to 0. If any blocks are cleared, count is reset to 0.
The generation of such blocks can be confined to a more advanced level by adding one more field in the block, let it be an integer called lv which show the level of the block when it is generated, the count field in existing blocks will increase by 1 only for those whose lv field is equal to the level at which a new block is generated.

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

Answer: We will use abstract class and concrete class to accommodate the possibility of introducing additional levels into the system, with minimum recompilation. We can add one more subclass that stores all necessary information of the new Level class to become a concrete class, under the abstract class. By this way, the recompilation only happens to the new Level class (Assume it is level5, only happens to level5.cc).

Question: 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?

Answer: We will use decorator design patterns to implement the subclasses, we will wrap up the multiple effects on the subclasses that need to apply multiple effects simultaneously. Since the subclasses are wrapped, we know that all the effects wrap up on it that can apply to it simultaneously. Also, we know that each of the effects are independent when processed, there is no influence on other effects. If we want to invent more kinds of effects, we can add more subclasses under the abstract class, SpecialAction. These subclasses just need to implement the field and method they need to achieve the new effects. We can prevent the program from having one else-branch for every possible combination, we just need to wrap up all the effects that it will have, and process all of them to the Player classes that have been wrapped.

Question: 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.

Answer: We can write a class which is called Command, and every command has their own method to achieve it. When we want to add a new command, we just add a new method in this class, and add an if statement in the controller method. Then, the only thing that will be recompiled is Command class if we add a new method. When we want to change the name of existing command, we can change the string on the right of if statement, if we want to change “counterclockwise” to “cc”, we just need to modify our code:
if (command == “counterclockwise”) {…}
Here we change it to:
if (command == “cc”) {…}
A controller method is given in Command class, so we just need to add an if statement when a new command is added. For example, we want to add “Giao” command, which may do something, we just need to modify controller method:

```
Void controller() {
…       // other command with if statement
  if (command == “giao”) {
    ...
  }
}
```

Extra Credit Features

RAII: we use smart pointers in the program, and there is no direct delete or new, this guarantees no memory leak can happen. The management of the board is using a unique pointer, and the management of blocks is also using a unique pointer.

Individual Highest Score: our program is available to check each player's highest score in the game, for a better experience of the game.

Change of plan

UML changes:

1. We removed the BaseBoard class, and we implemented all the methods in the Board class, and made connections between the Board class and other classes.

2. We directly set up the relationship between Graph class and TextDisplay class, Graph class and Game class, while TextDisplay class is kept with decorator design pattern. Graph class can draw the image according to Game class and TextDisplay class.

Coding strategy changes:

1. Originally, we decided to store blocks in a vector, so we can calculate the number of blocks that have been removed in each round by calculating the reduced length of the vector. In actual coding, we find another easier way to calculate the number of reduced blocks, that is giving a series (1,2,3,...,n, shown as int idx in Cell class and Board class) to the cell in the board when a block is dropped into the position. Then we can calculate the total number of different series to know how many blocks are left. This helps to calculate scores.

2. When we start the project, we want to change the board from bottom to top when some rows are removed, but then we decide to change the board from top to bottom, since we can change less rows of the board, further increasing the efficiency when the program runs.

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?

We worked in a team of 2 people. There are 3 things that I learned from this final project.

The first important lesson is communication. Communication can avoid a lot of "challenges" such as compiling errors, big changes to the structure of the program, working inefficiently, and so on. If the teammates just work in the way they like, it is always impossible to develop one program successfully, nobody can have exactly the same idea with others without communication. Also, just commenting out the changes a person made is not enough, the changes are still being easily ignored. When we develop the program, at least one call is guaranteed every day, we often discuss and exchange the ideas to make the best we can (AKA less change to current code), also,we made plans to work together and arrange the time (since CS246 is not the only thing in our life), last but not least, we made sure both of us can completely understand the change of code when make changes (we would like to listen to each other). Thus, we do not have a hard time on the project because of personal issues.

The second thing is to organize before writing code. This can be shown in the UML part of the project. A clear diagram helps with coding without unpredictable interruptions. We also shared the idea at the beginning of the project, and reached a consensus. Thus, when doing the project, anything like different opinions did not happen to us and we can always compile and run the code successfully with few adjustments.

Last but not least, making the rules but not restricted by the rules. Making rules can help people to do what they are supposed to do, to prevent unfair workload for everyone. However, following the rules is the most basic thing. Everyone should try their best to improve the performance of the whole team, when their work is done, if they have enough free time, they may help others. This is what a team can bring everyone, not only a program with the grade, but also friendship that lasts forever.

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

Generally speaking, our team achieved the goal and well followed the plan given in deadline 1, we finished the final project as expected. The only thing that we could change if we had a chance to start over is that we want to shrink the length of the plan. Both of us are not workaholics, and we enjoy the slow-paced lifestyle, however, some classes are actually pretty simple, and we gave too much time on it, so we worked for approximately 2 weeks on the project. If we can go faster at the beginning, then we only need 8 or 9 days to finish the project. Then the rest of days before deadline 2 can be considered as part of the vacation.

One more thing we want to change is that we want to have one more teammate, we missed the third teammate because of our slow-paced lifestyle (both of us were going to find one more person, but we did not until the deadline 1 had passed). The total workload is the same for every group, and does not depend on the number of teammates. But on the other hand, we have gained more knowledge because we have done more work.

Conclusion

The final project, Biquadris, is completed by Haoqi Zhang and Shengkang Zhang as expected. We made UML to help with coding, and we made a demo to help others to understand how the program runs, and we made a report to explain and summarize our results. We practiced most of the knowledge that we learned in CS246, we learned how to work as a team, how to use git command (Github), and how to communicate and understand each other. Finally, CS 246 helps people to find friends (if you need a friend, then new one for yourself).