BIQUADRIS

q5du r356wang y89yan

Introduction

Biquadris is a two-player tile-matching video game with special effect functionalities available to add fun to the game.

Overview

There are 24 different classes in our program, which are Board, Block, BlockI, BlockJ, BlockL, BlockO, BlockS, BlockT, BlockZ, BlockExtra, Cell, Position, cmdProcessor, Score, newBlock, SpecialEffect, DoNothing, seDecorator, Blind, Force, Heavy, TextDisplay, graphicDisplay and Xwindow.

Class Board represents a single player's game board. Board is initialized with a number to represent its player and a bool to check if it's in a text-only mode. Its init method assigns the Board a height of 18, a width of 11, a vector of vector of class Cell which contains 18*11 Cell objects in total, a weak_ptr to the Board of another player, a TextDisplay, a GraphicDisplay and a number called HeavyLevel representing the heavy level caused by the special effects. Class Board has methods to assign the Cells in the Board its content and the share_ptr of the Block the Cell belongs to, a method to check if a Cell is occupied, a method to clear the Board, a method to notify its TextDisplay and GraphicDisplay, methods to get and set its heavy level, a method to get the pointer of another Board and a method calculating the number of rows removed and the score added because of the removal of Block.

Each Cell object has a char to record its content, x and y values representing its position, a bool representing if it's occupied and a shared_ptr of the Block it belongs to. Class Cell has a method to check if it's occupied and methods that assign and get the Cell's content and the Block it belongs to and a method that clears the cell.

Class Block is the parent class of 8 subclasses representing different kinds of blocks: I, J, L, O, S, Z, T and BlockExtra (which is used in Level 4). Block has a vector of class Position to record each Cell's coordinates on the board, methods to move and drop the block and check if the block can be moved or dropped. The clockwise and counterclockwise methods are virtual in the Block parent class and implemented according to different cases in the subclasses. Each Block keeps

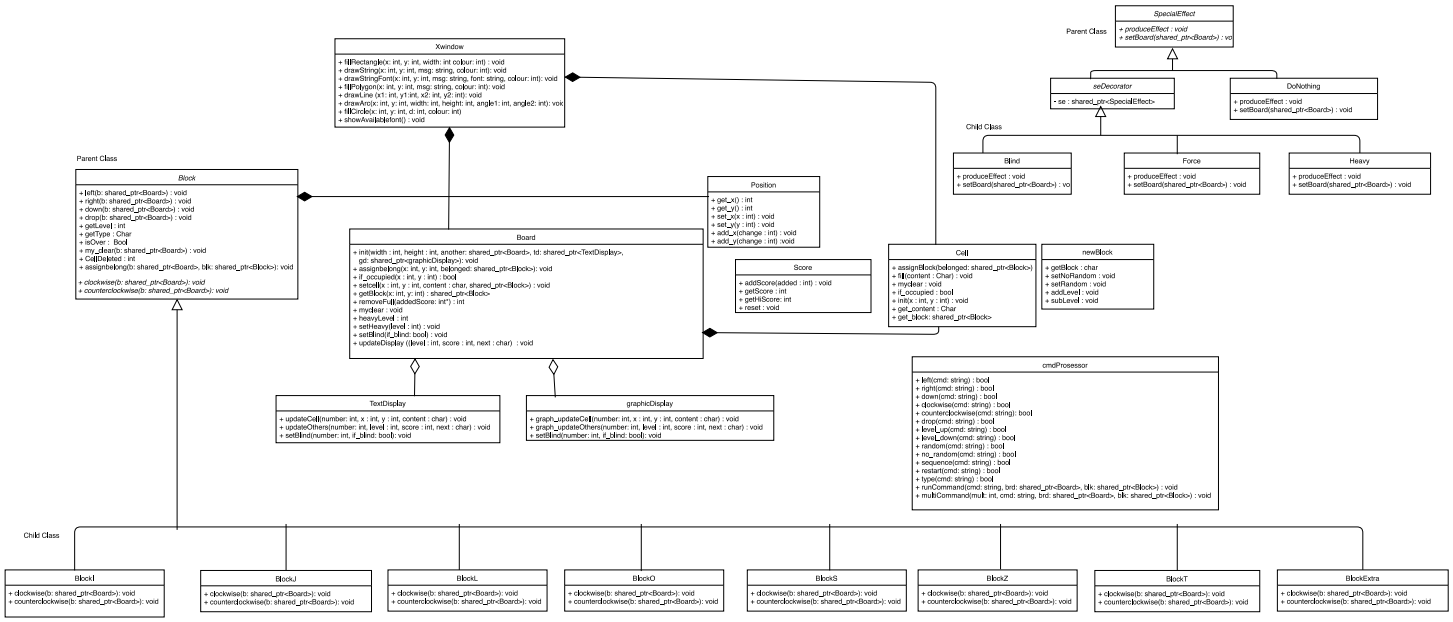
track of number of cells it owns to give us information about when a Block a completely removed.

Class cmdProcessor is responsible for checking the text command entered and processes regular movements. Class Score updates the current score and highest score of each Board. Class newBlock gets the next block's type based on the current level and sequence file. The next block's type is passed into a helper function initBlock() in main.cc that returns a pointer to the initialized Block.

3 special actions are under the abstract super class SpecialEffect using a Decorator Design Pattern. The subclass DoNothing has no effects on boards. Three subclasses of the abstract class seDecorator implement a method called setBoard(Board*) that passes a pointer to the current player's board to set the board producing the special actions and a method produceEffect() producing those.

Each Board owns a shared_ptr of TextDisplay and a shared_ptr of graphicDisplay. Every time the level, score, nextBlock's type or Cell's content of Board is changed, the Board notifies TextDisplay and graphicDisplay by calling their update functions. TextDisplay's output operator prints the current state of the game on the screen, and GraphicDisplay's update methods call the draw methods of Xwindow classes to modify the graphic display.

Updated UML



Design

The first challenge encountered is that nextBlock should be represented by a char or a pointer to Block. At first, we let the getBlock method in the newBlock class creates a new Block and return a pointer of that Block which represents the next Block. However, we found that when we initialized a Block, the constructor of class Block will put the Block in the top-left corner of the Board, which is not what we want for the next Block. Therefore, we decided to use a char represent the type of next Block (main.cc line186), and the return type of the getBlock method in the newBlock class should be a character (newBlock.cc line55). When the current Block in the Board is dropped, get the shared_ptr of a new Block by passing the character that represents the next Block type to the initBlock method (main.cc line33), which initializes a new Block and returns a share_ptr to the new Block, and call the getBlock method of the newBlock object to update the char representing the next Block type. After initializing a new Block object, we need to call the isOver method of that Block object to check if the Block can be put on the top-left corner of Board (main.cc line362). If it returns force, then another player wins, and the game is over.

The second challenge we encountered is how to recognize if a block is completely removed from the board in order to calculate the score correctly.

To solve this challenge, first we declare a field in each cell which is a share_ptr of the block it belongs to. We implement a method in Block called “assignbelong" (block.cc line11) which assigns each cell at the positions this block owns a shared_ptr of the block itself. This method is called immediately after a Block is initialized in the “initBlock” method (main.cc line33). We also declare a field in each Block indicate the number of cells it owns. In the “removeFull” method (board.cc line70), if a line is full and need to be removed, each cell in the line calls its “get_block” method to get the shared_ptr of the Block it belongs to. Then, call the “cellDeleted” method of the Block (board.cc line84). Everytime When the “cellDeleted” method is called, the numCell field in the Block class is reduced by one and returns the number of cells the Block contains. After call the cellDeleted method, check its return value. If the return value is zero, which means that the cell removed is the last cell in the block, and the block is completely removed. Then get the level of the block when it is created by calling the Block’s getLevel method to get the score added because of the removal of this block. After that, set the content of

the Cell to blank by call the Cell's fill method can set the parameter to be space character. In the Cell's fill method, if the content to be filled is blank, it will sets the shared_ptr of the block it belongs be nullptr(cell.cc line33). In this way, when the last the cell of a Block is removed, the share_ptr of that Block will be destructed automatically. After deleting the full line, go through the whole board, set the content and the share_ptr of the block it belongs of each Cell in the Board to those values of the cell above it, and set the line on the top of the board blank. (This part is in board.cc line 82 – line 111) In the process of removing full lines, we need to know both the number of lines removed and the score added because of the removal of Blocks. Therefore, we pass a pointer to int called addedScore which represents to the score added because of the removal of Block. (main.cc line575 & line582), by changing the value the pointer points to, we can get the score added because of the removal of Block. In the same time, since the return value of the removeFull method is the number of lines removed, we can get the score added because of the removal of lines.

Resilience to Change

We adopted the Decorator Design Pattern for creating special effects. Class seDecorator (Special Effect Decorator) is a decorator of the SpecialEffect class and can adapt to changes in new special effects that the design requires. For example, if now the biquadris needs a new special effect, say half-blind or super-heavy, only a corresponding class added under seDecorator will be sufficient for the change.

Our eight different block shapes, I, J, L, O, S, Z, T and *(the special extra block for level four) are subclasses from an abstract class Block, and inherit the concrete methods left, right, down, drop, getLevel and getType, and virtual methods clockwise and counterclockwise with their own implementation for clockwise and counterclockwise based on their own characteristics. This design simplifies the process of creating new shapes, for example, if a new shape, short I (which takes up two cells), is required, we can simplify create a new subclass under Block to adapt to the change.

For additions in commands, we designed a cmdProcessor class where all the existing commands are stored. All commands are first processed by our cmdProcessor class and then be actually

executed. If a new functionality is introduced, provided that the code for that feature is already coded in the corresponding board, cell or block, the only change we have to make is to add a new method (where we do the command validity checking and support the partial command/shortcut feature) for that command in the file. Renaming a command will simply be editing the cmdProcessor file.

In general, the whole project was designed in low coupling and high cohesion. Each class is responsible for only one task, and the degree of interdependence between classes is low. In this way, if we need to make some changes to our program, we only need to change the class that is responsible for that task, which will not cause a lot of recompilations.

Difference and Development

During the designing phase, we did not intend to use unique pointers or shared pointers for our pointers. We then discovered that using unique pointers, share pointers and vectors can not only save us time in memory management but also give a shorter and neater finished code, so we change our design to using these tools.

Classes Added:

We use the MVC Design Pattern and added classes called TextDisplay and GraphicDisplay to increase our cohesion level. Also, we reassigned the task of recording score to a separate class (originally done by Board), called Score, which achieves higher cohesion. For the class SpecialEffect, we added a subclass DoNothing to initialized the SpecialEffect object at first to use the Decorator Design Pattern, and a method setBlock which is essential for setting each Cell the Block it belongs to. For Block, we added a subclass called BlockExtra for level four.

Methods Added:

For Block class, we added the method isOver, which return true when the Block can’t be put in the top-left corner of the Board, which means that the game is over. It is essential in deciding the termination of the game. Also, we added the method CellDeleted for the purpose of determining when a Block is completely removed. For Class Board, we added the method myclear to clear

the board, updateDisplay for updating both the Textdisplay and GraphicDisplay and setBlind which is necessary for setting the blind mode.

Answers to Questions

Q1: 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 conned to more advanced levels?

We can have an integer field "count" for blocks to count the number of times a new block is being dropped after the block. It is initialized by default to be 0 and is being incremented every time a new block is dropped. Once the "count" field is greater than 10, we clear the block.
It won't be conned to more advanced level. Because higher levels only determine the randomness of generation of the next blocks and the "count" field is the same for every block being initialized.

Q2: How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?

We have a class "NewBlock" and a method "getBlock()" which returns the type of the next block based on the current level. In level 0, block types are returned from the block types sequence files; In other levels, the next block's type is determined by its probability distribution, calculated by the built-in function rand() from library <cstdlib>. Then in main.cc, initBlock(char) produces a shared_ptr <Block> to the next block. Since we divided different features into different classes, when new levels are introduced, only the affected classes will be recompiled.

Q3: 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?

We use decorator design pattern. Under the abstract superclass "SpecialEffect". The Core class, DoNothing, has no effect on board. Each kind of special action is a subclass of the abstract

Decorator class “seDecorator”. On line 157 in “main.cc”, a shared_ptr<SpecialEffect> is initialized to be nullptr. After reading in the command for special actions on line 274, the shared_ptr<SpecialEffect> is initialized to be the type of the decorator subclass and applied to the board using methods setboard and produceEffect. So that each action is applied to the board affected from the previous actions and therefore allows for simultaneous application and avoid having else-branch. If we invent more kinds of effects, just add a new subclass of “seDecorator” for the new effect could be created.

Q4: 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.

We will design an cmdProcessor class where all the existing commands are stored. All commands will first be processed by our cmdProcessor class and then be actually executed (cmdProcessor will provide a method to process a command). If a new functionality is introduced, provided that the code for that feature is already coded in the corresponding board, cell or block, the only change we have to make is to add a new method (where we do the command validity checking and support the partial command/shortcut feature) for that command in the file. Renaming a command will simply be editing the cmdProcessor file. To add a command that execute a sequence of commands (“macro” language) will simply be adding a new method to the cmdProcessor file and a else condition to the process command method we provide to other classes. If the new command added has a similar name as an existing command, we will edit the method for spelling checking to update the shortcut feature.

Extra Credit Feature

We did not use any delete or free in our code. No arrays but vectors, and all raw pointers are wrapped by unique pointers or shared pointers. Also, we created a cheat mode. If the command line has "-cheat1" or "-cheat2", which means the cheat mode is on for player1 or player2. If the cheat mode is on, when you can produce special effects, you can choose multiple specials effects and produce them at the same time. Be default, the cheat mode is off, and you can only choose one special effect each time.

Final Questions

Question: 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 together as a group of three people. From a technical perspective, we learned the importance of using tools like GitHub and real-time code sharing. At the beginning, we did not know how to use any of these tools, soon we realized how inefficient it is to email our code every time we make a small change. Then, we started to use GitHub and commit our code to our repository, which we found very convenient compared to our original method. GitHub also support version control with git, which saved us a lot of trouble from accidentally changing our code and being unable to revert to the previous version. Later when we coded together, we found an even more efficient way to share our code, which is to use the Live Share functionality supported by Visual Studio Code. It allows us to see the instant change in our companions' code and even to share our terminal, making debugging three times faster. From a communication perspective, we learned the importance of communicating accurately and concisely through technical terms. We first struggled to express our ideas on design, but then we review together on the terms we learned in class, like the decorator pattern, we then understood each other's ideas and was able to express ourselves better. Also, we learned the importance of making sacrifices and compromises. Since all our group members had different class schedules, it was very difficult to find a time to code together, we then agreed to code together on weekends though some of us were not willing at the beginning.

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

Through the process doing the whole project we've been using Visual Studio Code's LiveShare to write code together. Although LiveShare allowed us modifying codes and sharing terminals, we encountered problem when two teammates are changing pointers in our code to using shared_ptr, while the other teammate was adding "graphicDisplay" class to the program. It resulted in some duplicated works because we need to modify pointers from "graphicDisplay" and main function to smart pointer objects. Therefore, one thing we would have done is using Git to tracke changes, recoding issues and reducing time on duplicated work.
Also, the main.cc for our program is long and cluttered since the whole game and all the edge cases are presented by main.cc. If we can start over and plan smarter, we would design a "Game" class which wraps the whole main function and split each features and edge cases into public methods of "Game" class. Also, we did not start testing until two days before deadline. We tested out some memory errors and behavioral errors and it was stressful since there were not a lot of time left.

Conclusion

Biquadris is a good practice for object-oriented programming in a team. It provides a great opportunity to apply what we learn in CS246, and improve our skills in technical communication.