CS246 Project - Quadris

Course: CS246

Yuxuan Zhang (y2283zha)
Zhenni Gu(z28gu)
Xuemeng Dai (x33dai)

● Introduction

We work as a group of three to design a game project Quadris using object-oriented language C++. Quadris is non-real-time Tetris. When users run the file, they see both a plain text display on the screen and a colorful graphic display on a window. By inputting commands from the keyboard, users may enjoy the game. This program employs good object-oriented techniques. We create a variety of objects and build different relationship between those objects (e.g. has-a, owns-a, inheritance) based on their roles and project requirements.

● Overview

Our final design follows the "Model View Controller (MVC)" pattern. We have the separated presentations of data, class TextDisplay and GraphicDisplay for users to view. The class GamePlay works as a controller. It communicates with the users by reading in commands using cin. Based on the commands, the controller will either use or update the data in Model. We have many classes in Model. Each has its own responsibility. Take class Score and class NextBlock as two examples. Class Score is responsible for storing the highest and current score, score calculation and score updating. Class NextBlock is responsible for storing the information of the next block and deciding the type of the next block based on the level. It is also responsible for storing the level information and update the level as required. When Model is updated, changes will be reflected on "View" (displays) through notifications so that the users see the result of their commands on the screen. With the help of this pattern, the main function is only responsible for basic settings of the game (seed, text-only and startlevel and script file), calling start() in GamePlay to start the game and calling end() to end the game.

As the controller, GamePlay either "has -a" or "own-a" six objects: Board, Block, Command, Score, NextBlock and GraphicDisplay. Board class owns the Cell class and has the TextDisplay class. Whenever a board is created with rows and columns, it will set up a vector of cells so that when blocks are placed/updated on the board, the cells are occupied/updated. Cell class has six fields indicating its own coordinate, type, which indicates types of blocks ("T", "O", etc) that currently occupies the cell) as well as width, height, and a pointer to Xwindow for graphic display. The TextDisplay class that Board class "has" is responsible for printing the text display on the screen. The Board class also contains methods to check for deletion of rows, to clear the board and to determine whether the next block is valid to place on the board based on the block type and whether the cells are occupied or not.

Block class is the abstract class for IBlock class, JBlock class, TBlock class, OBlock class, SBlock class, ZBlock class, LBlock class and a special BarrierBlock class. A Block object has fields: self-level, type of the block, number of variants it has, current shape and a reference position (the usage of these fields are explained in Design section). It also has a vector of pointer to cells on the Board and a vector of positions (we have a class called Position), so Block "has" cells and "owns" positions. The Block class implements the method of moving block "left", "right", "down" because those methods are shared exactly the same by all the derived classes. It also has methods tgdisplay() and untgdisplay() which update the cell type based on the block type and updates the textdisplay and the graphicdislay of the cells the block has. The "drop" method is virtual because the BarrierBlock class acts a little different from the other 7 types on this method (more detail in Design below). It also has pure virtual methods "clockwise" and "counterclockwise" since different types of blocks have different number of variants and those variants have different shapes. There are two extra pure virtual method that updates cells and positions the block has as it moves. The abstract class also has protected methods like canRotateCW and canDown that tells whether a specific command can be performed by the current block or not.

It is worth mention that we have a class Command. This is one of the extra features we have. The fields of the command class are strings that represent standard command names. We will interpret the user input based on the standard ones that we store in this class. The class also has three important methods, including "putOnboard" and "NextonBoard" that put the new block on board when the previous block is dropped. The "execute" method interprets the user input and then executes the user input by calling corresponding methods implemented in different classes (details in Extra Feature section).

• Design

Board, Cell, Block

The first challenge we face in design stage is how we put different types of blocks on the board. Moreover, when we move/rotate/drop the blocks, how do we reflect those changes on the boards and show the result on the "View" for the users to see?

The solution is to build three objects Board, Block, Cell for storing and updating the data with the help of other three objects TextDisplay, GraphicDisplay and Xwindow for display. Both Board and Block have cells. Board owns cells because cells do not exist without a board. But when blocks fall off board, the cells are still there on board.

An board object is first created that initializes all the cells status to be of type ‘ ‘ to indicate that it is not occupied. When “putonBoard” method is called with block type as parameter, the corresponding block class’s constructor will be called. Each block class has a constructor that sets the type of the block (like ‘I’) and the selflevel of the block to indicate the level when they are generated (to calculate the score when blocks are removed) and the number of variants the block type has when they are rotated (‘I’ has two and ‘L’ has four). The constructor consumes two other parameters x and y which indicates the initial position of the block on the board (where to put it). This is important when the users issue the command to change the type of the current block so that we can put the new block to wherever needed and not just the top left initial position. In the constructor, the current shape of the block and the reference position of the block are set. The reference position we use for each block is the coordinate in the left bottom corner of the block. Whenever the move/rotate command is issued by the user, it will first use the four positions the block owns to check if it can be moved/rotated in that way. If yes, block will be moved/rotate by following. The left bottom corner coordinate of the block is first updated if necessary and then updateBcells() method (overwrite in each block class) is called. Based on the reference position and the current shape of the current block, all the four cells as well as the four positions of the cells are updated. Then, tgdisplay(Board *b) under block class will be called that updates the type of the cell and notifies the text display and the graphic display. We thought about using Observer Pattern for display in this case. However, we did not because if the graphic display and text display are the only two observers of a subject, it is not more efficient than just including notify methods in the block class. We need to keep track of the current shape of the current block and the number of variants different types of block has because based on the current shape and variant, we know what shape it will become when a rotation command is made. For example, the current shape of Block L is 0, after clockwise rotate, it will become1 and after counter-clockwise rotate, it will be 2. Based on this and the reference position, the position of all the four cells the block has are determined.

We need to delete the blocks when the game is finished, so the GamePlay class has a field of a vector of blocks. It has two purposes. It is used by the Score class to count which blocks are remove when rows are deleted so that the score is updated. Also, when the game is over, the destructor of GamePlay will delete each block in the vector to avoid memory leak.

BarrierBlock (Bombs)

The other challenge we face is that, in level four, we need to have time bombs. First, we need to create this type of block. Since it has a lot in common with the 7 normal block and we need to treat it as normal block as specified in the requirement, we decide to make it another derived class of abstract block class. It inherits fields of abstract block class, type, cells and positions. It also inherits the tgdisplay (Board *b) and undisplay methods. Inheritance is good since we do not need to link it to “Board” and “Cell” from the start. The drop method is different because it has only one cell and it only drops from the middle column. So, we overwrite the drop method.

Another challenge related to barrier block is that we need to count the number of blocks drops. We create another integer field in the Board class called bomb. When a block is dropped, it will first check if the level is 4 and then if no row is deleted, we increment bomb by 1. When a row is deleted, the bomb is set back to 0. When the bomb reaches 5, the barrierblock will be dropped and the bomb is set back to 0.

NextBlock

In the text display and graphic display, we need to show both the main board and the image of next block. Because they look similar, we actually create two boards of the Block object. One for the main board and the other one for showing the next block. We think this is efficient because the board for next block is just a smaller size of the main board showing only the initial position of one block.

Row Deletion and Score

Row deletion is a challenge because Board owns cells. When we delete the rows, we cannot just delete the cells because they are still part the board. What we do is that we implement the copy assignment operator in the Cell class. When row i is deleted, we copy the cell status(type) from the row above to the current row starting from row i. So each cell in row i column j from row i – 1 column j.

However, using this method, we did not actually update the cells that the blocks have. When a row deletes, part of the cells of the block might get removed. This affect the Score class because we need to update score based on the blocks removed. So, we add a method called updateBlocks. Every time a row is deleted, it will call this method and pass the deleted row number and the vector of blocks as parameters, it goes through each cell in each block. If the cell has the same row number as the deleted row number, the cell pointer will be removed from the block. For the cells with number above the deleted row, its row number will be reduced by 1 (point to one cell down).

- ## Resilience to Change

Accommodate changes

Because our program has a variety of classes and objects with each of them having their own role, it is easier for our program to accommodate changes, like:

1. If more types of blocks are come up with, the program allows the change by simply adding more subclasses of Block (similar as IBlock, JBlock…). The inheritance relationship allows subclasses to add their own features and have anything common in derived class (Block class). If some additional feature is added to one specific type of block, we can make those change in that block's class to minimize recompilation.
2. If we have more command/changes to existing command names or if we need to implement more advanced functions like allowing the users to rename or create their own command (as mentioned in the last question in project specification), we have the Command class where we could just do all the changes there to minimize recompilation.
3. Since we separate "View" GraphicDisplay and TextDisplay from the "Model", if we want the program to display in a different way, like if we want to add a background pictures/drawing or add the player's name, we could do that in these two class to minimize recompilation.
4. Since we have a separate class for score, if the rules for calculating the score changes or if we want to add another record to Score, for example, adding a record of "last-time score", we can simply add another method or field in Score class, and then notify the GraphicDisplay and TextDisplay.

Minimize Coupling and Maximize Cohesion

1. In consideration of coupling, we use the plmp idiom. By forward declaration of the impl class, we can minimize the dependency between classes. For example, in Board class, we forward declare class GraphicDisplay, class TextDisplay, class Xwindow, class Block, and class Score instead of including all of them in the header files. This minimizing the dependency between classes.

2. For low coupling and high cohesion, as we can see in the UML, instead of having each class related to each other, the controller class GamePlay has six pointer point to it. This is not a bad thing because we try to ensure that every module has only one responsibility. This help minimize recompilation. For Board, it only initializes the board, set up the cells, check deletion of rows and check if the board is full or not. It does everything related to what happens on the board, and nothing else. There is no other responsibility like printing on the screen, which is handled by and only by the display class. Same thing for Score, Command, NextBlock, Cell, TextDisplay and GraphicDisplay with roles illustrated in the overview and design section above.
3. Also, we do not use "friend" which allows low coupling. By avoiding keyword friend, and not allowing other classes to access the fields or methods, the program minimizes the relationship between classes.

- **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?

We can add a field to abstract block class to keep track of the lifetime of a block. The initial lifetime of a concrete block should be 10. In the abstract block class, there should already be a field that indicates the level of the block (which level they are in when they are created, not the current level). This field helps us to identify which blocks are subject to this "disappear" feature. There should be a public method in block that decreases the lifetime of a block by one. When a block drops, the public method will be called on the existing blocks (not yet completely cleared) with self-level greater than a given level. Their lifetime will be deducted by one. When the lifetime of a block becomes to 0, we can undraw it from the board. This process is likely coded in the main controller.

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

We could use abstract class. We could have an abstract class called "Level" that contains the common field and methods of all levels. We can make the methods virtual for those ones that differ by levels. In the subclasses, we overwrite the virtual methods. When we create different "level" objects and uses "level" pointers, the correct method will be called. We could create one class for each additional level.

What's more, it might worth thinking about using visitor pattern and decorator pattern for levels. If something depends on two factors, for example, if the effect of move or rotate depends on both type of block and the level, they it might be helpful to have visitor pattern with levels as visitors and blocks as the one who accepts the visitors. In this case, for each level subclasses, there should be method for each type of block (pass by reference as parameter).

Decorator pattern might be useful when advanced levels are based on some lower levels. They inherit all requirements from lower level plus new features. In this case, we could have a base level and a decorator class (in each class, there is a pointer to a level object to add on).

<u>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.</u>

We can create a class of commands with fields of strings that represent standard command names. All the things that has something to do with commands will take advantage of this class then. When we want to add new command names, or to change existing command names, we could change the default value of those fields (or just reassign through constructor). To rename a command, we could add an extra method in the command class called "rename" that takes in two parameters (one is the old command name and the other is the new one). We could use iostream to allow users to input the changes they want to make (old name and new name) from keyboard. In terms of supporting a "macro" language, we think we can use map<string, string> that takes the shortcut name ("llr") as key and the corresponding sequence of commands ("left left right") as value. Again, we could also use iostream here to allow users to input the sequence of commands and the name for it. When the user issues a shortcut command, the program can search the key from the map and get the corresponding value. Then, it just reads the sequence of command as usual. Since all the changes are done in the command class, we minimize recompilation.

- **Extra Credit Feature**

We have the class Command that store all the standard command in it. All the methods that are used to interpret the user input are put in there, for example, isLeft(), isRight(), isRestart(), isValid(). Because we build command as a separate class, we create a mehtod called execute. The user input string is passed as a parameter. It also consumes some other object pointer parameters (like block, score, nextblock, board, etc) because it is responsible for calling public methods from different classes based on user's command. When the input string is left, it will invoke left() in block class. The controller, GamePlay, will call execute method in Command class to execute the user command.

The benefit to have a separate class is exactly the last question on the project specification. When we want to add new command names, or to change existing command names, we could change the default value of those fields (or just reassign through constructor). To rename a command, we could add an extra method in the command class called "rename" that takes in two parameters (one is the old command name and the other is the new one), etc. All the changes are made in the Command class so that we can minimize recompilation.

- **Final Questions**

<u>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?</u>

With respect to developing software in teams, first of all, we found that writing comments and giving the variables as well functions names that made sense were very important. It improved the readability of the program. Secondly, UML was very helpful for designing the program and dividing parts between group members. When having clear function names and variable names, it was very helpful for members to implement the separate class by having sense of methods from other classes, and built the separate class from them; also, spelling mistakes could be minimized. Thirdly, discussion between group members were always necessary. Due to the fact that everyone had its own thoughts, it was difficult to understand the code written by others within a short time period even if the code was well-named and well-commented. By briefly explaining how the core functions were implemented, other members could have a sense of the overall code and wrote its own part easily.

With respect to writing large programs alone, first of all, diving the large program into several parts was important. The first thing that I should think of was to think about the UML (the relationship between classes and what classes). Without thinking about the details of how specific classes were implemented or what fields or functions were needed, I should think about what classes were needed. Secondly, even if I worked on my own, comments and naming were important. It was easy for a person to forget what exact implementation he or she did, so well-formed comments and naming could help save time and write the code effectively, efficiently and readably. Thirdly, searching for help online was always a good way when stuck. There were many hints and demos on the website to give you ideas of the problem.

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

First of all, we would build a abstract class for Level and put different levels as derived class. We did not do this because there are four levels for now. However, when we implement the time bomb block for level four, we realize some of the method is specific to a certain level, it is very difficult to add codes to the existing ones in the risk of ruining the existing ones. So, it is safer to build a separate abstract class for level. Secondly, we can use smart pointers to avoid memory leak. For example, we can use shared_ptr so that the memory is automatically deallocated when the last remaining shared_ptr owning the object is destroyed. Last, we will start the program as early as possible. If we start the project earlier, we may have time left for testing and add more features and functions to improve our code. Thirdly, we can use design pattern;