Tuesday, August 1, 2023

Design for CS246 Spring 2023 Project – Biquadris

Plans:

- Project breakdowns:

July 18~21: Completing implementing the Block class and its subclasses
July 22~23: Completing the gameBoard class, which manipulate the blocks
July 23~25: Completing the Level class and its subclasses, which uses a factory method and controls the actions, scores and the display of the game
July 26~27: Completing the Levelcontroller, which can increase, decrease the levels, display scores, and interpret the commands to interact with players
July 28: Completing main functions, as it will take in arguments while start

Test: throughout the process of development

Person of response: Minxuan Li, for all the work as a single-person team

The development process: (Look back)

The progress of development stuck to the original plan for the most of the time. However, as I underestimate the workload of this development, for a few days in the late July, I had to work for hours to implement the required features. As I didn't stick with RAII patterns in the beginning, it was impossible for me to change my program into this pattern in the last day of the development, and that is the thing I need to be careful with for other upcoming development.

Design of the program: UML attached below: (also a copy of png in the submitted files)

There are 5 individual super classes that helps make the program.

- Superclass LevelController:
  - This class is responsible for taking commands, interpret the command and manipulate the basic rules of the game, (i.e Gameover, High score, player turns). As well as the output to be seen by players. It takes initial parameters to determine if the function like graphical display is opened or not, and stores some initial data. (I.e. start level, default sequence text).
  - The class owns a vector of game boards, levels, and their data. Storing in its private field(in this specific condition, it has 2 of them each). It has a collection of setter and getter public member functions/methods, to make it easy but controllable to access its data by other classes.
- Superclass Levels:
  - This super class is pure virtual, but consists of a lot of basic functions/method implemented for concrete classes(i.e. manipulate game board, Motion of Blocks, Events

etc.). it has a pointer to its game board, but the level doesn't own a game board. When the user want to change his level, the levelController just make a new level and discard the old one, and attach the pointer to game board with the original level to the new level.
    - When generating blocks, since each different level have different rules, some biased random, some true random, and others generate blocks using a sequence file, the team uses a factory method pattern in order to complete this. Each level as a subclass (concrete levels 0~4) override the virtual method for generating of the super class, so when level changes, the generator change as well. The level class also stores data about some event state (i.e. blind, heavy, random), to record if they are on, and when the event will end.
- Superclass gameBoard:
    - This class is the board of the game, which owns a character map to store dropped blocks. It has a pointer to the level controller. When it comes to generate new blocks, it will call the levelController to give it the address of its level (since the level of gameboard may change frequently, it is not wise to store the level address here), and call the block generating function inside level, which differs when level is changed (so it has a pointer to the levelController). The generating function of levels will allocate a new space for the new block, and after making the block, it is given to the game board and stored in the Next Block block ptr. The original next block is moved to this block, and Is tracked by game board on the character map.
    - Class game board has a set of member method for moving blocks. It can move a block to left, right or goes down, by interacting with its blocks stored. After the block is dropped, it calls rows checking functions, to see if some lines are fully filled, and add scores according to the game rule. It is also responsible for the effect of runtime event, like heavy, blind or force. It can also start the opponents' event by obtaining the pointer from the levelController to the opponent level, and call its state setters and event apis. In this way it can indirectly influence the other gameBoard, which is not relevant to itself even a bit.
- Superclass Blocks:
    - This is also a pure virtual superclass, with a set of subclass as concrete blocks. The superclass stores common member method for all blocks, like move right, left, drop, down, or rotate in Both directions. It has a pointer to its game board, as well as a pair of coordinates on the board, making it able to determine if the block itself can be moved to some directs, and is there obstructions on the way that prevent it from moving. This is the most elementary method of moving and rotating, so class game board and levels relies on them to manipulate block motions. The rotation functions are designed to adapt different blocks. It rotate the block by doing math on a matrix, so that if we add new blocks it is also easy to implement without listing out its rotations individually. The character maps are stored in a vector of char, making it extendable for accommodating different kinds of blocks in the future. When we need the char on a given coordinate, the coordinate will be converted into linear position on the vector, make it easy to access.
    - Subclass are concrete, they are all kinds of blocks (I, J, L, S, Z, T, O), and the only difference of them is that they have different maps and different space to store the map.
- Superclass AbsBlock:
    - This is a class for storing the lines a block is on after being copied to the game board and lost its identity as a block. When it is turned into characters, we still need to know if this whole block is gone after a few lines has been fully filled. In order to do this, the instance of this class is created, maintaining the lines occupied by the characters of the block and the original level it is created, and is stored into a vector in game board. When a line is filled, the elements in that vector will check if this line is relevant and change or erase the lines stored inside the instance accordingly. For example, if a T block occupies line 1, 2, and 3, and created in level 3, when line 1 is filled, the remains lines of this block turns 1,

and 2. When all its remaining lines are gone, we consider the block as disappears from the board, and the score which is (3+1)^2=16 is added on to the total score.

Coordinates designes:

The left bottom corner is considered as the origin, with positive x coordinates to the top, and positive y coordinates to the right. Blocks denote their coordinates on the board also according to the position of its bottom left block.
When blocks rotates, it maintains the same position on its bottom left corner, which makes it easy to deal with the coordinate on the board when rotation.

Questions and answers:

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?

Instead of storing character arrays in gameboard, I can store a vector of ptrs to the used blocks, and the used block class has a list of coordinates which will change ever time when some lines are filled. Once the time has gone, and conditions are met, I can erase its characters remaining on the gameBoard with the vector, according to the coordinated that is maintained by the vector.

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

Make a pure virtual superclass and add levels as subclass of it. Each time you want to add new features, just override the generator and the moving functions of blocks. You can also have a new scoring system rewritten if you want. By using a factory method ,I can easily change the generator by changing the levels of the level controller to add or subtracting level difficulties.
(I have already have a level design to accommodate the adding of different levels, following the method I plan to use while planning)

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?

I was thought about decorator, but it would be hard to end an effect especially some of the effect are constant. So instead I just have boolean flags to indicate the appearance of each event. Each event have independent effects, so I can write them without considering there orders. However, if they can be applied simultaneously, I would think about implementing a decorator pattern

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.

Design a STL map for the commands, so that if someone want to rename, put his preferred name as a key, and the original command as a value. Then by checking the dictionary, the program will know which command the user want. For adding new commands, I will compress the commandInterpreter in the most powerful class (level controller) that can control almost all of the game. It will be easy to add new features in the more basic classes and call from the most powerful one. If I would have to support a macro language, I may put the sequence of commands in a stack, with its arguments, and then put it in the map. If some of the commands are renamed, check STL to find its original meaning, and give the original command to the interpreter.

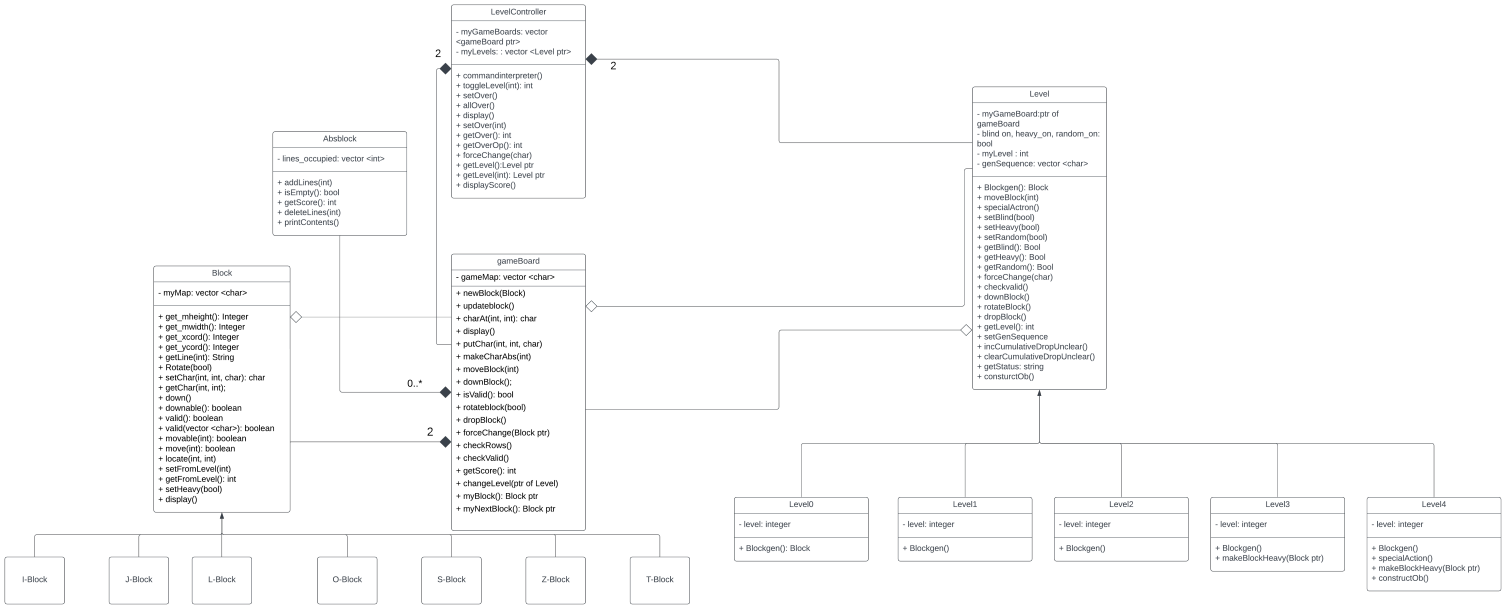
Overall questions in addition:

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?

As a programmer that works alone, I was taught the importance of planning ahead before writing such large program. Since I had developed some other program larger than this using python, I knew a little about planning, but C++ is such a language that is rigorous. The structure of this program became complex as the development progressing, and I had to adjust many inadequate initial designs, as sometimes I underestimate things.

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

I would have more comment while programming, since sometimes it is hard to understand some code written a few days before, especially when it becomes complex. I also need to implement RAII rules since the beginning of the development, since it is too time wasting and risky to fix it into this rule at the end.


LevelController
- myGameBoards: vector <gameBoard ptr>
- myLevels: : vector <Level ptr>
+ commandinterpreter()
+ toggleLevel(int): int
+ setOver()
+ allOver()
+ display()
+ setOver(int)
+ getOver(): int
+ getOverOp(): int
+ forceChange(char)
+ getLevel():Level ptr
+ getLevel(int): Level ptr
+ displayScore()
2
2
Level
- myGameBoard:ptr of gameBoard
- blind on, heavy_on, random_on: bool
- myLevel : int
- genSequence: vector <char>
+ Blockgen(): Block
+ moveBlock(int)
+ specialActron()
+ setBlind(bool)
+ setHeavy(bool)
+ setRandom(bool)
+ getBlind(): Bool
+ getHeavy(): Bool
+ getRandom(): Bool
+ forceChange(char)
+ checkvalid()
+ downBlock()
+ rotateBlock()
+ dropBlock()
+ getLevel(): int
+ setGenSequence
+ incCumulativeDropUnclear()
+ clearCumulativeDropUnclear()
+ getStatus: string
+ consturctOb()
Absblock
- lines_occupied: vector <int>
+ addLines(int)
+ isEmpty(): bool
+ getScore(): int
+ deleteLines(int)
+ printContents()
Block
- myMap: vector <char>
+ get_mheight(): Integer
+ get_mwidth(): Integer
+ get_xcord(): Integer
+ get_ycord(): Integer
+ getLine(int): String
+ Rotate(bool)
+ setChar(int, int, char): char
+ getChar(int, int);
+ down()
+ downable(): boolean
+ valid(): boolean
+ valid(vector <char>): boolean
+ movable(int): boolean
+ move(int): boolean
+ locate(int, int)
+ setFromLevel(int)
+ getFromLevel(): int
+ setHeavy(bool)
+ display()
gameBoard
- gameMap: vector <char>
+ newBlock(Block)
+ updateblock()
+ charAt(int, int): char
+ display()
+ putChar(int, int, char)
+ makeCharAbs(int)
+ moveBlock(int)
+ downBlock();
+ isValid(): bool
+ rotateblock(bool)
+ dropBlock()
+ forceChange(Block ptr)
+ checkRows()
+ checkValid()
+ getScore(): int
+ changeLevel(ptr of Level)
+ myBlock(): Block ptr
+ myNextBlock(): Block ptr
0..*
2
Level0
- level: integer
+ Blockgen(): Block
Level1
- level: integer
+ Blockgen()
Level2
- level: integer
+ Blockgen()
Level3
- level: integer
+ Blockgen()
+ makeBlockHeavy(Block ptr)
Level4
- level: integer
+ Blockgen()
+ specialAction()
+ makeBlockHeavy(Block ptr)
+ constructOb()
I-Block
J-Block
L-Block
O-Block
S-Block
Z-Block
T-Block