CS246-Assignment 5

Group Project

Quadris

Spring 2017

Buchen Dong

Xinyuan Fan

Xiaoxuan Wang

Introduction:

The Quadris is a traditional Russian game, which is favorable and challenged. The essential interesting and tricky part is there will never be an end – as long as the players can keep themselves from fulfilling the game board, the scores go on and on along with the ambiguous to hit even higher. It was rather complicate task to design such a game, especially working with different people. In the developing process, we had faced considerable amount of bugs, and it takes the majority of our times. As a result, we have this well organized and detailed report to address the issues we had faced, as a record for future developments.

Overview:

As a simple version of Tetris, the Quadris has no time limitation for dropping, player can spend as much time as they wanted for consideration where is the best location to maintain “alive” state (can keep play this game). There are 11columns and 15 rows, with 3 reserved lines for block rotation in order to not “fall out the board”. There are seven kinds of different blocks which are as following : IBlock, JBlock, TBlcok, OBlcok, ZBlock, LBlcok, and SBlock. They will all be initialized at the top left of the board, and they can be moved left, right, and downward direction. If the player is decided, they can use the drop function to let the blocks to “hit the ground”.

There are three main parts of the program, Grid, Blocks, and Level. Grid takes the responsibility as a board to hold those dropped blocks, by creating a XY vector (runs from 0 to 10 horizontally, from 0 to 17 vertically), by which every movement of the blocks would be well and clearly record and well as the board. This is our main gaming section, and next blocks would be provided following different rules (different Levels would have various rules). There are seven blocks to choose from, as demonstrated above. Moreover there are five levels, we decide to organized them by creating abstract classes called Block and Level, to set general rule of each block and level. Although these blocks are different, still a similarity definition can be found.

As a summary, the most basic class we have is simplely posn, which just the vector of x-y axis on gamming board, also we used info to tell what kinds of blocks and levels we have, and as well a observer to notify two display classes with a windows classes had been provided. Addition to that we have a subject class to keep the information and state the player is currently at, such as current level, score, the highest score, and next block coming up, associate with observer notifies. Last but not least, as we explained above, Block as a basic class has 8 subclasses: IBlock, JBlock, TBlcok, OBlcok, ZBlock, LBlcok, SBlock and StarBlock. Similarly, Level has 5 subclasses: Level0, Level1, Level2, Level3, Level4. In order

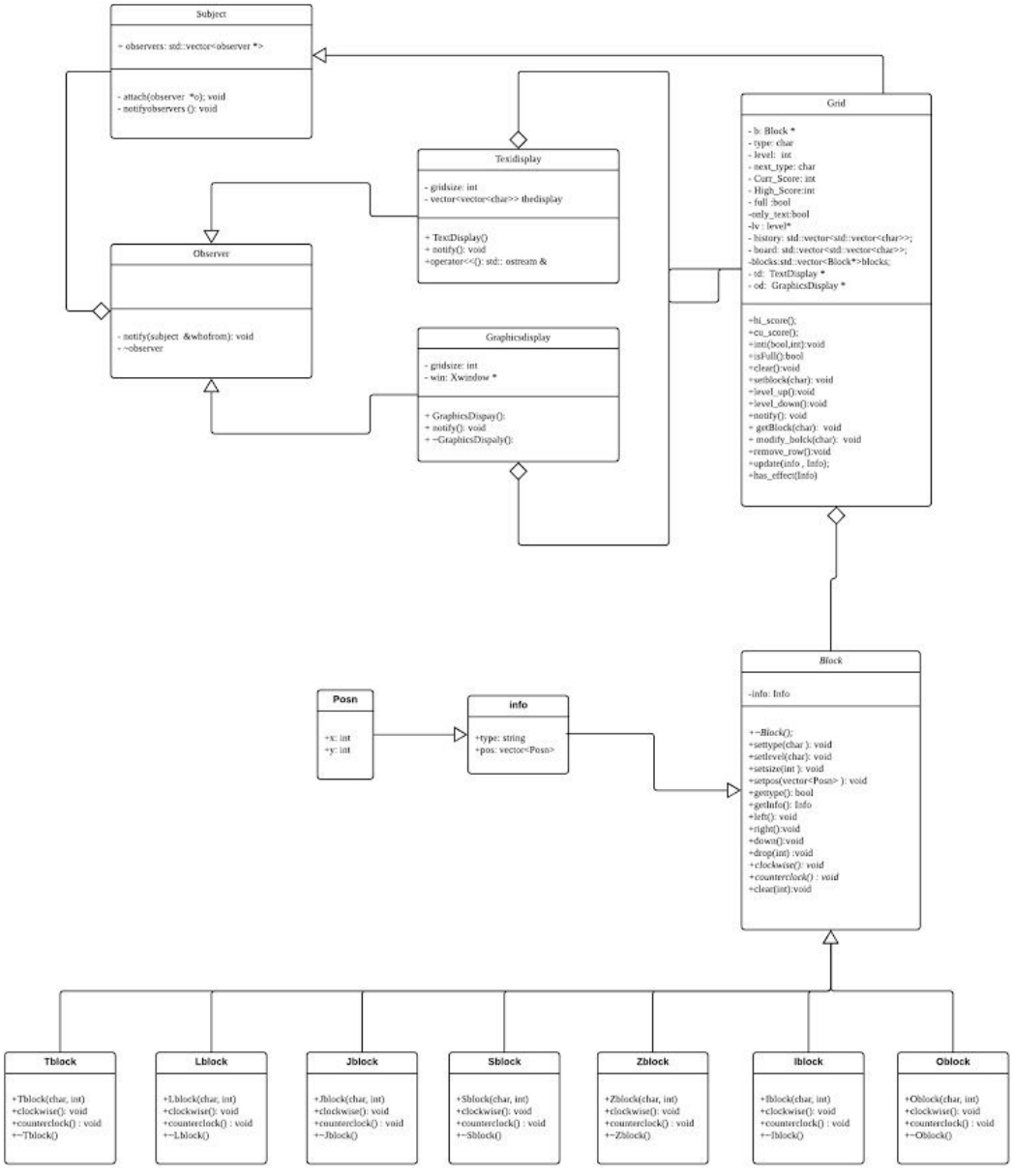
to glue them together, we designed grid taking care of most functionalities; it is a bridge between each component we just discussed (our UML graph have showed a clear relationships between them).

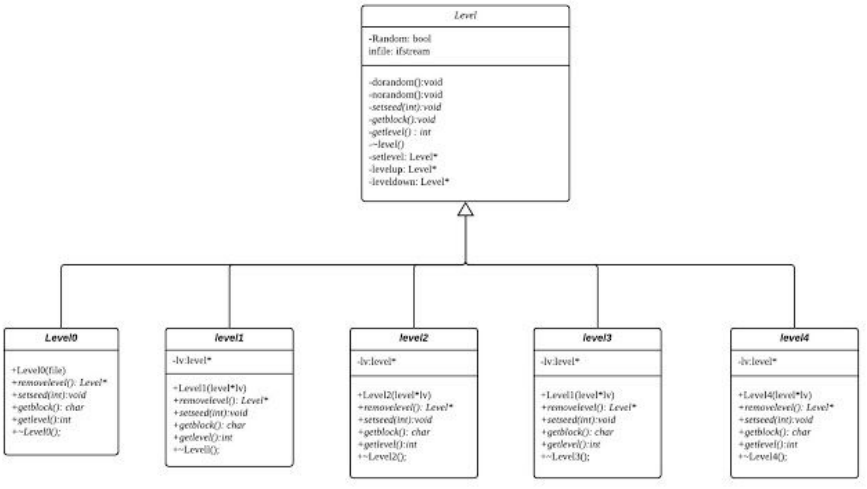
In order to achieve a low cohesion and high compiling design, so we need to separate our design, then grid is the heart of all, while block and level an all portable, that they can be used for any other game with an appropriate attach structures. More specifically, it looks dramatic to have so many classes, while actually it essential and necessary to do so – blocks are all different for initialize and rotation, not to mention each level has distinct rules. (we will discuss these clearly afterward)

A make file is also provided for building up game and./quadric command line to run each command.

UML:

did not change the basic design(the relationship between classes)
change in Grid:
add fileds only_text for the command line interface option
add a argument to inti for the command line interface the startlevel n option.
add a method setblock for the command line interface the enter”L””J” ext option.
change in level:
add field level and size for score and remove method;
change in block:
add setlevel, setsize method for grid to use;
change from virtual destructor to a method clear to free the space the block have used
change in level:
add method setseed for the command line interface the seed option;


Subject
+ observers: std::vector<observer *>
- attach(observer *o): void
- notifyobservers (): void
Textdisplay
- gridsize: int
- vector<vector<char>> thedisplay
+ TextDisplay()
+ notify(): void
+operator<<(): std:: ostream &
Observer
- notify(subject &whofrom): void
+ ~observer
Graphicsdisplay
- gridsize: int
- win: Xwindow *
+ GraphicsDispay():
+ notify(): void
+ ~GraphicsDispaly():
Grid
- b: Block *
- type: char
- level: int
- next_type: char
- Curr_Score: int
- High_Score:int
- full :bool
-only_text:bool
-lv : level*
- history: std::vector<std::vector<char>>;
- board: std::vector<std::vector<char>>;
-blocks:std::vector<Block*>blocks;
- td: TextDisplay *
- od: GraphicsDisplay *
+hi_score();
+cu_score();
+init(bool,int):void
+isFull():bool
+clear():void
+setblock(char): void
+level_up():void
+level_down():void
+notify(): void
+ getBlock(char): void
+ modify_bolck(char): void
+remove_row():void
+update(info , Info);
+has_effect(Info)
Posn
+x: int
+y: int
info
+type: string
+pos: vector<Posn>
Block
-info: Info
+~Block();
+settype(char): void
+setlevel(char): void
+setsize(int): void
+setpos(vector<Posn>): void
+gettype(): bool
+getInfo(): Info
+left(): void
+right():void
+down():void
+drop(int) :void
+clockwise(): void
+counterclock() : void
+clear(int):void
Tblock
+Tblock(char, int)
+clockwise(): void
+counterclock() : void
+~Tblock()
Lblock
+Lblock(char, int)
+clockwise(): void
+counterclock() : void
+~Lblock()
Jblock
+Jblock(char, int)
+clockwise(): void
+counterclock() : void
+~Jblock()
Sblock
+Sblock(char, int)
+clockwise(): void
+counterclock() : void
+~Sblock()
Zblock
+Zblock(char, int)
+clockwise(): void
+counterclock() : void
+~Zblock()
Iblock
+Iblock(char, int)
+clockwise(): void
+counterclock() : void
+~Iblock()
Oblock
+Oblock(char, int)
+clockwise(): void
+counterclock() : void
+~Oblock()


Level
-Random: bool
infile: ifstream
-dorandom():void
-norandom():void
-setseed(int):void
-getblock():void
-getlevel() : int
-~level()
-setlevel: Level*
-levelup: Level*
-leveldown: Level*
Level0
+Level0(file)
+removelevel(): Level*
+setseed(int):void
+getblock(): char
+getlevel():int
+~Level0();
level1
-lv:level*
+Level1(level*lv)
+removelevel(): Level*
+setseed(int):void
+getblock(): char
+getlevel():int
+~Level1();
level2
-lv:level*
+Level2(level*lv)
+removelevel(): Level*
+setseed(int):void
+getblock(): char
+getlevel():int
+~Level2();
level3
-lv:level*
+Level1(level*lv)
+removelevel(): Level*
+setseed(int):void
+getblock(): char
+getlevel():int
+~Level3();
level4
-lv:level*
+Level4(level*lv)
+removelevel(): Level*
+setseed(int):void
+getblock(): char
+getlevel():int
+~Level4();

Design:

Command Line:

1. -text runs the program in text-only mode, no graphics are displayed. The default behavior (no -text) is to show both text and graphics.
2. -seed xxx sets the random number generator's seed to xxx.
3. -scriptfile xxx Uses xxx instead of sequence.txt as a source of blocks for level 0.
4. -startlevel n Starts the game in level n. The game starts in level 0 if this option is not supplied.For instance:./quadric –scriptfile sequence2.txt

Command:

1. left, right, and down: Moves the current block one coordinate to the correspon direction if it is possible.
2. clockwise, counterclockwise: Rotates the block 90 degrees clockwise or counterclockwise.
3. drop: Drops current block to the lowest it can hit. If there are rows being filled completely, those rows need to be removed. (once drop is called, automatically check whether there are filled lines that need to be cleaned from the board, plus automatically calculate score)
4. levelup, leveldown: Increases or decrease the difficulty level of the game by one
5. norandom, random file: Relevant only during levels 3 and 4, this command makes these levels non-random or random, instead taking input from the sequence file, starting from the beginning
6. sequence file: Executes the sequence of commands found in file (automatically execute and print out the result after executing all command in the file).
7. I, J, L, etc.: Replaces the current undropped block with the new stated block. Heaviness is determined by the level number.
8. restart: Clears the board (delete grid — called destructor of grid which clears every allocated memory int class grid) and starts a new game(allocate new grid)
9. hint: Suggests the “best” place for the current block.

Levels:

1. Level 0: Takes blocks in sequence from sequence.txt, or from another file whose name is given on the command line

2. Level 1: there will be randomly chosen block wot probabilities skewed such that S and Z blocks are selected with probability 1/12 each, and the other blocks are selected with probability 1/6 each
3. Level 2: All blocks produced with same probability
4. Level 3: The block selector will randomly choose a block with probabilities skewed such that S and Z blocks are selected with probability 2/9, and the other blocks are selected with probability 1/9 each. Blocks generated in level 3 are “heavy”: every command to move or rotate the block will be followed immediately and automatically move downward by one.
5. Level 4: with Level 3 positive, Level 4 there is an external constructive force: every time you place 5(and also 10, 15, and so on., which the counting will be achieved in main.cc) blocks without clearing even one row, a 1x1 block is dropped onto game board in the centre column. Once dropped, it acts like any other block: if it completes a row, the row disappears. Hence, if player do not act quickly, these blocks will work to eventually split screen in two, raise the game difficulty.

Blocks:

These seven types of blocks with their own initialization, rotation, move. All types of blocks have similar state(top-left) as their initialized format on the board.

Resilience to Change:

Our goal is to minimize the coupling between class, so when we implement for a new lever, we will always create a similar subclass and modify it to adapt new features. Since there is a inheritance relationship between class and subclass, a rule is already well defined. While the subclasses have not relationships between each other, the relation is clear and straightforward.

As a result, we can easily find out which part is wrong, and had affected the whole program. There is a guarantee of convenient to add or delete new different difficulties (we don’t need change the any other part of the programs), the modification is also guaranteed with small changes. Our design for Block is identical, a new shape of block can be created following the abstract class Block. Specifictly write a rotate and move is sufficient in this case for any shape we want.

This is a great aid for debug, and the mantiantion of the clearness of our programs.

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 conned to more advanced levels?

we have the method called history to restore every block's infomation. Then for each block to disappear, block is given a invalid value which is not a valid board posn; just like level 4 we set up counter to determine did we successfully cleared one line. If not, we trick "the game helper" to disappear some exist block. Which is simple task for us, since we have a history record every block had been generated, and a for loop would do this job for us easily and simple.

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

In Level class, we use 5 separated module level 0-4. Then use abstract class to collect all the level. In these files there are different way to generate a block type. In order to reduce the interruption, we derived subclasses from level-collection Level Abstract Class. Then adding an additional level into the system, just define a new subclass from Level and implement its level's rule.

3. 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? 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? Keep in mind the effect that all of these features would have on the available shortcuts for existing command names.

Just add different options for command in main. In main, we separated the command interface and the interpreter and in interpreter the move command is separated from all other command. if a new command is adding , first category the new command the the three type we have. Then according to the purpose of the command, call the corresponding method in grid. rename a command is not hard at all. If it's not a move command just simply change the name. if it is a move command, we have a function to determine the command. for the short cut of the command, we have a function to determine it's full name.

Extra Credit Features:

:) we don't have much bug.

Final Question:

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?

The biggest issue we had faced is updating. We break the program into several different parts, so everyone in charge of different job. For many times, some members are holding a old version of file written by other members, it was rather confusing. We all had more than 10 files of similar but not exact programs. We spend too much time to find the right files, and could never find where is the bug in our own parts. Communication is essential.

Secondly we had huge trouble with observer and notify, we can not connect them very well, writing program in a group is for sure different from writing alone, consistency can be very challenging.

Lastly, our final design is not very different from our initial draft, this for sure is a time saver, we don't need to change our routine a lot. "The mian diraction" is correct, which is we are own a right track. A good planning is critical.

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

If we have a chance to start over, we will create a timeline and write down the issues that need to be considered. We will also better handle the relationship between modules, such as the relationship between grid and block. At the same time, we will spend more time thinking about algorithms and data structures. To sum up, if we start over, we hope to complete this project in a more efficient way, mabey find a online edit software that everyone can edit at the same time, so everybody is at the same page.

Plan of attack

Date	Activity	Bochen Dong	Xinyuan Fan	Xiaoxuan Wang	Note
JUL 15	Plan and design	√	√	√	:)
JUL 16	UML: due1	√	√	√	:)
	Plan: due1			√	:)
JUL 17	Grid	√			:)
JUL 18	Levels（4）			√	:)
	Quodris	√			X
JUL 19	Observer	√			:)
	Subject	√			
JUL 20	Blocks（7）		√		:)
JUL 21	Graphicsdisplay			√	partial provided
	Textdisplay			√	:)
JUL 22	Documentation			√	:)
JUL 22	main		√		:)
JUL 23	Test	√	√	√	:)
JUL 24	Run and debug	√	√	√	

UML is above.

First of all, we decide to break this game down to blocks, board, and levels for comprehensive and logical maintenance, since level is comparatively easier than the other two we decide who ever take that part would also in charge of documentation as well as display. When consider the Grid, we find an observer's pattern would be essential useful, so we developed Quodris, Observer, and Subject as assistances to update the game board. Moreover, we build our blocks individually; there are eight of them (including a special one for game level four) all have their very own version of initialization, movement, and rotation. And we want to use a abstract class Blocks to set the rules for them. As for levels we want to create a set of element and find a way to randomization for next blocks. The way we develop this game, all of its part are "portable", which means low coupling, every part can always be used for other implications and convenient for debugging. Last but not least, the documentation and test are also critical for the development process; we want to update (for instance, hint()) and simplify some programs in order to aim a higher cohesion design.