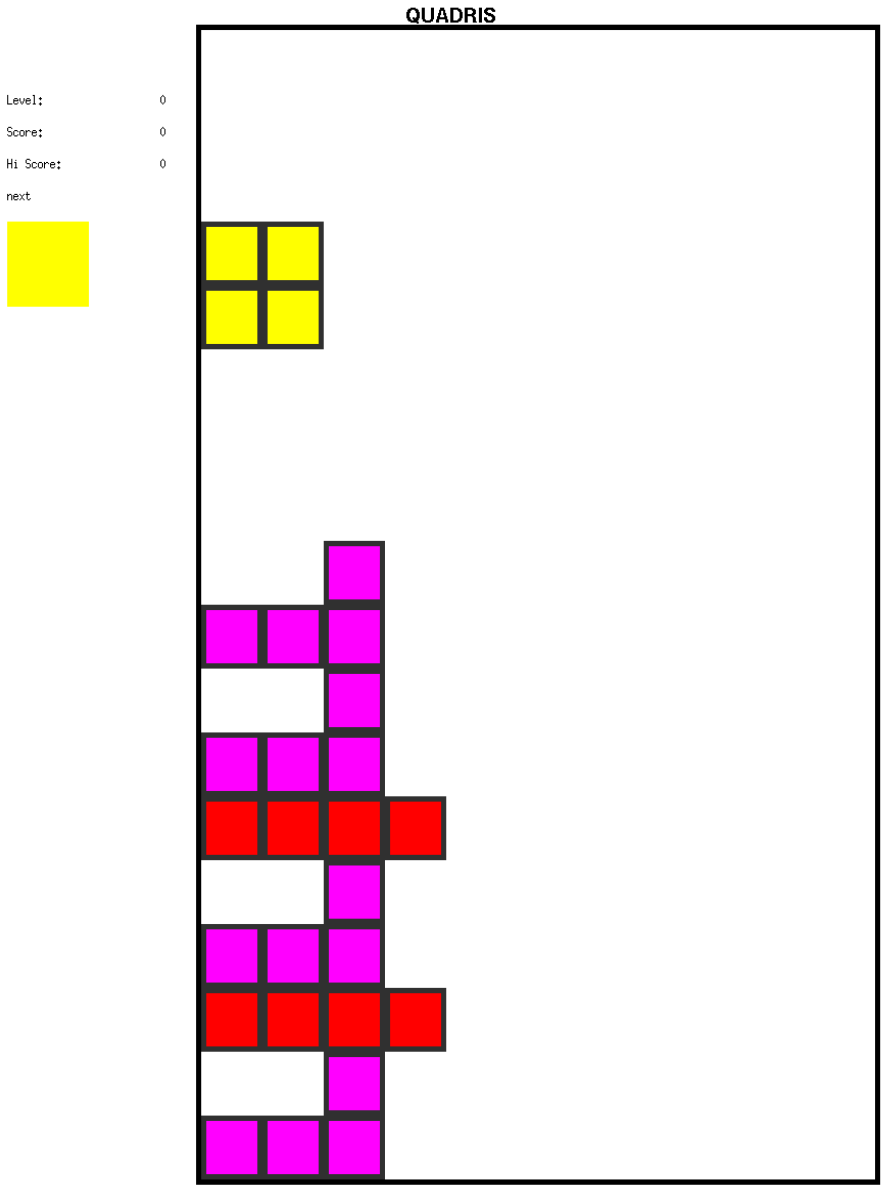
Quadris: CS246 Assignment 5

Due date: Wednesday July. 25th

Group Member: Owen (z745wang), Jimmy (z2di), Aaron (b6hou)



Introduction:

Tetris itself is a simple video game that has been popular for decades, the making of it isn't exactly easy. However, with great teamwork and cooperation, huge amounts of individual effort, and a passion and desire to build a presentable project, our group of three were able to overcome this challenge with hours of coding, testing, and a lot, a lot, A LOT of debugging.

Here, we want to thank all the TAs and Professors of this course. Without all the hard work you put in to help us and solve difficult issues, without the programming concepts and ideas you taught us throughout the term, without the helper files you provided. We are certain that this project will not be as polished and completed as it is today.

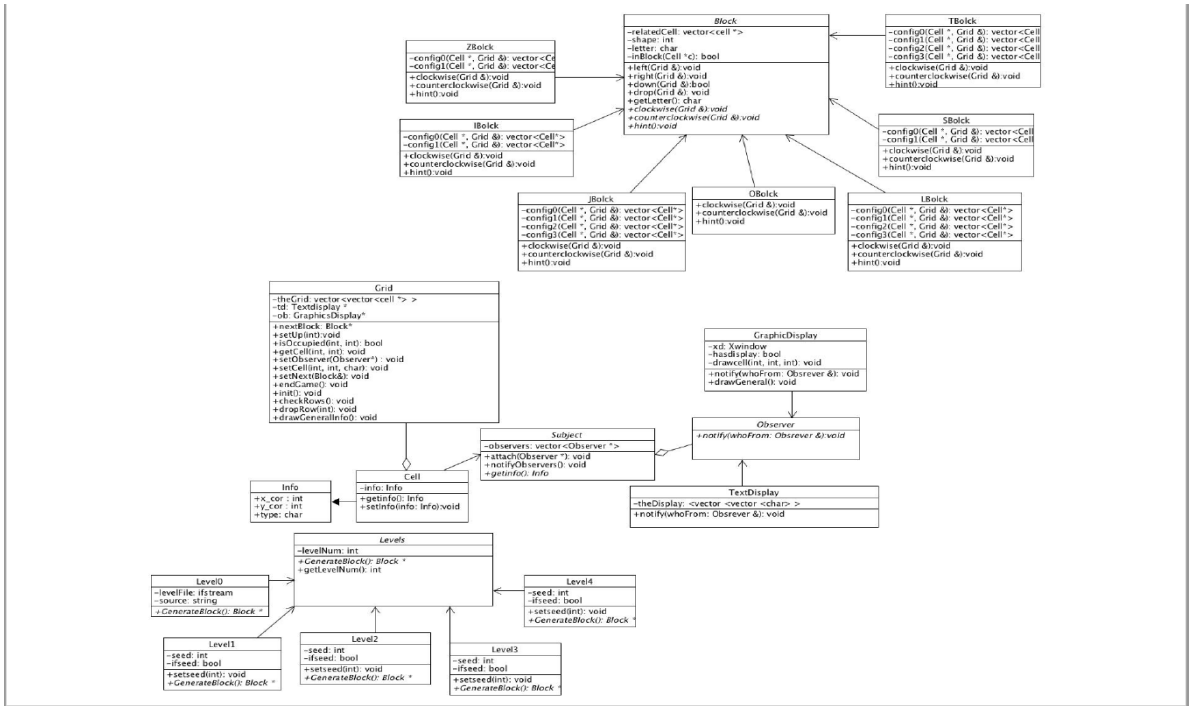
Overview:

This project contains 43 header and implementation files, around 2 MB in size. The main board, Grid class, is achieved using a vector of vector of Subjects. This grid is a very important component of the game since it controls many important aspects such as row clearing and checking if the player has lost (Game Over).

When a new block appears, while the super class Block controls the general movement of the block, such as drop, down, left, and right, each type of blocks, which is a subclass of Block handles the type-specific movement, such as clockwise rotation and counter-clockwise rotation. They do it through a vector of Subject pointers with fixed length of four.

The game has five different levels. We made Level as an abstract class and each level file is its subclass, so they control the percentage of different types of blocks that are dropped for different levels. Furthermore, different levels may implement their own distinct features, such as heavy blocks and extra block drops.

We created a Cell class as an instance of Subject and TextDisplay as an Observer. Furthermore, we created a GraphicsDisplay class using the provided Xwindow file and implemented it as an observer so both graphics changes when new block appears on the block. On the other hand, Subject itself is a very simple class that only attach / notify observers, and returns its Info, which contains a few important information regarding itself. This allows other classes to use the information to decide whether they can or should manipulate or change some aspects of certain cells.


Block
-relatedCell: vector<cell *>
-shape: int
-letter: char
-inBlock(Cell *c): bool
+left(Grid &):void
+right(Grid &):void
+down(Grid &):bool
+drop(Grid &): void
+getLetter(): char
+clockwise(Grid &):void
+counterclockwise(Grid &):void
+hint():void
ZBolck
-config0(Cell *, Grid &): vector<Ce
-config1(Cell *, Grid &): vector<Ce
+clockwise(Grid &):void
+counterclockwise(Grid &):void
+hint():void
TBolck
-config0(Cell *, Grid &): vector<Cell
-config1(Cell *, Grid &): vector<Cell
-config2(Cell *, Grid &): vector<Cell
-config3(Cell *, Grid &): vector<Cell
+clockwise(Grid &):void
+counterclockwise(Grid &):void
+hint():void
IBolck
-config0(Cell *, Grid &): vector<Cell*>
-config1(Cell *, Grid &): vector<Cell*>
+clockwise(Grid &):void
+counterclockwise(Grid &):void
+hint():void
SBolck
-config0(Cell *, Grid &): vector<Cell
-config1(Cell *, Grid &): vector<Cell
+clockwise(Grid &):void
+counterclockwise(Grid &):void
+hint():void
JBolck
-config0(Cell *, Grid &): vector<Cell*>
-config1(Cell *, Grid &): vector<Cell*>
-config2(Cell *, Grid &): vector<Cell*>
-config3(Cell *, Grid &): vector<Cell*>
+clockwise(Grid &):void
+counterclockwise(Grid &):void
+hint():void
OBolck
+clockwise(Grid &):void
+counterclockwise(Grid &):void
+hint():void
LBolck
-config0(Cell *, Grid &): vector<Cell*>
-config1(Cell *, Grid &): vector<Cell*>
-config2(Cell *, Grid &): vector<Cell*>
-config3(Cell *, Grid &): vector<Cell*>
+clockwise(Grid &):void
+counterclockwise(Grid &):void
+hint():void
Grid
-theGrid: vector<vector<cell *> >
-td: Textdisplay *
-ob: GraphicsDisplay*
+nextBlock: Block*
+setUp(int):void
+isOccupied(int, int): bool
+getCell(int, int): void
+setObserver(Observer*) : void
+setCell(int, int, char): void
+setNext(Block&): void
+endGame(): void
+init(): void
+checkRows(): void
+dropRow(int): void
+drawGenerallnfo(): void
GraphicDisplay
-xd: Xwindow
-hasDisplay: bool
-drawcell(int, int, int): void
+notify(whoFrom: Obsrever &): void
+drawGeneral(): void
Subject
-observers: vector<Observer *>
+attach(Observer *): void
+notifyObservers(): void
+getinfo(): Info
Observer
+notify(whoFrom: Obsrever &):void
Cell
-info: Info
+getinfo(): Info
+setinfo(info: Info):void
Info
+x_cor : int
+y_cor : int
+type: char
TextDisplay
-theDisplay: <vector <vector <char> >
+notify(whoFrom: Obsrever &): void
Levels
-levelNum: int
+GenerateBlock(): Block *
+getLevelNum(): int
Level0
-levelFile: ifstream
-source: string
+GenerateBlock(): Block *
Level4
-seed: int
-ifseed: bool
+setseed(int): void
+GenerateBlock(): Block *
Level1
-seed: int
-ifseed: bool
+setseed(int): void
+GenerateBlock(): Block *
Level2
-seed: int
-ifseed: bool
+setseed(int): void
+GenerateBlock(): Block *
Level3
-seed: int
-ifseed: bool
+setseed(int): void
+GenerateBlock(): Block *

Changes to UML and Design Pattern:

Although the original UML provided a good starting point of attack and blueprint, throughout the working process, we found that it contained a few errors and needed some adjustments for quality of life purposes. Therefore, our final UML is very similar to the first version with few minor additions and subtractions.

First, we got rid of the GeneralInfo class and Play class. Although some of the functions within these two classes would have helped when implementing additional features, we felt the general purpose of these two classes are not clear. Furthermore, we simply did not have enough time or resources to complete the implementation and put them to use.

Second, we clarified the relationship between classes. For example, we planned to have a Block pointer and a Level pointer in the original UML, and…well, the compiler complained so we had to get rid of that. This is also done to reduce the coupling between classes.

Lastly, the original UML did not include some finer details that are either used to complete the program requirements, or introduced to complement extra features, or to accommodate design philosophy. For example, we shifted the generateBlock () methods out from the Grid to increases the cohesion within the class.

Design

As indicated in the UML, our version of the Quadris game used the Observer pattern to complete the main bulk of the game. The subject we used in this project is a Cell class, as described earlier, each cell contains an info object, which includes the coordinates of the cell and the type of the cell. Although this field is private, but a client can call the getInfo method to retrieve this information, and then uses the setInfo method to mutate the information.

There are two classes that depend on the Cell class: grid and cell. The grid class possesses a vector of vector of cells. To access these cells, a client must use the provided getCell () method. A grid can also use the setCell() method which consumes an info and call the setInfo method in a particular cell to the desired type.

A block possesses a vector of cell pointers. Any movement commands will cause the block to first set the cell at its original location to empty, then set the cells at its new location to be the desired type. Any rotational movements are handled by its subclasses, based on the Block type. This allows us to add in new block types with minimum recompilation. A block is generated by a Level object.

The level class itself is an abstract class that does not own nor owned by anyone. The main purpose of the level class is to generate blocks. Every level generates the block according to a different set of rules. This allows us to implement new levels with minimum recompilation.

Observers, on the other hand, is responsible for the displays of the game. We implemented two displays: Graphics and Text. The graphical display uses the Xwindow class which is responsible to create rectangles and clear the rectangles from the display. The

GraphicalDisplay class, when notified, will tell the xwindow the coordinate of the rectangle it needs to clear or add.

The textdisplay class, on the other hand, contains a vector of vector of characters that matches the info of the cells in grid. When a textdisplay object is outputted, it prints the grid in a game-board like fashion with all the information such as score, high score, level, next piece, etc.

Resilience to Change:

One of the key point on our mind when we first designed the program is to make it, so the program can withstand changes. We believe with our design, it is very easy to introduce changes and expansions to the game. For example, if we want to introduce a new level5 to the game, all we need to do is to introduce the corresponding .h and .cc files and change the level cap in main. No other file needs to be recompiled. Same idea, if we want to introduce new block types, we will only need its header and implementation files and modify the corresponding rules of block generation in the level file, if necessary. Other changes can be implemented through a similar way. This is because our program follows the principle of low cohesion and all classes are only loosely depending on each other. Therefore, it's easy to introduce new changes with minimum recompilation.

One thing to point out is that the group tried to introduce a new level with different requirements and extra features. However, this was eventually not done due to time constraints. However, the process to revert the changes was very easy. We simply deleted the level header and implementation files, changed the upper level limit in main, and the program worked like the new level has never existed. This is something we want to highlight in our program. Not only is it resilient to change, it's as easy to revert a change as to implement it.

Extra Features:

One extra features we implemented is to allow the user to define shortcuts to existing commands. To implement this feature, we made a command interpreter using the multimap data type template provided by C++, which is very similar to a dictionary data type that stores a pair of strings. We made an iterator using the template and whenever a user command is fed in, we go to the command interpreter to find the according command. This is how we implemented the command-shortcut. Whenever the user wants to rename a command, a new entry is added to the back of the interpreter. This feature supports overwriting – if the user tries to create a shortcut using a previously used string, the later string will be used.

Quadris 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 confined to more advanced levels?

One of the possible solutions are to implement a queue using an array of Block pointers. Every time more blocks are generated, their pointers get push into the beginning of the array. Then we can set all the blocks in and beyond the 10^{th} element of the queue to not visible. To achieve this, we can simply add a visibility field (Boolean) for all blocks objects that are default to True (visible). This can be easily confined to more advanced levels since we don't have to implement the queue and leave visibility as defaults.

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

I believe our design is a good accommodation to introduce more advanced levels since all we need to do is to create a new class of instance Levels and modify the GenerateBlocks(). This way, we only need to compile new LevelN class to introduce new levels.

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? (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 these features would have on the available shortcuts for existing command names.

One of the possible ways is to implement an array of {key, value} pairs, where all the values are the original names of game-commands while default keys are the same as values. To rename game-commands, we could simply alter the keys in the array, and search for the proper game-commands in keys and use corresponding value to execute the correct game-command. To create shorthand for a series of commands, we can register the sequence of command to a new value while its shorthand name to be a new key. To ensure these features work properly even with renamed commands, we can create a placeholder string and loop the search function to construct a proper sequence of commands.

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 have learned many lessons during the assignments about developing software in teams, and the most important one is that the lack of proper communication will literally triple the work we need to do. For example, when we first started, everyone in the group wrote a copy of the own Level class without knowing someone else was working on the same project. Later, as we get to know each other, we finally started to have conversation with other group

members. Thank to that, we were able to divide and conquer Block and its sub-classes evenly (It would be a lot of work to put on one person).

Another lesson we learned is that we need to make contributions independently outside the meeting time or we must speak out. There are times after we assigned work, one group member couldn't finish his assigned part or did not have the time to debug after he writes the code (This is usually me, Jimmy). Instead of asking other group members to cover his parts or do the debugging, I always say "I will do it later". Then when the entire group meets and try to compile the program, obviously, the program doesn't compile. Now the whole group must sit in the meeting room and debug. This is not the most productive and efficient way to use our precious meeting time. The lesson we learned from this is that, if anyone has difficulties finishing their assigned work, other group members are more than willing to help. But since they are not mind readers, you must go ask them directly, so they know what you need.

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

One thing we would have done differently if we start over is that we would have more internal communication between the team. As I mentioned earlier, especially during earlier stages, the team lacks some communication and there were times we were working on the same file. Although we have tried to schedule team meetings, first, it's hard to find a time and a location that can accommodate everyone; second, we spend too much of the precious meeting time on debugging instead of planning and assigning workloads.

Another thing we should have done is to start coding a lot earlier. We have spent a lot of time on planning and creating the UML (and we think we did a fantastic job, frankly). However, we found out later that we are short of time to implement and make our blueprint a reality. It is depressing that we had to delete an implemented class because a class it depends on could not be finished on time.

Conclusion:

In the end, we were able to gain a lot of knowledge and enjoyment from working on this assignment. We were able to learn how to build a product from scratch, and not just sit in my room and code, but as a team. We were able to learn to play off each other's strengths and to specialize what we were good at. We were able to learn to use version control well with GitHub, which made collaboration so much easier. Again, we would like to say thank you to all the staff who made all this possible behind the scene. Good luck to ourselves on the final. Thanks Mr. Goose.