Quadris: Plan of Attack and Questions Answering

Group Member: Owen, Jimmy, Aaron

UML and Design Pattern:

As indicated in the UML, our group is planning to use the Observer pattern to implement a large bulk of this task. In A4, we designed programs / algorithms independently using distinct design patterns and were able to horizontally compare the advantages and disadvantages of each method. Based on A4, all our group members agreed that the observer pattern will be our best option for the following reason:

First, it allows us to introduce features much more easily with significantly less recompilation and overall coding efforts. For example, in A4Q5, we only needed a few lines of additional code and changes to introduce a new observer to the original code from A4Q4. We hope we can use a similar approach to implement a graphics for the game and add it as an extra feature. In addition, if we are not able to complete one or two extra features before the due date, we can simply remove the observer and ensure the rest of the program will still compile.

Second, we believe that another advantage of implemented using observer pattern is that the program follows the principle of Low Coupling, as introduced in class. We felt this will be handy since low coupling essentially means different modules of code are less depended on each other. This enables efficient and cohesive teamwork by significantly reducing the amount of repeated code each team member must write to ensure the individual part is, in fact, functional. We are hoping that this method will also reduce the amount of time we will need to put together code implemented by different people, as this might be debatably the most difficult part of a project.

- Look into the possibility to use decorator pattern to implement rotate.

Plan of Attack:

We are looking to spend approximately 35-40 hours on the main body of the assignment. Thanks to observer pattern, we can divide the task evenly. We hope that each member will be in-charge of one of Block class, the Subject class, and Observer class separately while each member should not spend more than 10 hours on individual part. On Sunday Jul. 22nd , we will get together and fully implement main function and debug possible errors. We should spend rest of the time before the due date on implementing extra features and finalizing documentation.

Monday, July 16, 2018:

- Set up UWaterloo git (PRIVATE)

Tuesday, July 17, 2018:

- Finalize all header files.
- Create a skeleton for int main();

Wednesday, July 18, 2018:

- Implementing Observer pattern
    - Subject (Owen)
    - Observer (Jimmy)

Thursday, July 19, 2018:

- Implement Block and Grid classes.
    - Block (Owen)
    - Grid / Levels (Jimmy)

Friday, July 20, 2018:

- Implement Block and Grid classes.
    - Block (Owen)
    - Grid / Levels (Jimmy)

Saturday, July 21, 2018:

- Implementing General Info and Play
    - General Info (Undecided)
    - Play (Undecided)

Sunday, July 22, 2018:

- Fully implement main() (All together)
- Make sure program compiles

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.