CS246 A5: Quadris (Fall 2017)

Sebastian Hothaza, Hamza Usmani and Miguel Mesa

INTRODUCTION

For this last CS246 assignment we had three different programs we could implement, Quadris, Watan and Sorcery. We chose to implement Quadris, a non-real-time version of Tetris, mainly because of three reasons:

- Compared to the other options, we were most familiar with the game and the logic behind it, as we all had played it before.
- We considered that having this previous experience and being more familiarized with the game would make us feel more comfortable implementing it.
- We also liked the idea of implementing Quadris, as it was really similar to Tetris, a game that when released, had a huge impact on our childhoods

During the selection we also considered the Sorcery game. Though some of us were familiar with similar games such as Hearthstone, there were many warnings and little details that could lead to big errors. Due to all these warnings, we thought that it would be more sensible to avoid Sorcery. Finally, we were not that familiar with Watan or any similar kind of game. This could lead to misunderstanding the requirements as well as making testing very difficult, so we just discarded it.

Quadris is a game that implements a board, cells, blocks, and many movements such as down, left/right and drop. Many other features such as the Level randomness toggling, and the hint system are also implemented in this game.

OVERVIEW

For the final design, we have included some new classes, as well as added new methods and variables to the original classes to adapt them and be able to implement all the functionalities. However, the main design structure was preserved.

Our design consists of a board where the game takes place, this board is partitioned by cells, which in turn specify their state. This board can also contain different types of blocks, which can be any block derived from the virtual block class. It can be rotated both clockwise and counter-clockwise, moved left, right, down and dropped upon user's request. Finally, the Level classes specify different difficulties in which the game can be played. Probabilities involved in block generation are dependent on level as well as the ability to take in a seed for random generation

The Levels and the Board classes employ the factory method pattern and the Cells and both Text Display and Graphical Display interact following the observer pattern. The cells behave as the subject meanwhile the other two classes observe and wait to be notified.

A more detailed description of all the classes used in this project and a brief description of their functionality is outlined below:

Board

Principal class of our design and all the others are built around it. It is formed by cells, which are the pieces forming the grid. The board also contains previously dropped blocks, the current block being manipulated as well as a preview of the next block to come on screen. Lastly, it will also keep track of the level it is currently in as well as the score/high score of the user.

Functionality:

- Check when a row is full using "isRowFull" and when needed remove it in "removeLine".
- Add a block to the board or erase it. We use "addBlock" and "eraseBlock" respectively.
- General moving actions:
  - move: down "moveDown"
  - horizontal: "moveBlockHorizontally". Positive/negative parameter of this method determines right/left movement accordingly
  - drop: "dropBlock" method drops a block and clears any full lines
  - rotate: "rotateClockwise" and "rotateCounterclockwise" simply rotate a block
- Check if a move is legal.
- Add a 1x1 block every 5 moves without clearing any line, "addBlankBlock", in Level 4
- Board also implements "getHint" method, which returns the positions for the best placement of the current block. Players can call this to get a good hint.

Cell

Units that form the board where the game will be taking place. Each cell will have a position of type Pos, and a state that will determine whether they are free or occupied by a specific block. It is also the subject of our observer pattern, so it can attach and notify observers.

Functionality:

- Attach and notify observers. "attach" and "notifyObservers" used for this.
- Update the state of the cell. A cell can have any state that is in the State enum (any of the block characters is a state, such as I, j and l, as well as hint and none)
- Each Cell has a pointer to a block, which can be nullptr or the block which currently resides in the current cell

Pos

Convenient way of publicly storing x and y coordinates for each cell of the board. It is implemented in Cell.h.

Observer

Abstract class that all the observers of our board will have to derive from. An observer must implement the notify method, which essentially tells it the new state of a Cell on the board.

Text Display

Cell observer responsible for printing to console the current state of the board.

Functionality:

- Be notified and get updated when a cell is modified, “notify”
- Overload operator<< so that we get the desired output.

Graphic Display

Cell observer responsible for graphically displaying the current state of the board.

Functionality:

- Be notified and get updated when a cell is modified, call “notify” method
- Print the board, level, score, and blocks as graphics objects. Blocks and cells are rectangle graphics objects, whereas everything else is simply graphics string

Window

Implementation of the window class which allows a graphical interface to be show. Graphics display owns a window and creates an XWindow object, which displays all of the graphical components of the game

Commands

Input received by the user is interpreted and processed.

Functionality:

- Read the command with its multiplier, “readCommand” is invoked
- If “sequence” command received, switch input from user to given file. Once file is read, give control back to user
- Similarly, if “norandom” command is read, the block generation is taken over by the file provided, and will continue in this manner until “random” is called
- Vector of strings where all the possible commands are stored: “masterCmdList
- Interpret command shortcuts, call “autofill” function to autofill the commands, so users do not have to type in the entirety of the command, simply the amount needed to be unique

Global

Storage of all the global variables needed in our program, such as score, high score, current level.etc

Block

Abstract base class from where all the specific types of block are derived. Each block creates itself and implements its rotate methods. All of the different letter blocks are defined as subclasses of Block, as well as the * Block, which is the 1x1 block in Level 4. In the graphical interface, this 1x1 block is a black coloured block.

Functionality:

- Rotation works as the pure virtual methods that all concrete blocks will have to implement. "rotateClockwise" and "rotateCounterclockwise"
- Implements moving a block down and horizontally. "moveDown" and "moveHorizontally" methods deal with this
- Delete a position of a block when it disappears while removing a line using the "removePosition" method
- GetHeavy() method returns if a block is "heavy" or not, and if it is, the game will move the block down one row upon every successful transformation

Level

Abstract base class from where all the different levels will be derived. Handles the creation of the next block depending on the current level of the board.

Functionality:

- Calculate and create next block in the game using pure virtual method "createBlock"
- Probabilities specified for block creation depending on the level as per assignment
- There are 5 subclasses of Level (levels 0 to 4), and all of them can support Block creation from a sequence file. Level 0 can only do block creation from a sequence file. Hence the Level base class provides this sequence file as a vector parameter

UPDATED UML

We can clearly differentiate two parts in our UML; the one regarding the Level and the one containing the blocks, board and cells.

Starting with the levels part of the UML, the structure remained mainly the same, we only included some more specific methods and variables we didn't consider at first, such as adding a "randomOn" or "randomOff" method to facilitate the creation of block being chosen randomly or not.

Regarding the Blocks class we changed some variables from private to protected to be able to directly use them in the subclasses. We also decided to merge both move right and left options to the same method "moveHorizontally". We did this because at first, we didn't consider introducing any parameter to these methods. However, we later realized that we needed to also handle the multipliers that were introduced, as input, in front of the command. Receiving an int we would only need to introduce it positive or negative depending on if we want them to move right or left. We also included a new subclass BlankBlock which will be the 1x1 block.

Regarding the graphical part, in the initial UML, as we had not worked much with Xlib, we only had a basic idea of what elements would be needed. During the actual implementation, we completed this part in the UML as well as included some relations; the Board class owns a Graphical Display and the Graphical Display owns a Window, which will be used as the display pane. Finally, Graphic Display also has a Block, the “next” block, because it is not part of the board yet, but we still want to display it.

TextDisplay class stayed mainly the same, we only realized we also wanted to have the next block to be able to display it.

Finally, the Board class may look as if it had increased a little because of some new methods we had to include but the changes made were mainly trivial or because of some displaying necessities. Some things we added here were; a method to check if placing a block somewhere in the board is legal through the method “canPlace”. This is useful because we are going to check it in a lot of situations and we don’t want to have to have duplicate code. We’ve also added some more specific methods for hint, specifically “clearHint” and “getHint”.

We are really satisfied with the work we made in the first UML because even though we had to make some changes to it, the core remained the same.

DESIGN

The Factory method Pattern

This pattern was useful for the design of this program because this way we would only need to define the interface for creating an object and then let the subclasses decide which class to instantiate. We applied this pattern to Levels so that when you call the createBlock method on a specific level object, the appropriate block would be generated, depending of variable probability.

The Observer Pattern

We decided to include and implement this behavioural pattern in our design because this allows us to define a one to many dependencies between objects, so that when one changes state, all the dependent ones are notified and automatically updated. This ensures the grid is always updated.

The generic implementation of the observer pattern requires having both “attach” and “notify observers” methods in the subject class and a “notify” method in the Observers that will update the state when called. In our particular case Cell will be our Subject and both Text and Graphic Display will be its observers. Cell knows its observers as we have attached them to it, when the state of a cell changes the observers will be notified. If we only want to work in text mode or graphic mode then only the specified observer will be notified. It would be easy to add new ways of displaying the program, as a new observer could simply be added to Cells of the Board. For example, lets say now we want our game in a web application, this would simply be added as an observer to the cells of the Board. When notify is called, we will update the grid at that position to its new state.

Inheritance

We used inheritance in our program mainly in two cases, Blocks and Levels. In blocks we decided to use inheritance because there was a significant amount of code that is common amongst all blocks; inheritance would avoid duplicate code. For example, all block children have identical “moveDown” and “moveHorizonatally” functionality. Thus, it makes sense to only code those movements in the base class, “Block”.

Inheritance also makes the code much more flexible. If the return type of a method is the superclass or a pointer to it, then all the subclasses will be valid options.

In Levels, all block creation follows the same pattern; a block is created and then returned. It would be senseless and tedious to implement each block return type in each level class and as such we simply implement returnXBlock in the superclass, where X is a specific block type.

‘Owns a’ relationships

We decided to use “owns a” relationship between out board class and both cells and block classes because we considered that destroying the board should also destroy the cells and blocks.

RESILIENCE TO CHANGE

Due to the use of design patterns in our design we managed to make pretty flexible implementation. This way we can easily add new features or modify partially some requirements without having to make a drastic change to our code.

The block implementation

We decided to make a base class block that will apply to all the different concrete blocks. To do this we looked for the feature that all blocks had in common but that had a different implementation for each type, so a pure virtual method to be able to generate inheritance. This was their capacity to be rotated and to create each type of block we will need to define the positions we want it to take, in other words its shape and the new state its going to represent. By implementing the rotation methods clockwise and counterclockwise we will have created a concrete block. This way each time we want to add a new type of block, we will just need to create a new class that inherits from block and implements the previously described requirements (such as rotate clockwise and counterclockwise).

Commands

Commands are handled in an “else if” chain in main. Adding a new command is fairly simple and modular. One would simply need to add the command to the master_cmd vector for autofill functionality. The second, and final, step would be to add the corresponding “else if” statement where they can insert the desired behaviour of the newly implemented command.

* Note that we exclude the discussion of the actual command implementation in the core classes as that is completely dependent on which command you’d like to support.

Factory Method Pattern

Level follows the factory design pattern which allows easy adding on of new levels. One would add the new Level_X class (as a child of the Level base class) and create their blocks however they choose. This modularity is provided by the virtual Level superclass.

Observer Pattern

The observer pattern through its attach and notify observer methods allows modular behaviour in the context of adding on new observers. For instance, in the future if one wishes to add HTML support for this game for a web version, they can attach the web interface as an observer.

ANSWERS TO ASSIGNMENT QUESTIONS

Please note that the answers to the following questions have not changed since due date 1, since our answers were relevant and we decided to implement our code following them as well.

Question 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?

The approach we thought would be more appropriate to solve this requirement is to add a private member to all blocks. It would work as a counter; before a new turn starts, we add one and check if it is equal to 10. If so, it means that it has been in the board for 10 turns already and we just delete it and update the grid accordingly.

With this solution, it would be trivial to generate the blocks in any level as the counter is initialized in the block constructor.

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

A solution we found to this problem was having an abstract class, Level, and all the other levels build from this base class. This design would allow us to be able to add new levels in a very simple way, namely by following the factory method. Each Level subclass must simply concretely implement the pure virtual methods in Level base class such as "addBlock". Regarding the compilation, the new Level subclasses we will create will never affect the already existing ones, so when recompiling our program, we will only need to compile the new level added and not recompile the entire code.

Generic techniques which are simply best-practices in code should also be done, such as using forward declarations rather than unnecessary include statements. Including header files should only be used when the actual implementation in the included files is required. This way we will be avoiding unnecessary compilation dependencies. Another way of reducing dependencies and therefore re-compilations, would be to take all private members out into another class and replace them with a pointer to the implementation. This way we will not need to re-compile when the private members change.

Question 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?

Having a file such as "commands.h" that is included in main.cc can make this possible. This commands file can handle all user-input and commands, and do the appropriate commands. Since Level/Block are abstract classes, new commands could be implemented all the way down to each block of each level, just by implementing new concrete subclasses.

Similarly, renaming an existing command could be done fairly easily as well, in the commands.h file. If the "rename" command is passed in by the user, the variable name which stores the command such as "counterclockwise" can simply be changed. This implementation will require that the original commands are stored in variables as strings (for example).

Also, because of the fact that commands are in main.cc or in the commands file, and not coupled with any actual implementation of the program, it will be easy to change commands or rename them without overly affecting the actual board or levels. Furthermore, these variable strings of command names can be stored in an initial array/vector, named "defaultCommands" for example, and then a "macro" language can be implemented by simply calling different commands in the array/vector in a specified order.

FINAL QUESTIONS

Question 1: What lessons did this project teach you about developing software in teams?

For 2 of our 3 members, this was the first time using version control (Git) software which was an excellent learning experience as to how software development is conducted in industry. The most significant lesson we learned as a team working together on one major deliverable is the importance of communication of not only which member is tackling which task, but also on the requirements and behaviours of produced functions. Since all the functions in the program coexist and interact together, it is paramount that we define expectations for each function. For example, a team member working on a rotate method must communicate what parameters they require when their function is called by, perhaps, another team member. Planning properly beforehand is just as important as actually writing the code.

Another important lesson we learned was the importance of only pushing code that compiles. Since there are many components to our final project, each being worked on by different people, it makes testing near impossible if others are pushing code that does not compile. If we enforce this rule amongst ourselves, it makes implementing new features and thus building the program much easier.

Question 2: What would you have done different if you had the chance to start over?

- Divide work more sensibly: The way we divided work for this project was primarily based on how many classes each person was to implement. If we had the chance to do it over again, we would attempt to divide workload by logical flow of the program. By doing so, we would have less confusion of whose function provides/requires a function created by another user. It would allow for a more fluid design.

- Started command skeleton earlier on: Our group experienced some challenges implementing the user interface system which required going into previously-finished classes to make modifications. Had we done all the command handling first, it would have been easier to structure the following methods/functions accordingly.

- Make more frequent pushes to Github during the development: Often times a member of the group could code several classes and push them after a considerable amount of time, and although this did not cause any major problems this time around, pushing code incrementally and frequently would be better in communication between members and seeing progress.

CONCLUSION

This last CS assignment was a great learning experience for all of us. We got to experience what collaborating towards a single deliverable was like all while employing various design strategies we learned in class. We got to work with the standard library in C++ as well, and use things in it that were not covered as much in class, and see how powerful the standard library is.

Also, it was very beneficial for us that we used industry-standard version control (git and Github) for the development of this project as that is something we can directly carry outside the classroom. Debugging the program was also much easier as having a group of 3 allowed for more angles of attack and different ideas. We are quite proud of our work and hope you enjoy running it as much as we enjoyed building it.

Below are just some screenshots of the graphical interface when different commands are called.

Figure 1: the hint system being called

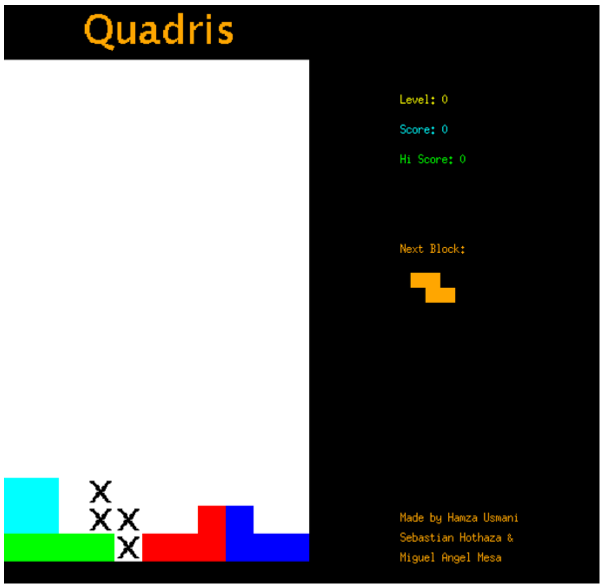
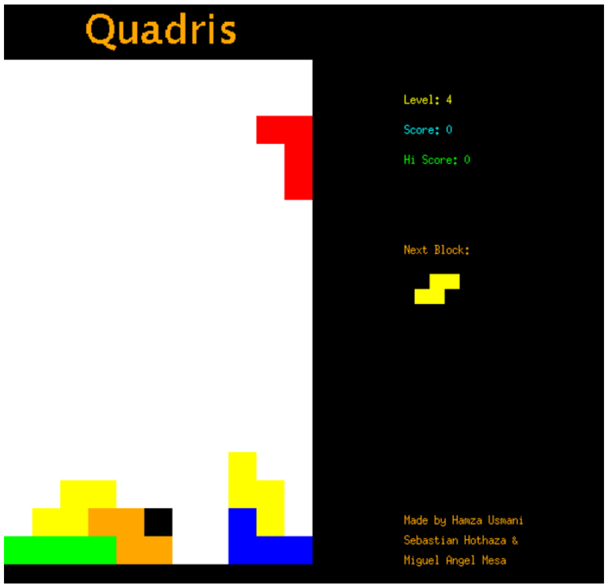


Figure 2: Level 4 of the game


Quadris
Level: 4
Score: 0
Hi Score: 0
Next Block:
Made by Hamza Usmani
Sebastian Hothaza &
Miguel Angel Mesa