

CS246 A5 - Quadris

12.04.2017

Ryan Harrs, Michael He, and Tim Gibson

rcharrs, tj2he, te2gibso

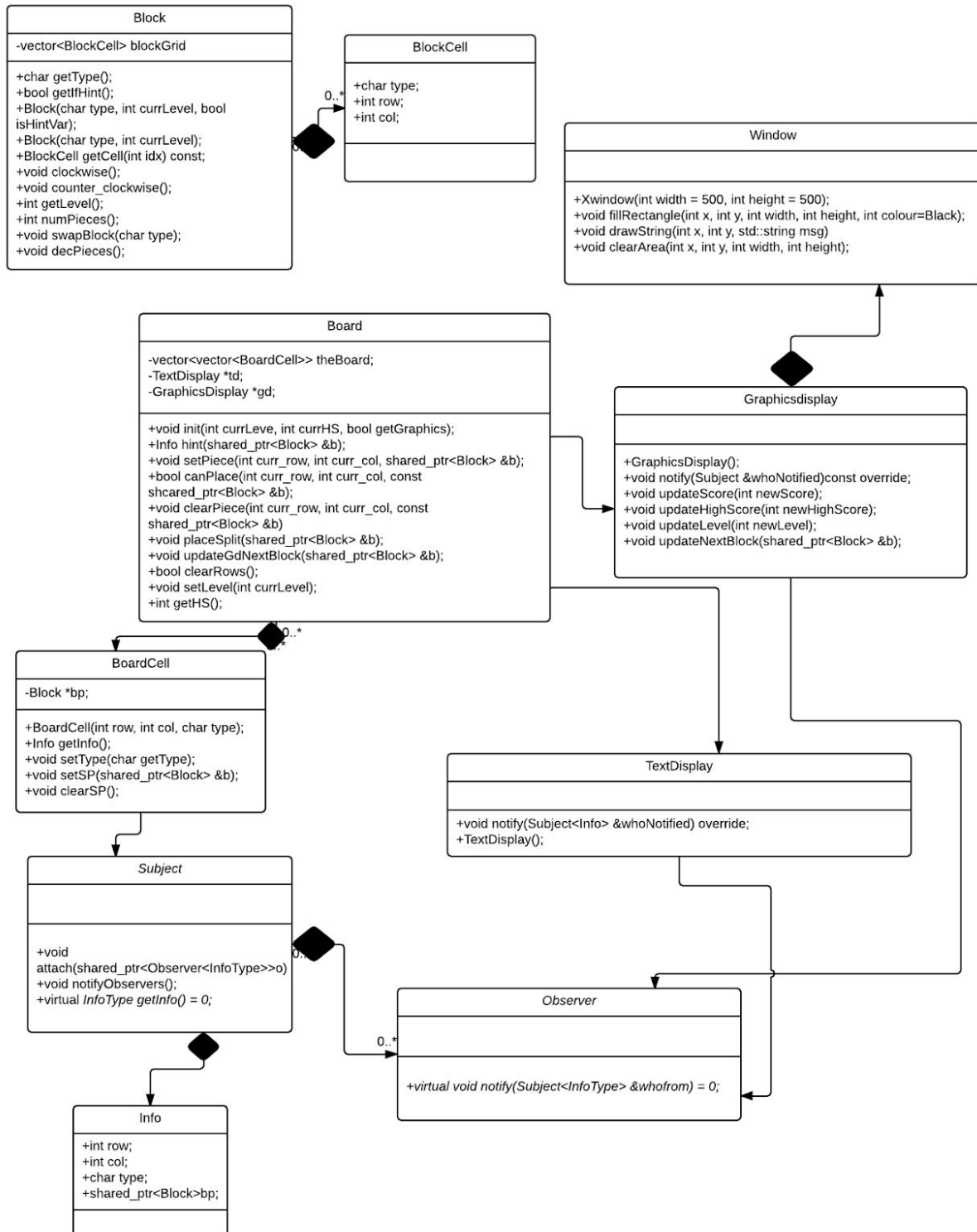
Introduction

Building this game of Quadris from scratch was no small task. Through many hours of coding, much trial and error, and lots of debugging, the three of us came together to build a program we can all be proud of. With great teamwork, lots of individual effort, and a drive to build something start to finish, we managed to piece together a fully functional game of Quadris.

Overview

The structure of our quadris game is all built to surround a board which is a vector of vectors of board cells. This board is the playing field on which we place and/or move quadris blocks using the cells as a grid to place the blocks. Blocks are initialized based on what type they are and are built into a vector of block cells (which they own themselves). Blocks can then be manipulated and then drawn onto the board. When a block is dropped, the board cells get all sorts of info from the block and store that info for later use. In order for us to see what the board looks like, we keep a text display and a graphics display both observing each individual cell in the board. These displays are then being updated every time a cell is changed with the text display being re-printed every time a command is entered, and the graphics display being shown in an XWindow. As the displays are observing the board cells, the displays are observers, which have notify functions, and the board cells are subjects. Subjects are fairly simple classes that control a subject's observers and have an info object attached to it, which can be accessed through `getInfo()`. This method however is virtual and must be implemented by a concrete subject. The Info object is a structure with a few important details about a subject. Board cells have manipulatable data so that we can set/change them to store important information. Board cells differ from block cells are simply structures with a few fields to store information about a cell in a block.

Updated UML



Changes to Original UML

Our final UML diagram is both different and similar in a lot of ways to the original UML diagram. With the original UML diagram we thought up a good plan of attack with regards to the different classes we would need and how they would interact with each other. Thus, the final UML diagram from afar (without the methods and fields) looks very similar to the original one. However, while planning for the original diagram, we failed to account for a lot of the smaller details that would require many fields and methods to be added here and there and even more methods often to accommodate for the new enhancements and so on. As a result, the final UML diagram has a lot of additional methods and fields in each class. There is also a change in the way we manage cells, we separated BlockCells from BoardCells just so that it was easier for us to distinguish between the two and not get confused, as well as BlockCells could be public and we would have more access to manipulate them as we may, whereas BoardCells it is important to keep private so that a user can't change the board.

Design

I. The Game Board

Board objects contain a vector, of BoardCell vectors. This represents our grid of BoardCells. The board also deals with interactions with blocks. Including whether or not you can place a block in a certain position, setting a block to a position on the board, and moving a block around on a board. The game board also deals with creating a hint for where to place a block by finding the lowest spot of where to place a given block. The strategy for finding the right block was too try every rotation if you were to drop it from start in every column and see which version was able to get the closest to the bottom of the screen. This way, you fill up the bottom of the screen faster and will clear more lines and run into less trouble. A good quadris strategy. The board keeps track of the level, the score, and will clear a row when it's full of blocks and update the rest of the board accordingly by taking the row that was cleared, and getting the info of the cell above it and copying this and moving it down, and repeating for each row above the row cleared.

II. Blocks

Our Block objects represent one of the 7 available block pieces in Quadris. Each block object has a vector of 4 cells representing its components. Each cell contains coordinates, allowing for use to keep track of our block. A block object also contains, a position (position of the entire block), a level and a type. This allows for

our block to be easily drawn on our text and graphic displays. Our class also has method `getType()` and `getLevel()` allowing for access to their respective fields. Our blocks can rotate them selves, and keep track of how many pieces of the original block are still left on the board so that when the last piece of a block is deleted, the designated score is allocated.

III. Cells

There are `BlockCells` and `BoardCells`. A `BlockCell` is a struct with the most basic info of a cell for a block: it's row, col, and type. It's main purpose is just to store information we need when looking at a block. A `BoardCell` however is much different. A `BoardCell` is a class that privately contains the same info as a `BlockCell`, and has a pointer to the block from which it came. You can change the info of a cell through different implemented functions. `BoardCells` are subjects, and as such have a `getInfo()` function that returns an `Info` object with the characteristic values of a cell (row, col, type).

IV. Info

`Info` is a simple struct that subjects own. `Info` has fields: row, col, type, and `*bp` which is a pointer to the block that owns a cell (can be `nullptr` if cell hasn't been set yet).

V. Subject and Observers


We used the observer design pattern. The abstract subject class has no special functions other than to deal with observers, and the one virtual `getInfo()` function that needs to be overridden by concrete subjects. The only subjects in our project were cells. For abstract observers, the only thing that an abstract has is it's virtual `notify` method that each concrete observer must have. The two displays are both observers, and they observe each and every `BoardCell`.

VI. The Displays

The displays both start off blank as they are created when the board is initialized and when that happens there should be no blocks on the board. The displays then constantly watch each `BoardCell` and if they ever change, the displays will be updated accordingly.

A. Text Display

When printing the text display, the board actually prints some of the information such as score, level, and high score itself, and then following that



uses the `TextDisplay` class to print the cells. `TextDisplay` has a vector of vectors of chars that contain the type's of each cell on the board, and when the class is printed, outputs the cells in a grid like fashion that would resemble a tetris board. When a cell is changed, the `TextDisplay` is notified and changes its vector to match the cell type that the cell that notified it was changed to.

B. Graphics Display

The graphics display uses an `XWindow`, which is defined by the window class. The window class deals with drawing rectangles and string and clearing areas. Graphics display has functions to update the score, level, and high score as well as drawing the next block on the right side and clearing the areas to update each of the things previously listed. Graphics display creates a grid of its own with dimensions of rectangle blocks, and uses this math to create a grid of its own. Thus, when a cell notifies the Graphics display, the display knows exactly which coordinates to change, how big of an area it needs to change, and to which colour it needs to change it to and does so accordingly.


VII. Overall Implementation

Finally our game is completed in our main file. This file creates a board and alters it according to various commands. For example, blocks are created based on the current level and drawn to the board (utilizing `block/board` methods). Our main function also implements features such as partial and multiple commands.

Resilience to Change

I. Already Implemented Adaptability/Recompilation Minimization

One way that we've implemented adaptability is by allowing simple addition and removal of commands. Commands are stored in a single vector, so adding a command would simply be adding it to the vector (similarly with removing commands). Another addition to adaptability is the `getblock` function. This function fetches a block taking a corresponding string as input. If we were to introduce a new block into the game, only this function and block's initializer would have to be altered. If we had not taken these precautions, altering block types would require much of `main.cc` to be rewritten. Not only specific code, but our approach to this project also has many adaptable components. For example, our board and block classes can be reused for any games with pieces that consist of cells (eg. battleship). Adapting our program to another game would only involve changing the available



type of blocks (which is easily done as mentioned above), and then reconfiguring the logic (to the specific game). This is evident as much of our project was derived from A4Q4/5, the game of Reversi. We used a similar design pattern, and used the same observer state relationship with the cells and displays. We've also implemented a block class that easily handles different block types, and makes creating new block types or adding functions/methods/fields that would mutate/change a block easier. The displays are another example of implemented adaptability. Creating the displays separate from the board allows us to better handle the displays, and add/delete/modify the displays in a separate class. Changing features of the display were very easy as it was easy to access and the code was clean. Conclusively, we as a team set out to try and build a program with high cohesion and low coupling. We believe that the classes we implemented in the ways previously discussed help us to accomplish that goal.

II. Room for future potential additional improvements on minimizing recompilation and increasing adaptability

While creating some terrifically adaptable functions, our main file also has some quite specific code. While this isn't completely avoidable, there was definitely room for improvement. For example, all of the move commands have quite similar code. This code could've been converted into a function, reducing repetitive code in our program. Another function that could've been implemented slightly more gracefully is the "printblock" function. This function is hardcoded to print for each of the seven blocks in Quadris. If we were to add a block it would have to be adapted accordingly. We could improve it by only taking a reference to the current block and printing based on the block's coordinates and type. Since each class has its own module it is relatively easy to adapt to changes and minimizes the need for recompilation. Depending on the functionality of the additions, we can immediately figure out which classes to add changes to. For example, if we desire to change the state of a block we can just add the respective function to our block class or if we'd like to alter the board we'd add a function to our board class. By grouping all functions with similar functionality to a single class allows our program to have high cohesion and makes it easier for us to introduce new features to it. These new features can be flipping the block horizontally or vertically, which can be done by using the already provided `Block::clockwise()` function twice. Because we've grouped functions with similar purposes to the same class it also makes each of our classes independent of each other. This can be seen with our board class, it does not rely on any other class in our program. The board keeps track of our blocks, but it can easily be reused as the board class for similar games such as reversi or battleship. This shows that our classes have low coupling between each other. Coming into the

project we've set a goal to aim for a program that has low coupling and high cohesion and we believe we've done a pretty good job with our finished product. Of course there are many improvements that can be made to increase the low coupling, mainly our main class, it can definitely be seen by the users of our modules that our code is reusable and fairly adaptable to new features.

Answers to 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?

Similarly to the answer for DD1, we would add a bool value to the Block class, but to ensure encapsulation, it would be private and we would create a public method to get the value of the bool (the bool being set when the block is initialized). Then, instead of what we originally proposed of keeping a vector of recently placed block pointers, we think it would be better to add a field in the block class that keeps track of how many blocks have been placed since this block has been placed. Since the board's cells have pointers to the block it owns, each time a block is dropped, all the other cell's can tell their blocks a new blocks been placed and to increment the value that keeps track of how many blocks have been placed since by one. This way, every time a block is placed, we can just go through all the cells and check if each block is a generating block and then if blocks have been placed, to clear the generating block.

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

We think that our original answer is not very efficient nor effective. Having the variables codes into main and keeping track of them in main is quite messy. We think that in the future it would be better to create a level class that main accesses, and the level class would have variables and functions that would help implement the different levels. This would make introducing addition levels into the system much easier since we could do the implementation in the level class, and main would only have to deal with calling and checking values in the level class and/or using the level class methods. Keeping all the additional methods and variables out of main and more accessible in the level class.

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 (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.

We believe that a similar answer to our original answer would be a good solution. By simply having a vector of vectors of strings which are acceptable commands would organize the commands well. Each vector of strings would be different command nicknames, and each vector of vectors would represent a different command. This way, when you wanted to rename/add a new name to an existing command, you can go into the corresponding vector of strings and either `emplace_back` or edit a string in the vector. Then, when if you wanted to add a new command, you can simply add the name of the command as a string to a vector then add that vector to the vector of vectors. Then when main gets a command, it can look through the vector of vectors pretty easily to find the correctly associated command.

Answers to Final Questions

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've learned many important lessons about collaborative software development, most of which revolve around communication. Many issues arose in the earlier stages of this projects due to members working on the same files and overriding each other's work. We quickly learned to convey our additions as well as when we needed others to update the main branch. This allowed for effective teamwork, while minimizing time spent rewriting code. Another important lesson we learned was dealing with group members that wanted to do their own thing. Often people would think they had a solution that would work and go off for a day trying to do that, while someone else would be trying to find another solution to the same problem, and essentially we would lose a lot of time doing things over. We weren't efficiently using our time and resources well. We realized this when we kept getting merge conflicts as we were working on the same thing. After the second time we thought we ought to organize our tasks better and allocate jobs more distinctly. This taught us a valuable lesson on working as part of a team, and overcoming issues among group members. Finally, we learnt how to contribute independently, as a team. Even though we worked together on many occasions, we also learned how to make



efficient use of the time when we could not meet up. Through effective communication, we could determine which classes and functions could be implemented independently, and work accordingly. These learned skills allowed for productive, and enjoyable work.

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

If we were given the chance to start over, we would try to distribute the work slightly more evenly/make sure members were involved with more aspects of the project. Despite doing a good job to make a cohesive product, our members' responsibilities were quite distinct. For example, Tim worked mostly on the main function, resulting in a lesser understanding of certain class implementations. While this is not necessarily a bad thing, this may not be ideal from a learning perspective. Knowledge of other group members "specialties" could also have made for easier implementation of classes, ie. Tim being more hands on with the board class, would have made it easier for him to implement it in his main function.

We also think that planning more in advance would have really helped us complete things with fewer revisions. Due to poor planning and unforeseen problems that often arose, we found that when creating different parts of the program we would need to modify and sometimes completely rebuild/restructure other parts of the project in order to accommodate for new functions, fields, and methods. Had we spent a lot more time and really figured out how we were going to implement everything, and made sure we knew how all the pieces were going to fit together, it would have saved us a lot of time. Instead we just figured out a rough idea of how the classes were going to be related to each other, and how they might interact, and did everything on the fly. This caused a lot of issues and a lot of backtracking, which could have been avoided. Next time, we should definitely plan a lot better. It saves so much time in the long run.

Conclusion

All in all, it was a quite enjoyable experience for all of us. This project really taught us all a lot about building a product from start to finish, and not just by yourself, but as a team. We learned to play off each other's strengths and to specialize what we were good at to work efficiently and effectively in completing the project. We learned to use version control well with GitHub, and learned to make sure we weren't overlapping code. All these skills came together and in the end we became very good at working together and worked as a well-oiled machine.