CS246 | Final Design Document | Quadris

This document provides an overview of the CS246 final project (Quadris). It outlines the structure of the implementation, design patterns used, provides level of configurability on program specifications, and overall insights on the delivered project.

Overview

Quadris is a static-timed version of the Tetris game. Similar to Tetris. It contains the following main components:

- Board
- Block
- Cell
- Level

The following are used to display components:

- Text Display
- Graphical Display

The mechanism by which our game operates is as follows:

- The **main** file creates a board object and initializes it
- The **Board** is created with text and graphical display options turned on, sets the current and high scores
- The **Board** is then initialized with 11 x 18 cells. As well as either a user inputted level or default level for the game and other program start flags such as setting the displays shown.
- **Level** is in charge of generating the appropriate **Block**
- User interacts with the block by entering commands and **main** will pass them onto **board** to interpret
- **Board** will then pass on the changes to **Display**, and display will output to users

Quadris has one board which contains Blocks, Cells, and tracks Level. The board is initialized at the start of the program and destroyed when the program terminates.

The board is initially composed of 11 x 18 cells. Each block also contains 4 cells indicating their position and shape on the board. The available shapes for blocks are: I, J, L, O, S, Z, T, as well as Divider and Hint blocks that will not be shown unless under specific level and commands.

The current level of the game will dictate the next provided block. For levels 0-2, randomness for certain blocks vary. For levels 3-4, obstacles such as a Divider and increasing heaviness for blocks generated in levels 3-4 are introduced.

In order to update board, text display and graphical display relies on the updated cells and blocks every time the user makes a move.

Updated UML

The main structure for our UML did not change. We still have all of the classes we added previously, but we added a subclass Hint for . Below are some major changes include:

Board Class:

- Added a string fields called lvl0File that saves the file name for level 0 to minimize the number of places we need to change if we decided to use a different name for sequence.txt.
- Added three integer fields that saves the seed, random number, counter for divider blocks
- Added a Block pointer that saves the next block
- Added 2 Boolean fields that checks if graphical display and bonus features are enabled
- Added addHint() and removeHint() functions that adds a hint block to the vector and notifies the observers, and removes the hint block from the vector
- Added getLevel() function that passes the current level to main
- Changed the public method clear() to private. Clear() is only called when a block is dropped, and not called by any other class. To maintain privacy, we changed it to private.

Block Class:

- Added integer fields count that saves how many blocks have been fetched since this block is created.
- Added Boolean fields isHeavy to save the block's heaviness.
- Removed field colour, because fetching colour from Block to update graphical Display causes the display to be very slow.
- Added reverseFillState() function that reverses the isFilled field for all the cells of the block. This is used to reverse the state of last block before hint is called, so that hint can move and rotate without being blocked. It is also used to reverse back the state after hint is removed.
- Added setHeavy(bool isHeavy) function that sets the block's heaviness to isHeavy. This is used to set the heaviness of the next block when level changes.
- Added calculateScore() function that calculates the block's score based on the level it is generated in. This eases the process of adding scores when a block is cleared.
- Added updateCount() function that adds the count fields for the block by one. Since block is not a friend of board, board cannot access or modify any block's fields. This function maintains the privacy of block.
- Added getInfo() function that returns a BlockInfo struct containing the current version of Block's private fields. This is a getter methods that lets other classes use the data of the private fields without actually having access to them.

Information Structure:

- Added a new structure BlockInfo that saves a copy of the Block's private fields: type, score, count, isHeavy and cells.
- Changed the old Info structure to BoardInfo since we now need to differentiate between the two info structures.
- Added level and cells to BoardInfo since TextDisplay and GraphicalDisplay need to access the Board's private fields via the return BoardInfo structure from Board's getInfo() function.

Level Class:

- Added private fields for Level 0, 3 and 4.
- Removed protected field level and getLevel() function since the only class that will need this field is the Board class, but now Board has its own level field.

Cell Structure:

- Added Colour field that saves the colour of the cell according to what block it is in. Since now graphicalDisplay updates the cells that changed, it will need the colour of the cell
- Added an overloaded ==operator to check if two cells are the same. This helps the graphicalDisplay to check if a cell has changed, and update it if necessary

Relationship between classes:

- Changed that relationship between block and cells as well as board and cells from "own-a" to "has-a" since when blocks and the board is deleted, the cells still exist.
- Board now uses BlockInfo since it can only access its blocks' data using the block's getInfo() function which returns a BlockInfo structure.
- GraphicalDisplay now uses BoardInfo since it needs to access each cell to see if they have been modified.
- Cell do not need to use BoardInfo anymore since it does not need to know anything about Board. It just needs to change when Board and Block tells it to.

Design

In order to solve various design challenges, we have incorporated several design patterns.

Observer Pattern

Graphical display and text display are set as observers to board. If a change is made in board (i.e. user makes a move), board, as the subject, will notify its observers about such a change, and its observers (text and graphical display) will adjust to the changes.

Factory Pattern

Quadris game has levels 0-4 which increases level of difficulty of the game and enhances user interactive experience. Therefore, we adhered to the Factory Design Pattern to implement levels. Level is the base class for the final product (LevelZero, LevelOne, LevelTwo, LevelThree, LevelFour). The products are the subclass of Level. Depending on the current level, blocks are generated with a different probability and increasing challenges appears in level 3 and 4.

Throughout our implementation, we complied with object oriented programming style using encapsulation, abstraction, inheritance and polymorphism.

In our implementation to generate and store existing blocks on the game board, board contains a vector of blocks. The abstract block class does not exist, instead its subclasses (I, J, L, O, S, Z, T) blocks inherits from the abstract class with additional fields.

Resilience to Change

Since we have used abstract classes for level and block, added an extra level and block design can be easily accomplished simply by creating a new class that corresponds to the new object with its own specifications. In addition, we created vectors which store the possible user commands. Hence, adding new commands can be easily implemented simply by emplacing to the back of the vector. Our original design was a create a command class that performs the translation from a keyword to a program command, as well as supporting user specified “macro” language to indicate a sequence of commands.

However, this was not implemented due to time constraint. In our original design, the command class will contain a method commands(). In the case where the user would like to either rename or add a command, the method will take in the user specified keyword and its target command, and store or modify the information in a map. The map contains a string and a vector of strings. If the user wish to store a “macro” language, this map can also be used to store such information. Moreover, the algorithm to perform each command, as well as to calculate score is stored in a separate method. Hence, any need to modify one component of the program will affect every little of the rest of the program implementations.

Answers to Questions

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?

In our design, board contains a vector of Blocks which indicate the existing blocks currently on the board. We can add an extra counter field to Block which add 1 every time a new block is generated. If any blocks in the board exceeds 10, the block is erased from the vector and the corresponding cells become unfilled. Since board itself will know the current level of the game, we can easily confine this feature to more advanced levels.

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

In our design, level is an abstract class and each level is a concrete subclass. If additional levels are introduced, we will need to add more subclasses and the methods associated with levels. This design will allow minimum recompilation and little modification to other components.

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.

We can create a command class that has one field; a map of string to a vector of strings. The command() method can take in user specified keyword and the string of commands that the user wish to associate with the keyword. The command class can pass this information to main, where corresponding board methods are called. This implementation will support deletion, rename, and addition of keyword commands.

Extra Credit Features

We implemented a feature where blocks will disappear from the board if it does not get cleared before more than 10 blocks have fallen. In our implementation, we followed our original design prior to solving the problem. We added an extra counter field to the blocks which increments by 1 each time a new block is dropped. If any block exceeds 10, the block is erased from the vector and the corresponding cells become unfilled.

We have confined this feature to level 4 only. Since board know about the current level of the game, we simply need to add an extra condition before performing operations for this feature. Also, users to explicitly turn on or off this bonus feature with bonusOn/Off commands. We implemented a field for board to check if bonus features are turned on.

Also, we implemented this project without explicitly managing our own memory and without any leaks. We handled all memory management using vectors and smart pointers. It was initially challenging because we did not have much experience using smart pointer previous, but we looked at the course material and cppreference and eventually gained familiarity.

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 worked in a team of three. This project taught us the importance of communicating prior to coding. Drawing the UML for due date 1 was definitely very helpful. Through discussing about the design, we were able to have a clear idea of the basic structure of our project and each member were on the same page with how we plan to develop the game. Moreover, we each were responsible for a specific class and we communicated either in person or online whenever one class needs to interact with another class. This also helps us to keep track about cohesiveness and coupling level in our design. Overall, this was definitely a unique and valuable experience for us.

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

We would read the project specification together and more carefully. Initially, we each read the requirements and started discussing about our design. However, in the later stage of our implementation and during debug stage, we realized we have either missed or misunderstood some specifications. If we had the chance to start over, we would read the document more carefully, outline all the requirements and start from there. Having a more solid understanding of the rules will reduce the chance of re-implementing a portion of code later on.

Conclusion

Overall, this project enhanced our understanding about object-oriented programming and how various design patterns can be used for a specific part of a program. We were able to gain hands-on experience in developing abstraction and encapsulation in a program. We also gained experience in prospecting possible exceptions and handling them appropriately. Having to develop a large scaled program relative to regular assignments in a team is definitely very interesting. We enjoyed working on this assignment. ☺