<u>THE GAME OF QUADRIS</u>

A NON REAL TIME LATINIZED VERSION OF THE POPULAR VIDEO GAME TETRIS

Prepared by

Siddharth Malik
2A Computer Science
ID - 20642177

Sacchit Chadha
2A Computer Science
ID – 20645242

Table of Contents

1.0 Introduction

The game of Quadris is a non real time Latinized version of the popular video game Tetris. A game of Quadris consists of a board which is 11 columns wide and 15 rows high. Blocks consisting of four cells (tetrominoes) appear at the top of the screen, and you must drop them onto the board so as not to leave any gaps. Once an entire row has been filled, it disappears, and the blocks above move down by one unit. The game consists of 7 different types of blocks, which can be rotated according to the user input. The game consists of both a graphical and a text display, and the user can decide which mode he/she wants to play the game. The game also consists of 4 levels, each with increasing level of difficulty. The user could start a game on a level of his choice, or begin playing from level 0. Further, the user could switch between the various levels while playing the game. There are several predefined commands which the user could use to play the game, or rename them as per his/her liking. The score, high score, and current level are displayed in both the graphical and text modes for the user. The user could play the game until he/she wants with no restrictions on time limits.
From a work oriented perspective, we started by making a rough draft of the various classes and modules we would require for the completion of this project. We soon realized that this task wasn't easy, since multiple classes were dependent on each other, and this only made the task harder. After a few brainstorming sessions, we finally came up with a plan that could be implemented.

2.0 Overview

In our project, the main function of the program starts the game of Quadris. Further we have a Game class, from which the entire control of the program is handled. The Game class is responsible for initializing a new game. Initializing a new game typically consists of initializing a new board of Quadris (18 rows x 11 columns) in both the graphical and text modes, initializing a new block type and displaying it on the board, setting up the score, high score and level for the user, and interpreting the user commands. The many functionalities of the Game class include moving and rotating the blocks on the board, increasing / decreasing / setting the level of the game according to user input or use a default value if a user doesn't specify which level he/she want to start the game from, setting up of the file name (if the user specifies a file from which characters would be read for generating blocks in higher levels, or setting up the file by default for level 0), for generating the next block. Further the Game class also handles the cases, when a game of Quadris is lost and when the game is restarted. This class is responsible for updating both the graphical and text displays after the user inputs a particular command / set of commands. From the Game class, we further handle and interpret the various flags the user passes on the command line (if the user passes any flags), and then pass these to the appropriate classes to handle them.

3.0 Updated UML

Since our UML was quite big, we couldn't fit it here and hence have submitted it directly on Marmoset. In our UML, we have only shown public methods for our classes apart from accesssors and mutators of the particular class. Also we have only shown those private and protected fields of the classes, when we to signify a "has-a", "owns-a" relationship, or when we needed to emphasize a particular design pattern.

4.0 Design Methods & Implementation Methods

- **Factory Method Pattern**
  - We implemented the various levels for the game of Quadris by using the factory method pattern. We decide to use this pattern since depending upon the level, different types of blocks had to appear on the board. For this reason, in our level subclasses for various levels, we generate the character corresponding to a specific block type, and return this character in our Game class. From the Game class, we then generate the specified block type by calling the methods of the Block subclasses and set it up on the board.
- **Observer Pattern**
  - The second pattern which we implemented was the observer pattern. In our program, the Game class is the subject and is responsible for updates the board according to certain events, and hence the score, high score, level, next block type and both the displays are updated accordingly, thus the subject notifies its observers whenever its state changes, and thus we used an observer pattern.
- **Drawing and Updating the Displays**
  - The text display and the graphical display have their respective classes, and both of these classes inherit from the View class, which is an abstract superclass. The View superclass, consists of 2 pure virtual methods, notify and update which are overridden by its subclasses.
  - The text display is implemented as a vector<vector<char>>. The text display is initialized in the board format as specified in the

given guidelines in the constructor of the TextView class. The notify method of the TextView class updates the particular coordinates on the board with a specified character. The update method of the TextView class updates the level, score and high score as well as draws out the board and the next block.

  – According to the flags specified by the user, the graphical display could be displayed along side the text display, or could be turned off. The graphic display is implemented using the Xwindow class. The constructor of this class, initializes an Xwindow pointer to a Xwindow on the heap. The notify method of the GraphicView class draws out the blocks in various colors on the board. The update method updates the score, high score level, and next block type and displays them on the Xwindow.

- **Updating the Board**
  – For implementing our board for the game, we made a Board class which handles the board logistics. For keeping our code simple and clean, we implemented the board as a vector of vector of Cells, where each cell contains a character. Thus, if a cell is empty on the board it contains an empty character, and if it’s filled up, it holds a particular character of a block type. Analogously, we can define if a row is filled or is empty since each row is simply a vector of cells in our program. The Board class has the following features:
    1. Checks whether a row on the board is filled up or not, and if it is filled up, it clears up the row, and moves the blocks above it one unit down.
    2. Updates the score, when a filled up row is removed from the board and when a block is completely removed from the board.
    3. Passes the starting coordinates to a block, hence indicating from where a block should start appearing on the board.
    4. Clears the entire board once a game is restarted.
    5. Checks whether a particular cell is empty or not.
    6. Maintains a vector of all the blocks which appear on the board, and removes those blocks from the vector which are removed from the board
    7. Updates the cells with a particular character when either a new block appears on the board, or block(s) are moved and rotated on the board. The text and graphic displays are also notified at this point.
- **Levels**
  – We implemented our levels using the factory method pattern. We made an abstract superclass of Level and each of the individual level subclasses inherit from the Level superclass. In our implementation each of the level subclasses return a character for

the next block type according to the algorithm specified in the guidelines. In our Game class, we check which character is returned and accordingly update the board and the displays. Also, in our Game class we check whether the blocks are heavy or not, and if they are, we move them one unit down (if possible) for movements and rotations.

- **Command Interpreter**
  - For interpreting the text-based input which a user would enter for playing, we made an Interpreter class. According to the input that a user enters to play the game, this class handles the user input and determines if the input is valid or invalid. If the input is valid, the methods of other respective classes are called, which handle the board, block and display logistics.
- **Scoring**
  - For managing the scoring of the game, we made a Score class in which we made getters for getting the current score and high score. Further, we also made methods for resetting the score and setting the high score equal to the current score if the current score exceeded the previously set high score.
- **Block Movement & Rotations**
  - The game of Quadris consists of seven different types of blocks, and their various rotated configurations. We decided the best way to implement this would be to make an abstract superclass of Block, and make a class for each type of block. The classes for the individual block types inherit from the Block superclass. Each block is implemented as a vector of Coords, where each coord contains a pair of integers, which represent the x and y coordinates. The Block superclass provides the following functionality to its subclasses:
    1. For movements of blocks, it checks whether the required cells on the board are free or not (by running a for loop over the isFree method of the Board class for the required coordinates of the block). If the cells are free, the block is moved to the new cells and both the displays are notified. If this isn't possible, the block isn't allowed to make the movement and the displays are not notified. For rotations, the coordinates of the block are updated first and then we check whether the cells at the updated coordinates are empty or not. If they are, the rotation is done, the block rotates into the new cells and the displays are notified. and if they aren't, the block returns to its original configuration and the displays are not notified.

2. According to the rotations of each block, we made methods for each individual configuration of a block. In each of these methods we update the coordinates of the block as per its new configuration. Our Block base class provides 2 pure virtual methods, clockwise() and anticlockwise() which each of its subclasses override. Now when a block is rotated clockwise or counterclockwise, we check the current configuration of the block, and then according to its current configuration we update it to its next configuration by calling the configuration methods.

5.0 Resilience of Change

We have implemented all our classes in such a way that adding / removing features from most of these classes is quite easy to do, with minimum recompilation. We tried our best to make our program as flexible as we could, without losing encapsulation and invariance.
For implementing various levels, we made an abstract Level superclass, and each level has a corresponding class, and all of the level classes inherit from the Level superclass. Thus if we want to add / remove a level we can easily do so, without affecting the functionality of the remaining levels. To illustrate this point, we've implemented an additional level, Level 5, which returns characters to certain new types blocks (C, N type blocks). Similarly, we have created 2 new block types, the C block and the N block to the existing block types. Creating new block types didn't require much work since each block type has its own class, and all of these classes inherit from the Block superclass. Thus, we can conveniently add or remove block types with minimum changes and recompilation of our program. On a similar note, our program is capable of supporting more than 2 types of views, since each of the text and graphic display have their respective classes which inherit from the abstract View superclass. One could create a mix of graphical and text based views, without affecting the functionality of the other views types.
In our program, the level, views, score, cell, and coords classes all have high cohesion, this is because these classes consist of functions which work on the data members and fields of the class itself. In addition, these classes provide an interface for other classes to access their information. For example, in our coords class we have 2 integers and this class has setters and getters for the 2 integers. Each of the block and the board classes "has-a" coordinate or a set of coordinates. The board and the block could use the objects of the coords class

easily due to the interface provided by the coords class. We have also tried to minimize coupling buy implementing OOP features such as encapsulation and invariance. We haven't declared any friend class in our entire code, since doing so weakens encapsulation and makes a particular class dependent on other classes. Further, in our block subclasses, we decided to make the methods for the various configurations of the block as private, since these methods were only being called by the public members of the class and the objects of the class didn't need to call them explicitly.

6.0 Answers to Questions of Due Date 1 & Changes

We prepared a schedule for us follow for the completion of this project, and we followed it religiously. We made sure that we didn't miss any deadlines and stayed in constant touch with each other. We made a few changes to our UML, which we have updated.

Question: 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?

Answer: We could make a vector of Block * in our Board class. As soon as a Block appears on the Board we would assign it a health of 10 and add a corresponding Block * in the vector which we created. Now, whenever a new Block appears on the Board, firstly we would assign it a health of 10, secondly we go over through all the Block * in the vector which we created and decrease the health of each Block they are pointing to by 1, and lastly we add the new block (with health 10) to our vector of Block *. In this way, when a Block's health reaches 0, we remove it from the Board and hence from our screen and thus it gets removed from the vector. When the Block is removed from the Board, we could fill those Cells on the Board with an empty character, which means that these Cells have now become empty. Further, we move each column of alphabets for each Cell (for each of the four Cells that became empty by the disappearance of the Block), 1 unit down. This way the Cells which were emptied now get filled with the character above it. Hence in this way we move each column of characters 1 unit down to fill up the emptied Cells. Further, we don't end up splitting any Blocks since at any time a Block becomes "dead" i.e. 4 Cells get empty on the Board and hence 4 columns move down by 1 unit (this would also gets updated on the display). The generation of such Blocks can easily be confined to advanced levels since in the Level subclasses, all we are doing is setting the level, incrementing / decrementing levels, and getting the character for the next block type, so this doesn't affect the health of an alive block in any way. At advanced levels, whenever a new block would be generated, we would assign it a health 10, and when blocks are added to the board we decrease the health of each alive block by 1, and once the health of an alive block reaches 0, we remove it from the board and hence from the display and subsequently from the vector of Block *

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

Answer: We have implemented an abstract superclass for Level. Each individual level has a concrete derived class which inherits from this superclass. Using this design pattern, we can easily add additional levels into the system, with minimum recompilation. For each new level added, we can implement a new derived class which inherits from the abstract Level superclass. For other modules which depend on the various levels, adding new levels won't be a problem since the fields of various level classes in those modules would be updated accordingly as per the specified level (e.g. level 5, level 6 etc.) whenever a new level is introduced. Also, various features which are present in one level and aren't there in other levels can be implemented using this design pattern. Further, the functionality of individual levels wont get affected.

Question: 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 new 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.

Answer: Any new command that is added to the system must also be added to the interpreter and corresponding functionality must be added to the game class, without knowing the actual command being added, it is difficult to tell how the game class would change, however for the Interpreter class, it would just amount to adding another private field for that command. Since the interpreter class has private string variables for each of the commands (e.g. string left = "left" initially before renaming) and when a command is passed into it, it checks if the command is valid and calls the corresponding method in the game object. If the left command is renamed to "l" then we would just have to update the left variable in the interpreter object to be equal to "l" instead of "left". For a sequence of commands we can add some more private string variables, like string specialMove1 = "ldwrdp", if the user enters this command, then the Interpreter will first detect it and then call the corresponding function in the game object (in this case it will call left, down, right, drop).

7.0 Extra Credit Features

We have implemented the following features for bonus marks:

- Made an additional level, Level 5 with 2 new types of blocks – “C block” and “N block”, this was done by creating an additional subclass for level 5 which inherited from the Level superclass.
- Made 2 additional types of blocks – “C block” and “N block”, and included their rotations, again this was done by simply creating 2 additional classes for these block types which inherit from the abstract Block superclass
- User can rename the predefined commands for playing the game
- We have provided the functionality of reading input from a file and turning the randomness off for all levels instead of just level 3 and 4.

8.0 Answers to Final Questions

Question: 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?

Answer: We learnt a lot of things about software development while completing this project. We familiarised ourselves with version control (we used BitBucket for this project) and learnt it to use it properly. We also understood the importance of effective communication while doing this project. We would often visit each other’s house and work together. Before we started coding particular modules / classes, we would sit together and discuss the logic of how we would implement specific features and guidelines. Doing this project together, made us realise the importance of teamwork and collaboration. We made a plan and we struck to it and ensured that we never missed any of our set deadlines. By finishing our work on time, we learnt the importance of managing our time effectively and efficiently without stressing too much. Overall, successfully completing this project was an enriching experience for both of us.

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

Answer: Initially we thought that scoring would be extremely easy to implement since we only needed to keep track of the number of rows cleared at a particular level and doing managing the scoring for the blocks would be simple too. However, when we started to implement the scoring for our game, we released it wasn't that easy. First, since we needed to know which level a block was generated on, we had to include this field in our block classes. Second, we implemented a vector which kept track of all the blocks on the board and as soon as a block was removed the screen, we would remove it from the vector and update the score. The problem was we were leaking a lot of memory and took us quite some time to figure out to fix it, because of this reason we made some last minute changes our UML too. This is one thing, we wished we could have done better. Thinking clearly about how the scoring works could have saved us quite some time.

9.0 Conclusion

In conclusion, we learnt a lot of new stuff about Object Oriented Programming, and got an opportunity to practice it while implementing this project. We also learnt that teamwork is a key ingredient to develop large software programs successfully. We tried our best to make this program as modular and flexible as possible while adhering to the designs of OOP.