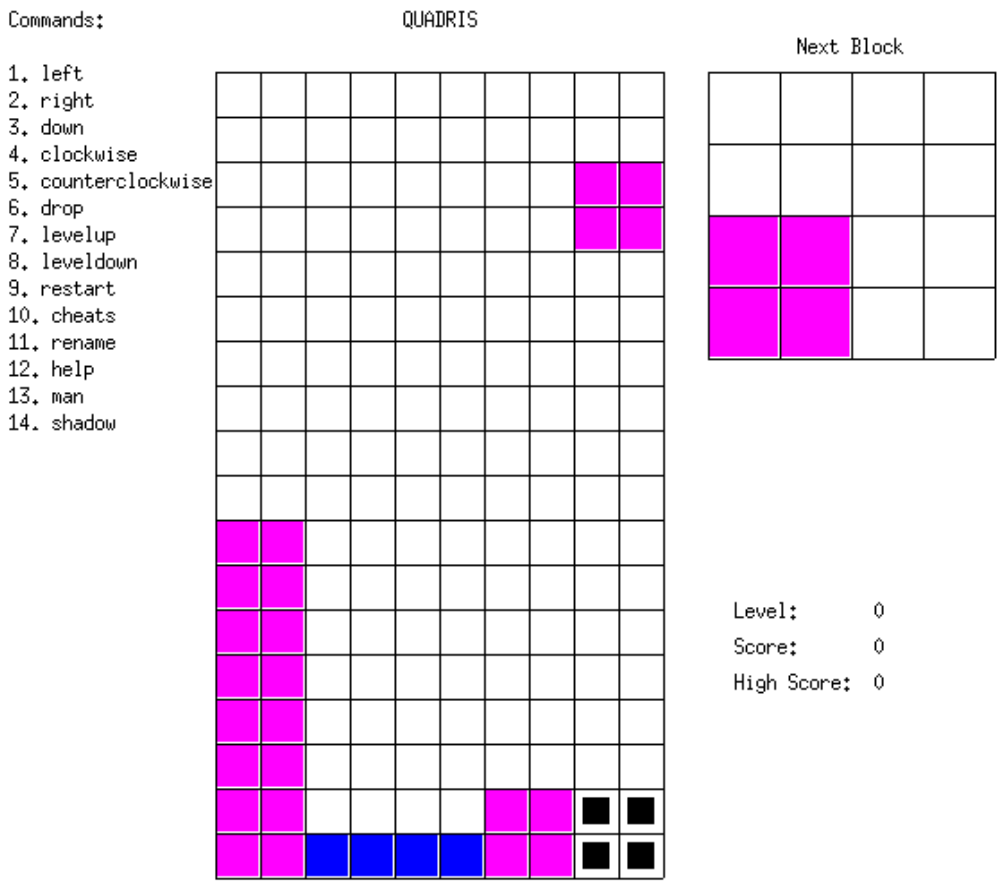
Quadris Design Document

CS246 – Yuan Gao (y65gao) – 20429092

4/4/2013



1 ABSTRACT

This document summarizes the plan of implementation for the game of Quadris, and breaks down the process. The sole contributor to this project is Yuan Gao, with no other team member.

2 DIFFERENCES BETWEEN PLAN DOCUMENT

When creating the planned document, the program didn't seem very complicated. I believed that it would have been very similar to the game of Light's Out, except where we had to generate blocks and drop them down on the grid.

With that in mind, I didn't spend too much time on the UML design, and thought that only a few methods in addition to the groundwork already done with Lights would have been sufficient. This was definitely not the case. I soon realized that the scope of the project was much larger than anticipated, and that I would have had to rewrite the entire UML and design from scratch in order to make it work.

The core of the design still stands, the main classes: Board, Block, Interpreter, Cell are all still there. A key class that was added was the Trie, to help manage commands.

Further changes can be observed in the completely overhauled UML.

After looking at some other versions of Tetris around the net, I saw a few features that were pretty cool implement. The first was the ability to turn on a shadow mode, which toggles the drawing of where the block will be after it's put down, this is exclusively for the graphics version and not the text based version.

The token system originally planned was implemented, but then removed because it didn't feel like it was of any use to the user, and defeats the purpose of actually playing the game. Perhaps in a live version of the game tokens can be purchased with micro-transactions, but for this project, that wasn't necessary.

3 IMPLEMENTATION OVERVIEW

This section will go through each of the major classes and how the design is put together.

3.1 BLOCK

Blocks are basically the rough equivalent to tetrominos. They are represented by a 4x4 grid of Boolean values, either on or off.

Blocks are generated using the PRNG class, as supplied by the course notes. The pseudo number generator takes a seed, and subsequently the tetrominos are generated with a given probability using the RNG.

Based on the project specifications, the probabilities for level 1 (our base level) was as follows:

S and T Blocks – 8.33%

All other blocks – 16.66%

PRNG randomly generates a number between [0, 11], which means there's 12 numbers to choose from. Thus, each number corresponds to a certain block.

For movement handling, we rebuild the block every time it moves. First we check if the move is legal, and if it is, we proceed. If not, we move the block back to its "original" position, and don't do anything, and pretend the command never happened.

For rotation handling, we swap the x and y coordinates. If clockwise, (x, y) becomes (y, 3 – x), if counterclockwise, (x, y) becomes (3 – y, x).

3.2 CELLS

The game board is composed of 180 individual cells, with 10 columns and 18 rows. There are actually 15 rows of cells, with 3 rows reserved on top for rotation. For scoring purposes, each cell stored some information of the previous tetrominos.

If a cell contained a tetromino, it stores the block's ID, the level it was generated in, as well as the character type.

To expand, the block ID is a unique number between zero and the constant integer BLOCKS, the upper bound of the number of cells we can have. Right now it's set at 180, which logically follows from the boundaries (18 rows x 10 columns). The reason IDs are associated with each cell is that to check if a block was deleted, we simply check if that ID no longer exists on the board.

We use the level it was generated in to calculate the score when a particular cell was cleared. If the cell had an ID that was unique and is no longer present after the row was cleared, we factor this into the score using the scheme given to us.

Score = (current level + number of lines)^2

3.3 BOARD

The board ties up Block and Cell classes together, and most of the handlers for block movement, block logic (whether a move was legal), and most of the commands are performed by the Board class.

3.4 INTERPRETER

The Interpreter is the handler that recognizes user input commands, and then calls the respective methods in the Board class to interact with Blocks. To recognize commands, we store commands in TrieNodes, first seen in Assignment 3.

Using TrieNodes simplify two processes. Firstly, commands can be easily searched and matched. Secondly, we can easily implement auto-completion for commands (i.e. non-ambiguous entries automatically, if they are the prefix of a valid command, recognized).

To facilitate the process for the user, the following Interpreter behavior was implemented:

- If a valid command was entered, that is, a non-ambiguous prefix of a valid command, that command is executed
- If a valid command was entered, but was ambiguous (ex. d), which translates to either down or drop, a prompt tells the user the matching commands.
- If a non-valid command was entered, i.e. wasn't found in our TrieNode, we prompt and advise the user on using the 'help' command which accesses the manual of commands.

For multipliers, only numbers attached to the beginning of commands would be interpreted as multipliers. To handle this, the Interpreter class reads in a command, and will recognize if the prefix had a number. If that number was present, it executes the command the number of times that was requested.

For command renaming, using TrieNodes also helps with the process. When an old command is being renamed, we find the old command in the TrieNode, delete it, and put the new command in at that index. Due to the limitations of TrieNode (when it was written in Assignment 3 we were limited to lower case characters), the same condition applies here, commands can only be lower case. It would be possible to implement the ability to recognize all commands by taking all commands entered by the user and convert it to lower case but by the time I realized that it was too late.

4 FEATURES

4.1 CHEAT MODE

The cheat mode is a fairly simply implemented cheats toggle, which allows the user get access to some extra features that aren't available in standard quadris.

These cheats currently include:

- Clear [line] (clears any line, starting from the bottom)
- Up (moves a block up)
- Softreset (clears the board and saves the current and next available block)

Due to the nature of cheats, upon activating them, all future high score counts are eliminated, until cheats are turned off.

For the implementation of the cheat mode, we need to modify the list of commands. This is done by adding cheat commands to the list of 'Commands' – an array that contains all the available commands to the user that gets cycled through every time the user enters a command.

Also, note that the grid needs to be redrawn when cheats are activated and deactivated due to the current limitations of xWindow, which doesn't allow the direct removal of strings (wasn't implemented). This creates some lag in command execution time.

4.2 SHADOW GUIDANCE

Shadow guidance is a system that places a shadow of where the block will eventually land at the bottom. It is also toggleable, and is disabled by default. To graphically simulate a shadow, we use smaller black squares.

4.3 USER MANUAL

The user manual is accessed with the 'help' command, and it is coupled with the 'man' command to give users a better idea of what commands are available, as well as provide usage details.

5 PROJECT SPECIFICATION QUESTIONS

How could you design your system to make sure that only these seven kinds of blocks can be generated, and in particular, that non-standard configurations (where, for example, the four pieces are not adjacent) cannot be produced?

Instead of using an abstract block class as originally planned, we have an array of block configurations. To generate a block, we construct a currentBlock and a nextBlock, and every time a block drops, we load a new block. Because we defined the possible block configurations in our array, non-standard configurations aren't possible to generate.

Another thing to consider was the level 0 case, where it had to read a sequence of characters. To handle letters that weren't in our array of block configurations, we use a placeholder, in this case, a tilde (~), to notify the block generator to skip it if that placeholder was seen. This was not considered in the original planning document.

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?

This answer remains the same - there could be a global game counter that counts the number of blocks that has fallen, and each block is assigned a number as soon as it's generated. In this design, blocks are given IDs. The special blocks that disappear will have an additional Boolean that indicates if it can disappear or not. Every time a new block is created, the special blocks on the grid check if the number of the block that just got generated and the difference between them is larger than 10. If it is, then the older special block gets removed from the grid. This was not implemented.

Could the generation of such blocks be easily confined to more advanced levels?

It can be easily confined to more advanced levels. The method for the block being deleted will only be called if the Boolean of level being larger than [x] for example is true. This is the same answer as in the planning document. This is not implemented.

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

The only changes that need to be made with this design is in Block.cc, where we generate the next block type depending on the level. An additional else clause would be needed with each additional difficulty level which modifies the probability that each block can appear (assuming this is the only thing that changes when the difficulty goes up.) This is not too different from the original answer, except now it's handled by the Block class instead of another sub-class.

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.)

To add a new command, because the all commands are in the Trie, we simply insert the command and Trie handles the rest. The Command interpreter would also need to be modified, and then the Board would need to be modified because all methods that interact with Blocks are there (the non-trivial amount of code that needs to be added). Thus, we'll only need to modify the Interpreter class and the Board class. This was not implemented.

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)?

In the previous planning document it was stated that it wasn't too difficult to implement, and it was true to some degree. It wasn't hard to implement – with the usage of TrieNode, it became very easy to sort through commands, remove commands, and add commands. However, because of the way TrieNode was written, only lowercase characters could be stored. At the time of writing, I didn't realize that I could just convert all user input to lowercase and store them that way, so that all commands entered would be case insensitive, but this ultimately was not implemented. Thus, all commands had to be in lower-case in order to be renamed.

If you worked alone, what lessons did you learn about writing large programs?

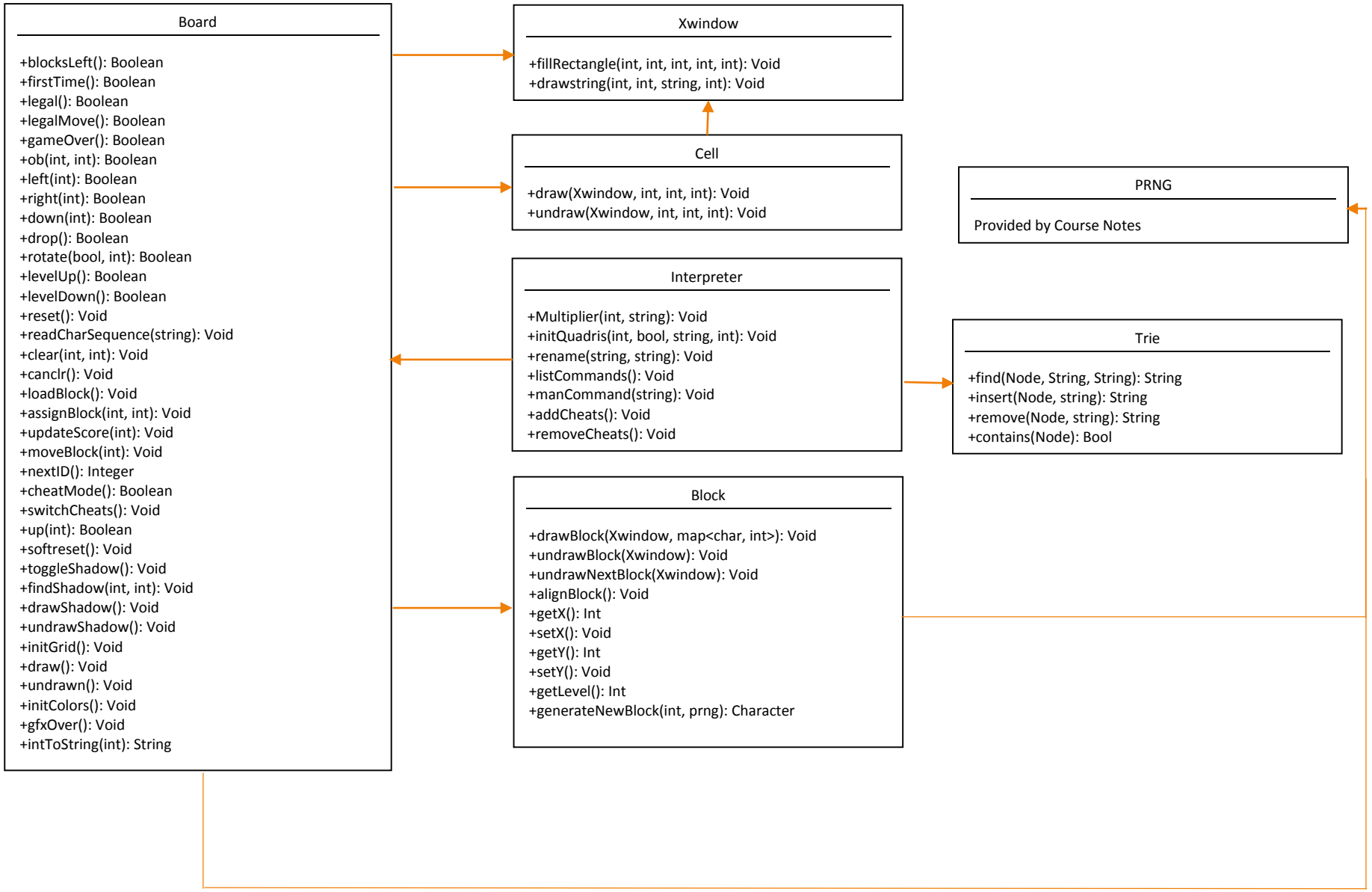
Working alone meant ultimately that all the code wouldn't have to be shared and heavily commented for another party to read through – that was definitely the major benefit. It also allowed for more scrapping of ideas and starting from scratch when something didn't work without having to consult anybody else beforehand.
However, it also introduced a lot of other difficulties. For example, due to the size of some of the classes, it soon became very hard to manage writing all of the code. Also, instead of having one person being able to continuously test the program, all test cases had to be written by myself, which meant a lot of time was spent on writing this project. The advice given on the project specifications definitely helped – building the program from the ground up, having only the text version then slowly implementing the graphics allowed for the project to be built in steps.

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

I definitely would have spent more time planning. The original planning process was very rushed, and thus not a lot of the code was considered, and as such, when it came to actually writing it, most of the plan was scrapped and a start from scratch was necessary. This is why the UML is so different.

Also, extra features and making the game look better would definitely been a plus.

6 UML


Board
+blocksLeft(): Boolean
+firstTime(): Boolean
+legal(): Boolean
+legalMove(): Boolean
+gameOver(): Boolean
+ob(int, int): Boolean
+left(int): Boolean
+right(int): Boolean
+down(int): Boolean
+drop(): Boolean
+rotate(bool, int): Boolean
+levelUp(): Boolean
+levelDown(): Boolean
+reset(): Void
+readCharSequence(string): Void
+clear(int, int): Void
+canclr(): Void
+loadBlock(): Void
+assignBlock(int, int): Void
+updateScore(int): Void
+moveBlock(int): Void
+nextID(): Integer
+cheatMode(): Boolean
+switchCheats(): Void
+up(int): Boolean
+softreset(): Void
+toggleShadow(): Void
+findShadow(int, int): Void
+drawShadow(): Void
+undrawShadow(): Void
+initGrid(): Void
+draw(): Void
+undrawn(): Void
+initColors(): Void
+gfxOver(): Void
+intToString(int): String
Xwindow
+fillRectangle(int, int, int, int, int): Void
+drawstring(int, int, string, int): Void
Cell
+draw(Xwindow, int, int, int): Void
+undraw(Xwindow, int, int, int): Void
PRNG
Provided by Course Notes
Interpreter
+Multiplier(int, string): Void
+initQuadris(int, bool, string, int): Void
+rename(string, string): Void
+listCommands(): Void
+manCommand(string): Void
+addCheats(): Void
+removeCheats(): Void
Trie
+find(Node, String, String): String
+insert(Node, string): String
+remove(Node, string): String
+contains(Node): Bool
Block
+drawBlock(Xwindow, map<char, int>): Void
+undrawBlock(Xwindow): Void
+undrawNextBlock(Xwindow): Void
+alignBlock(): Void
+getX(): Int
+setX(): Void
+getY(): Int
+setY(): Void
+getLevel(): Int
+generateNewBlock(int, prng): Character