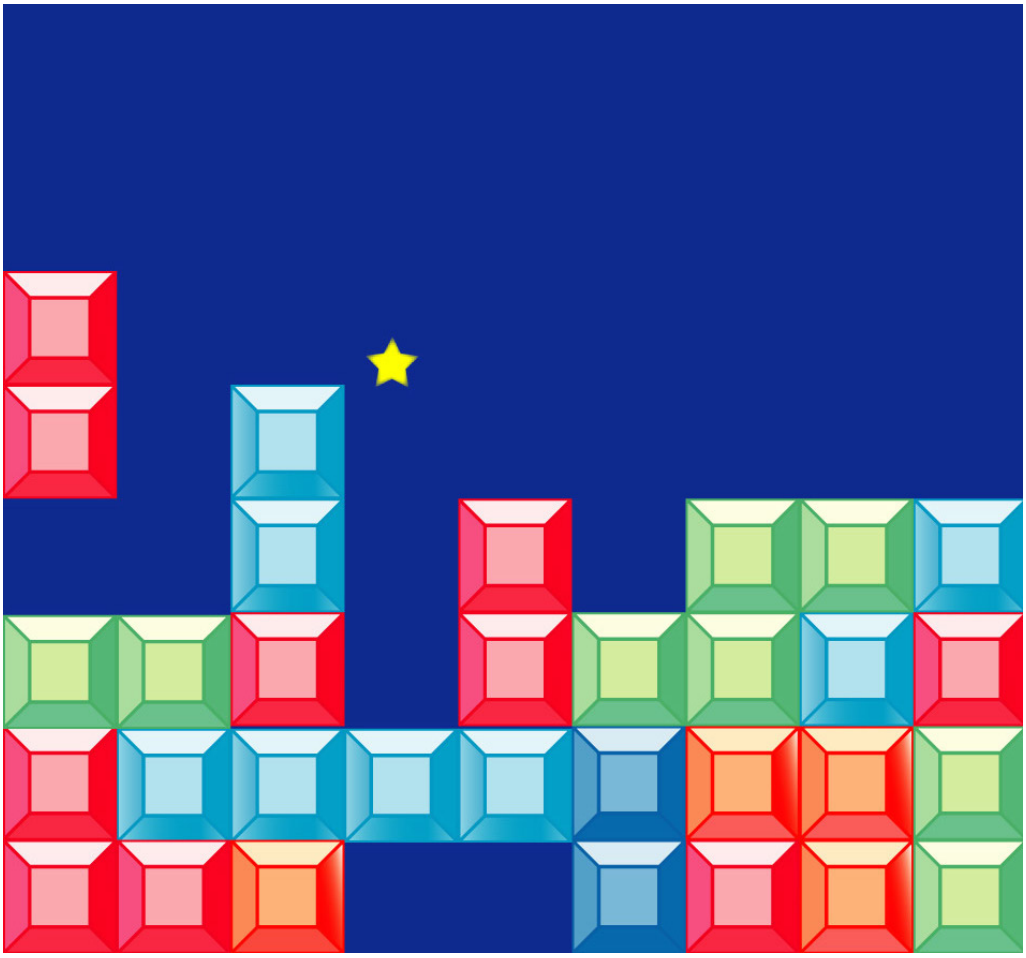

Final Project (Quadris)

Prepared for: Trenz Pruca, Title
by: Qian Qiao, Yunhao Xue
July 30, 2014
Spring 2014

Difference between Plan and Actual Design

-There are two major difference:

(1). On due date1, we planed to implement the Block class with 7 derived classes, (i.e. IBlock, JBlock, etc) by using Inheritance. But on the actual design, we realize that it's sufficient to use a private field called blockType to identify the type of the block which avoids more complex codes using pure virtual method and inheritance. So we decided to make a change, using one concrete Block class rather than an abstract Block class.

(2). On due date1, we did not have a class called Cell to store many useful information. But later, we recognized that the scoring rule is if a block is totally cleared, then we should get additional score, which means we need to know which cell belongs to which block, so only a 2D-char array is not enough.

High-level implementation specification

- Class Cell

The Cell class is the fundamental components of both Block and Board Class. To be precise, each block has 4 Cells and each board has 18 * 10 = 180 Cells.

The Cell class has the following functionalities:

Firstly, the cell class has a private field called cellType. This gives us chance to determine what character the cells hold. This is essential for the displaying of the board.

Secondly, each cell has fields called r and c, which are abbreviations of rows and columns. This records the relative position of the cell in the board (which has a 18*10 two-dimensional cell array).

For Example, if cell.r = 10, cell.c = 6. Then we can access that cell through board[10][6].

Lastly, one thing must to mention is an assignment operator must be overloaded, that is crucial for the removeRow() operation (which we will explain latter).

• Class Board

The Board Class is consists of a 2D-Cell array (18*10). It's used for the creation, destruction of the Block. Thus, we need a pointer to the current block and a pointer to the next block generated by the random number generator.

Additionally, the board must observe the individual motion of the current block. For example, if we want to drop the current block, we need to call Board::drop and Board::drop will call currentBlock->drop(). This is needed for the graphical display part. We undrawn the current block, and draw the current block at the new position.

There are also two vectors called vlevel and vcleared for each generate block recording its genereting level and whether the whole block has been cleared, which is used for scoring.

• Class Block

There are only seven kinds of block. We can thus create a Block class. This class is responsible for the individual motion of current block. Namely, left, right, down and drop. In order to reflect the outcome of each individual motion, we need the Board Class to become the observer of the Block Class. Thus for each Block object we need a pointer to the Board.

To achieve each motion, we need to first determine whether if these operations can be performed.

For example, when one block hit the leftmost edge of the board, we can't go left anymore. And when one cell of the block has an adjacent, nonempty cell (which is not in the current block), we can't move the block into that direction.

- Control (Main Function)

In main function, we read the input(command) from the keyboard. And we accept the following command line argument: -text, -startlevel –scriptfile –seed. We specify this in the Constructor of the Board Class.

- Miscellaneous1:
- Probabilistic-based Random Number generator

According to the question, there should be 4 levels implemented. And for the level 2 and 3, we give probability to each type of Block. We can use random generator rand() to achieve this.

- Miscellaneous2: Grade Calculating

We will only calculate the score every time after a block dropped. If a block is dropped, we will check how many rows should be cleared, and for each block in vcleared false, check whether it is cleared this time. Then we can calculate the total score.


Board
-seeded : Boolean
-HighScore : Integer
-theboard : Cell **
-gameStart : Boolean
-level : Integer
-score : integer
-seed : integer
-numblock : integer
-ff : ifstream
-vlevel : vector<int>
-vcleared : vector<bool>
-currentBlock : Block *
-nextBlock : Block *
-window : Xwindow *
+Board(OnlyText : Boolean = false, seed : integer = 10, file : string = "sequence.txt", startlevel : integer = 0)
+genNextBlock() : Block *
+isFullRow(n : integer) : Boolean
+clearRow(n : integer) : void
+isWon() : Boolean
+isLost() : Boolean
+returnNextBlock() : Block *
+SetLevel(n : integer) : void
+getlevel() : void
+clearCell(r : integer, c : integer) : void
+setCell(r : integer, c : integer, type : char, numblock : integer) : void
+getState(r : integer, c : integer) : Boolean
+scoreplus(n : integer) : void
+left() : void
+right() : void
+down() : void
+clockwise() : void
+counterclockwise() : void
+drop() : void
+levelup() : void
+leveldown() : void
+restart() : void
+getType(r : integer, c : integer) : char
+iscleared(i : integer) : Boolean
+getvcleared() : vector<bool>
+getvlevel(i : integer) : integer
+setvcleared(i : integer, result : Boolean) : void
Block
-rotationType : integer
-cellArray : Cell*
-type : char
-number : integer
-bd : Board*
+getr1() : integer
+getr2() : integer
+getr3() : integer
+getr4() : integer
+getc1() : integer
+getc2() : integer
+getc3() : integer
+getc4() : integer
+getcell(i : integer) : Cell
+Block(type : char, number : integer, myboard : Board *)
+getBlockType() : char
+canleft() : Boolean
+canRight() : Boolean
+canDown() : Boolean
+left() : void
+right() : void
+getRotationType() : integer
+Clockwise() : void
+counterclockwise() : void
+down() : void
+drop() : void
+draw(w : Xwindow *) : void
+undraw(w : Xwindow *) : void
Cell
-cellType : char
-r : integer
-c : integer
-blocknumber : integer
-level : integer
+Cell()
+Cell(r : integer, c : integer, type : char, blocknum : interger)
+setX(n : integer) : void
+setY(n : integer) : void
+setType(type : char) : void
+getX() : integer
+getY() : integer
+incrementX() : void
+incrementY() : void
+decrementY() : void
+getState() : Boolean
+getType() : char
+getBlockNumber() : integer
+draw(w : Xwindow *) : void
+undraw(w : Xwindow *) : void
+setCell(type : char, numblock : integer) : void
+operator=(old : Cell &) : Cell &
Xwindow
-d : Display*
-w : Window
-s : integer
-gc : GC
-colours : long*
-width : integer
-height : integer
+Xwindow(width = 500 : integer, height = 500 : integer)
+fillRectangle(x : integer, y : integer, width : integer, height : integer, colour = Black : integer)
+drawString(x : integer, y : integer, msg : string, colour = Black : integer)

Assignment Questions

- Q1:

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?

There is private field called type in Block. Then define a constant char array {I, J, Z, ...}, which means we can only construct these seven kinds of Blocks. If we want to add new Block then change the char array.

- Q2:

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?

For each cell in the board, there is a field in cell recorded it belongs to ith block. And the field in Board recorded how many blocks has been dropped. Every time we drop a block, we will check whether the (numBlock minus 10)th block has been cleared.

- Q3:

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

Using visit design pattern to add new features without changing the original codes. Or use the Decorator pattern to dynamically add the additional feature to the program. In this case, when we have a new level. We just add an derived class of the abstract decorator class.

- Q4:

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.) 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)?

Define each command name as constant string before implementation. Then if we need to change the name of command, we could just change the string defined. For new command name, use decorator pattern or visit pattern to implement it.

Lessons Learnt

- 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?

By doing this project, both team members haves the opportunity to write a large problem that employed object-oriented development techniques. And during these two weeks, we realize that good documented code is crucial to the project itself. Writing code in poor manner can lead to several problems when doing debugging & rewriting. And the collaboration between two people is key to the whole process.

From that project I realize how important the structure of the project is, in the first due date, we decide not to implement the Cell class, but in the second week we realize that it's far more convenient to implement Cell. And it takes a lot of time to make a change. So the structure of the problem is important and carefully thinking of the structure at high-level can reduce a lot of time debugging & rewriting.

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

If I had another two weeks, maybe I will using Inheritance to implement the Block Class. Additionally, I would consider to implement some other feature like:

1. give a hint on where the block should drop?

2. using <time> library to let the block drop gradually, like the game Tetris.

3. make a more beautiful user interface. The one we made is simple but not elegant.