CS246—Assignment 5, Group Project (Fall 2014)

Due Date 1: Friday, November 21, 11:55pm
Due Date 2: Sunday, November 30, 11:55pm

This project is intended to be doable by two people in two weeks. Because the breadth of students' abilities in this course is quite wide, exactly what constitutes two weeks' worth of work for two students is difficult to nail down. Some groups will finish quickly; others won't finish at all. We will attempt to grade this assignment in a way that addresses both ends of the spectrum. You should be able to pass the assignment with only a modest portion of your program working. Then, if you want a higher mark, it will take more than a proportionally higher effort to achieve, the higher you go. A perfect score will require a complete implementation. If you finish the entire program early, you can add extra features for a few extra marks.

Above all, MAKE SURE YOUR SUBMITTED PROGRAM RUNS. The markers do not have time to examine source code and give partial correctness marks for non-working programs. So, no matter what, even if your program doesn't work properly, make sure it at least does something.

Note: If you have not already done so for Assignment 4, make sure you are able to use graphical applications from your Unix session. If you are using Linux you should be fine (if making an ssh connection to a campus machine, be sure to pass the `-Y` option). If you are using Windows and putty, you should download and run an X server such as XMing, and be sure that putty is configured to forward X connections. Alert course staff immediately if you are unable to set up your X connection (e.g. if you can't run `xeyes`).

Also (if working on your own machine) make sure you have the necessary libraries to compile graphics. Try executing the following:

```
g++ -L/usr/X11R6/lib window.cc graphicsdemo.cc -lX11 -o graphicsdemo
```

The Game of SquareSwapper5000

In this project, you and a partner will work together to produce the video game SquareSwapper5000, which is a match-three style puzzle game[1] that draws elements from popular games, such as Candy Crush Saga and the Bejewelled series.

A game of SquareSwapper5000 consists of a board, 10 columns wide and 10 rows high. When a game begins, the board cells are covered with randomly generated squares. The game progresses by the user choosing to swap two neighbouring squares to cause a **match**. A **match** is defined to be three or more squares of the same colour aligned in some fashion (See Section Squares). Any attempted swap that does not result in a match is disallowed. Points are awarded each time a

[1] `http://en.wikipedia.org/wiki/Tile-matching_video_game`

match is made. Levels in the game define their own goal. Advanced levels may pose additional challenges such as a bigger/irregular board, or making matches at certain locations of the board (similar to the concept of clearing jelly on a Candy Crush board).

The major components of the system are described in the following sections:

Board

A basic SquareSwapper5000 board is 10 columns and 10 rows. Each cell of the board holds one square.

Squares

BasicSquare

This is your standard square with no special abilities. BasicSquare can be white, red, green or blue. A match comprising entirely of BasicSquare results in those squares disappearing from the board. When a square disappears, it causes a hole to appear in it's stead on the board. A hole is immediately plugged by the square to the north falling down to cover the hole thereby leaving a hole to the north which is then plugged by the square north of it. If there is no square north of a hole, a new randomly generated square is created to plug the hole. Square movement or the generation of new squares might automatically trigger further matches.

LateralSquare

A LateralSquare is generated any time four squares are matched horizontally. The newly generated LateralSquare has the same colour as the matched squares and takes the place of one of the middle squares that were matched. Other squares disappear leaving behind holes that are plugged in the same manner as discussed above.

If a LateralSquare is involved in a subsequent match this causes the entire horizontal line of squares on the board to disappear (as if they were all part of the match even if they are of different colours). The holes caused by the disappearance of the entire row are plugged in the standard fashion.

UprightSquare

An UprightSquare is generated anytime four squares are matched vertically. The newly generated UprightSquare has the same colour as the colour of the matched squares and takes the place of one of the middle squares that were matched. Other squares involved in the match disappear to leave behind holes which are plugged as discussed above. (Note: the newly generated UprightSquare might itself fall south one or two places to plug holes caused by other squares part of the match that have now disappeared.)

If a UprightSquare is ever involved in a match this causes the entire vertical line of squares on the board to disappear (as if they were all part of the match even if they are of different colours). All the squares in the vertical line disappear leaving a very deep hole that is plugged by creating and dropping squares all the way through to the bottom of the board.

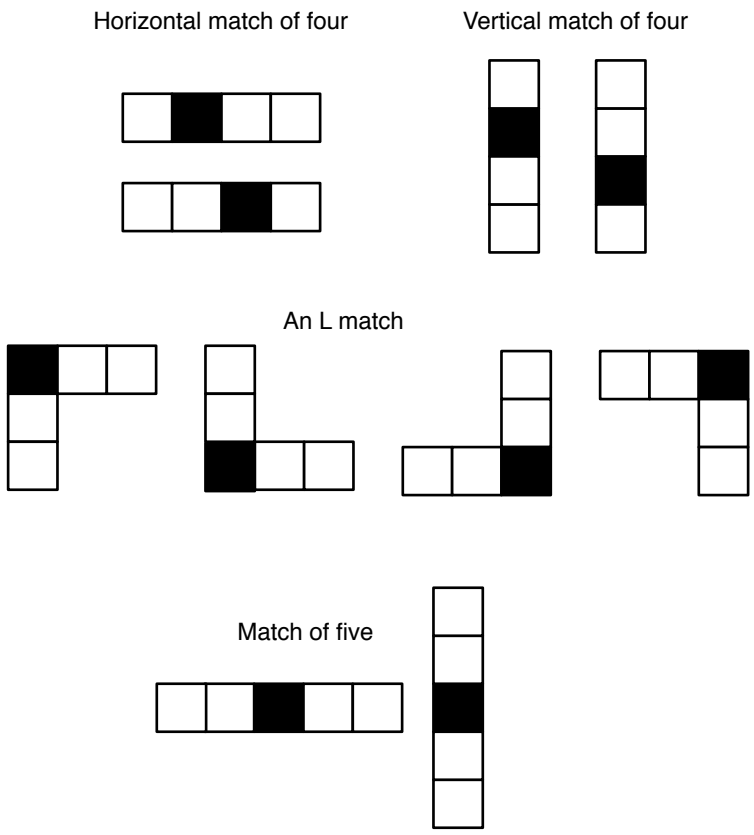

Figure 1: Different types of matches

UnstableSquare

An UnstableSquare is generated anytime a match in the form of an L is made. An L is defined as a match comprising of three squares matched horizontally and three squares matched vertically with a shared square that must be one end of the horizontal and vertical match (See Figure 1). All squares are of the same colour. Note that the letter L can be in any of its rotated forms. The newly generated UnstableSquare has the same colour as the matched squares and takes the spot of the square where the horizontal and vertical match intersects. All other squares involved in the match disappear leaving behind holes to be plugged.

If a UnstableSquare is involved in a subsequent match it causes a number of its neighbours to explode leaving a big hole. If the match is a three square match then a 3x3 hole is created centred at the UnstableSquare. A four square match causes a 5x5 hole to be created, centred at the UnstableSquare. The hole created is plugged in an analogous manner to that previously described.

PsychedelicSquare

A PsychedelicSquare is generated whenever there is a horizontal or vertical match of five squares of the same colour. The PsychedelicSquare has the same colour as the matched squares and takes the spot of the middle square in the match. All other squares disappear leaving behind holes that are plugged in the manner discussed above.

If a PsychedelicSquare is involved in a subsequent match it causes all squares from the board

of the same colour as the PsychedelicSquare to disappear leaving behind holes. These holes must be plugged in the manner discussed above.

The following diagram illustrates the different types of matches that are possible. In the diagram, we have used a black squares to show where a newly generated square would be placed. The white squares will cause holes to appear:

Question: How could you design your system to make sure that only these kinds of squares can be generated?

Question: How could you design your system to allow quick addition of new squares with different abilities without major changes to the existing code.

Display

You need to provide both a text-based display and a graphical display of your game board. A sample text display follows:

```
Level:      1
Score:      0
Moves Remaining: 7
Hi Score: 100
----------

__0 __1 __2 __0 __1 __3 __0 __0 __1 __2
_h1 __2 __1 __0 __3 __1 _b2 __2 __3 __3
__1 __0 __2 __1 __3 __2 _v0 __1 __1 __2
__2 _v1 __0 __1 __2 __3 __2 __1 _b2 __2
__1 __2 __1 __3 __3 _b1 __0 __2 __1 __3
__1 __3 _p0 __0 __3 __1 __2 __2 __3 _v3
__0 __1 __2 __0 __1 __3 __0 _h0 __1 __2
__2 _h0 __0 __1 __2 _h3 __2 __1 __1 __2
__3 __1 __0 _h1 __0 __0 __1 __2 __2 _v3
__0 __0 _p1 __2 __3 __1 __0 __2 _v3 __3
```

Each square is represented by 3 characters. The leftmost character is used for any advanced features you implement (see ahead for one such advanced feature).

The middle character is used to represent the type of square. The following encoding shows the characters to use for each type of square:

BasicSquare	underscore (_)
LateralSquare	h
UprightSquare	v
UnstableSquare	b
PsychedelicSquare	p

The rightmost character is used to represent the colour of the square. The following encoding is used

```
White: 0
Red:   1
Green: 2
Blue:  3
```

Notice we have not used the black colour. We will use it later for an advanced level feature.

Your graphical display should be setup to show the entire board. BasicSquare are straightforward, draw squares of that colour. You should come up with a visually pleasing way to display special squares. One basic way could be to write a character inside a square to indicate what type of square it is (e.g. P inside a yellow square would represent a PsychedelicSquare). Perhaps a slightly more visually pleasing graphic would be to use some combination of colours to indicate special squares. Do your best to make it visually pleasing[2].

Levels

Part of your system will encapsulate the decision-making process regarding which square is selected when creating the initial configuration of the board and when generating new squares. A board should start with no matches. The level of difficulty of the game depends on the policy for selecting the next square. You are to support at least the following difficulty levels:

- Level 0: (**Make sure you get at least this working)** The initial board configuration for this level is read from a file `sequence.txt`. During the progress of the level, if new squares need to be generated, the colours for the new squares are also read from the sequence file. A level 0 sequence file must always have 5 LateralSquare, 5 UprightSquare, 3 UnstableSquare and 2 PsychedelicSquare. (A sample is provided)

  The level is completed when the score is at least 200 more than the score at the start of the level. At this point, the player automatically levels up and a new board satisfying the requirements of level 1 is created.

- Level 1: The square selector will randomly choose squares with probabilities skewed such that White and Red squares are selected with probability $\frac{1}{3}$ each, and the Green and Blue squares are selected with probability $\frac{1}{6}$ each. In addition, every 5^{th} square is a special square. All special squares have equal probability of being generated. Level 1 is completed when the score is at least 300 more than the score at the start of the level. At this time the player automatically levels up and a new board satisfying the requirements of level 2 is created.

- Level 2: All squares are selected with equal probability. No special squares are generated (from now on special squares only appear as discussed in the Squares section). This level also introduces a new feature; certain cells on the board are marked as locked. For the text-based version of the game, indicate a specially marked cell by displaying the character l (lower case L) in the first of the three characters representing a cell. When a match involves a square sitting in a locked cell, the cell is unlocked. For the graphical version, you may use the reserved black colour to draw a border around locked cells (or you could come up with something more visually pleasing).

  The board configurator randomly locks 20% of the cells. To complete the level (a) the score must be at least 500 more than the score at the start of the level and (b) there should be no locked cells remaining on the board.

[2]Feel free to do some research and add additional colours to the Window class

For example, the following board represents a valid initial configuration at this level (note the absence of any special squares and the presence of 20 locked cells). If the player swaps the square

```
__0 __1 __2 __0 __1 __3 l_0 __0 __1 __2
__1 __2 __1 __0 l_3 __1 l_2 __2 __3 __3
__1 __0 __2 __1 __3 __2 __0 l_1 __1 __2
l_2 __1 l_0 __1 l_2 __3 l_2 l_1 __2 __2
__1 __2 __1 __3 __3 __1 __0 l_2 __1 l_3
__1 __3 __0 l_0 __3 __1 __2 __2 __3 __3
__0 __1 __2 __0 l_1 __3 __0 __0 __1 __2
__2 __0 l_0 __1 __2 l_3 __2 __1 __1 l_2
__3 l_1 __0 __0 l_1 l_0 __1 __2 __2 __3
__0 __0 __1 __2 l_3 __1 __0 __2 __3 __3
```

at (8,3) (a 1 square) with the square at (7,3) (a 0 square) the following *intermediate* board configuration will result: The move caused a L match which led to the creation of a PsychedelicSquare.

```
__0 __1 __2 __0 __1 __3 l_0 __0 __1 __2
__1 __2 __1 __0 l_3 __1 l_2 __2 __3 __3
__1 __0 __2 __1 __3 __2 __0 l_1 __1 __2
l_2 __1 l_0 __1 l_2 __3 l_2 l_1 __2 __2
__1 __2 __1 __3 __3 __1 __0 l_2 __1 l_3
__1 __3 __0 ___ __3 __1 __2 __2 __3 __3
__0 __1 __2 ___ l_1 __3 __0 __0 __1 __2
__2 ___ ___ _p0 __2 l_3 __2 __1 __1 l_2
__3 l_1 __0 __1 l_1 l_0 __1 __2 __2 __3
__0 __0 __1 __2 l_3 __1 __0 __2 __3 __3
```

Prior to the swap two cells (those at location (7,2) and (5,3)) had been locked. Since they have now been part of a match, the cells are now unlocked.

The above configuration is only an intermediate configuration shown to illustrate the effect of the match. In particular, the L match has caused holes to appear which are plugged by the squares to the north, resulting in the *intermediate* configuration: New squares are then generated to be

```
__0 ___ ___ ___ __1 __3 l_0 __0 __1 __2
__1 __1 __2 ___ l_3 __1 l_2 __2 __3 __3
__1 __2 __1 __0 __3 __2 __0 l_1 __1 __2
l_2 __0 l_2 __0 l_2 __3 l_2 l_1 __2 __2
__1 __1 __0 __1 __3 __1 __0 l_2 __1 l_3
__1 __2 __1 __1 __3 __1 __2 __2 __3 __3
__0 __3 __0 __3 l_1 __3 __0 __0 __1 __2
__2 __1 __2 _p0 __2 l_3 __2 __1 __1 l_2
__3 l_1 __0 __1 l_1 l_0 __1 __2 __2 __3
__0 __0 __1 __2 l_3 __1 __0 __2 __3 __3
```

placed at the holes (0.1). (0,2), (0,3) (1,3)

You are free to create more complex/fun levels. Read below for some suggestions.

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

Note on level changes: Notice that levels do not change when reaching a fixed score. This is to facilitate the ability to move up or down levels. Therefore, it is conceivable that at a score of lets say 535 the player decides to switch down to level 0. To complete level 0 they then have to achieve a score which is 200 greater than their score at the start of the level i.e. 535 + 200 = 735. For more on level shift commands read ahead.

Scoring

Players earn points by matching squares. Each move is scored as follows (note that a valid move always leads to a match):

- For a move that matches 3 squares, the player gets 3 points.
- For a match of 4 squares, the player gets 2 x 4 points.
- For a match of 5 squares, the player gets 3 x 5 points.
- For a match of more than 5 squares, the player gets 4 x number of matched squares e.g. a match that involves a LateralSquare is viewed as a match of 10 squares and would earn the player 4 x10 = 40 points.
- If a match causes a chain reaction, each chain reaction is worth 2^N times the score for the match where N is the N^{th} chain reaction for that move.

It is possible for a single move to cause two matches. Consider the following board. If the move is to swap (9,9) with (8,9) then this leads to two matches, a match of three squares involving the locations (9,7,), (9,8) and (9,9) and an L match centred at (8,9). This is not considered a chain reaction. In particular the score obtained by this move is 3 + 15 = 18 points.

```
__0 __1 __2 __0 __1 __3 __0 __0 __1 __2
_h1 __2 __1 __0 __3 __1 _b2 __2 __3 __3
__1 __0 __2 __1 __3 __2 _v0 __1 __1 __2
__2 _v1 __0 __1 __2 __3 __2 __1 _b2 __2
__1 __2 __1 __3 __3 _b1 __0 __2 __1 __3
__1 __3 _p0 __0 __3 __1 __2 __2 __3 __3
__0 __1 __2 __0 __1 __3 __0 _h0 __1 __2
__2 _h0 __0 __1 __2 _h3 __2 __1 __1 __2
__3 __1 __0 _h1 __0 __0 __1 __2 __2 __3
__0 __0 _p1 __2 __3 __1 __0 __3 __3 __2
```

Command Interpreter

You interact with the system by issuing text-based commands. The following commands are to be supported:

- `swap x y z` swaps the square at the `(x,y)` co-ordinate with the square in the `z` direction (0 for north, 1 for south, 2 for west and 3 for east) e.g. `swap 4 5 3` results in an attempt to swap the square at (4,5) to the square to the east (right) i.e. the square at cell location (4,6).
- `hint` the game returns a valid move (x,y, z as above) that would lead to a match.

- `scramble` Available only if no moves are possible, this command reshuffles the squares on the board (no new cells are created)
- `levelup` Increases the difficulty level of the game by one. You may clear the board and create a new one suitable for that level. If there is no higher level, this command has no effect.
- `leveldown` The same as above, but this time decreasing the difficulty level of the game by one.
- `restart` Clears the board and starts a new game at the same level

End-of-file (EOF) terminates the game.

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? (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 swap s`)?

The board should be redrawn, both in text and graphically, each time a command is issued. For the graphic display, redraw as little of the screen as is necessary to make the needed changes.

Command-line Interface

Your program should support the following options on the command line:

- `-text` runs the program in text-only mode. No graphics are displayed. The default behaviour (no `-text`) is to show **both** text **and** graphics.
- `-seed xxx` sets the random number generator's seed to `xxx`. If you don't set the seed, you always get the same random sequence every time you run the program. It's good for testing, but not much fun.
- `-scriptfile xxx` uses `xxx` for this level's initial board configuration. You can assume that the specified file will contain a valid board configuration for that level in the format discussed above. Files **may** contain a last line indicating a pre-determined order in which coloured BasicSquare should be generated. If a specific sequence is provided this overrides any random generation of squares for that level. If no such sequence is provided, the default behaviour for that level should take place. This command line argument will be used to ease testing. You will lose many marks if this option does not work as required.
- `-startlevel n` starts the game in level `n`. The game starts in level 0 if this option is not supplied.

Note: multiple command line options can be specified at the same time e.g. -scriptfile checkLocked-Cells -startlevel 2, should start the game at level 2 and read in the initial configuration of the board from the file checkLockedCells.

The above command-line interface will prove useful in testing your program during the live demo. We will have ready a number of initial configuration files (for different levels) which we will ask your program to read in, so that we can quickly test whether you have correctly implemented the required features without having to make large number of moves.

Grading

Your project will be graded as follows:

Correctness and Completeness	60%	Does it work? Does it implement all of the requirements?
Documentation	20%	Plan of attack; final design document.
Design	20%	UML; good use of separate compilation, good object-oriented practice; is it well-stuctured, or is it one giant function?

Even if your program doesn't work at all, you can still earn a lot of marks through good documentation and design, (in the latter case, there needs to be enough code present to make a reasonable assessment).

If Things Go Wrong

If you run into trouble and find yourself unable to complete the entire assignment, please do your best to submit something that works, even if it doesn't solve the entire assignment. For example:

- can't do actions for matches involving special squares
- program only produces text output; no graphics
- only one level of difficulty implemented
- does not recognize the full command syntax
- score not calculated correctly

You will get a higher mark for fully implementing some of the requirements than for a program that attempts all of the requirements, but doesn't run.

A well-documented, well-designed program that has all of the deficiencies listed above, but still runs, can still potentially earn a passing grade.

Plan for the Worst

Even the best programmers have bad days, and the simplest pointer error can take hours to track down. So be sure to have a plan of attack in place that maximizes your chances of always at least having a working program to submit. Prioritize your goals to maximize what you can demonstrate at any given time. We suggest: save the graphics for last, and first do the game in pure text. One of the first things you should probably do is write a routine to draw the game board (probably a `Board` class with an overloaded friend `operator<<`). It can start out blank, and become more sophisticated as you add features. You should also do the command interpreter early, so that you can interact with your program. You can then add commands one-by-one, and separately work on supporting the full command syntax. Take the time to work on a test suite at the same time as you are writing your project. Although we are not asking you to submit a test suite, having one on hand will speed up the process of verifying your implementation.

You will be asked to submit a plan, with projected completion dates and divided responsibilities, as part of your documentation for Due Date 1.

If Things Go Well

If you complete the entire project, you can earn up to 10% extra credit for implementing extra features. These should be outlined in your design document, and markers will judge the value of your extra features.

The following are rough ideas for features you could implement. Do NOT attempt these until you have the base game running:

- The ability to do T matches. Similar to L matches. Could produce a different special square or an UnstableSquare .
- The required levels all use a 10x10. It would be interesting to have irregular grids
- Some levels in the game could require the player to reach a certain score in a limited number of moves (e.g. get to 500 in 50 moves)
- More special squares with different abilities. One possibility is the presence of "Rooted Squares" with the restriction that a level cannot be completed until all Rooted Squares have disappeared.
- Use a curses[3] library to make this like an actual game with WASD controls. We offer no support for this though.
- More complex levels

If you do implement advanced features, make sure that you have a way to turn off these features for the initial part of the demo.

Submission Instructions and Due Dates

See **project_guidelines.pdf** for instructions about what should be included in your plan of attack and final design document.

Due Date 1 (November 21, 2014): Due on due date 1 is your plan of attack and initial UML diagram for your implementation of SquareSwapper5000.

Due Date 2 (November 30, 2014): Due on due date 2 is your actual implementation of SquareSwapper5000. All .h and .cc files should be included in ss5k.zip. In addition, you must submit your design document and final UML diagram.

A Note on Random Generation

To complete this project, you will require the random generation (or rather, pseudo-random) of numbers. You have two options available to you. In `<cstdlib>`, there are commands **rand** and **srand**, which generate a random number and seed the random generator respectively (typically, seeded with **time()** from `<ctime>`). Alternatively, you can use the **prng class** from `prng.h`, which provides an encapsulated random number generating algorithm. Either is fine for the project; it is not required to use one over the other.

[3] `http://en.wikipedia.org/wiki/Ncurses`