Summary of Design Choices

For Assignment 3, Spring 2015 Ravi Woods – rbw@ic.ac.uk

Data Structure Choice:

When choosing the data structure for the 4x4 grid as a whole, there were two obvious options:

- *A vector*
  This would give the possibility of consolidating algorithms that work in four directions (such as pushing and merging the grid in a particular direction) into a single function, using different indexing for each.
  However, since the vector index is defined as a single number, this may cause confusion, as the grid has two dimensions.
- *A 2D Array*
  This would be the easiest to visualise, and make the code easier to read.
  However, as column and row indexes cannot be switched, any algorithm which works in four directions could only be consolidated into two directions – one handling left & right, the other handling up & down.

From this, I decided on more understandable code, so I chose a 2D array.
However, when I needed a structure to store the indexes of empty elements, I then needed to store two elements (row and column indexes). So, here I made a "point" structure, and stored the empty element indexes in a vector of "points".

"Push and Merge" function:

My first approach for pushing and merging elements, according to the commanded direction, was two separate functions – one to push and one to merge. However, there are some cases where this is problematic, such as if we pressing "a" (i.e: Left) on [4,4,4,0] , the output should be [8,4,0,0], rather than [4,8,0,0].
These and other problematic cases led me to an implentation with two indexes (Index A – the Back Index – and Index B – the Front Index), with each index pointing to an element (Element A and Element B) in the 4x4 grid. Depending on the values at each index, either the index or element could be changed.
You can see how the indexes move across a line of elements, after "a" (i.e: Left) is pressed, finishing after 7 stages.

1	4	4	4	0
2	4	4	4	0
3	8	0	4	0
4	8	0	4	0
5	8	0	4	0
6	8	4	0	0
7	8	4	0	0

GREEN = Both pointers at that element
BLUE = Pointer A at that element
YELLOW = Pointer B at that element

Figure 1: Example showing movement of pointers

There are two advantages with this implentation:

- With a few conditional statements, the code works with any set of 4 elements.
- In addition, just by swapping where the indexes start, and whether they are incremented or decremented, the code can easily handle a bidirection pair (Left & Right, or Up & Down).

WHEN SHOULD I CHECK FOR GAME OVER? :

Since I knew that the game over check will involve a large amount of element comparison, I knew that, to make the code efficient, this game over checking should occur only when necessary. From analysis of the game over condition, I came up with four points that increase this efficiency:

- If the grid hasn't been changed, the game over condition shouldn't be checked (as nothing as changed). The code works out if anything has been changed from the "push and merge" function.
- In addition, if there are more than one empty elements, the game over condition needn't be checked, as there is always a move that can be taken.
- If there is exactly one empty element, the extra two must be added before the game over condition, because the only time game over can be reached is if a 2 is added, as the grid is then filled.