Report 6g

[Niels Dreesen, Mustafa Hekmat Al Abdelamir, Jakup Højgaard Lützen]

November 5, 2022

1 Forword

This report is about the challenges and implementation decisions our group was faced with, when designing a version of the game 2048 in the programming language F#. This took place at the end of block 1, which is approximately the middle of the PoP course.
At this time, our programming skills were best described as modest, where it's possible to design simple applications. The game 2048 has an appropriate difficulty level for this stage of our education, and it presented us with several challenges further discussed in this report.
We would like to thank Fritz Henglein for his contagious enthusiasm, Jon Sporring for teaching us the wonders of recursive recursion, and above all Peter Kanstrup Larsen for his emotional and academic support through all these weeks of seemingly insurmountable programming challenges.

2 Introduction and problem formulation

The game 2048 (as presented at `www.2048.io`) is conducive to functional programming, as each move can be separated into discrete functions, when done properly.
Our task was to create a version of this game, but where the values of the tiles are represented with colors in the following sequence: (Red, Green, Blue, Yellow, Black).
With the help of the task requirements, we took advantage of a new structure of coding where we are using multiple files. We are using the different files to separate the code between the specific implementation we are working with and the general purpose functions we are creating to facilitate the specific implementation. Furthermore, we had to tackle list manipulations, with regards to the game pieces, which has somewhat arbitrary properties and numbers, which makes them a straightforward choice for storing and manipulating through lists.

3 Problem analysis and design

First we need to represent the objects on screen in variables. Each piece on screen has 2 properties. A color, and a position. According to the assignment requirements this means that we are defining a position, consisting of an integer tuple, a color, a piece which consists of a color and a position, and a state which is a list of pieces, ie. representing the board as a whole. From there we have to think about the functionality.

- We need to be able to push all the pieces in a certain direction as far as they go and
- We need to merge consecutive pieces of the same color when they are pushed "into" each other and
- We need to add a random red piece every turn.

To be able to push pieces as far as they go, we need to somehow gather the pieces from a single row or a column in a list. This we can do with a filter function, which takes n and a state as an input and outputs all the elements in the state which have a matching row/column number to n:

We can use this function for rows as well, if we can transpose the board on its diagonal. This effectively makes the rows of the original state into the columns of the new state. After running the calculations needed on the state, it can be transposed back. We can also make the horizontal and vertical manipulations interchangeable in the same style by making a flipping function which can flip the board.

Once we have gathered a list of the pieces on a row/column, we have to give them new coordinates to push them as far as possible to the edge of the board. To do this, we need to know the sequence they appear on the row/column. So we need to sort the list of pieces, according to the vertical or horizontal coordinate. Once we know the sequence we can just give them the coordinate $e + n$ where e is the edge coordinate and n is the number of the piece in the sorted sequence.

As for the problem of merging pieces, we need two things; first to compare sequential pieces on a row/column to see if they have the same color, and then to give them a new color of double the value.

Lastly, to solve the problem of adding a random piece we must first know all the empty positions on the board. We can do this with a function, which takes a state and returns all the possible positions on the board which weren't in the state. Once we have a list of empty places, we can simply choose a random one and add a new piece to the state with with that coordinate.

With this functionality we should be able develop the program, but as for the GUI we will need to translate the state into a graphical representation, and we need to handle user input to call the different functions.

To begin with we have to create a square window. Each piece is going to be a square as well, with the side-length $\frac{wh}{3}$ where wh is the window side-length. This means that we can place the pieces at $x * \frac{wh}{3}, y * \frac{wh}{3}$ to translate the coordinate of the piece into pixel coordinates on screen and 3x3 pieces can snugly fit on it.
As for the user input, we have to check whether its an arrowkey, and then we need to shift all the pieces in its direction and add a random piece.
Due to the inconsistencies in the assignment paper, a decision needed to be made regarding the directionality and coordinate interpretation of the tile positions. We decided on the following form as shown in Figure1, which mirrors how the Canvas module draws on an actual canvas.

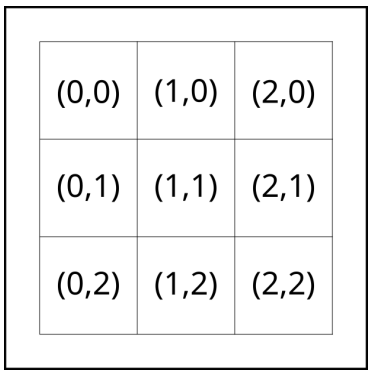

Figure 1: Our implementation of position values.

4 Program description

fromValue

This function simply matches our colorvalue with different colors and transforms each of them into the corresponding color in canvas. This is easily done with a match case.
An example follows, where fromValue Red returns:

```
val it: color = { r=0uy; g=255uy; b=0uy; a=255uy}
```

However - looking at the source code for diku-canvas/canvas.fs, we find this:

```
// colors
type color = {r:byte;g:byte;b:byte;a:byte}
let fromArgb (r:int,g:int,b:int,a:int) : color =
```

```
  {r=byte(r); g=byte(g); b=byte(b); a=byte(a)}
let fromRgb (r:int,g:int,b:int) : color = fromArgb (r,g,b,255)
let red : color = fromRgb(255,0,0)
```

Which means we can conventiently return Canvas.red, which is equivalent to the requested behaviour.

nextColor

The nextColor function defines which colors two tiles of equal color should merge into. This is also done with a match case which points every input color to an output color, in the sequence as shown in figure 2.

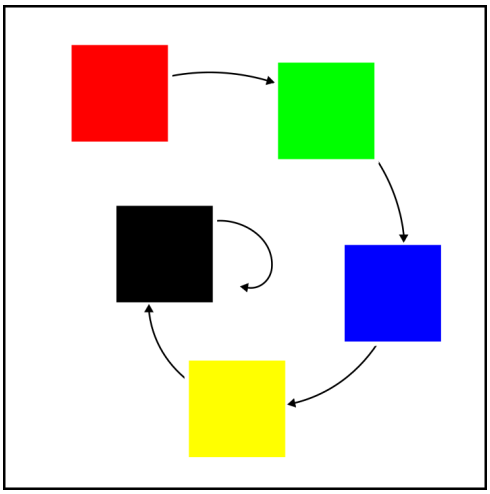

Figure 2: The sequence of colors when merged using nextColor

filter

The filter function uses pattern matching to extract the vertical coordinate from every piece and adds the piece to a new list if the coordinate matches with the input n:

```
let rec filter (n:int) (s:state) : state =
    match s with
        | ele::rest ->
            let (_,(x,_)) = ele
            match (x=n) with
            | true -> [ele] @ (filter n rest)
            | _ -> filter n rest
        | _ -> []
```

shiftUp

This is the function that does all the heavy lifting. The assignment was not very clear in explaining exactly, what this function should be able to do, but simply described that it should "tilt all tiles to the left". We assumed that the mismatch in directions between the specified function name, and the explanation in the comment was a mistake, and as such, we chose that it should indeed "tilt all tiles upwards." Furthermore, in "tilting", we understood it as first merging all vertically adjacent tiles with the same color, and then moving every tile as far up as possible. The output of this function should in other words correspond to pressing the Up-arrow.
To fully comprehend this function, we wrote the following pseudo-code, breaking this action into smaller and simpler parts:

```
for each column (using filterfunction):
    sort by row
    for every tile:
        if tile color = adjecent tile color:
```

```
            replace the two tiles with a tile of (nextColor,(this tile position))
    for every tile:
        move tile number n to position 0+n
```

The shiftUp function processes one column at a time. So it uses the filter function to extract a column. Then it sorts the elements in the column according to their vertical coordinate. This means that we are left with a sorted list of a column.

```
s  |> filter column |> List.sortBy (fun (_,(_,y)) -> y)
```

This is passed into the mergeTiles subfunction which compares the two first elements to see if they are the same color. If they are they get replaced by a single piece of the next color on the position of the first element. If theyre not the same color, the function recursively calls itself without the first element to check whether the next two elements are the same. This is done using the .Head and .Tail properties of lists to extract the relevant elements. And notice the function will check if there is just one element in the list, and won't do any further tests if its true, since there are no merging possibilities:

```
let rec filter (n:int) (s:state) : state =
    match s with
        | ele::rest ->
            let (_,(x,_)) = ele
            match (x=n) with
            | true -> [ele] @ (filter n rest)
            | _ -> filter n rest
        | _ -> []
```

```
let rec mergeTiles (s:state) : state =
    match s with
    | s when s.Length = 1 -> s
    | elem::rest -> if (fst elem) = (fstrest.Head) then
                        [(nextColor (fstelem),snd elem)]
                            @ rest.Tail else  @ (mergeTiles rest)
    | [] -> []
```

This is then passed to a shoveTiles subfunction which assigns each piece its index in the list as it's vertical coordinate:

```
let rec shoveTiles (n:int) (s:state)  :state =
    match s with
        |(col,(x,y))::rest ->(col,(x,n))::(shoveTiles (n+1)rest)
        |_ -> []
```

flipUD

The flip upside down function uses list.map to reorder our pairs according to our index of vertical coordinates. The new vertical coordinate value is simply the old one decreased by 2 in order to change the position of the tiles in the corresponding direction.

transpose

The transpose function uses list.map as well in order to transform the position values, simply by flipping the x and y coordinate values.

empty

This function returns a list of empty positions on the board by creating a list of all possible position, a list of all taken positions and then find the "difference" between the lists, with the List.except function:

```
s  |> filter column |> List.sortBy (fun (_,(_,y)) -> y)
```

recklessAdd

This function was not strictly necessary, but it made creating the beginning state much cleaner. It does exactly the same as addRandom, except it returns a state instead of a state option. **This function will crash** if the board is full (when addRandom would return None), so it should only be used when a free space is guaranteed (e.g. when populating the board at game onset).

newTileIfMoved

We noticed that the original game does not add a random tile, if the board is unchanged following a keypress, even when there are empty spaces for tiles.
We implemented this by checking if the state had changed after a keystroke, and only adding a tile if it indeed had changed.
This resulted in a high degree of repetition of code in the react function, so we decided to abstract it away into a seperate function, which takes the old and new states as arguments, and then returns the output of addRandom, if and only if the state has changed. Otherwise it returns None.

Once in the root directory of the program it can be run with the command:

```
dotnet run \Program.fsx
```

Numerous test results to check the validity of the program will be printed. When the GUI has lunched, it is possible to control the game with the arrow keys. Press the arrowkey in the direction you want to tilt the pieces.

5 Testing and experiments

In order to test if our program works correctly, we implement a unittest in our program.fs, which prints the outcome of various tests. In these tests we use multiple testlists and compare the state after the different functions with the expected outcome.
We chose to test the functions (shiftUp, flipUD, transpose, empty, addRandom, fromValue and nextColor), because these are used to create the new boardstate after a given move. To further illustrate the functionality of our program, we include a figure of random moves in a random game below.

6 Discussion and conclusion

This assignment presented some unexpected challenges. One of which being simply keeping track of the x and y coordinates.
It turnes out that these coordinates mentally get easily mixed up, for instance when sorting a list, or in the filter function, where we quite counter intuitively need to evaluate the x coordinate to get the correct column.
It might seem like a banality, but drawing a grid with the coordinates (such as figure 1), and looking at it when in doubt can greatly help minimize bugs when coding after a long and tiring day.
The unit test had its own set of challenges.
How does one check for equality between game states? The state is unfortionately a list, and the order of elements matter in lists. We therefore opted to check for equality using Sets which eliminate this problem as demonstrated here:

```
> let aList = [1;2;3]
- let bList = [3;2;1]
- let aSet = Set.ofList aList
- let bSet = Set.ofList bList
- let listEquals = (aList=bList)
- let setEquals = (aSet=bSet)
- printfn "The lists are equal: %A" listEquals
- printfn "The sets are equal: %A" setEquals;;
The lists are equal: false
The sets are equal: true
```

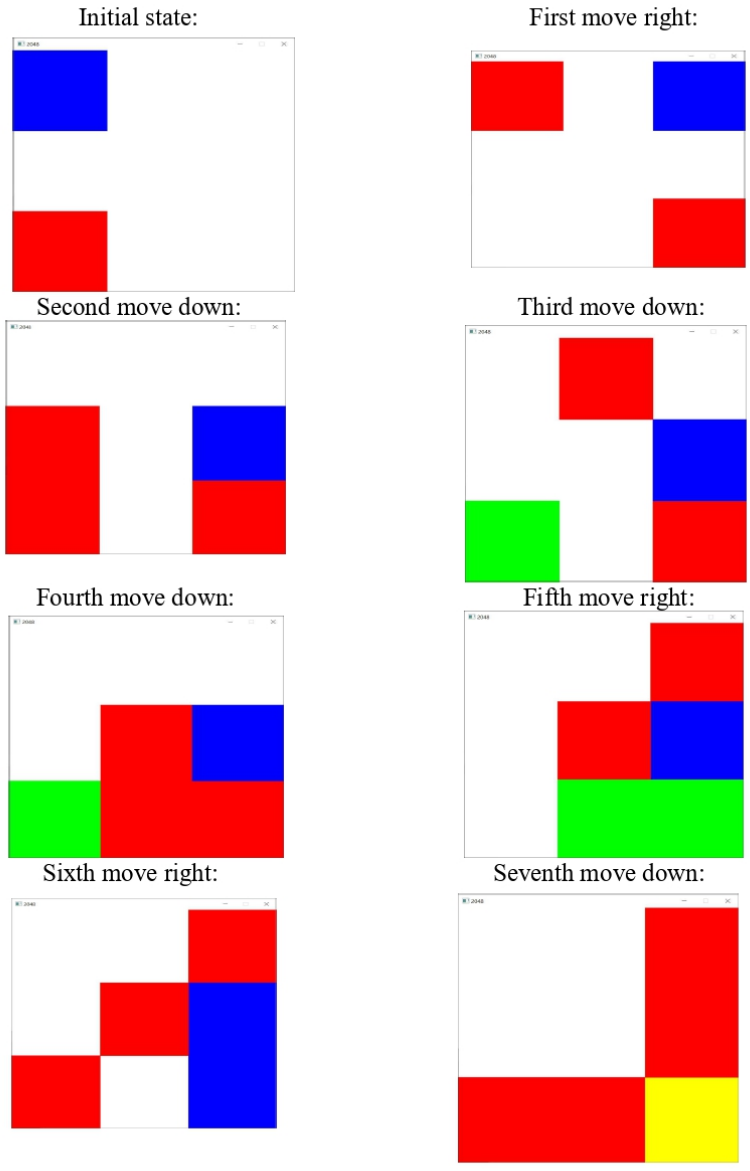

Figure 3: An example of random moves on our gameboard.

Alternatively, one could resort to using elaborate sorting algorithms, which are very dependant on datatypes etc. Using sets elegantly eliminates this concern. Using sets however has one major drawback. This method would not catch if a function creates duplicate identical tiles, as these would be eliminated when converting the list into a set as demonstrated here:

```
> let aList = [1;2;3]
- let bList = [1;1;2;2;3;3]
- (Set.ofList aList = Set.ofList bList)
-     |> printfn "These sets are equal: %A";;
These sets are equal: true
```

This could be checked for, by making sure that the lengths match every time a state is turned into a set.

This very problem of duplicate tiles could also be avoided by designing the game entirely around a different data type.
It would intuitively make sense to represent the grid as a 2-dimensional list where each place in the grid matches the indices of the list. The "value" data type could in this case be expanded to include an "Empty" value, making it easy to count the empty tiles.

Overall we developed our skills through the project work for 6g and are satisfied with the result of our program. Some things in the code could possibly be solved more smoothly, but our solutions seem practical for the different tasks.