Københavns universitet

Programmering og problemløsning - Jon Sporring - 05/11/2022

Rapport 6g

Thor Stie Gregersen mxb267

Alexander Bang dkg213

GitHub repo: https://github.com/F1dsy/PoP/tree/master/6g/src

Indholdsfortegnelse

Forord

Dette er en rapport til opgave 6g i pop på KU skrevet af Alexander Bang og Thor Gregersen. Rapporten omhandler hvordan vi har lavet spillet 2048 i fsharp vha. Canvas biblioteket.

Introduktion

Vi har skrevet programmet 2048, dette er et eksisterende program, som vi har taget udgangspunkt i til vores programfunktioner. Selve opgaven giver os dog nogle andre betingelser end det originale program. Rapporten går igennem vores arbejdsprocess og beskriver hvordan vi har løst de opgaver og problemstillinger vi er blevet stillet undervejs, samt kommer med et endeligt spil.

Problemformulering

Opgaven giver os nogle type funktioner, som vi bruger i vores implementationsfil. Spillet benytter også en anderledes form end det originale spil, vi har til opgave at lave spillet på en 3x3 plade istedet for en 4x4 plade, og vi bruger farver til at repræsentere spilbrikkerne istedet for tal.

Problemanalyse og Design

Programmet har vi skrevet ud fra de givne typer, og vi har brugt eksemplerne samt tipsnene gemt i kommentarene til at lave vores signaturfil

```
module Lib

type pos = int * int // A 2 - dimensional vector in board - coordinats (notpixels )

type value =
    | Red
    | Green
    | Blue
    | Yellow
    | Black // piece values

type piece = value * pos //
type state = piece list // the board is a set of randomly organizedpieces


// convert a 2048 - value v to a canvas color E.g. ,
// > fromValue Green ;;
// val it: color = { r = 0uy
// g = 255 uy
// b = 0uy
// a = 255 uy }
val fromValue: v: value -> Canvas.color

// give the 2048 - value which is the next in order from c, e.g. ,
// > nextColor Blue ;;
// val it: value = Yellow
// > nextColor Black ;;
// val it: value = Black
val nextColor: c: value -> value

// return the list of pieces on a column k on board s, e.g.
// > filter 0 [( Blue , (1 , 0)); (Red , (0 , 0))];;
// val it: state = [( Blue , (1 , 0)); (Red , (0 , 0))]
// > filter 1 [( Blue , (1 , 0)); (Red , (0 , 0))];;
// val it: state = []
val filter: k: int -> s: state -> state

// tilt all pieces on the board s to the left ( towards zero on the first coordinate ), e.g. ,
// > shiftUp [( Blue , (1 , 0)); (Red , (2 , 0)); (Black , (1 ,1))];;
// val it: state = [( Blue , (0 , 0)); (Red , (1 , 0)); (Black , (0 ,1))]
```

```
41 val shiftUp: s: state -> state
42
43 // flip the board s such that all pieces position change as
44 // (i,j) -> (2 -i,j), e.g.
45 // > flipUD [( Blue , (1 , 0)); (Red , (2 , 0))];;
46 // val it: state = [( Blue , (1 , 0)); (Red , (0 , 0))]
47 val flipUD: s: state -> state
48
49 // transpose the pieces on the board s such all piece positiosn
50 // change as (i,j) -> (j,i), e.g. ,
51 // > transpose [( Blue , (1 , 0)); (Red , (2 , 0))];;
52 // val it: state = [( Blue , (0 , 1)); (Red , (0 , 2))]
53 val transpose: s: state -> state
54
55 // find the list of empty positions on the board s, e.g. ,
56 // > empty [( Blue , (1 , 0)); (Red , (2 , 0))];;
57 // val it: pos list = [(0 , 0); (0 , 1); (0 , 2); (1 , 1); (1 , 2);
58 //(2 , 1); (2 , 2)]
59 val empty: s: state -> pos list
60
61
62 // randomly place a new piece of color c on an empty position on
63 // the board s, e.g. ,
64 // > addRandom Red [( Blue , (1 , 0)); (Red , (2 , 0))];;
65 // val it: state option = Some [( Red , (0 , 2)); (Blue , (1 , 0));
66 //(Red , (2 , 0))]
67 val addRandom: c: value -> s: state -> state option
```

Problembeskrivelse

Vi satte fs filen op ud fra vores signatur fil. Selve funktionerne varierede meget i sværghedsgrad, så vi startede med at lave de simple funktioner, det var f.eks. 'fromValue' og 'nextColor'. De fleste af funktioner kunne laves vha. en enkelt linjes kode, som vi kom frem til ud fra kommentarene.
Dog var der nogle mere komplicerede funktioner, 'empty' krævede mere forståelse af hvordan lister virker i fsharp end hvad vi havde på stående fod. Vi fandt ud af det vha. microsofts beskrivelse af lister, og endte med en rimelig simpelt funktion.

```
1  module Lib
2
3  type pos = int * int // A 2 - dimensional vector in board - coordinats (notpixels )
4
5  type value =
6      | Red
7      | Green
8      | Blue
9      | Yellow
10     | Black // piece values
11
12 type piece = value * pos //
13 type state = piece list // the board is a set of randomly organizedpieces
14
15 let fromValue (v: value) : Canvas.color =
16     match v with
17     | Red -> Canvas.red
18     | Green -> Canvas.green
19     | Blue -> Canvas.blue
20     | Yellow -> Canvas.yellow
21     | Black -> Canvas.black
22
23 let nextColor (c: value) : value =
24     //red 2-> green 4-> blue 8-> yellow 16-> black 32
25     match c with
26     | Red -> Green
27     | Green -> Blue
28     | Blue -> Yellow
29     | Yellow -> Black
30     | Black -> Black
31
32 let filter (k: int) (s: state) : state =
33     List.filter (fun (v, (x, y)) -> y = k) s
34
35 let flipUD (s: state) : state =
36     List.map (fun (v, (x, y)) -> (v, (2 - x, y))) s
37
38 let transpose (s: state) : state =
39     List.map (fun (v, (x, y)) -> (v, (y, x))) s
40
41 let empty (s: state) : pos list =
42     let emptySpots: pos list =
```

```
        [ (0, 0)
          (0, 1)
          (0, 2)
          (1, 0)
          (1, 1)
          (1, 2)
          (2, 0)
          (2, 1)
          (2, 2) ]

    let positions = List.map (fun (v, pos) -> pos) s
    List.except positions emptySpots

let addRandom (c: value) (s: state) : state option =
    let emptySpots = empty s

    if (List.length emptySpots) = 0 then
        None
    else
        let rnd = System.Random()
        let nrn = rnd.Next(List.length emptySpots)
        let npos = emptySpots.[nrn]
        let ns = (c, npos)
        Some(s @ [ ns ])
```

I ovennævnte implementationsfil har vi udeladt vores shiftup funktion, da denne kræver lidt mere forklaring. Funktionen er opbygget med to skridt. Det ene er `compress` og den anden er `merge`. Funktionen gør det at den starter med at "komprimere" alle brikkerne opad for hver kolonne. Derefter tjekker den for par og fletter dem sammen i `merge` funktionen og til sidst komprimerer den igen. Dette sikrer at alle brikkerne ender så langt oppe, som mulig og at de rigtige brikker bliver flettet sammen.

```
let shiftUp (s: state) : state =
    let mutable ns: state = []

    for c = 0 to 2 do
        let fltd = filter c s

        // Checks if an element with x coordinate exists in list
        let find lst x =
            List.exists (fun (v1, (x1, y1)) -> x1 = (x)) lst

        // Checks if an element with x coordinate and value v exists in list
        let findv lst v x =
            List.exists (fun (v1, (x1, y1)) -> v1 = v && x1 = x) lst

        let compress lst =
            List.map
                (fun (v, (x, y)) ->
                    if (find lst (x - 1)) || x = 0 then
                        (v, (x, y))
                    else
                        (v, (x - 1, y)))
```

```
            lst
        let merge lst =

            List.choose
                (fun (v, (x, y)) ->
                    if (find lst (x + 1)
                        && findv lst v (x + 1)
                        && not (findv lst v (x - 1))) then
                        Some((nextColor v, (x, y)))
                    elif (find lst (x - 1)
                          && findv lst v (x - 1)
                          && not (findv lst v (x - 2))) then
                        None
                    else
                        Some((v, (x, y))))
                lst
        ns <- ns @ (compress >> merge >> compress) fltd
    ns
```

Til den sidste del af opgaven bliver vi bedt om at gøre det muligt for brugeren at bevæge felterne vha. tasteturet. Hertil tilføjede vi en react funktion, som tjekker om brugeren klikker på en piletast og derefter rykker på brikkerne alt efter hvilken piletast der blev klikket, og tilføjer en rød brik på et tilfældigt felt.

```
#r "nuget: DIKU.Canvas, 1.0.1"
#load "6g0Lib.fs"

open Lib

let draw (w: int) (h: int) (s: state) : Canvas.canvas =
    let boardsize = 3
    let dw = w / boardsize
    let dh = h / boardsize
    let c = Canvas.create w h

    for (v, (x, y)) in s do
        Canvas.setFillBox c (fromValue v) (dh * y, dw * x) (dh * (y + 1), dw * (x + 1))
    c

let react (s: state) (k: Canvas.key) : state option =
    let reacted =
        match Canvas.getKey k with
        | Canvas.UpArrow -> Some(shiftUp s)
        | Canvas.DownArrow -> Some((flipUD >> shiftUp >> flipUD) s)
        | Canvas.LeftArrow -> Some((transpose >> shiftUp >> transpose) s)
        | Canvas.RightArrow ->
            Some(
                (transpose
                 >> flipUD
                 >> shiftUp
                 >> flipUD
                 >> transpose)
                    s
```

```
30              )
31          | _ -> None
32
33      if (reacted = None) then
34          reacted
35      else
36          addRandom Red reacted.Value
37
38  let w = 600
39  let h = w
40
41  let s: piece list =
42      [ (Red, (0, 0))
43        (Blue, (2, 0)) ]
44
45  Canvas.runApp "app" w h draw react s
```

Afprøvning og eksperimentation

Vi kunne ikke afprøve programmet før vi havde en fsx fil, så vi tjekkede sådan set koden efter ved at se om syntaksen passede. Da vi satte vores fsx fil op kunne vi så se alle de fejl der var i implementationen. Til første del af opgaven havde vi ikke nogen react funktion så den fungerede uden problemer.

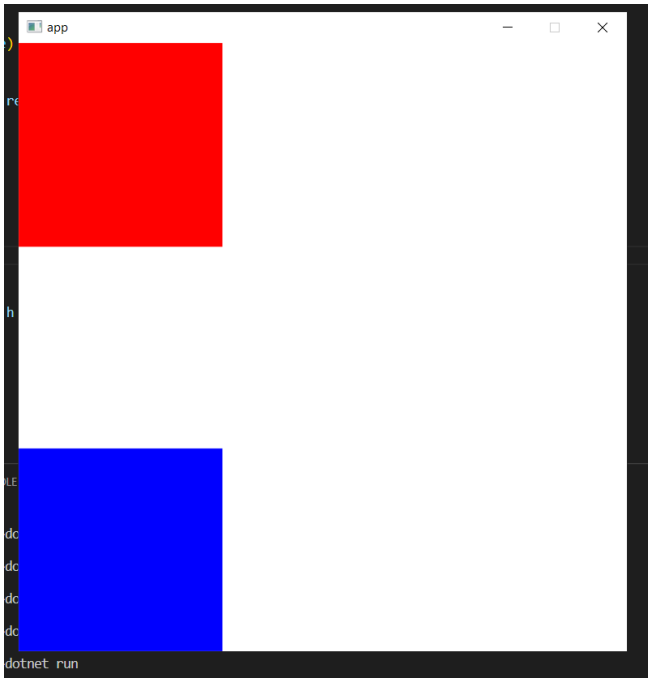

Figure 1: Færdige resultat af 6ga

Da vi satte vores react funktion op til opgave 6gb, opdagede vi at shiftup funktionen var meget kompliceret og blev nødt til at ændre i den. Da vi havde funktionerne på plads tjekkede vi spillet igen og fandt ud af at, hvis der var 3 ensfarvede brikker på række, ville de flette sammen til 2 nye farver, så 3 røde blev til 2 grønne, hvor de burde blive til 1 grøn og 1 rød. Dette løste vi og endte med et færdigt program.

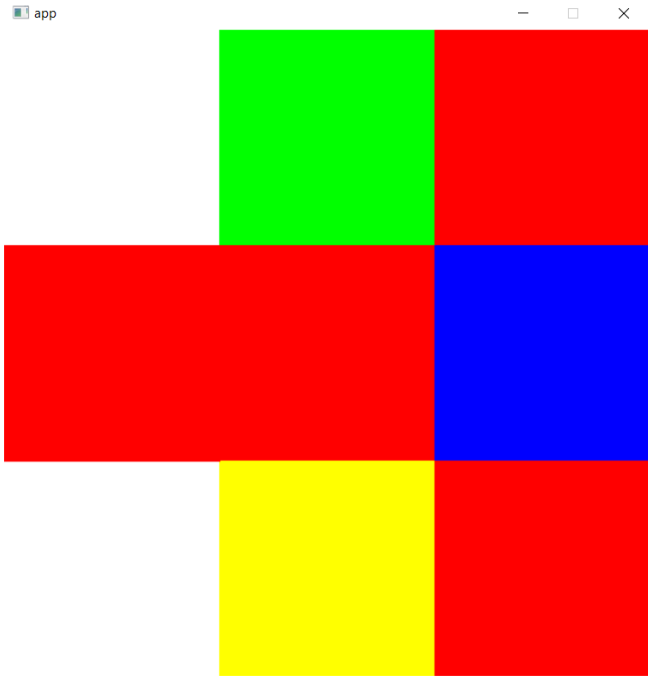

Figure 2: Resultat efter tilfældigt tryk på piletasterne

Diskussion

I det originale 2048 spil var der nogle regler som vi ikke har implementeret i vores spil. Nogle af de ting vi ikke fik implementeret er:

- Når spillet slutter (når man har tabt) laver vores program intet og bliver nødt til at blive stoppet manuelt af brugeren.
- Når man laver et træk i en retning hvor der ikke sker ændringer på positionerne af brikkerne, tilføjer det originale spil ikke en ny brik. Vores spil tilføjer dog altid en brik uafhængigt af om `state` har ændret sig. For at implementere dette ville vi tjekke, om den forrige og den resulterende state er den sammme, ved at tjekke hvert element igennem `state`.
- Når spillet starter bliver det originale spil altid startet med brikker på tilfældige positioner. Vores spil starter altid ens.

Vi havde problemer til at starte med da shift up funktionen egentlig var halvdelen af spillets funktionalitet. Det virkede underligt at vores library, som bestod af en masse enkle funktioner som gjorde en ting, havde en så stor og kompleks funktion midt i det hele. Vi blev derfor lidt i tvivl om vi havde forstået opgaven rigtigt og om logikken egentlig skulle implementeres et andet sted som f.eks. i react funktionen.

Konklusion

Vi har til denne opgave lavet en forenklet implementation af spillet 2048. Programmet består af et 3x3 bræt som starter ud med en rød og en blå brik, man kan bevæge sine brikker rundt vha. piletasterne. Der kommer en ny rød brik efter hvert tryk og hvis 2 ens farver rammer hinanden fletter de sammen til en ny farve. Spillet slutter når man ikke kan bevæge en eneste brik længere.