PROGRAM DESIGN AND DATA STRUCTURES, 2017/2018

FLAPPY BIRD

February 26, 2018

Group 18:
Alexander Sellström
Johan Andersson
Erik Granberg

Contents

Introduction

Having done no graphics and almost only pure functions in haskell we were curious as to see what it was like. That is why we decided to make a graphical game. We were recommended to use a graphics library called Gloss [1]. Their slogan "Painless 2D vector graphics, animations and simulations." really made it sound like the perfect fit for our little project.

Flappy bird was a pretty tactical choice. It is a well known game that has really straight forward rules. We didn't want to create a new game because then we would have to spend a lot of time with the concept of a game as opposed to just actualizing an existing concept.

We encountered several challenges while creating this game. One of the biggest challenges was understanding the graphics library gloss. Not because it is hard to use but, because of the lack of documentation and examples. Here Haskell's type declarations proved to be incredibly useful.

1.1 FLAPPY BIRD

Flappy bird is a side-scrolling game developed by Vietnamese programmer Dong Nguyen. It was released in late May 2013 and is heavily inspired by retro games such as super mario. By the end of January 2014 it was the most downloaded free game in the iOs App Store. The game supposedly had average daily earnings as high as $50,000 from in-app ads during its peak. [3]

1.1.1 Gameplay

The objective is to direct a bird that is constantly pulled down by gravity between a set of pipes. The length of the pipes vary but the gap between them is always of the same size. If the bird comes in contact with a pipe or the ground/ceiling it is game over. The points keep stacking up until the player messes up and it's game over.

1.1.2 Software requirements

To compile the game you need to install the Haskell Platform [2]. This includes Glasgow Haskell Compiler, commonly refered to as GHC, and the Cabal package manager. You also need to install the graphics library Gloss and Gloss-Game. To install these, open up your terminal and type:

```
cabal install gloss gloss-game
```

Use the terminal and navigate to the directory where "Main.hs" is located and type:

```
ghc Main.hs
```

then type:

```
./Main
```

Now you should be presented with the game menu.

1.1.3 How to play

The game is incredibly simple to play. When you run the program, you are greeted with a splash screen.As soon as you run the program the game start and the bird gravitates towards the ground. Press space in order to keep the bird in the air. As mentioned above the objective is to direct the bird between the pipes so alternate between letting gravity pull the bird down and pressing space in order to move the bird up so that you can pass through the gap.

Program description

This section will go in to more detail and explain the most important functions and data types that make up the game.

2.1 Data types

```
data FlappyBird = GameOver {finalScore :: (Integer)}
                | MenuScreen {}
                | Bird {birdLoc :: (Float, Float)
                , birdVel :: (Float)
                , pipeLoc1 :: (Float)
                , safeZone1 :: (Float, Float)
                , pipeLoc2 :: (Float)
                , safeZone2 :: (Float, Float)
                , pipeLoc3 :: (Float)
                , safeZone3 :: (Float, Float)
                , score :: Integer} deriving (Show, Eq)
```

This is the data type we choose to use to represent the state of our game. It has three different constructors for the different states of the game. GameOver, MenuScreen and Bird. GameOver is used when the bird has died, it's only element being the amount of points accumulated during the game. MenuScreen has no elements and is used to greet the player upon launching the game. Bird is used during the actual gameplay and contains the position and velocity of the bird, the positions of all the pipes and safe zones, and the current score.

`birdLoc (Float, Float)`

Represents the birds coordinates (x,y).

```
birdVel (Float)
```

Represents the birds velocity i.e at which rate the aforementioned birdLoc changes.

```
pipeLocX :: (Float)
```

Represents the x-coordinate of the Xth pipe

```
safeZoneX :: (Float, Float)
```

Represents the coordinates (x,y) of the Xth safeZone. The safeZone is what you would see as the gap between the pipes in the game.

```
score :: Integer
```

Represents the score in the game.

2.2 FUNCTIONS

```
main :: IO ()
main = play window background 60 menuscreen render handleKeys update
```

Play is a function from gloss that we modeled our entire game after. As you can see it takes seven argument. Here we will list all the arguments in order that the play function wants and further down is the explanations of the actual arguments that we passed.

1. A window
2. A colour for the background
3. The number of simulations steps to take for each second of time.
4. The initial game state
5. A function that renders the current game state into a single picture
6. A function that handles input events

7. A function to step the game one iteration. It is passed the period of time (in seconds) needing to be advanced.

```
window :: Display
window = InWindow "Flappy Birb" (width, height) (offset, offset)
```

InWindow is a function predefined in gloss that take a string and two tuples of Ints. Which results in a window with the name of the input string, the height and width from the first tuple and the offset from the second.
Example: width = 900 height = 600 offset = 100 window = InWindow "Flappy Birb" (width, height) (offset, offset) will produce a window with the name "Flappy Birb" that is 900x600 pixels.

```
initialState :: FlappyBird
initialState = Bird
  { birdLoc = (-350, 0)
  , birdVel = (-100)
  , pipeLoc1 = (-100)
  , safeZone1 = (-100, 0)
  , pipeLoc2 = (200)
  , safeZone2 = (200, 150)
  , pipeLoc3 = (500)
  , safeZone3 = (500, -150)
  , score = 0}
```

Like the name suggests this is the initial state of the game, the positions of everything when then game starts.

```
mapNumbers :: Integer -> Picture
mapNumbers x
          | x == 0 = numba0
          | x == 1 = numba1
          | x == 2 = numba2
          | x == 3 = numba3
          | x == 4 = numba4
```

```
        | x == 5 = numba5
        | x == 6 = numba6
        | x == 7 = numba7
        | x == 8 = numba8
        | otherwise = numba9
```

Takes in a single digit Integer and returns a picture representing that digit.

```
scoreAslist :: Integer -> [Integer]
scoreAslist int
            | (int `div` 10) == 0 = [int]
            | otherwise = (scoreAslist (int `div` 10)) ++ scoreAslist (int `mod` 10
```

Takes an integer and returns a list of every digit in that integer

```
scorePicture :: Position -> [Integer] -> [Picture]
scorePicture (_,_) [] = []
scorePicture (x,y) list = [translate x y (mapPicture (head list))] ++ (scorePicture (
```

Takes a position and a Integer list and returns a picture of the Integers starting at the position with each digit offset by 30 pixels.

```
ripscreen :: Integer -> FlappyBird
ripscreen x = GameOver {finalScore = x}
```

Used to set the current game state to GameOver whenever the bird is killed. The argument x is the amount of points accumulated by the player during the game.

```
checkState :: FlappyBird -> Integer
checkState MenuScreen {} = 1
checkState GameOver {} = 3
checkState Bird {} = 2
```

This function checks the state of the game by using pattern matching on the data constructors of FlappyBird

```
moveBackground :: Integer -> [Picture]
moveBackground x = [translate (-(realToFrac y)) 0 wallpaper] ++ [translate (-(realToF
                    where y = x `mod` 1800
```

This function translates two pictures of the background to follow eachother using the score of the game as argument and puts them in a list.

```
render :: FlappyBird -> Picture
render game
  | (checkState game) == 1 = pictures ([wallpaper] ++ [menu] ++ [translate (-350) 0 b
  | (checkState game) == 2 = pictures ([pape] ++ pipes ++ [bird] ++ counter)    --
  | otherwise = pictures ([wallpaper]
                      ++ [translate 20 (-100) restart]
                      ++ [translate (-20) 150 finalscore]
                      ++ [birb2]
                      ++ (scorePicture ((103), (140)) (scoreAslist (finalScore game)
    where
      pape = translate (pipeLoc1 game) 0 wallpaper
      bird = uncurry translate (birdLoc game) $ mapFlaps (birdVel game)
      counter = scorePicture ((-40), 200) (scoreAslist (score game))
      pipes = [pipe1, zone1, pipe2, zone2, pipe3, zone3]
      pipe1 = translate (pipeLoc1 game) 0 pipe
      zone1 = uncurry translate (safeZone1 game) $ zone
      pipe2 = translate (pipeLoc2 game) 0 pipe
      zone2 = uncurry translate (safeZone2 game) $ zone
      pipe3 = translate (pipeLoc3 game) 0 pipe
      zone3 = uncurry translate (safeZone3 game) $ zone
      pipeColor = dark green
```

This is the function that draws the all frames in the game. It first calls on checkState with the current game state. If checkState returns 1, the menuscreen is rendered, this includes the background, a welcome message telling the player how to start the game and a picture of the bird. If checkState returns 2, the current game state is rendered, this includes the bird, the pipes, the safe zones, the background and the current score. If checkState returns something else the game over screen is rendered, this includes the background, the bird, the final score and a message telling the player how to start over.

```
moveObjects :: Float -> FlappyBird -> FlappyBird
moveObjects time game
 | (checkState game) == 2 = game {birdLoc = (x, y')
                                  , birdVel = (vy')
                                  , pipeLoc1 = (a')
                                  , safeZone1 = (c', d)
                                  , pipeLoc2 = (e')
                                  , safeZone2 = (g', h)
                                  , pipeLoc3 = (i')
                                  , safeZone3 = (k', l)
                                  , score = (z + 1)}
 | otherwise = game
  where
    z = score game
    -- Old location and velocity for the bird.
    (x, y) = birdLoc game
    (vy) = birdVel game
    -- New velocity
    vy' = vy - 350 * time
    -- New location
    y' = y + vy * time
    -- Old locations of pipes and safe zones
    (a) = pipeLoc1 game
    (c, d) = safeZone1 game
    (e) = pipeLoc2 game
    (g, h) = safeZone2 game
    (i) = pipeLoc3 game
    (k, l) = safeZone3 game
    -- New locations of pipes and safe zones
    a' = a - 150 * time
    c' = c - 150 * time
    g' = g - 150 * time
    e' = e - 150 * time
    i' = i - 150 * time
    k' = k - 150 * time
```

This is where all the movement takes place. Change in the birds velocity, movement of pipes, everything that moves in the game. The function takes a Float that represents the time since the last iteration and a FlappyBird which is the current game state and then outputs a modified game state with respect to the time passed.

```
handleKeys :: Event -> FlappyBird -> FlappyBird
handleKeys (EventKey (SpecialKey KeySpace) Down _ _) game@(Bird {birdVel = (z, w)}) =
handleKeys (EventKey (SpecialKey KeyF1) Down _ _) _ = initialState
handleKeys _ game = game
```

This is the function that handles the keyboard input. In our case it's just space which changes the birds velocity to 200 and F1 which restarts the game.

```
checkCollision :: FlappyBird -> FlappyBird
checkCollision game
  | (checkState game == 1) || (checkState game == 3) = game
  | wallCollision (birdLoc game) = ripscreen (score game)
  | pipeCollision (birdLoc game) (safeZone1 game) = ripscreen (score game)
  | pipeCollision (birdLoc game) (safeZone2 game) = ripscreen (score game)
  | pipeCollision (birdLoc game) (safeZone3 game) = ripscreen (score game)
  | newPipe (pipeLoc1 game) = game {pipeLoc1 = (450), safeZone1 = (450, 0) }
  | newPipe (pipeLoc2 game) = game {pipeLoc2 = (450), safeZone2 = (450, -150)}
  | newPipe (pipeLoc3 game) = game {pipeLoc3 = (450), safeZone3 = (450, 150)}
  | otherwise = game
```

This function takes the game state as input and uses pattern matching to check if certain events are happening in the game. It checks if the bird is colliding with a pipe or the floor/ceiling in which case it will call on the ripscreen function with the current score in the game. The final thing that this function checks is if the pipes have reached a x-coordinate smaller then (-490), which in our case would mean that the pipe is 40 pixels outside of the screen. If this happens the pipes location is changed to (450).

```
wallCollision :: Position -> Bool
wallCollision (_, y)  = topCollision || bottomCollision
  where
    topCollision =  y + 10 >= (300)
    bottomCollision = y - 10 <= (-300)
```

As you saw above this is function is called on by the checkCollision function with the current location of the bird. What it does is that it check if the y-coordinate of the bird plus it's hitbox is greater or smaller then the height of the window, that is if the bird is colliding with the ceiling or the ground.

```
pipeCollision :: Position -> Position -> Bool
pipeCollision (x,y) (a,b) = topCollision || bottomCollision
  where
    topCollision = interval && y >= (b+40)
    bottomCollision = interval && y <= (b-40)
    interval = x >= (a-20) && not(x > a + 20)
```

This function is also called on by checkCollision but with the coordinates of the bird and the coordinates of the safe zone, which we explained earlier is the gap between the set of pipes. The function checks if the birds x-coordinate is within the interval that is the width of the safe zone and pipes and if its y-coordinate is larger then the y-coordinate of the safe zone plus half of the safe zones length. It of course also checks if the y-coordinate of the bird is smaller then the y-coordinate of the safe zone minus half its length.

```
newPipe :: Float -> Bool
newPipe x
  | x < (-490) = True
  | otherwise = False
```

This function is called on by checkCollision with the current x-coordinate of the pipe. If value if the x-coordinate is lower then -490 i.e 40 pixels outside of the game window it returns True.

```
update :: Float -> FlappyBird -> FlappyBird
update seconds = checkCollision . moveObjects seconds
```

This is just a composition of functions that we explained before. This is the format in which the play function wants all functions that are invoked once each iteration.

Known shortcomings

One thing that we did not manage to implement was the randomly generated pipe lengths that exists in the original version of the game.

In our version of the game we have 3 sets of pipes of different length which makes the game predictable.

We could easily fix this if we were to use unsafePerformIO. But choose not to because we were recommended not to use it.

We also tried a pseudo random algorithm called linear congruential generator that takes a seed and produces a number, you could then use that number as a new seed and thus creating a sequence of numbers. However this proved to be very calculation heavy. We thought about creating maps outside the game in a separate program and then using them, but decided it was pretty much as predictable as the current game is.

Bibliography

[1] Gloss. http://gloss.ouroborus.net/wiki. visited on 2018-02-26.

[2] The haskell platform. https://www.haskell.org/platform/. visited on 2018-02-26.

[3] Ellis Hamburger. The verge. https://www.theverge.com/2014/2/5/5383708/flappy-bird-revenue-50-k-per-day-dong-nguyen-interview. visited on 2018-02-26.