G54RFP Coursework Part III

Jack Ellis
psyje5@nottingham.ac.uk
4262333

Contents

1 Project Overview **1**

2 Technical Background **2**

3 Implementation **3**
3.1 Programming 3

4 Reflections **5**
4.1 What was learned 5
4.2 Project Topic from a real-world perspective 5

A Figures **6**

1 Project Overview

Breakout is a video game published by Atari in 1976, where the player controls a paddle and must, by using this paddle to bat a ball back and forth, destroy a number of bricks arranged in rows at the top of the screen. This project aims to create a *Breakout* clone using Haskell and the Threepenny GUI library.

2 Technical Background

This project is by and large written in standard Haskell, using Prelude functions. The only external library featured is the Threepenny GUI framework. This is a GUI framework written in Haskell that uses the web browser as a display. A program written with Threepenny is a small web server that:

- Displays the UI as a web page
- Allows the HTML Document Object Model (DOM) to be manipulated
- Handles JavaScript events in Haskell

It works by sending JavaScript code to the client as a Foreign Function Interface.

As well as the above web interface features, Threepenny also provides a framework for handling Functional Reactive Programming (FRP). It introduces a type `Event a`, which is a type synonym for `[(Time, a)]`, that captures and represents a stream of events (of type `a`), carrying timestamps. `Event` is an instance of *Functor*, but not *Applicative*, mostly because the type for `<*>` would be `Event (a -> b) -> Event a -> Event b`, which makes no sense as event streams in general are not synchronised. This type can be combined with another type introduced by Threepenny, the `Behavior a` type. This type represents a value that varies with time, and is the type that will encapsulate the "whole value" which changes in an FRP application.

Threepenny provides support for recognition of a number of types of DOM event, including keyboard keypresses and mouse clicks. It introduces a new kind of event, the `Timer`, which gives us the ability to have a constantly occurring event. This will be used in this project to allow the ball to move continuously. The timer constructor takes an `Int` as an argument, this is used to denote the interval (in milliseconds) between ticks. Functions can be bound to events using the `<$` and `<$>`, which differ only in that the former feeds the same value into a function where the latter, used in this project for the `keydown` event, feeds in the value returned by the event (in our case the keycode of the key pressed).

3 Implementation

3.1 Programming

I first defined 4 data types and two type synonyms, shown in Figure 1. `Pos` and `Dim`, meaning position and dimension respectively, are simply synonyms for `Double`, and will be used to avoid functions with type signatures like `Double -> Double -> Double -> Double -> Double -> Bool`, which are somewhat confusing and difficult to read. The `HitWhere` type will be used in collision detection to determine where on a either the paddle or a brick the ball has hit, in order to make the ball "bounce" appropriately. `GameState` is the "state" of the game, be it before/during a level, paused, complete, or the player has died. `Block` is the type that will be used for the bricks on screen, containing their position in x and y, the amount the player's score will increase by when the block is broken, a value of type `HitWhere` which will be used to break the block if hit, and the colour the block should be drawn. Finally, `Env` is the "whole value" type used in the Reactive part of the code: it contains all relevant information about the game, including the ball's position and velocity in x and y, the paddle's position, and a list containing all of the bricks. I then defined various values "globally" - the width and height of the canvas, the number of blocks in x and y, the widths and heights of the blocks and paddle, and the size and initial speed of the ball (in y). This I did because various functions later in the program (mostly to do with hit detection and drawing) will make use of these and it makes more sense to me to simply define them as values in their own right. If I were to extend the program (for instance incorporating multiple balls or paddles) I would likely define record-style types for both of those things and include their height, width, and initial velocity within those.

Using these I defined an initial list of `Block`s (called `initBlocks`) and an initial `Env` (called `initEnv`), with the ball in the centre of the canvas "aimed" downwards at the paddle. The blocks' initial values are generated by a list comprehension, with their colour starting out red and getting darker as their `y` value increases. The score value of a block is directly proportional to how far up it is; 100 more points per level up initially, starting at 100. At each level the score value of the blocks is equal to the initial value defined here multiplied by the current level, using the `blocksForLevel` function.

To handle keyboard events I defined a function `keyboardHandle`, of type `Char -> Env -> Env`, code for which can be seen in Figure 2. This function can be thought of as a "router" function, given that all it really does is call the correct function depending on the `gameState` of the `Env` passed in as an argument. It uses the keycode of the keydown event rather than the character to allow both the arrow keys and 'a' and 'd' keys to be used for moving the paddle.

`updateEnv` (Figure 4) is the most complex function defined here, updating the current `Env` every tick. The main part of this is updating the current position of the ball, which requires determining whether or not the ball has hit either a brick or the paddle. If it has, the `HitWhere` type is used to determine where in general the ball has hit and bounce it properly off in an elastic collision. If the ball has hit a brick it will simply bounce off following the same laws of physics that light does: the angle of incidence (the angle the ball comes in from) equalling the angle of reflection (the angle it goes out from). If the ball hits the bat however, its behaviour is slightly different: the ball's angle of reflection depends not on its angle of incidence but where along the paddle it strikes. Striking the paddle in the middle will cause the ball to bounce directly upwards, and striking either side of it will cause it to "reflect" further in that direction until, if it could, striking at the extremity would cause it to "reflect" entirely side-to-side. As well, the ball will only bounce if it hits the top of the bat; while less "realistic", this would be essentially useless to the player and fewer checks per function call will aid performance. I felt this would make the game somewhat more interesting than a direct "reflection", given that such a setup would in fact lead to every game being the same (assuming the same starting conditions). The list of `blocks` is then filtered to remove those that have been hit, and the `scoreVal` of those removed bricks is added to the player's score. I have not included the source code for this function in this report because I feel it is too large to be helpful.

Within `setup` I created a `timer`, setting the interval of ticks to 10ms, as I felt that 100 times per second is a reasonable rate to be updating frames and the game at. I then created an HTML5 Canvas, and a `div` to wrap it in. I then bound `keyboardHandle` to any keypress event within the window, and `updateEnv` to each tick of the timer, and - using `unionWith` and `accumB`, bound all of that to `initEnv`. After that I created a "status line" to be displayed within the wrapping `div` above the canvas onto which the game will be drawn, which will (depending on the state of the game) show the current level and score, or prompts to go to the next level or resume/restart the game.

Finally I used the `on` function to, every tick of the timer, update the canvas to reflect the current `Env`.

Figures 5-9 show the game in the 5 possible values of the `gameState` element of `Env`.

4 Reflections

4.1 What was learned

Over the course of this project I learned a great deal about the Threepenny suite, as well as using functional programming as a medium of video game development. Normally this kind of program would be made in an object-oriented fashion: the paddle, bricks, and ball being separate objects (likely a subclass of a more general object), and the mapping of these concepts to a functional environment has been something of a challenge. I believe that an environment variable is the best way of going about this, and Threepenny's timer allows for this. In terms of ways the project could be made better, a more general data type for the drawn objects would likely serve to make the code cleaner, in particular a `Rect` type that could encapsulate both the paddle and blocks, and would open the project up to using something like `Data.Lens` to modify values. Unfortunately I did not have the time to implement this.

4.2 Project Topic from a real-world perspective

I believe that the project topic was a realistic one from a real-world perspective; Haskell-based video games are becoming more popular of late and this example, while not trivial, was simple enough to demonstrate a proof of concept. As well as this, the use of an external API is something that is done often in the real world of programming, and the heavy use this project makes of the features of the Threepenny Suite again makes it representative of such a project.

On the other hand, the project only really makes use of the Threepenny Suite, and consequently many real-world Haskell functions (failure states provided by the `Maybe` monad being the main one) do not feature. This I feel is a consequence of the nature of the game; failure states are not very varied as a result of the nature of the program, and those that exist are easily accounted for, so the `Maybe` monad is not really required.

A Figures

Figure 1: The defined data types HitWhere, GameState, Block, and Env, and the type synonyms Pos and Dim.

```
type Pos = Double

type Dim = Double

data HitWhere = InX
              | InY
              | InBoth
              | Not
              deriving (Eq, Show)

data GameState = PreStart
               | Going
               | Paused
               | LevelComplete
               | Dead
               deriving (Eq, Show)

data Block = Block {x         :: Pos
                   ,y         :: Pos
                   ,isHit     :: HitWhere
                   ,colour    :: UI.Color
                   ,points  :: Int
                   } deriving Show

data Env = Env {ballX     :: Pos
               ,ballY     :: Pos
               ,balldX    :: Double
               ,balldY    :: Double
               ,paddleX   :: Pos
               ,paddleY   :: Pos
               ,blocks    :: [Block]
               ,gameState :: GameState
               ,level     :: Int
               ,score     :: Int
               } deriving Show
```

Figure 2: Keyboard handling.

```
keyboardHandle :: UI.KeyCode -> Env -> Env
keyboardHandle c e
  = case gameState e of Going ->          if (c == 39 || c == 68)
                                          && (paddleX e <= canvWidth - paddleWidth/2)
                                            then e {paddleX = paddleX e + 10}
                                          else if (c == 37 || c == 65)
                                               && (paddleWidth/2 <= paddleX e)
                                            then e {paddleX = paddleX e - 10}
                                          else if (c == 27 || c == 32)
                                            then e {gameState = Paused}
                                          else e
                        LevelComplete -> if c == 32
                                           then let level' = level e + 1
                                                    dBall' = sqrt ((balldX e)^2 + (balldY e)^2) * 1.1
                                                    blocks' = map (\b -> b {points = points b * level'}) initBlocks
                                                 in initEnv {gameState = PreStart
                                                           ,level = level'
                                                           ,balldY = dBall'
                                                           ,score = score e
                                                           ,blocks = blocks'
                                                           }
                                         else e
                        Dead ->          if c == 32 then initEnv else e
                        otherwise ->     if c == 32 then e {gameState = Going} else e
```

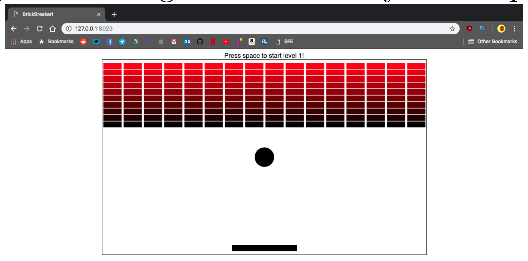
Figure 3: Collision detection, both general and applied to a Block.

```
hitCheck :: Pos -> Pos -> Pos -> Pos -> Double -> Double -> HitWhere
hitCheck cx cy bx by bw bh = let deltaX = cx - (max (bx-bw) (min cx (bx + bw)))
                                 deltaY = cy - (max (by) (min cy (by+bh)))
                              in if (deltaX ^ 2 + deltaY ^ 2) < (ballSize ^ 2)
                                   then if abs deltaX > abs deltaY
                                          then InX
                                        else if abs deltaY > abs deltaX
                                          then InY
                                        else InBoth
                                 else Not

hitBlock :: Pos -> Pos -> Block -> Block
hitBlock cx cy b@(Block {x = bx, y = by}) = let hc = hitCheck cx cy bx by (blockWidth/2) blockHeight
                                             in if hc /= Not then b {isHit = hc}
                                                             else b
```

Figure 4: updateEnv and its helpers: paddleOffset (which calculates how far from the center of the paddle the ball has struck), and splitList (which splits a list into a tuple of values that satisfy the predicate p and those that don't).

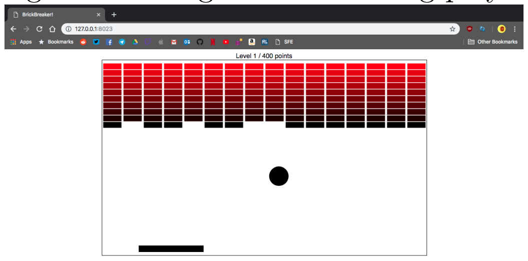
```
paddleOffset :: Pos -> Pos -> Double
paddleOffset paddleX ballX = (2 * (paddleX - ballX))/paddleWidth

splitList :: (a -> Bool) -> [a] -> ([a],[a])
splitList p as = splitList' p as ([],[])

splitList' :: (a -> Bool) -> [a] -> ([a],[a]) -> ([a],[a])
splitList' _ [] t            = t
splitList' p (a:as) (ys, ns) = if p a then splitList' p as (a:ys, ns)
                                      else splitList' p as (ys, a:ns)

updateEnv :: Env -> Env
updateEnv e = case gameState e of Going -> let x = ballX e
                                               y = ballY e
                                               dX = balldX e
                                               dY = balldY e
                                               pX = paddleX e
                                               pY = paddleY e
                                               hit = map (hitBlock x y) (blocks e)
                                               hitB = map isHit hit
                                               (nh, ih) = splitList ((==Not) . isHit) hit
                                               hitP = hitCheck x y pX pY (paddleWidth/2) paddleHeight
                                               ballV = sqrt ((balldX e)^2 + (balldY e)^2)
                                               bdx = if x < ballSize
                                                     || x > canvWidth - ballSize
                                                     || elem InX hitB
                                                     || elem InBoth hitB
                                                     || hitP == InX
                                                     || hitP == InBoth
                                                       then negate dX
                                                     else if hitP == InY
                                                       then negate $ ballV * sin ((pi/2) * paddleOffset pX x)
                                                     else dX
                                               bdy = if y < ballSize
                                                     || elem InY hitB
                                                     || elem InBoth hitB
                                                     || hitP == InBoth
                                                       then negate dY
                                                     else if hitP == InY
                                                       then negate $ abs $ ballV * cos ((pi/2) * paddleOffset pX x)
                                                     else dY
                                               gs = if y > (canvHeight - ballSize)
                                                      then Dead
                                                    else if null nh
                                                      then LevelComplete
                                                    else Going
                                               dS = (sum . map points) ih
                                           in e {balldX = bdx
                                                ,balldY = bdy
                                                ,ballX = x + bdx
                                                ,ballY = y + bdy
                                                ,blocks = nh
                                                ,gameState = gs
                                                ,score = score e + dS
                                                }
                                  otherwise -> e
```

Figure 5: The game on a freshly loaded page.



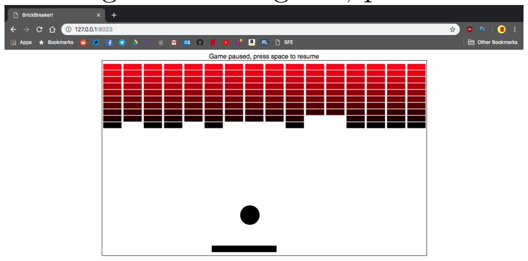
Press space to start level 1!

Figure 6: The game while being played.



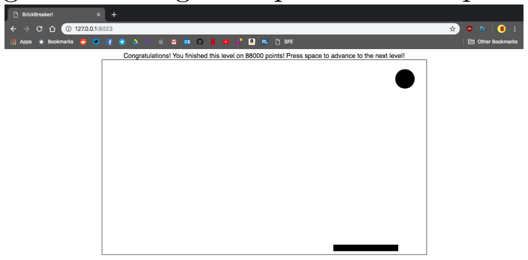
Level 1 / 400 points

Figure 7: The game, paused.



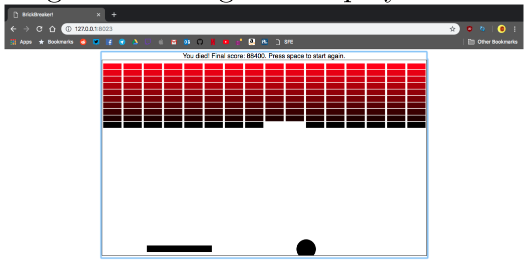
Game paused, press space to resume

Figure 8: The game upon level completion.



Congratulations! You finished this level on 88000 points! Press space to advance to the next level!

Figure 9: The game on player death.



You died! Final score: 88400. Press space to start again.