FIT2102 Assignment 1 Report

Zi Li Tan 30623030

Table of Contents

Overview

This section contains a general overview of the code, the main design principles I tried to keep, and some of the points that I felt needed some justification or explanations.

General Overview (and managing states)

In keeping with a functional programming approach, this code relies mostly on one observable. This observable contains a stream of 'State' objects, which contains the information to draw the game. The State objects is mainly composed of 'Body' objects, which contains the information for each element of the game. Namely, Body objects tell the position, size and velocity of the object. At the subscribe, the observer gets a function called 'update view', which drew and updated the elements on the SVG canvas, limiting the impure function call into the subscribe call.

To create the stream of states, we had a ticker observable to run a tick every ten milliseconds, which was then merged with all the key watching observables. I made some general functions to create the state objects, so then we used scan on the observable to convert all the key inputs (which results in an action class) into states. The reason for using scan instead of map is because we often need knowledge of the previous state.

Going past all the function, types, and constant definitions, the flow of the program is simple:

1. Draw static objects (walls)
2. Starts a 'waiting' observer to wait for spacebar press (to start the game)
3. When spacebar is pressed, stop the 'waiting' observer and call 'run game'
4. 'run game' creates a 'game' observer that produces a stream of game states
5. When game is over, stop the 'game' observer, then go to step 2.

Movements were handled by having functions that took a Body object, read its velocity, then return a new Body object with the velocity added onto the position (and everything else usually kept the same). The most complicated part of movements would be handling the ball's collisions, since this required knowledge of the location of both paddles and the walls. Since walls do not move, we could just check constants to find these. However, for the paddles, they also needed to be passed into the function 'check collisions' in order to avoid having to look up the elements from the SVG canvas.

Design Principles

- Referential transparency: the majority of functions used were pure, returning new 'copies' of objects, and not affecting anything outside the function, with the few exceptions having it clearly written into its comments. Doing this is mainly to make it easier to use functions as we do not need to worry about side effects until we get to 'update view' to draw on the canvas.
- Immutability: all variables used are immutable, this mainly helps us ensure we keep referential transparency.
- Avoiding repetition: I overall tried to make functions generalised in a way that allowed me to reuse them. This included some uses of high order curried functions, which mainly helped in code readability (both looking back at it and while writing it).
- Avoiding overloading functions: I tried to keep all the functions containing only the tasks they had to do, such as having 'make paddle state' focus on doing the check for the action passed in, and handing it off the other functions like 'moving paddle state' to actually handle the movement and collision checking.
- Avoiding excessive use of literals: There is a long list of constants defined at the very start of the code. This is so that when I decide to change certain values, it is much easier to change it throughout the whole code, as well as easier to understand (for example, when I want to fine tune the size of paddles, I just need to change the constants. And when I refer to the width of the walls, I use named constants instead of mysterious numbers).
- Smallest scope possible: For most cases, I tried to keep the everything in the smallest enclosing scope possible, placing constant definitions within the function they are used, and defining most of the impure functions within other impure functions to avoid collisions with variable names and herd most of the impure functions inside of one.

Notable Point 1: Impure function usage

While it is preferable to avoid impure functions for referential transparency, it is obviously unavoidable- otherwise, how would we draw onto the canvas? However, I attempted to limit this by keeping most of the impure functions within the scope of 'run game' and having it all only come into effect at the subscribe call. But, a major pitfall in my method was that I wanted to separate static objects from the rest of the elements, to have a separate function for drawing the walls, since that only needs to be done once. The issue then came when I wanted to implement the ability to restart the game. As walls should only be drawn once (otherwise things will start to lag given enough restarts, as there will be too many elements), I could not put it inside of the 'run game' function, which is called on each restart. So, that meant I had to keep an impure function outside of the 'run game' scope to draw the walls.

Notable Point 2: Key up key check

Something that may look strange is that all the other key processing observers used the 'key check' function, whereas the observer that looked at 'key up' instead of 'key down' had it explicitly defined. The purpose of this is simply because the 'code' property of 'key up' is undefined, but 'key check' compares the ‘code’ property with the expected key. I did not want to pass undefined into the types of expected keys, hence having to rewrite almost the same code as the ‘key check’ function.

As for why I included a key check for ‘key up’, in order to have smooth movement, we needed the key inputs to set a velocity rather than directly changing position (the issue is that the rate which it takes key inputs from a press and hold is quite slow), but then this left us with the problem of not being able to stop the movement. So, including a check for ‘key up’ was to solve this problem.

On Following the Functional Reactive Programming Style

The main things I tried to do to follow the functional reactive programming style is having different generator functions for the player and CPU things, and have the functions that update them take in these generator functions instead of requiring a bunch of other inputs to be passed in. However, to keep some functions pure, this meant I had to place a lot of optional parameters that defaulted to the constant values defined at the start of the code.

The other main use of the reactive programming style is in using a scan function to map each item of the observable into a game state, which is then passed into another function- update view- by subscribe.

Additional Features

Rage (Energy point system)

The additional features added all relies on this system: ‘rage’. The player starts off with four rage points and gains one every time the CPU scores on them. With this, the player can then use multiple abilities. Initially, I was going to make it called ‘energy’, where the player gains one every time they scored, but I felt a comeback system was more fun than a win-more system, so I switched it to this (but some points of the code still says ‘energy’ instead of ‘rage’). The ‘rage’ points are stored within the rectangle body object of the paddles. Whilst that does lead to an unnecessary property being ‘stored’ in the walls (since walls also share the same body type), this actually has no effect because the rectangle bodies of the walls are not actually stored after being drawn.

The main thing to note for this, is that we ended up double storing the ‘rage’ level, once in the rectangle body, once in a separate text body. The reason for this is so that we can keep the same pattern with drawing objects, where each body in state is a drawn object (as just having a text body would make it hard to track energy levels inside of ‘make paddle state’, and just having it stored in the paddle would make it hard to draw a display).

Another thing to note is that I did not implement the system for the CPU paddle to use, but the general framework of it should be there (since the CPU paddle also uses ‘make paddle state’, which is where the checks are done, so, you just need to implement a choosing mechanism- such as a random number each tick- that produces the action classes and a few display aspects).

Ability 1: paddle growth

The first ability for using ‘rage’ is ‘paddle growth’, which increases the height of the player’s paddle until a point is scored. This was the most straight forward to implement, as the only really major change required was to add an extra parameter into the generator functions for creating paddles to change the size. As the rectangle body was already being passed into ‘make paddle state’, we had no issues with getting the energy levels.

Ability 2: teleport

‘Teleport’ is an ability that instantly moves the top of the player’s paddle to the same y-position as the ball. In a lot of ways, it is as simple to implement as paddle growth, so much that I could group the handling of these two together in the scan part of the observable. The main hurdle for this is that ‘make paddle state’ will now need to know the ball’s position in order to move the paddle to it.

The ‘dragging’ effect after a teleport was made by accident, but I liked it, so I kept it in. Basically, what happened was that I mistakenly also passed the ball’s velocity to the paddle in ‘make paddle state’, instead of making it zero. But I found it quite fun to play with that effect, as it allowed slightly more leeway with timing teleports.

Ability 3: shrink ray

‘Shrink ray’ is an ability that makes the CPU’s paddle smaller. This was the most annoying one to implement, since it required a special case to be considered in the scan part of the observable, as it just so different to cause a change on the other paddle. The main issue was that the part that because we are trying to keep functions pure, the functions that generate a body for the paddles are not really distinct from each other apart from their position. So, that means the thing that actually handles distinguishing them apart to put them in ‘State’ is the function inside of scan. Basically, we need to run an effect on the CPU’s paddle, not the player’s paddle, meaning the action needs to be passed into ‘make paddle state’ for the CPU not the player. However, the player still needs to lose one ‘rage’ point, we need to create another action that removes ‘rage’ points. Finally, since the shrinking action is being passed into the ‘make paddle state’ that has the CPU’s paddle- and not the player’s- then the check for whether there is enough ‘rage’ points needed to be done outside of ‘make paddle state’.

While, I could potentially change ‘make paddle state’ to also get the opponents paddle as input for energy checking, I did not want to add even more parameters since it was already feeling quite bloated. So, the idea is sacrificing a little bit- as ‘make paddle state’ is no longer the only thing that checks energy, as the function in scan does it a bit- for being able to run an action on both paddles- drain energy on the player, and shrink on the CPU.