Assignment 1 Functional Reactive Programming

Design Rationale

Function Reactive Programming Style / High Level Overview)

In this assignment, I aimed to thoroughly implement a FRP style of programming throughout my pong application. The first step I took to implement this style into my game was to first create the required Observable streams I would need throughout my game. I started with creating the gameEvent function which was based on the FRP Asteroids observable. This function allows us to create different Observable streams for each unique action a player can make during the game. We then map each of these Observable streams to a class which is used for state management which we will discuss later. The next Observable we created was “game” which is the core of our application. It merges all our above Observables and uses them to create an accumulated state with the selectState function. With the Observables completed I had the “reactive” portion of my goal completed, next I started creating immutable types which defined the structure certain game objects and the state itself could take. By doing this I maintained the absolute immutability of all data within the program adhering to a functional style. Then in order to connect my Observable streams to my defined types without actually mutating or causing side effects, I used the selectState function which is seen in the scan function in our main game observable. This is where all our state management in our program takes place (discussed in depth later). The selectState function allowed us to use the input streams which were mapped to classes such as Move() to transform the current state to another and then return an entirely new state. By following this functional style, we are able to create new states without actually mutating any data or causing any side effects, but still modifying our programs behaviour. Finally, the scan function in our game Observable accumulates our state every time Move or Increment is called. By doing so we are able to iterate through a “game loop” in an entirely FRP style using accumulated state and entirely immutable types/objects.

State Management / High Level Overview)

For this assignment the “state” was the central focal point of the design. All functions in the program use the state in one way or another to calculate or transform entirely new states or objects whilst maintaining the general immutability of the given state. To begin with we defined a “state” type which defined a structure which our state would take, within it held game mechanic objects such as game paddles, balls, scores etc. It is important to note however the state defined was a constant readonly and as such could not be modified in any way as to maintain immutability. Now with our immutable state how are we supposed to actually change the game? Simple, we use our Observables in conjunction with functions that can transform and create entirely new states! As our game observable detects player interaction, it creates the appropriate class eg: Move(3). This class is then passed into our selectState function which then reads the given class and based on it does different transformations. selectState passes the current state of our program, to a multitude of the other functions who then perform transformations on this state to create entirely new objects based on defined types. These newly created objects are ultimately

combined to create an entirely new different transformation of the originally given state. This state is “accumulated” in the scan function in the game loop allowing us to “store” an ongoing state of the game. This state is finally passed to updateView which updates the according elements to match the given state.

This is the overall way the state works in our game. We start with an initialState which is never mutated, we “accumulate” this state based on the players actions via Observable, selectState then calls all other functions to do the appropriate transformations and we create a new state which is returned and viewed.

Summary of Workings)

As we have already thoroughly discussed the design philosophy, state management and general process our code takes, here we can discuss some of the interesting parts of the code and other functions.

The first interesting code we can discuss would be the doCollision function. As you can probably guess this function is responsible for calculating the math behind ball collisions in our game. When a ball collides with a top or bottom barrier we simply negate its x velocity which is quite straight forward, however when the ball collides with a paddle we have to calculate the exact position it struck the paddle, the angle it struck at and more. We first calculate the point of impact so we know where on the paddle the ball hit, this is important because this changes the trajectory of the ball greatly! Next we calculate an angle with some trigonometry to get an appropriate direction the ball should go in range(-1,1 where -1 is 45 degrees up and 1 is 45 degrees down). Lastly we calculate a velocity vector based on all of these which is used to create a new ball object with the newly calculated y position and velocity.

The doCollision function itself is called by another function moveBall which checks whether a collision has occurred or will occur based on the current path. If not it simply returns a new ball with the x,y positions being x + x velocity and y + y velocity.

The rest of the functions aren’t very interesting, they consist mainly of small transformations for score, or bounds checking.

Conclusion)

To conclude, I think this overall assignment went extremely well. I felt that I was able to implement pong in a very functional style which allowed for very easy and proficient management of state. I also liked how by using an FRP approach, the game design itself was much more fluid and linear or pipelined. By simply doing transformations on Observable streams to accumulate a game state, the game design is both dynamic, yet rigid due to state immutability.

Tarun Menon 29739861