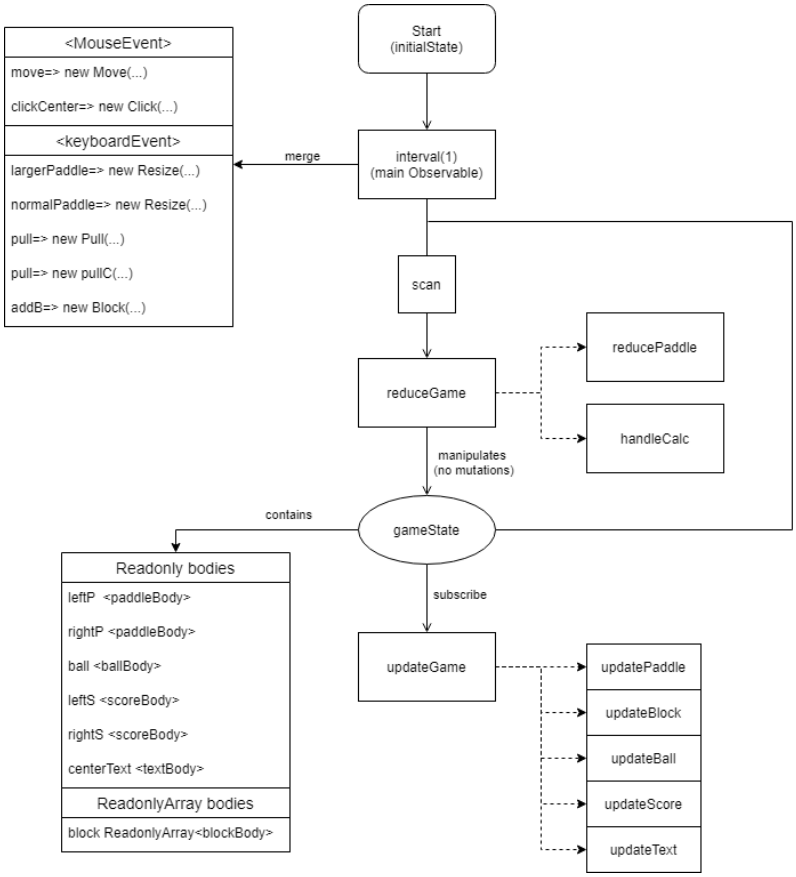
Assignment 1: Report

Diagram created to help visualize the overall design:



Design decisions

The diagram above is a rough representation of the program design and helps visualize the program's processes. There are a few design decisions which I would like to point out:

- Every distinct view object type has their own body model
  While certain models share certain properties such as ID, I believe a single body model would not be appropriate as certain objects do not require certain properties. Example x and y positions are required for the ball move but not required by scores as they are always in a fixed position.
- Helper functions usage
  Certain functions have large computations required such as the reduceGame function, helper functions will perform part of the function and help improve readability.
- Ball speed change when colliding with paddle
  The ball's x-velocity and y-velocity will vary based on how far it is from the centre of the colliding paddle. The nearer to the centre will have a greater x-velocity and a lower y-velocity. Furthermore, the max angle it can bounce back from the centre of paddle will be 80 degrees from its centre.
- Helper function handleCalc
  The helper function handleCalc is responsible for all the calculations required for update. It has access to all readonly members of gameState passed from reduceGame which allows it to check for collisions, update the score, change the speed and position of objects and determines whether the game should end.

Functional programming techniques

The pong game consists of objects which require properties to define their state and require ways to handle asynchronous events such as mouse and keyboard events. Hence, Observable streams were used as it:

1. Has the capability of handling asynchronous events
2. Utilizes transformation instead of mutations.

The following components were used to follow the functional programming style:

Usage of pure functions

The entirety of the program computes solely through pure functions which eliminate side effects. One of the main traits of functional programming is that side effects are prevented and one of the key components to maintain this is to remove all impure functions, and have only pure functions as it has no side effects and also always produces the same results with the same input.

Interfaces for states/bodies/constants

The program component's states and bodies were defined through interfaces which contains read-only members to prevent mutations. The bodies are interfaces which represent the properties of the objects. The gameState is the main interface which stores these bodies to allow the transform functions to have access to the properties of all objects within the program.

Usage of pipe operators

Map and flatMap were used effectively to maintain the functional approach and eliminate loops. Furthermore, the main operators used for the transformation in pipe are the merge, and scan. Merge allows streams to combine while scan is responsible for managing the state of the game. Scan will start with an initial state, and during computations this state will be reduced and transformed which is what allows the game to run.

Reduce for appropriate computing behaviour

During run-time, the events are asynchronous, and the program needs to perform different transformations based on the events current present. To handle this the reducing is required. The pong program's reduce function is named reduceGame() and takes two arguments, the state and event. The function encapsulates all the possible transformation and performs the appropriate state transformation based on the event provided.

Transformation to create new output state for changes

Data structures were updated through the usage of pure functions, but the properties are also not overwritten. The functions will create a new object with all the properties transformed, this is what allows the state to change but maintain the functional programming style as it prevents the need of mutations.

Update to ensure game visually changes

The game has to animate which means the game view has to change after the transformation is done. To do this we need to subscribe the Observable and pass an appropriate function which updates the game. The updateGame method does exactly that and utilizes the setAttribute method to update all objects individually based on the transformed state.

Extension 1 (Power-ups)

The first extension includes power-ups which improves the gameplay and emphasizes my understanding on observables, states and keyboardEvent. When deciding on the power-ups, I was hoping for different power-ups which focuses on improving different gameplay aspects:

Larger paddle and Normal paddle

The first and second power-up are connected as one can only be triggered after the other. The larger paddle power-up is triggered by the x key which makes the player's paddle larger whereas the normal paddle is a complement power-up triggered by the c key which reverts the paddle back to its original size. The main gameplay aspects which these power-ups improve are simplicity and understanding. A larger paddle helps players understand the physics of the ball and eases gameplay whereas the normal paddle allows them to test these new findings for a normal paddle.

Pull

The pull power-up is triggered by the z key, it highlights the player with a black fill and brings the opposing paddle at your current paddle's y-position. This power-up focuses to improve the challenge and opens new strategy possibilities. Fully utilizing the pull function may provide new winning strategies for the players.

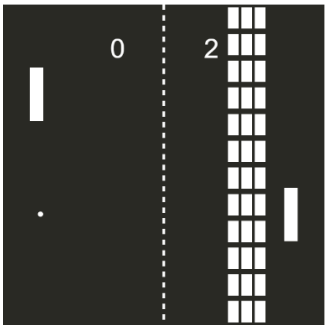
Implementation

The Power-ups implementation fully utilizes the functionalities available from the creation of the base game along with some addition functional programming implementation ideas. An interesting feature would be how the implementations utilize reusable functions along with how it lets the program identify its occurrence.

- Usage of simple classes
  There are simple classes which are recognized by the reduceGame function. The reason it is possible is due to the "instanceof" pattern-match. They are responsible for allowing the reduceGame function to identify what the event has occurred and may hold appropriate value to define the power-up features, such as a read-only member showing the fill to change to.
- Reusable functions
  In the core of the power-ups implementation there are long codes which would reoccur for the creation of each power-ups. By utilizing reusable functions these codes are not repeated and future changes and updates will be easier.
- keyCode
  The power-ups are triggered when specific keys are pressed. keyCode is used to identify the specific key that was pressed as every key has its own distinct keyCode value.

Extension 2 (Incorporate gameplay from classic arcade games):

Selected game: breakout



When the v key is pressed, the game creates three columns of blocks in front of the enemy paddle. This incorporates the breakout game's gameplay where the ball can destroy these blocks but bounces the ball back to the player. Furthermore, new blocks cannot be produced until all the blocks are destroyed.

Implementation

The implementation is very similar to that of Extension 1 where it utilizes features such as:

- keyCode
- Simple classes

But an interesting feature which differs greatly from Extension 1 is the way the blocks are produced while maintaining the functional programming style.

Replacing loops with a functional programming method

In total there are 36 different positions for each block which are as such: [0,0],[0,1]…[2,11]. Instead of hard coding all 36 positions, two functions were used to remove such need.

Transformation through map() and flatMap()

The map and flatMap functions were used along with two constant arrays named rangeX and rangeY containing [0,1,…,11] and [0,1,2]. The idea is that with these two arrays we can produce a new array with all 36 position in a pure way.

The following line of code is part of the function which help produce the new array with all 36 positions:

```
rangeX.flatMap(c=>rangeY.map(r=>[c,r]))
```

The reason why flatMap and map are appropriate are due to the elimination of side effect. It does not mutate the original array and utilizes transformation to create a new array which is used for the creation of the wall of blocks.

Filter function used to remove the blocks from view and state

The filter function is another important feature as it is responsible for both removing the blocks during transformation and all filtering the blocks to remove from the game's view. Like map() the filter function is also a pure function which helps maintain the overall functional programming style and offer the required component to complete Extension 2.