PONGVADER Design Report

Angeline Tanvy – 30930227

Overview

Pongvader is a mixture of Pong and Space Invader. The goal of this game is to reflect the ball using paddle like usual Pong and shoot at enemies before it reaches player's goal like Space Invader. In order to code Pongvader in Functional Reactive Programming way, this whole game is run within 1 subscription. This subscribtion is a merged from several observables, which will be scan and reduce into a single state, so that this state will be a parameter for subscription. Inside the subscription will be, a function that will move all elements/objects of this game according to the given state. This function is called updateView. State here is a type, which is a status of current game: position of paddles and ball, active bullets and enemies, etc. Later, inside the subscription, this state will be passed in into a function called updateView, which will set all elements attributes according to the state continuously, so it seems like all elements are moving smoothly. Therefore, the key idea of the coding is, this gameplay will be conducted by moving objects, which every movement will be through creating state and let updateView to display them. By creating a new state every time we want to update elements' attributes, scan and reduce the states and update them through subscription, it will not mutate any variables and objects, therefore, it remains pure.

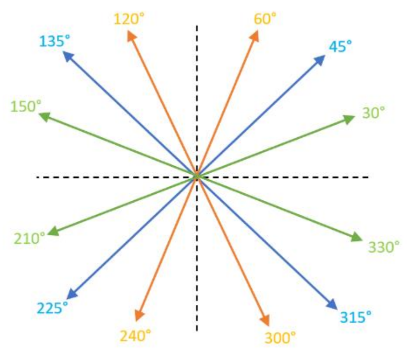
Paddles' movement

There are two paddles in this game, player's paddle at the left-hand side and computer's paddle at the right-hand side.

1. **Player's Paddle** will be controlled by player by moving mouse up and down. An observable is made for this by observing mouse's x and y position. Then a state with updated paddle's y position will be created and scan before subscribing it.
2. **Computer's Paddle** will be automatically move by an observable. If the ball is close to computer's paddle, computer's paddle will start to follow ball's movement. A state with computer's paddle y adding up or subtracting down to ball's y will be created and scan before subscribing it.

Ball's movement

Balls movement is determined by its direction degree where it is heading. To simplify the movement, the graph beside is used. For example, when the ball is hitting a vertical wall with 30°, it will be reflected to 330° (360°-30°). Else if the ball is hitting a horizontal wall with 120°, it will be reflected to 60° (180°-120°). Concluding, if it hits vertical wall, x degree will be reflected to 360°-x°, else if it hits horizontal wall, y degree will be reflected to 180°-y°. For each degree, there will be a function called degreeConverter, which will convert degree to x's rate and y's rate of a degree. Then, these rates will be added to ball's positions of x and y correspondingly.



Enemy and bullet

Enemy and bullet will be declared as Body type. Body is a type for an object which can expired after certain times.

1. **Enemy** will be spawned by observables in 3 different time intervals. For each enemy, the spawn position will be randomized using a class instead of Math.random to ensure our code is pure. Every time an enemy is spawned, it will be appended to state's enemy, so that it can appear on the canvas through updateView. Enemy will be moving to the left until it reach player's goal (player's life will decrease by 1) or it will be shot by bullet before it reached.
2. **Bullet** will be created every time user press space. An observable is created for this which will observe whether space key is pressed or not. If it is pressed, similar to enemy, this bullet will be appended to state's bullet to display it on canvas through updateView. The initial position of this bullet will be the position of player's paddle at the moment, as player's paddle is the one shooting out the bullet. Bullet will start of at the left-hand side and move to right direction.

Shooting

In the gameplay, enemy is supposed to die if a bullet shoots it. Therefore, collision check is needed between enemies and bullets. By using shootingAction function, it will check all bullets whether they collide with any enemies. If they collide, both bullet and enemy that collide will be set to expired = true and enemy will be removed from the canvas and from the state's enemy as it has died.

End game

If either player's life (hearts) or computer's life is finished, a message will be shown indicating the result of the round. If player's heart = 0, then computer will win, and YOU LOSE message will be displayed and vice versa. Moreover, it also allows user to retry by pressing enter key. There will be an observable for this that is watching whether enter is pressed by player or not. Once enter is pressed, the game will restart by returning initialState to the observable, so it can be set by updateView.

Overall

As explained above, state is the key to this Pongvader code. By using this state, there will be no side-effect and minimize bug as we do not mutate any variables and objects. Instead, we keep creating a state, then scan and reduce it before subscribing it to updateView. Inferring from the code, there are many classes and functions for each feature such as player's paddle movement and computer's paddle movement (instead of paddle movement), moveEnemy and moveBullet (instead of moveObject), etc. are mean to keep it well encapsulated, where each class/function will be only dealing with their own concern and limit mutation. Moreover, following FRP style, for-loop is never used in this code. Instead, map, filter, forEach and many others high-order functions are used.