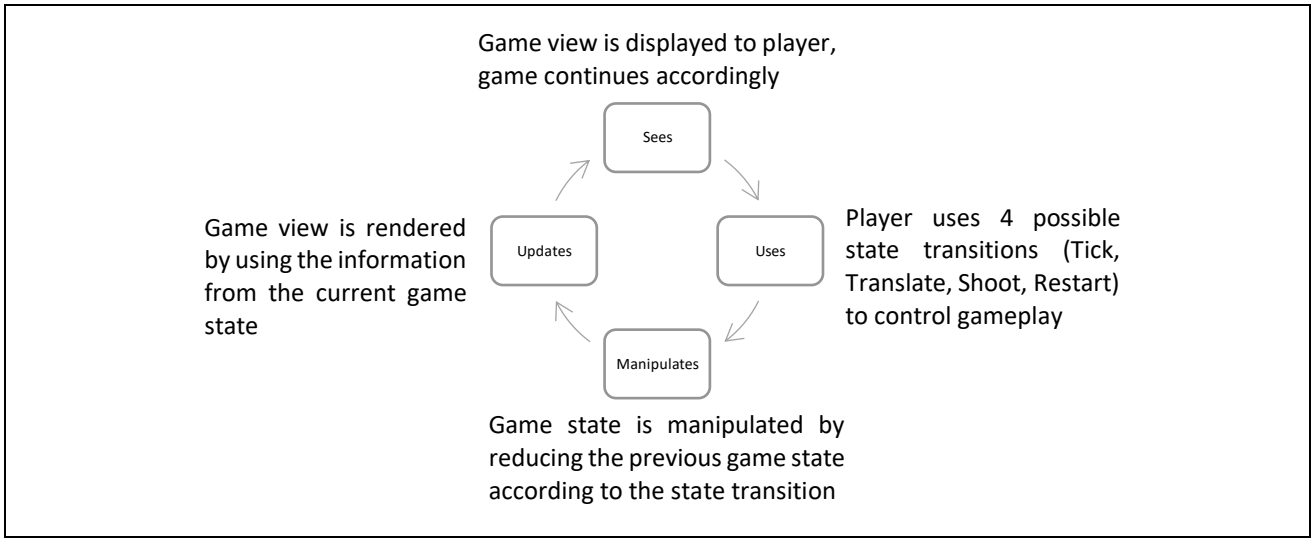
Introduction

Space Invaders game implementation is built with functional reactive programming style using rxjs as observable and game display rendered in HTML using SVG. The game is built based on the Model-View-Controller (MVC) architecture as diagram below shows.




Initial State and Setup

State object is used to store information in its properties regarding to the current state of the game. The state will be used together with observable streams and state reduction to maintain gameplay and build code with functional reactive programming style. To be able to reduce state, first an initial state object must be created before game stream starts by initializing properties with necessary information of starting game point. Several properties in the initial state are object of body type or array of bodies (ship, aliens, bullets, aliens' bullet, shields, and holes) containing properties describing their display, position, and more which are important for game state and display. To aid creation of body objects in both initial state and later state, createShape function is created in form of curried function to compose more detailed, reusable functions following FRP style (createBullet, createAlienBullet, createAliens, createShields, and createHoles) which sets different property values for different bodies. An additional createShip function is created as well which returns a ship body by default without the need to use the curried function as properties of ship is already known.

After defining initial state, observables and classes are created to enable change of states in the game without altering state object. An observable named gameClock is created as timer in game that emits stream of instances of Tick class which contains current time elapsed and a random number generated by nextRandom function with a provided seed. Note that the use of Math.random in RNG class used by nextRandom when no seed is given is impure but the rest of the code in the RNG class is pure given a provided seed, this is part of the setup of the game to provide randomness in bullet spawn and create deterministic gameplay onwards. Other observables relating to user control through keyboard events are created - startLeftTranslation, startRightTranslation, stopLeftTranslation, and stopRightTranslation observables emits stream of instances of Translate class which contains a direction to enable and disable ship's movement when user presses or release arrow right or left, shoot observable emits stream of instances of Shoot class to provide ship's mechanism to shoot when player presses arrow up, and restart observable emits stream of instances of Restart class to restart game play when player presses space bar.

Managing State

Observables created are merged together and piped with scan using reduceState function as accumulator and initial state as initial value. This allows encapsulation of transformations of state in a function and avoid redundancy of observables having their own subscription as all of them will subscribe to the same updateView function. The use of observables and scan maintain purity in our code as the transformation does not alter state but return a new state instead, thus maintaining the FRP style. For each observable triggered, it returns an instance of a class we have created earlier which reduces the previous state inside the reduceState function to return a new state depending on the instance given.

We first evaluate instances created from user's interaction: translate, shoot, and restart. When translate instance is given, a copy of the previous state is manipulated such that ship's position is updated with the given direction. When shoot instance is given, a copy of the previous state is manipulated such that the bullet property is concatenated with a new bullet with position initialized to ship's current position using createBullet function. When restart instance is given, a copy of previous state is manipulated such that the restart property is updated to true, the actual restart itself is later handled in tick function under the condition that player lost the game.

When instance is none of the previous class type, then it is of type Tick and tick function is called to build a new state with properties including mapping bodies other than ship to the next position, filtering expired bullets exiting SVG to be added to exit property, adding alien bullet object shot by a random alien if shoot interval is met, and checking game status which is set to be game over if aliens passed ship. Some interesting points in tick function are the aliens' movement and random shoot mechanism. Both of these use properties that contains the time for the next move and shoot which can be updated during state management instead of creating separate observables like tick. This is due to design decision to separate observables created to deal with user input and gameClock observable to handle all non-user input situations including alien movement and random shoot mechanism. In addition, to make a more complex gameplay during the later state of the game, aliens' speed is increased when aliens' number reached a certain limit, meaning move interval is reduced thus using property to keep track of the next move is easier than using interval observable. Aliens' movement is mapped accordingly only if current time hits the time set for next move which in turn sets a new time for the next move with the required move interval. The same concept is applied to random shoot mechanism whereby an alien's bullet body created with position taken from a random alien using random number argument is concatenated with the rest if shoot time is reached. Other points to note are setting aliens' direction and determining if aliens' passed ship, both of these uses reduce function to find the most bottom alien and leftmost or rightmost alien which position is compared to determine the next direction and game over status. The reduce is needed to keep things pure and find the correct alien every time state is updated and avoid the need to store a specific alien which is useless as aliens keep moving and can be possibly destroyed during game play.

At the end of tick function before returning, the new state object is handed over to handleCollision function, returning another manipulated copy of the state which filters out all bodies that needs to be removed depending on which body interacts with which. Between these body interactions, interaction between shields and bullets is different from others as shield cannot be entirely removed but rather disintegrates per each damage from bullet. To solve this, instead of removing shield, hole bodies are created and placed in the position where shield is damaged by bullet with the condition that no hole has been placed there. If another bullet hit the shield in a damaged spot which already placed a hole body, then the bullet is disregarded and continues to move like normal. Using hole bodies to implement this allow us to reuse existing function for collision interaction. The only cases when both shields and holes are when alien hits shield (assume that alien is stronger and can instantly break shield) and when game restarts or advances to next level. The manipulated copy of state removes collided bodies from each state properties and concatenate them to exit property, add new hole bodies to holes property, update game score according to number of aliens destroyed, and game over status in case ship is destroyed.

Updating View

So far, observables have been piped to reduceState function but the state change is not shown yet to the player. To do this, the merged observables is subscribed to updateView function. For every state reduced from the observables, the updateView function takes information from state object and renders the game play in HTML page. The updateView function is the only impure function in the program as we need to alter attributes of existing elements, add elements, or remove elements in SVG to display the game, therefore all side effects in the game are contained inside this function. All bodies and necessary elements' display are updated here, starting from ship movement, alien movement, bullet movement, shield disintegration, scoreboard, level, generating bullets from ship and aliens, removing collided and expired bodies moving outside of SVG, as well as displaying or hiding a game over text. For bodies which does not exist

by default in the HTML page, updateBodyView function is created and reused to add these bodies as elements to HTML or update display for existing elements.

Continuing Game

At some point, our game will face two situations, either player managed to destroy all aliens on screen or ship is destroyed. Our game does not really end regardless if player loses or shoots every alien on screen as we can restart or proceed to next level. Therefore, observables do not unsubscribe and mechanisms to restart or continue to next level are needed instead. To handle this and aligning with previous decision of separating observables between handling input and non-input situations, we can make use of tick function to perform several checks that immediately returns a new state object without going through the usual flow of the function. At the entry of tick function, check if all aliens are destroyed, player asked to restart with a condition that ship is destroyed, or ship is destroyed and player has not restarted yet. If all aliens are destroyed, game continues to next level by returning a manipulated copy of the initial state with the previous level incremented, score maintained, all objects except ship are removed, and time and timing for aliens' move and random shoot updated. If player pressed space to restart but ship is not destroyed yet, the tick function flows like normal and at the end of the function the restart property is reset back to false. If ship is destroyed, a manipulated copy of the previous state is maintained with update only on time and timing for aliens' move and random shoot to ensure both will not happen up until a point where player choose to restart, in this case function returns a manipulated copy of initial state with update on time and timing for aliens' move and random shoot.

Conclusion

Throughout implementation of space invaders game, code follows functional reactive programming style. Variables are displayed as constant which are immutable and functions are used to transform container into a new container. Several functions inside the code are reusable and composed from other function. State is managed and reduced to a new state using observables and instance of classes. Other objects like bodies and arrays in the game also make use of functions that transform into a new object (e.g., map, filter, reduce, concat, etc.). The functions used are pure with exception for updateView function subscribed by observables which is needed in order to append, remove elements, or alter attributes in SVG canvas to display our game, hence side effects are contained inside this function.