FIT 2102 ASSIGNMENT 1 REPORT

Summary

My implementation of the Guitar Hero FRP game has drawn great inspiration from FRP Asteroids provided in the course notes, as well as the solution from FIT2102 Workshop4 and Applied 3. The game revolves around a Tick based system, where during every tick (a constant), the game updates its State by using a scan operator to accumulate the state progressively. The state of the game is then rendered in the subscription of the observable stream, maintaining a pure and functional paradigm.

State management

The way states are managed in the game is through Action classes, where each action class handles the logic and updates the state accordingly. The implementation of an Action into the class requires an apply() method, used by updateState which accumulates emitted actions to be updated. Each observable in the game that wants to update the State must call an action to invoke the changes. This allows for great separation of concerns when building/maintain the codebase.

Functional reactivity is demonstrated as state is updated in the scan operator, where only the action emitted is applied to the state.

Action

As stated above, Action classes are used to handle state changes in a functional reactive way. This is to prevent state changes from happening outside of the defined action classes so that the code is pure, acting as a controller of the states (in MVC terms). Static methods are implemented in some action classes to break sub-actions (actions that are applied depending on the state) into small chunks, ensuring purity and readability.

Render

Renders the entire game state and handles all the side effects in the game. Creating note SVGs, updating note SVGs, playing audio, removing SVGs are all handled here, as these are side effects that provides feedback to the player on the game canvas. The render function strictly processes the values in the state, in simpler terms, it blindly renders what it is provided. This ensures the purity of the function, as it solely handles view without affecting the state.

Daryl Boon Yang 32836899

Observables

The game loop is implemented using Observables, where Observable streams are created for handling different parts of the game asynchronously, such as emitting notes based off start time, handling asynchronous user key inputs, updating the game state every tick using interval(). All these observables are merged into the main game observable, which handles all the streams asynchronously to update the state based off emissions from the different streams and applies the required actions. The game stream is then subscribed to handle side effects, which is rendering the state changes to the canvas to make sure that functional programming paradigm is followed.

The way to handle notes from the CSV file according to emission time is to make each note an observable, which is flattened into one observable using mergemap. When the note observable is created, a timer is piped to control release the note based off its start time, acting as a fuse. Each note observable is emitted once using the take(1), where the source observable completes after.

The Game

The note in the game is generally managed in 2 criteria, active and expired. Active notes are notes that would have side effects, expired notes are notes that would be removed and cleared.

All notes become expired when they reach the bottom of the column, where they either play a note or is ignored. Separating these note types into different lists allow for rendering in a pure way, where each function processes the exact input of notes that it is provided without altering state by any way.

Additional Requirements

Restart feature has been implemented as an additional requirement. The feature takes advantage of the observable switchmap() in RXjs. Switchmap allows the current observable to be replaced with a new observable, which can be utilized to reset the game stream. The restart subject acts as an observable that encompasses the observables that are affected by timing, mainly note emission. A subject can also act as an observer, when it receives a next() emission, the switch map in its piped observables triggers the switchmap to replace the current observable stream with a new identical stream. The main game stream described above wraps the merging and scanning of the game loop, which is subscribed to. Only the game stream is subscribed to, this makes the whole implementation functional and pure. An action to restart the game is used to reinitialise the state of the new game but carrying over the old high score. This is still

pure as it is handled by triggering an observable, like updating state in every tick, just that it reinitialises the state that is passed into the next game stream.

References

https://tgdwyer.github.io/asteroids/ , Tim Dwyer (n.a.)

Background video retrieved from: https://www.vecteezy.com/video/2259460-colorful-heart-beats-animation, (Pongthanin Thanasantipan)