FIT2102 Programming Paradigms Assignment 1 Report

Design decisions

- FRP as the main paradigm
- Immutable state management
- Game logic is divided into distinct functions, only responsible for their own part
    - Render class to update appearance and movement of notes
    - HitOrMiss class to take user interactions
- Loop driven by observables that emit events, so that it remains efficient and responsive as updates are processed reactively instead of imperatively
- Using "merge" to process multiple events smoothly as observables manage streams in a declarative manner

Code Summary

The game starts in this order, converting the csv contents to array of notes and fill it up in state, then convert notes into circles and tails to fill it up in state. Now the state is filled up with notes, circles and tails. This state will be used as "initialState" as it consists of everything needed for the game loop to work. The loop is driven by "tick$" as it constantly controls the appearance and movement of the notes and managing the game state. Apart from that, user input managed by observables that takes in "ASKL" as input will allow the game to react to user interactions such as playing the right note or playing a random note. The observables will be merged so that they can run together when subscribed without delay. The state will also be managed immutably by the pure "reduceState" function. Scores and multipliers are updated based on user interactions, but missing a note is managed by class "Render".

FRP style and Observables

The code follows FRP style by using pure functions to manage game state transitions and handling asynchronous data streams reactively. The approach ensures that the game is running around reactive streams, allowing it to be easier in handling events and state transitions. For example, time of the game, user interactions and note rendering are all represented by using observables. The values emitted by them are what keep the game running until gameEnd is true. An example of pure function is "reduceState" function where it takes the current state and returns a new one without modifying the old one, it ensures that the state transitions are free from side effects.

State management

The game state is represented by the State type, which consist of time, score, multiplier, consecutive hits, notes, circles, tails, noteCircleTail and gameEnd. A function named "reduceState" will apply actions without much side effects to the state to ensure purity and generate a new state to replace the old one. For example, the "HitOrMiss" class are representing user interactions and will update the state based on timings. The "apply" in the class will take in the current state and returns a new state based on timing and key

pressed. Same goes to “Render” class, but it is updated by intervals instead of user interactions. This ensures that the state transitions are handled immutably.

Observable beyond simple input

- “tick$” manages the game loop by emitting values at a fixed interval
- “action$” merge multiple streams, so that they can run concurrently and are processed together
- “scan” is also used as it is accumulating state changes over time, allowing the game to build up a new state from the old one
- “timer” is used to delay actions in triggerRelease (settimeout is not allowed)

Additional Feature

- User can choose song
    - I did not do much here but just creating buttons in html and connected it to my main.ts, and will take in the button pressed song and fetch the song

Teh Yee Hong
33591938