Assignment 1 Functional Reactive Programming

This game is designed using the Model-View-Controller (MVC) architecture to separate concerns, making the codebase modular and easier to maintain, as seen in Asteroids (Dwyer et al., 2021).

Overview

The `main.ts` file serves as the controller. It initializes observables that emit actions such as game ticks, note generation, and actions that reflect the user input, passing them to another observable that continuously updates the state based on actions. For example, the `keyAction$` function is curried to create observables that emit actions based on key presses, providing a modular approach to handling user inputs. Chaining these observables together create a reactive system, where actions react to user input, state reacts to actions, and the view reacts to state changes.

The model is split between `state.ts` and `circle.ts`. The former processes actions from the controller and updates the game state, while the latter encapsulates logic related to circle actions, keeping the state management modular and extensible. All functions and classes within these files are pure, ensuring that they produce no side effects and depend solely on their inputs. This makes the code more testable and predictable, aligning with the deterministic nature of functional programming. The encapsulation of game logic within the model adheres to the MVC pattern.

The `view.ts` file is responsible for rendering the game. It includes a single function that receives the game state and updates the view accordingly. This function is only invoked by the observer in `main.ts`, ensuring that side effects are confined to subscriptions, thus maintaining the purity of the rest of the code.

Utility functions are organized within `util.ts`, while `types.ts` defines the types used throughout the game.

Features

Reset

The key to making reset work is to contain the `scan` that updates the state in a `switchMap`. This way, every time the restart key is pressed, it switches to a new observable, causing the `scan` to start over with the initial state.

Pause/Resume

The logic of pause was inspired by Lei (2020). The crucial steps in the implementation are: an observable that emits the pause state, `concatMap` instead of `mergeMap` to emit the notes, and a filter to only emit the notes when the pause state is false.

Pause State Observable

I decided to not use a `BehaviorSubject` to store the paused state as seen in the answer, as it felt similar to using a mutable variable. Instead, I used an observable with `shareReplay` to replay the last emitted value to new subscribers. This way, the pause state is not stored in a "mutable variable", but in the observable itself.

concatMap instead of mergeMap

The initial implementation of the note observable used `mergeMap`. This emits all the notes in parallel, just with a delay of its starting time. Problem is, you can't pause the notes if they are already emitted and waiting to be played. To solve this, `concatMap` must be used to emit the notes sequentially. This way, the pause state can be checked before emitting the next note.

Pausing the notes

To stop notes from being emitted, `delayWhen` is used. This operator delays the emission of the notes until the game is resumed.

Star Circles

Fundamentally, the star circles are just the default hit circles with a different look. The difference is that they trigger a star phase when hit. The star phase is a 5 seconds period where the multiplier is increased by 3 and the song is sped up. Information about the star phase is stored in the state with the help of actions. To speed up the song, the note observable

retrieves the current delay from the state. This delay is added to the delay of the notes, effectively changing the speed of the song. This again highlights the importance of using `concatMap` instead of `mergeMap`, as the delay can be changed before the notes are emitted.

NOTE: Star phase can be modified to slow down the song instead by changing the delay to a positive value. It is currently a negative value to speed up the song.

Game Speeds

Uses an observable to react to button presses and calculates the amount of delay between notes to speed up or slow down the song by 1.5x and 0.5x respectively. This delay is stored in the state as the base delay using actions. The note observable retrieves this delay to calculate the new delay between notes.

NOTE: Slow mode can make some songs unplayable because a lot of notes become hold notes and they start overlapping.

References

Tim Dwyer, Nitin Mathew, Garvin, Mariusz Skoneczko, baker-mg, Alvin Zhao, haotongwang, Angus Trau, James Sully, Yusuf Ades, rhys-newbury, Blake, Jackson Wain, Jin Heng Yap, & aiman. (2021). tgdwyer/tgdwyer.github.io: Class of Semester 2, 2021 (v1.0). Zenodo. https://doi.org/10.5281/zenodo.5226211

Lei, J. (2020, October 6). How to make rxjs pause / resume? Stack Overflow. https://stackoverflow.com/a/64226025