FIT2102 PROGRAMMING PARADIGMNS A1

GUITAR HERO

By: Aaron Lam Kong Yew

ID: 33521808

Summary

This Guitar-Hero game reads note data from CSV files, such as RockinRobin.csv and SleepingBeauty.csv, then converts them into Note objects and use the Tone.js library to play corresponding sounds. The game State is managed inside *reduceState* using observables, and side-effect updates will occur inside *updateView*.

Interesting Parts

- An interesting aspect of the code involves the function *triggerNote*, which needs to handle both correct notes and random notes. Correct notes have a concrete duration calculated as (note.end - note.start), while random notes, which can be played at any time, do not have a defined start time. To handle this, I set the start of each random note to -1, effectively using it as a Boolean flag variable.

```
start: -1,
end: duration,
```

  This allows *triggerNote* to automatically determine whether the Note passed to it was a correct or random note, enhancing the function's reusability.

```
const duration = end - (start === -1 ? 0 : start);
```

- Another interesting part is the random note generation. To create random notes with different instruments, pitches, and durations, I initially considered a using an external random number generator (like the RNG Dots Stream from Applied 4). However, I found a non-trivial alternative, utilizing State.time as my "randomizer"

was simpler and more effective. Since State.time only ever increases, it can serve as a random number generator when passed into a hash helper function.
One tradeoff to consider; if a user presses a random key at precisely the same *State.time* each game, the audio generated for each might be the same.

Design Decisions

- One key design decision was to separate tail notes into three different arrays: *activeTails*, *collidedTails*, and *tailExit*. This separation of concerns allows the *tick* method to control the movement and appearance of the SVGs in *activeTails*, *collidedTails*, and *tailExit*, while the *handleKeyPress* method focuses specifically on *collidedTails* to handle the score and audio updates. By organizing the modification of state objects into separate classes, the objects' modifications are managed by *reduceState,* and will only be reflected in the *updateView* section, maintaining clear separation between state management and side effects.

FRP Style and Interesting Usage of Observables

- My code follows the Functional Reactive Programming (FRP) style by merging all 11 input Observables into a single stream. This approach allows the asynchronous flow of values emitted by each Observable to only require one *reduceState* function to handle all of them. The use of Observables is further exemplified in the decision on how to end the game and unsubscribe from the game engine. This is achieved using BehaviorSubject and asObservable, which are essential for managing the game's state and timing. *currentState$* holds and emits the current game state (which is updated by each .next call), while *stateChanges$* allows other parts of the game to react to state updates.

- Additionally, *exitEmpty$* and *gameEnd$* detect specific game conditions, such as when all objects are exhausted or when the game ends. This allows the input stream

to halt when the game ends while still being able to display the "Game Over" SVG as the final emission into the subscription call.

- I inserted an *auditTime* observable too, to prevent audio-stacking (by limiting the refresh rate of *updateView*)

State management and Purity

- The main subscription game engine is the core of the game's design, mimicking FRP Asteroids and combining multiple observables to manage the game's state while handling user interactions. This reactive approach helps separate state management from side effects. State transitions are handled in a 'predictable' manner, making the game more testable and easier to debug.

- The game's state is accumulated over time using the scan operator, which applies the *reduceState* function to the latest event (return { ...s}) , emitting the updated state to the *currentState$* BehaviorSubject. This ensures that the latest state is always available for subscription, which is critical when the game ends. By terminating the subscription using the *takeUntil* operator, which listens to a *finalState$* observable, the game effectively stops when all notes are exhausted.

Observables Beyond Simple Input

- The declarative use of Observables goes beyond simple input handling, extending into the core mechanics of Guitar-Hero's state management and timing. This is widely adopted in modern web development frameworks like Angular and Ext JS. Understanding the use of Observables in this context is essential for managing state, timing, and events in real-time world applications. For my assignment, the use of *BehaviorSubject* to manage each game state update and *auditTime* to throttle state updates are examples of how Observables provide more than just a simple way to handle the four key presses; they are central to controlling the flow of notes and timing of my entire game.

Additional Features

Song Library

Figure 1: Song Library and Start Button

To run Guitar-Hero, I added an input text box and a start button for the user to select an exisitng CSV song (in html and main.ts). Then, I changed the startGame function to directly call main(contents, samples) instead of adding an event listener for mousedown. I have also modified the fetch-response segment by enclosing the existing "fetch" inside a new *fetchSong* function with a string parameter to fetch the CSV file based on user input, as explained below.
Since I had previously created both elements already, I can retrieve the input text box (songNameInput) and start button (loadSongButton) using *document.getElementById*. Finally, the rxjs fromEvent that has been added, this will create an observable for any click Event on loadSongButton. The filter operator ensures that the input is not empty before proceeding to find the inputted song based on its string name. When the button is clicked, the subscribe callback is executed, hence calling *fetchSong* and subsequently calling *startGame*.

Reset Button

Figure 2: Reset Button

I have added a button in the html file and connected it using *document.getElementById("resetButton")* in main.ts. An observable *resetButtonClick$* is created using fromEvent(resetButton, "click"). This observable is then subscribed to, triggering the *resetCanvas* function when the button is clicked. In the *resetCanvas*, it queries all descendant elements of the main svg node that matches the static circles and static line selectors, and then destroys them.

Thus, clicking the resetButton triggers the *resetCanvas* function, removing all mutable svgs, and soft-resetting the game state (without needing to invoke *windows.onload*).

Break that Streak

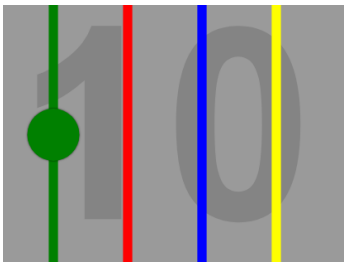

Figure 3a: a 10 Note Streak!

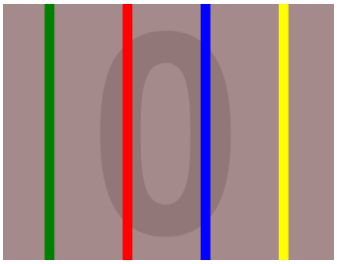

Figure 3b: User breaks the streak…

In main.ts, the streak count is managed by an extra attribute in State, that increments with each correct key press. The streak count is displayed using an SVG text element, which is created and updated in the *updateView* function in view.ts.
The streakText element is updated whenever the state changes, reflecting the current streak count.

For handling a broken streak, the *showFlash* function in state.ts is used to flash the svg background light-red. This function is called whenever a wrong key is pressed (inside *handleKeyPress* method), indicating a visual “break” in the streak.
This *showFlash* function makes the red background visible for a short duration and then hides it. Whenever a wrong key is detected, the showFlash function is called, and the streak count display is reset back to ”0”.

In conclusion, this feature ensures that the streak displayed is easily visible and the red background flashes on wrong key presses, punishing the player for breaking the streak.

Long Note Score Chain

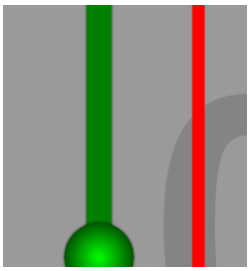

Figure 4: Score increases with duration

Originally, the user must time his keydown at the exact note.start and release at the exact note.end (also not humanly possible) to increase the score by 1. I have upgraded the functionality of long notes to instead implement a continuous “score chain” whereby:

- the score increases repetitively by a miniscule amount (0.05), which accumulates the longer the note is held down, and the "score chain" ends when the key is released prematurely, or when the Tail note has been fully consumed.

To implement this, I used observables to track key press and release events for the relevant keys (e.g., *KeyH, KeyJ, KeyK, KeyL*). For each key, I created observables like *pressKeyH$* and *releaseKeyH$* that accumulates the score chain based on the timing difference of (*releaseKeyH$ - pressKeyH$)* in reduceState.
All in all, this approach appears more logical (and cool) compared to the original base implementation requirements.

Keypress Colors

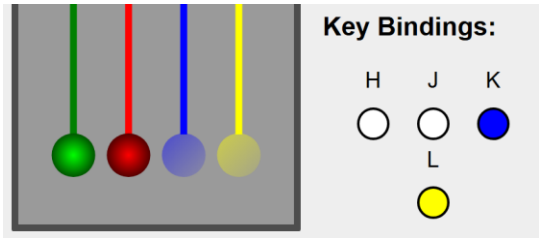

Figure 5: Mapping Keypress colours

Since the game implementation requires us to add visible instructions to play the game (via displaying letters H,J,K,L on the svg canvas), I have decided to also include visually-dynamic interactions with the user, whereby each keypress:

- will show up on the Key Bindings control-rows, and will match the correct fill colour for each one of the four keys pressed

- the static circles at the bottom of the column lines are mapped to the user's four keys, and will change their gradient colour from solid to translucent each time its respective key is held down.

Using rxjs's *fromEvent<KeyboardEvent>,* I am able to filter all keypresses to only accept the 4 user-playable keys and invoke *pressCircle* and *highlightControlCircle* in their subscribe calls for each keydown and keyup Event taken from *showKeys()*. This approach enhances the gameplay and interaction between the user and screen.

The end

Pls enjoy my Git Graph of the longest VsCode assignmemt done to date: