FIT2102-Assignment 1 Report

Enrico Tanvy | 33641668

Overview

The code follows the principles of Functional Reactive Programming (FRP) by implementing observables from the RxJS library to handle asynchronous events. In this Guitar Hero-style game, user inputs are treated as asynchronous events that need to be processed in real-time. The game is structured using the Model-View-Controller (MVC) architecture, which decouples the logic for handling user input (controller), updating the game state (model), and rendering the UI based on that state (view). In this game :

The Controller is represented by the observable streams, which listen for keydown and keyup events from the user. These observables detect specific key presses and map these inputs to corresponding actions, which are then processed by the game

The Model in the game is represented by the State object, which encapsulates the current state of the game. The state is updated through actions produced by the controller. The reduceState function applies each action to the current state to get the new state of the game when an action is emitted. The apply method is coded such that it adheres to functional programming principles. For example, employing techniques such as function composition, higher-order functions, and currying to transform elements within arrays of the state. Instead of mutating the state directly, the apply method of an action creates and returns a new State object. This approach maintains function purity and ensures that state transformations are predictable and manageable.

Observable

In addition to handling user inputs, the notes in the game are also represented by an observable stream. The note$ observable represents a sequence of note (which is parsed from the csv) objects that are emitted at specific times based on their start times and delay.

Notes are designed as observables to treat them as a continuous stream, allowing each note to be handled asynchronously and in sequence. This design ensures that notes are emitted and processed at precise times, and it integrates seamlessly with other asynchronous events, such as user inputs. By managing notes as observables, we simplify the process of updating the game state in response to user actions. This approach aligns with the principles of (FRP), which emphasizes the use of streams and reactive data handling to manage dynamic and asynchronous data flows effectively.

Pseudo-random number generator

To generate random numbers in the game, we use the RNG class from the applied. This class provides a random number generator that maintains purity, In the code, I create a lazy sequence for generating random numbers so that it would produce the same sequence of numbers when initialized with the same seed. The initialState initializes this sequence with a seed value. By using a lazy sequence, random numbers can be generated on demand.

Design Decision

For both Normal Notes and Tail Notes, I decided to use a single type, Note, due to their similar properties and handling within the game. The Note type encompasses information about the instrument and how the note should be displayed in the UI, including properties such as color and position. To differentiate between Normal Notes and Tail Notes, the Note type includes an isTail property.

An array in the state serves as a container for notes to be played, with each element being of the Note type. In the updateView function, the isTail property is checked to determine whether to call triggerAttack or triggerAttackRelease. However, since the array only accepts Note types, notes not displayed in the UI but still played are also forced to be of the Note type. This results in creating Note instances with unnecessary UI-related properties when only instrument information is needed