NAME: ER JUN YET **ID: 33521026**

FIT2102 - Programming Paradigms

Assignment 1: Functional Reactive Programming

Guitar Hero Game Project

Introduction

The Guitar Hero game project is an interactive web application that simulates a rhythm-based music game, leveraging TypeScript, RxJS and Tone.js. The project encompasses a variety of features including note generation, note pressing, music playback and game state management. By utilizing Functional Reactive Programming (FRP) principles, the project aims to provide a robust, maintainable, and scalable solution for real-time game interactions and state management.

The core of the project involves creating an engaging user experience where players interact with notes falling on a screen by pressing them by rhythm and the game responds to these interactions with music playback in real time. The application is designed to handle asynchronous events, manage state immutably and provide a declarative approach to game logic through observables and reactive streams.

Design Decisions

The project is organized into several TypeScript files to maintain a clean separation of concerns:

- `main.ts`: The entry point of the application, where the core game loop and observable subscriptions are managed. Manages the game's core loop and subscribes to observables for real-time updates. This file integrates various components of the game, ensuring that the application state is updated and the view is refreshed accordingly.
- `state.ts`: Defines the application state and related actions. It handles the state transitions and interactions. Contains the game state management logic. It defines the state structure and handles transitions based on user actions and game events. This separation ensures that state management is isolated from other concerns such as rendering or user input.
- `types.ts`: Contains type definitions used throughout the project, ensuring type safety and code clarity. Defines TypeScript interfaces and types that are used across the

application. This ensures type consistency and helps prevent errors related to type mismatches.

- `util.ts`: Provides utility functions that support various operations within the game. Implements helper functions that are used across different parts of the application. These functions support operations like generating random notes or handling mathematical calculations.
- `view.ts`: Responsible for updating the visual representation of the game, including rendering notes and managing user interactions. Handles the rendering of game elements and the update of the user interface. It subscribes to observables to receive updates about the game state and reflects these changes visually.

Reactive Functional Programming

Reactive Functional Programming (FRP) principles are central to the design of the Guitar Hero game project. The application of FRP enables immutability, declarative programming, and effective handling of asynchronous events.

A. **Immutability Management**

The project adheres to immutability principles by ensuring that state changes are handled through immutable updates. This is facilitated by managing state transitions in the `state.ts` file, such as `Tick`, `PressKey` and `PlayNote`, where each state change creates a new state object rather than modifying the existing one using `let`. This avoidance of side effects leads to more robust and reliable code, allowing easier comprehension and justification on the code, as they do not rely on or alter external state, reducing the likelihood of bugs that arise from unexpected changes.

B. **Pure Functions**

The project's FRP style is evident in how it manages pure functions and state. Pure functions are used extensively to ensure that state transformations are predictable and side-effect-free, enhancing code readability and maintainability.

For example, each state transition is captured in a pure function in the `state.ts`, ensuring that the game logic remained consistent and easy to reason about. The state was accumulated over time using the `scan` operator, which allowed for complex state transformations to be expressed declaratively.

Besides, all side effects such as `play` and `display` notes are contained in the `updateView` function that is subscribed by the observables to update the current game state and visual representation.

C. Observables Beyond Simple Inputs

Observables are used beyond simple inputs to manage complex game logic. Observables represent continuous streams of events, such as user inputs or game state updates. This approach allows developers to specify what the application should do in response to certain events, rather than how to do it.

For instance, user interactions like key presses are managed through observables. For instance, instead of using traditional `addEventListener` methods to handle key presses, the code utilizes RxJS to create an observable stream of key events with `fromEvent`, `filter` and `map`.

Moreover, complex game logic allows `merge` combines various observables like `ticks$`, `notes$` and `playKeys$` that are handled declaratively, to create a unified stream of events driving the game.

D. **Event-Driven and Asynchronous Handling**

The game handles real-time events and asynchronous operations using observables. User key press, note press and music playback are managed through reactive streams, allowing the game to respond to events in real time and ensuring that asynchronous operations are handled efficiently.

A key aspect of this approach is the use of ticks, a time-based event that drive the game's update cycle and synchronise gameplay with time-dependent tasks, such as moving notes and game state updates. By integrating `ticks$` observable, the game maintains a smooth and consistent experience, enhancing the overall responsiveness and fluidity of the gaming experience.

Conclusion

The Guitar Hero game project demonstrates a deep understanding of functional reactive programming principles. By adhering to functional programming styles, maintaining code quality and effectively using Observables, the project was able to achieve a high level of functionality while remaining clean and maintainable. The design decisions made throughout the development process were driven by the desire to create a game that is both reactive and easy to understand, showcasing the power and flexibility of FRP in game development.