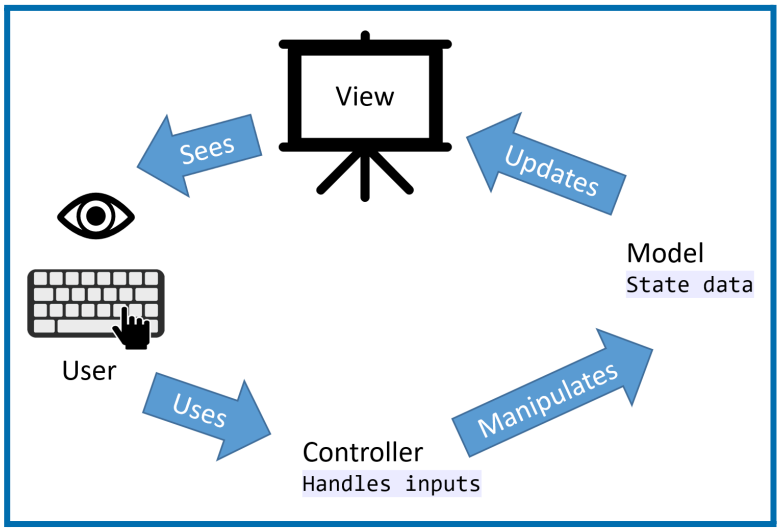
FIT2102 Assignment 1: Report

1. Introduction

The report outlines development of the Guitar Hero game using Functional Reactive Programming (FRP) principles.

2. High-Level Overview

- Relies on Observable/Observer pattern to linearise and capture asynchronous behaviour.



[Souce: FRP Asteroids | https://tgdwyer.github.io/asteroids/]

- Uses an Action interface that sets a contract for each type of Action in game to have an apply() function that updates the current state of the game
  - Abstracts away behaviour while making code open for extension.
- All Action type observables are merged at the end and a single reduceState function, used by scan(), updates the state through modifications caused by Tick and User Interactions.

3. Design Decisions

- **Tick Action Stream:**
  - Responsible for maintaining:
    - Flow of Time

        - Whether a Note is active or not (yet to play)
        - Score Multiplier and updation of current score
        - Updating the properties of each NoteBody object for animation
        - Indication removal of expired notes by moving them in the exit array
    - moveBody() function in Tick class, adopted from FRP Asteroids, updates properties of each NoteBody object stored in noteViews in the current state to animate each note through the flow of time.
    - posY property of each note view is updated by calculating how many pixels a note will travel. Each note starts at 0px and takes 2 secs to travel to the bottom of the guitar (distance: 350 px). Thus, in order for each note to cover 350px in 2 secs, each note moves (350 px / 2000 ms / TICK_RATE_MS)px.
    - Notes with tails are updated in a similar way. Only the end of tail property is updated when current time reaches the end time of a note. Initially, the property is set to 0 px and stays 0 px, and once it reaches the threshold, the property moves down at the same speed explained in the above point.
- **Animation Action Stream:**
    - Responsible for adding new note bodies to animate, received from the animate$ observable, by inserting them into noteViews array in State.
    - The animate$ stream is forced to start with 0s delay, whereas the autoNote$ and userNote$ (used for playing notes), starts after a delay of 2 seconds.
    - Since, I know that each note must take 2 secs to travel down the guitar columns, by the time the note body reaches the end, a corresponding state from userNote$ will be emitted for playing that note. Thus, the 2 sec delay.
    - If a note needs to be played at 0s, animating will not be possible, so delaying the userNote and autoNote streams make both animation and playing of a note possible.
- **User Interaction Streams - Normal & Tail Notes:**
    - Uses accumulation of user notes that have the same start time, and accumulation of key presses that are pressed around the same time, in order to play multiple notes at the same time.
    - Instead of using switchMap, these streams use withLatestFrom(), as with my current design switchMap causes a delay. combineLatest() was previously used, but in order to get missed notes, using partition(), withLatestFrom() was implemented.
    - The idea is similar across all streams for user interactions. From each user note/s, get the latest corresponding key press/es.
    - Then, find the notes that match the keypresses.

- ➢ Filter out the matched notes from the missed notes using partition.
- ➢ Use UserPlayNote & UserPlayTailNote action classes to update the state using the matched notes.
- ➢ Use the missed notes stream to decrement score and reset multiplier using MissedNoteScore action class.
- ➢ To play random notes of random duration, the negated logic is used.
- ➢ For each key press/es, get the latest user note/s, then check if they are within the leniency threshold. If they are not, use the PlayRandNote action class using the RNG stream to get random numbers to play a random note.