Vincent Wesley Liem 34278109

Code Design Decisions

- Code separated into *main.ts*, *state.ts*, and *view.ts* to separate responsibilities, using MVC architecture as shown in Asteroids.
- All impure code is contained in *view.ts* as *view.ts* is responsible for updating the view and playing audio (side effects) since it is only called from inside of a *subscribe()*.
- Our state is completely handled by the *state.ts file*, while the UI is handled by *view.ts* by just reading values from the state. Additionally, *view.ts* does not modify the state in any way, be it by directly modifying the state or creating new "random" circles for misclicks, it is all handled in *state.ts* so as to not break the rules of the MVC architecture.
- All utility functions are contained in *util.ts*, while types and constants are in the *types.ts* file to maintain cleaner code, modularization, and reusability.
- *main.ts* is responsible for initialising and "running" our game, handling streams for user input, game ticks, and processing the song CSV, although all the updating happens in functions in *state.ts*.

Code Follows FRP Style

- The code adheres to the FRP style due to how all of the variables in the game are immutable while also minimising and containing side effects of functions when running the game, with the side effects strictly being contained within a *subscribe()*, ensuring pure code.
- Everything in the game is handled by using functions and higher order functions, making the code "functional".
- Whenever there is a change in our stream, such as when the user hits a key, our code is able to react to this change, making the code reactive.
- This is all possible due to using observables, where almost everything in the game is put within observables, and whenever something new happens, such as a new circle is created or the user hits a key, the stream will react and emit a new value.

State Management & Purity

- As mentioned above, state is managed entirely in *state.ts* which is within the *pipe()*. This means that our state management itself must be pure since it is not inside of a *subscribe()*.
- Although the state is technically "changing" throughout the whole game, we avoid actually changing the state object by returning a new state object with updated properties every time a new value is emitted from our *action$* stream. This allows us to keep the state pure and immutable at all times.

- Not everything is pure however. Impure features such as updating the UI and playing audio are contained in *updateView()* which is only called inside our main *subscribe()*, and since impure code is allowed in *subscribe()* this meets the requirement.

Usage of Observable

- We use observables to constantly update and keep track of things that happen in our game.
- To constantly update our game, we create an observable based on *interval()* which would emit a new value every few milliseconds, with this value being the new state of our game.
- We also have observables that control user input such as mousedown, keydown, and keyup, and every time the user does a user input, it will also emit a new value (state).
- Whenever a new value is emitted by our observables, it updates our game state (backend) by returning a new state object which then gets thrown into the *subscribe()* which will update our frontend (UI and audio).
- Using observables allows for us to remove redundant code that would be needed if we don't use observables, such as having to separate our logic and functions that handle updating UI for different user inputs. Using observables along with the MVC architecture allows us to combine it all into one, making the code cleaner and also easier to debug.

[Additional Feature] Song Selection

- We must first render a UI for the user to see what songs are available, and have them be clickable which would then load the CSV for that song.
- To render the UI, I grabbed a list of song names from my *Constants* and mapped each one to a div, giving them an *id* of the song name. This approach has a problem where if we add a CSV, we must also add the list in the *Constants*, and if we mistype/misspell the name of the song in our *Constants*, it will fail to load the song.
- To fix this, we can attempt to read the name of the files in the */assets* directory and display those instead.
- After rendering the UI, we must assign a mousedown event to each of the divs which will return the *id* of the div, and fetch the CSV based on that *id*. To achieve this, I mapped all my divs to a mousedownEvent stream which returns the *id*, and then in the *subscribe()* we can simply fetch from the CSV and run our game.
- The game must be run in the *subscribe()* due to the fact that our game is impure, as we must update the UI and play audio within our game.