First we create an observable named **tick$** which emits a value every 25 milliseconds.. Inside the pipe, we will merge with different streams that correspond to user input. If the user taps, **fromKey** will filter by keycode and emit one object, if user holds, **fromKeyhold** will filter by keycode then emit multiple objects when user is holding, **switchMap** is then used because we only want the most recently projected observable, so that takeUntil and endWith are only executed once when the user lifts up the key. Each stream emits an object that is defined as a **class** specific to its own use case (e.g. Tap H to emit a **Tap** object). This is done so that the **reduceState**(controller) function will be able to handle different inputs from user by differentiating objects passed and to change our **State**(model) accordingly.

If a **Tick** object is passed to **reduceState**, then we will modify properties of **State** that are subjected to change in every time frame(i.e. every time **tick$** emits). If a **Hold** object is passed to **reduceState** the function will recognise the user is actively holding down a key, and thus will tackle cases where the user is playing a tail note. If the user is holding down a tail note, a new state object will be returned with the **hold** property set to true so that the tail note can be played inside the subscribe function, if the user lifts up halfway, then new state object is returned with **hold** set to false and **score**,**combo** and **multiplier** decreased. Otherwise if the user lifts up at the end of the tail note(tail note successfully played), then the score and combo will be increased with **hold** set to false. **hold** is set to false to stop note from playing. If a **Tap** object is passed to **reduceState** the function will first determine whether there are notes close to the bottom, if the user taps the correct key aligning the note at the correct time, a new **Body** object will be created with **tapped** set to true and then returned together with a new **State**; this is to prevent the **tick** function from “recognising” the note as a missed note. If the key press does not correctly align with a note, “nearby” notes close to bottom will be updated by having **distorted** set to true, so that the note will be played randomly between 0 to 0.5 seconds. Otherwise, if the user taps a key and there are no notes near the bottom then a random note will be played by returning an object with **play_random_note** set to true. In all cases above, every time we update our **State**, we do so by returning a new state object instead of changing a global state variable, by doing so we can ensure pure functional code.

Inside subscribe is responsible for playing the note sound and also to update the view of the canvas. Depending on our **State** and **Body**, a random/distorted/normal note will be played. The **render** function is responsible for updating the view. Inside **render** new circles and tails are created by calling **createSvgElement**, this function creates a new element then calls **setAttribute** on that new element, and then appending the new element to canvas. Movement of circles and tails are updated by continuously calling **createSvgElement** to create a new element with updated properties, then “overlaps” the old element with the new element by appending to canvas before removing the old element from canvas. This is done instead of directly updating the previously existing global element using **setAttribute** to contain side effects as much as possible. Circles that are “expired” will be removed from canvas and scores will be updated by creating and appending new textnode instead of directly changing the **textContent** property of score due to immutability.

```
merge(R$, of(true))
.pipe(delay(1000))
.subscribe(() => {

    startTimer();
})

function startTimer() {
    const source$ = tick$
        .pipe(
            map(elapsed=>new Tick(elapsed)),
            mergeWith(H$,J$,K$,L$, H_hold$, J_hold$, K_hold$, L_hold$),

            takeUntil(P$),
            takeUntil(R$),
            scan(reduceState, initialState),
        )
        .subscribe((s: State) => {
```

An additional feature implemented is the capability to pause and restart the game. Pause is done by simply adding **takeUntil(P$)** and returning the **State** in **reduceState** without modification, so that the observable will complete. Implementing restart requires wrapping the **source$** observable in a separate function so that we can continuously call the function whenever **R$** emits a value. Every time R is pressed, **R$** inside **startTimer** will emit a value that causes the **source$** observable to complete. Inside **reduceState**, the **exit** property of **State** will be updated to include all notes on canvas so that all circles will be removed on **render**. The **R$** stream outside **startTimer** will be delayed by 1 second to allow canvas to “clear up” before restarting the **source$** observable, **of(true)** is merged with **R$** because we want the game to start automatically when refreshing.If **of(true)** is not included game can only start when R is pressed.