Feature: Timers

Students parameterize other parts of their game, so that the experience changes as a new data field, a timer, changes. This track delves deeper into conditionals and abstraction, offering students two possible uses for a timer feature, and a chance to customize their games further while applying those concepts.

Product Outcomes	• Students add a splash screen to their game • Students use a timer to add a collision animation to a simple animation
Materials	• Moving Character Starter File • Watermelon Smash Starter File • Slides are not yet available for this lesson • Printable Lesson Plan (a PDF of this web page)
Prerequisites	• Simple Data Types • Contracts • Simple Inequalities • Piecewise Functions and Conditionals • Compound Inequalities: Solutions & Non-Solutions • Introduction to Data Structures • Structures, Reactors, and Animations • Key Events • Build Your Own Animation • Refactoring

Glossary

constructor :: a function that creates instances of a data structure

Adding a Splash Screen *30 minutes*

Overview

One of the simplest uses for a timer is a *splash screen*: something that is shown for a few seconds when you first run a game. Students implement a splash screen for their games, as a way of getting comfortable with the idea of passing a timer into their Reactor.

Launch

Timers are a key component in many video games: players may need to reach a certain objective before time runs out, or keep from losing a game for longer and longer periods of time to reach a high score. In this feature, we'll cover two possible uses of a timer in your game: adding a "splash screen" at the beginning to give instructions to the player, and adding a short animation when two characters collide.

Of course, a timer is a piece of data in our game that will be changing, meaning it should be added to our data structure. We'll be adding a timer to the completed the Moving Character Starter File file from Unit 5, and you can follow along using the same file, or your own video game project.

Investigate

Add a field called "timer" to the data structure, represented by a Number. Then, go through your code and add that field to each ***constructor*** call in your code. Once complete, run your program to make sure there are no errors, then move on.

The next step is to find (or make!) the image you want displayed as your splash screen when the game begins. We've made a simple image of instructional text overlayed onto the background, and defined it using the name `instructions`.

```
instructions = overlay(text("Press the arrow keys to move!", 50,
"purple"),
  rectangle(640, 480, "solid", "white"))
```

Encourage students to get creative here: In addition to giving instructions to a user, they can also use their splash screen to provide a backstory for their game, include names and images of their characters, and of course, note who created the game!

As of now, our Moving Character file doesn't have a `next-state-tick` function, but if we want our timer to increase or decrease, we'll have to add one. If you already have a `next-state-tick` function with the timer added to the `State` it produces, make it so the timer increases by 1 on every tick. Don't forget to add `on-tick: next-state-tick` to the reactor once you finish! Our `next-state-tick` function looks like this:

```
# next-state-tick :: CharState -> CharState
fun next-state-tick(a-char):
  char(a-char.x, a-char.y, a-char.timer + 1)
end
```

(Note that the position of the character doesn't change in `next-state-tick`. It only changes in response to keypresses, which is already handled in the `next-state-key` function.)

Now we have a timer added to our `CharState` structure, and it increases as the reactor runs. But how do we display our instructions screen *based* on the timer? The `draw-state` function handles how the game looks, so we'll have to add some code to this function. In our starting `CharState`, which we named `middle`, we had the timer start at 0: `middle = char(320, 240, 0)`. Since we made the timer increase by 1 every clock tick, we'll display the `instructions` image as long as the timer is 100 or below.

By default, the computer's clock ticks 28 times each second, so the instructions screen will be up for a bit less than 4 seconds.

We'll need to change `draw-state` so that it becomes a piecewise function. *If* the given `CharState` 's timer is less than or equal to 100, (the very beginning of the game) our splash screen should be displayed. Otherwise, the image of Sam the butterfly should be displayed at the correct position on the background, which is what the current code already does. To change `draw-state`, we add one new `if`: branch, and add the original code to the `else:` clause.

```
# draw-state :: CharState -> Image
fun draw-state(a-char):
  if a-char.timer <= 100:
    instructions
  else:
    put-image(sam, a-char.x, a-char.y, rectangle(640, 480, "solid",
"white"))
  end
end
```

Synthesize

Have students explain what's going on in their own words. (We only want the splash screen to appear at the very start of the game, when the timer is below a certain amount. All other times, we should see the game itself.)

Click "Run", and test out your new feature! You may want to increase or decrease the amount of time your splash screen is displayed, or make changes to the image itself.

Following these steps, students should end up with something similar to this completed Moving Character file.

Timer-Based Animations *45 minutes*

Overview

Students implement a timer-based animation, to create a temporary effect when a collision occurs.

Launch

Another way to use timers in a game is to add a short animation when a collision occurs. In this example, we're going to add a timer to a simple animation, but you could extend this to add an animation to your game when two characters collide, when the player reaches a goal, etc.

Note that if students have already used a timer to add a splash screen to their game, they will not be able to use the same timer field to display a collision animation. Instead, they could implement a collision animation in a different game, or add another, seprate field to their data structure: animation-timer and instruction-timer, for instance.

Open the Watermelon Smash Starter File and click "Run".

Our goal is to make a complete animation of a watermelon getting smashed by a mallet. When the mallet reaches the melon, we should see some sort of pink explosion! We've gotten you started by including a data structure called `SmashState`, which contains the y-coordinate of a mallet and a timer. When the reactor begins, the initial state (defined here as `START`) defines the mallet at 250 and the timer at 0.

To start, let's look at the `draw-state` function.

```
# draw-state :: SmashState -> Image
# draws the image of the watermelon and mallet on the screen.
fun draw-state(a-smash):
  put-image(MALLET, 275, a-smash.mallety,
    put-image(WATERMELON, 200, 75, BACKGROUND))
end
```

Currently, this function uses the images we've defined above (`WATERMELON`, `MALLET`, etc.) and draws the image of the mallet at x-coordinate 275 and the given `SmashState`'s current `mallety`, on top of the image of the watermelon, placed at the coordinates 200, 75 on the background. This code works for most of the animation, before the mallet hits the watermelon, but we want to see a pulpy explosion once it does.

- When should we see a watermelon pulp explosion in this animation? What must be true about the given `SmashState`?
- Which image should we replace to show the explosion animation? The mallet, or the watermelon?

Once the mallet reaches the watermelon (around y-coordinate 140), we should replace the watermelon image with one representing an explosion. Here, we'll use a radial star, whose contract is written below:

```
# radial-star :: Number, Number, Number, String, String -> Image
```

Practice making a few radial stars of different colrs and sizes in the Interactions Area. See if you can determine what each of the Number arguments represent.

Most importantly for our purposes, the second argument to `radial-star` represents the outer size of the star. Since we want this star to represent the exploding watermelon, and grow larger as the animation progresses, we can't use a static number for the size. Instead, we want to use one of our changing values from the `SmashState`.

Which field should we use to represent the size of the growing explosion? `mallety`, or `timer`? Why?

`mallety` only represents the y-coordinate of the falling mallet, whereas the timer can be set and reset based on certain conditions to represent the changing size of the star image.

Investigate

Change the `draw-state` function to make it piecewise: when the mallet's y-coordinate is 140 or less, draw the following image of the radial star `(radial-star(20, a-smash.timer, 25, "solid", "deep-pink"))` at the watermelon's current coordinates. In all other cases, produce the current body of `draw-state`.

The updated `draw-state` function should look similar to:

```
# draw-state :: SmashState -> Image
# draws the image of the watermelon and mallet on the screen. When the
# mallet's y-coordinate reaches 140, draw the explosion
fun draw-state(a-smash):
  if (a-smash.mallety <= 140):
    put-image(radial-star(20, a-smash.timer, 25, "solid", "deep-pink"),
200, 75,
       BACKGROUND)
  else:
    put-image(MALLET, 275, a-smash.mallety,
    put-image(WATERMELON, 200, 75, BACKGROUND))
  end
end
```

Note to students that we haven't done anything to change the value of a-state.timer yet! If the timer's value is still 0, as it begins in our START state, we won't see any star at all, even if our code is correct. We'll work on changing the value of the timer in response to different conditions within the next-state-tick function.

Now take a look at the `next-state-tick` function defined below.

```
# next-state-tick :: SmashState -> SmashState
# Decreases the y-coordinate of the mallet every tick
fun next-state-tick(a-smash):
  smash(a-smash.mallety - 2, a-smash.timer)
end
```

Currently, this function decreases the mallet's y-coordinate to make it fall, and doesn't change the timer. However, if we want the size of our explosion to increase, at some point we'll have to start increasing the timer (since the timer's value also represents the size of our explosion animation).

When should we start increasing the timer, thereby increasing the size of the watermelon's explosion animation?

For help, we can look back at our `draw-state` function. We only wanted to start drawing the explosion (the pink radial star) when `mallety` was less than or equal to 140. So we can check the same condition in `next-state-tick` to tell us when to start increasing the `SmashState`'s timer.

Turn `next-state-tick` into a piecewise function: once `a-smash.mallety` reaches 140 or less, continue decreasing it's y-coordinate, but also *increase* the timer by 2. Use the original body of `next-state-tick` as your `else` clause.

The final version of `next-state-tick` should look similar to:

```
fun next-state-tick(a-smash):
  if (a-smash.mallety <= 140):
    smash(a-smash.mallety - 2, a-smash.timer + 2)
  else: smash(a-smash.mallety - 2, a-smash.timer)
  end
end
```

Run your program, and watch that watermelon get smashed!

For a challenge, change the `draw-state` function so that once the mallet has passed below a certain threshold, an image of the smashed watermelon (we've defined one called `SMASHED`) appears. **Hint:** *Where* within the `draw-state` function will this new condition need to be placed in order for it to work properly?

Closing

We've shown you a couple ways to use timers in your games and animations, but there are many more possibilities. You could extend the timer animation to add a short animation when two characters have collided, or display an ever-increasing timer on the screen to show players how long they have ben playing your game. What other uses for timers can you come up with?