# Making Pong

Students use the Animation Design Worksheet to decompose a 2-player game of Pong, and implement it as a Reactor-based program.

| | |
|---|---|
| **Product Outcomes** | • Students create the first stage of a game of Pong, including a game board and two paddles<br>• Students build interactivity into the game, allowing each paddle to be controlled by keypresses.<br>• Students extend their pongState data structure to include a ball, tracking both its position and direction<br>• Students add collision detection, allowing the ball to bounce when it hits a wall or paddle. |
| **Materials** | • Slides are not yet available for this lesson<br>• Printable Lesson Plan (a PDF of this web page) |
| **Prerequisites** | • Simple Data Types<br>• Contracts<br>• Simple Inequalities<br>• Compound Inequalities: Solutions & Non-Solutions<br>• Piecewise Functions and Conditionals<br>• Introduction to Data Structures<br>• Structures, Reactors, and Animations<br>• Key Events<br>• Refactoring |

# Setting up the Paddles *45 minutes*

## *Overview*

Students decompose a complex problem (implementing Pong) into simpler sub-problems, and implement the paddle portion of the game.

## *Launch*

In Unit 3, you practiced decomposing simple animations into their data structures and functions. Let's consider how a 2-player game of Pong works: There are two "players", each represented by a paddle on either side of the screen. Each paddle can move up and down, as long as they remain on the screen. There is also a ping-pong ball, which moves at any angle and can be on or off the screen. Let's start out by adding the paddles, making sure they can move up and down, and then we'll add the ball later.

Using a blank Animation Design Worksheet, figure out how the paddles behave throughout the game, and decide what Data Structure you'll need to represent those behaviors.

Students should realize that each paddle is simply a y-coordinate, since neither paddle can ever move left or right.

Here is one possible structure that we could use to model the two players:

```
# a PongState has the y-coordinate
# of paddle1 and paddle2
# (no x-coordinate needed, since
# the paddles only go up/down!)
data pongState:
 | pong(
     paddle1Y :: Number,
     paddle2Y :: Number)
end
```

We can imagine a few sample `PongState` instances, in which the paddles are at different locations on the screen. If you haven't already, it would be a good idea to define a sample state for when the game starts, and maybe two other states where the paddles are at other locations.

We'll need to answer some questions, in order to write our `draw-state` function. - What will the paddles look like? - What does the background look like? - How wide is the background? How tall is it? - Define the function draw-state, and try drawing your sample PongState instances to make sure they look the way you expect them to.

The paddles don't move on their own, so right now there's no `next-state-tick` function. However, they DO move when a user hits a key! That means we'll need to define `next-state-key`, and answer a few questions in the process:

## *Investigate*

- What key makes `paddle1Y` increase? Decrease?
- What key makes `paddle2Y` increase? Decrease?
- How much does each paddle move when it goes up or down?
- What happens if some *other* key is pressed?
- Use the Design Recipe to write the code for `next-state-key`

Have students discuss their answers to these questions, before moving on to next-state-key.

At this point, we know how to change the `PongState` in response to a keypress and how to draw that `PongState` as an image. Let's build a `reactor`, which uses a `PongState` instance as the starting state and hooks up these functions to the `on-key` and `to-draw` event handlers.

```
pong-react = reactor:
  init: pongState(200, 200),
  on-key: next-state-key,
  to-draw: draw-state
end
```

When you run this reactor with `interact(pong-react)`, you should see your initial instance drawn on the screen, and the paddle positions should change based on the keys you press! Do all four keys do what you expect them to do? What happens if you hit some *other* key?

Right now, what happens if you keep moving one of the paddles up or down? Will it go off the edge of the screen? We should prevent that!

Take a few minutes and discuss with your partner: what needs to change to stop the paddles from going offscreen? You can use an Animation Design Worksheet if you want to be precise. Once you have a strategy that you feel confident about, take 15 minutes to try it out!

## *Synthesize*

Give the class 2-3 minutes to discuss, and then have different teams share back before they start to implement.

# Adding the Ball *45 minutes*

## *Overview*

Students modify the game State to add a ball, which can move in two dimensions.

## *Launch*

Now that we've got our paddles set up, it's time to start thinking about the ball. What do you notice about the ball? Have students volunteer lots of observations, and write them on the board. Only add the questions below to spark discussion if students run out of ideas:

- When does the ball move? On its own, or only when a key is pressed?
- Does the ball's position change? If so, by how much?
- What do we need, to keep track of the ball's position?
- Does the ball's direction change?
- What do we need, to keep track of the ball's direction?
- When does the ball's direction change?

## *Investigate*

Use an Animation Design Worksheet to add one part of the ball's behavior to your game.

**Did your `PongState` change as a result? Chances are, you needed to add `ballX`**

Number` and `ballY :: Number` fields to your State, to make sure the ball could move in any direction. Did your `draw-state` function need to change? What about `next-state-key`? Did you need to write `next-state-tick`? If so, what did you do?

Some students will hard-code numbers for moving the ball. That's okay! Once they start thinking about changing direction, those numbers will have to become fields in pongState, which change in response to paddle collisions.

Now the game is starting to come together! We've got two paddles moving up and down, and we make sure they stay on the screen. Meanwhile, we have a ball that can move in any direction…but so far the ball doesn't know how to bounce! It's time to plan out what bouncing will look like, and wire it all together.

- How do you know when the ball has hit the top or bottom wall of the screen?

- Write `is-on-wall`, using the Design Recipe to help you.

The goal of this activity is to have students get their collision-detection working, in preparation for the bouncing behavior.

- When a ball is moving up and to the right, what is happening to ballX and ballY?
- When that ball hits a wall, what should happen?
- How does the ball's direction change after it hits a wall?
- After it's changed direction, how does the ball's position change?
- Use the Animation Design Worksheet to plan out the bouncing behavior

**Watch out!**

This activity is pretty sophisticated! You'll want to make sure there are plenty of visual scaffolds for students, or (even better!) have them generate these diagrams themselves.

By now, you may have noticed that the direction of the ball itself needs to change, which means it needs to be added to our `PongState` structure. There are lots of different ways we could represent *direction*: it could be a String (e.g. "north", "southeast", "west", etc), or it could be a pair of Numbers that represent how much the ball is moving in the x- and y-direction from frame to frame.

What other ways could you represent direction? What are the pros and cons of each representation?

Here is one example of a way to represent this, during Numbers to keep track of direction:

```
# a PongState has the y-coordinates
# of paddle1 and paddle2,
# x and y-coordinates of the ball,
# and x and y-coordinates
# representing the direction of the ball
data pongState:
 | pong(
     paddle1Y :: Number,
     paddle2Y :: Number,
     ballX    :: Number,
     ballY    :: Number,
     moveX    :: Number,
     moveY    :: Number)
end
```

When the game begins, we can start out with moveX and moveY being specific numbers that move the ball up and to the right. We can change these later, or even make them randomized every time the game starts!

Before we worry about the paddles, let's start by thinking about the top and bottom walls of the game screen.

- What should happen if the ball hits the top of bottom of the screen?
- How would you detect a collision with the top or bottom wall?
- Make the ball bounce off the top and bottom, using the Animation Design Worksheet and the Design Recipe to help you if you get stuck!

Now let's make some sample instances for when the game begins, when the ball is about to hit a paddle, and then immediately after:

```
# an instance where the paddles are
# at the starting position,
# the ball is in the center (300, 200),
# and moving to the right by 20
# and up by 10 on each tick
pongStateA = pong(200, 200, 300, 200, 20, 10)

# an instance where the ball (x=150, y=280)
# is about to hit the top wall
pongStateB = pong(200, 300, 150, 280, 20, 10)

# an instance after the ball (x=550, y=280)
# hits the top wall
# it's still moving right (20),
# but now it's moving down instead of up (-10)
pongStateC = pong(200, 300, 550, 320, 20, -10)
```

The ball starts out moving up and to the right, but once it hits a wall the direction needs to change. Instead of moving up (adding 10 each tick), it's now moving down (adding -10 each tick) after bouncing off the wall (it's still moving up the screen by 10 each time, so we leave that unchanged). **Note:** Once the ball hits the wall, its y-position needs to change! If the ball stays where it is, it will still be considered to have "hit" the wall on the next tick. This will cause the ball to jitter back and forth, as it constantly hits the same wall over and over.

Change `next-state-tick` so that it generates the next `PongState` using the ball's previous position and the `move` fields. Then, add conditionals to `next-state-tick` so that it will *change the direction* of the ball when it's hit a walll

Let's walk through our new `next-state-tick` function, and make sure we understand it:

```
# next-state-tick :: pongState -> pongState
# move the ball, based on direction fields
fun next-state-tick(w):
  if (is-on-wall(w)):
    pong(
      w.paddle1Y,
      w.paddle2Y,
      # the paddles don't change position
      w.ballX + w.moveX,
      # the ball keeps moving in the same x-direction
      w.ballY + (w.moveY * -1),
      # but it bounces off the wall (move backwards by moveY)
      w.moveX,
      # the x-direction stays the same
      w.moveY * -1)
      # and the y-direction is reversed
  else:
    pong(
      w.paddle1Y,
      w.paddle2Y,
      w.ballX + w.moveX,
      w.ballY + w.moveY,
      w.moveX,
      w.moveY)
  end
end
```

If a collision with an upper or lower wall occurs, we need to do two things. First, we need to move the ball to it's next position, and make sure that new position is far enough away from the paddle so that it won't be considered another collision. Second, we need to flip the y-direction so that the ball is moving in the opposite direction. This is easy to do, by multiplying the `moveY` by −1.

Now it's time to start thinking about a different kind of collision: what happens when the ball hits a paddle?

- How do you know when the ball has hit `paddle1`? `paddle2`?
- Write `hit-paddle1` and `hit-paddle2`, using the Design Recipe to help you.
- Change `next-state-tick` so it checks for a paddle collision in addition to the wall collision.

# Closing *5 minutes*

You've got the beginnings of a very nice Pong game! What are some features you might want to add?

Let students brainstorm ideas. Some suggestions: keeping score, a game-over event, a splash screen…