Lab: Pong

You now know everything you need to know in order to make a working Pong game.

The project will be divided into several phases to make it manageable.

Save this work in a file named `pong.rkt`.

Part 1: Defining our Data

Start by identifying the data in Pong. You can do this on paper or on the computer; it doesn't have to be in DrRacket.

- What data is *static* — meaning, what data doesn't change over time?
- What data is *dynamic* — meaning, what data does change over time?

For this part of the project, your task is to write a complete data definition for your dynamic data (your *world*) and a set of constant definitions for all your static data. It must include:

1. Commented constant definitions for your static data.

2. Data definitions for your dynamic data, including:

    - A **description** of what a `World` is, including comments describing the **types of data** in the fields (showing **how to construct it**). For example, in our running & jumping demo we had something like this:
      ```
      ;; A World is a (make-player Real Real Real Real)
      ```
    - A `define-struct` line for each structure type you define. Our example in class was:
      ```
      (define-struct player [x y dx dy])
      ```
    - A description of **how to interpret** each type of data you define. Our example in class was:
      ```
      ;; where (x,y)  is the player's location, and
      ;;   dx and dy are the player's horizontal
      ;;                   & vertical velocities
      ```
    - **Examples** of each type of data you define. In our example:
      ```
      ;; Player at bottom-left corner, not moving:
      (define player0 (make-player MIN-X GROUND-Y 0 0))
      ;; Player jumping rightwards:
      (define player1 (make-player 150 (- GROUND-Y 20) 4 -6))
      ```
    - **Templates** for functions that process each type of data you define. In our example:
      ```
      (define (func-for-player p)
        (... (player-x p) ...
             (player-y p) ...
             (player-dx p) ...
             (player-dy p) ...))
      ```

Checkpoint 1

When you've completed this much, call me over to check your work. Save a copy, and email it to me and your partner (subject: **Period N: Pong part 1**, replacing **N** with your period number).

(continued on reverse)

Part 2: Model & time

Now that we have defined our **world**, we need to start planning and defining functions. Our data definitions already constitute a big chunk of our *model* (of the game world); what's left is the rules governing how the ball can move. We're not going to get into the *user controls* (via keyboard or mouse) or the *view* (i.e., the graphics) yet — we'll just describe how the world should behave. You may find it interesting to know that we're following a pattern called the *model-view-controller pattern*, and that if you say that phrase to any programmer, they will know what you are talking about.

Informally, what we'll do is create functions to

- set up the start of a volley, with the ball moving in some random direction from the center of the screen
- tell if the ball should bounce
- make it bounce
- tell if the ball has scored a goal
- restart play (with new scores) if the ball has scored a goal

Your task is to design the following functions:

3. **start-play**, which takes two scores, two paddle coordinates, and a velocity (along both axes) for the ball, and produces a **world** with the ball at the center of the screen, moving at the given velocity (hence in a particular direction).
4. **pick-direction-and-start**, which takes two scores and two paddle coordinates, chooses a random direction (i.e., a reasonable random velocity) for the ball, and calls **start-play** to begin play. (Since this function is random, it cannot be tested automatically.)
5. **ball-hit-side?**, `#true` iff the ball has hit the side of the playing field (i.e., top or bottom of the window, since the paddles are at the left and right) and should rebound. Things to consider:
   - The ball has hit a side iff it is in contact with the side and moving towards the side.
   - No more than two pixels' thickness of the ball should ever be off the top or bottom of the screen. (So, at all times during play, what are the min/max values of the ball's *y*-coordinate?)
   - Likewise, no more than two pixels' thickness of the ball should overlap a paddle.
6. **ball-hit-paddle1?** and **ball-hit-paddle2?**, `#true` iff the ball has hit (the face of) the named paddle.
7. **reverse-x** and **reverse-y**, each of which reverses the direction (along the horizontal or vertical axis, respectively) of the ball.
8. **move-ball**, which moves the ball at whatever speed (in px/tick) you have decided. It must account for the ball hitting the side or a paddle. It should *not* deal with scoring goals, since they involve replacing the ball (not moving it); deal with goals in the three functions described below, instead.
9. **goal-for-1?** and **goal-for-2?**, `#true` iff the ball has hit the right or the left side, respectively.
10. **handle-tick**, which will be your tick handler. It will use all of the above functions — some indirectly, via calling others that call them as helpers, and some directly.

Checkpoint 2

Again, call me over to check your work. Save it, and email it to me and your partner (subject: **Period N: Pong part 2**, replacing **N** with your period number).

Part 3: View and Controls

In this part, we finish the game by implementing the view and the controls. Remember to make a copy of your completed "Part 2" code before starting to work on this part.

View

Here are the steps for writing the graphics code. Note that it's OK to define additional helper functions if it would make your program simpler or clearer. (*Also, remember that you should have a function template for **world** data, and that you should copy and paste it to start your new function bodies.*)

11. Design a function **draw-paddle** that takes an *x*-coordinate, a *y*-coordinate, and a background image, and draws a paddle onto the background image at the given coordinates.

12. Design a function **draw-ball** that takes the ball's coordinates (either as separate numbers or as a **posn**, whichever is more convenient depending on your **world** definition) and a background image, and draws the ball onto the given background at the given coordinates.

13. Design a function **draw-scores** that takes the two players' scores (in order) and a background image, and draws the two scores onto the background.

14. Design a function **draw-world** that takes the world and produces an image of it, using the first three functions you just defined.

Run your code, and make sure those functions are working properly, before going on.

Controls

Let's have player 1 (on the left) use the "a" key to move their paddle upwards and the "z" key to move it downwards, and player 2 can use the up- and down-arrow keys.

15. Develop an itemization — i.e., **write the data definition and function template** — to describe the key events that your program can handle. After you list the four command keys I mentioned above, the last variant should be "or some other **KeyEvent**". Your function template should reflect this possibility by including an **else** clause.

16. Develop a function **handle-key** as your key event handler. (Remember the signature that key event handlers have?) It will move one of the paddles iff one of the four command keys has been pressed.

Finally, develop your **main** function to start the world running. If all is well, then you should have a working two-player Pong game at this point. When done, call me over to demo your working program. Save it, and **submit it via DrRacket**.

Bonus tasks

Bonus option 1: In our implementation, the player controls the paddle's position. Redesign it — which includes redesigning your data definitions as well as your functions — so that the player controls the paddle's *velocity* instead. This gives the game a little more challenge, since the movement of the paddles will be somewhat "slippery" as a result.

Bonus option 2: Make a one-player Pong game. To do this, develop a function **move-computer-paddle** that takes the current **world** and produces a new world; it should decide whether to move the paddle up or down in order to intercept the ball. Then, use your **move-computer-paddle** function to develop a tick handler that moves the computer paddle in addition to the other functionality, and develop a modified keyboard event handler that ignores the keystrokes that corresponded to the second player in your original version.