Problem Set 11

OUT: Monday 3/30/2015, 9pm EST
DUE: Monday 4/6/2015, 9pm EST

Last updated: Tue, 31 Mar 2015 12:44:20 -0400

Preliminaries

- Use the full Racket language to complete this assignment. Enable it by adding `#lang Racket` at the top of the file.
- Put all your solution files in a directory named `set11` in your repository.
- Download extras.rkt to this directory (right-click and choose "Save As"; don't copy and paste) and commit it as well.
- Use `begin-for-test` and `rackunit` to define your examples and tests.
- Don't forget to tell us how many hours you spent working on the assignment. This should be a global variable called `TIME-ON-TASK` in each file. For example:

```
(define TIME-ON-TASK 10.5) ; hours
```

- So each solution file must have at least the following at the top:

```
(require "extras.rkt")
(require rackunit)
(define TIME-ON-TASK <number-of-hours-you-spent>)
```

- After you've submitted your solution, use a web browser to go to https://github.ccs.neu.edu/ and check that your repository contains the following files:
  - `set11/draw.rkt`
  - `set11/extras.rkt`
- **Git Commit Requirement**: For this assignment, you must have at least three well-labeled git commits (including the final commit). A well-labeled git commit accurately and succinctly describes the changes since the previous commit. Something like `"commit2"`, or `"home work 3"` is not an acceptable git commit label. Failure to meet this requirement will result in loss of points.

1 Drawing Application

1.1 Additional Preliminaries

Save your solutions for this problem to a file named `draw.rkt`.

Run the following expression (you must have required extras.rkt) to check that your file is properly named and is in the proper directory:

```
(check-location "11" "draw.rkt")
```

Add these additional provides at the top of your file (below the requires), so that we can test your solution:

```
(provide INITIAL-WORLD)

(provide handle-mouse)

(provide Shape<%>)

(provide get-world-shapes)

(provide create-rectangle)

(provide create-circle)
```

1.2 Problem Description

Create an interactive drawing application where the user can create and manipulate shapes on a canvas, including rectangles and circles.

We will talk about "gestures." A gesture is a sequence of individual mouse events that have a meaning when viewed together. In particular, we care about "drag gestures." A drag gesture consists of a mouse `"button-down"` event, followed by zero or more mouse `"drag"` events, followed by a mouse `"button-up"` event. Note the difference between a `"drag"` event (an individual event) and a "drag gesture" (a series of events).

Requirements

The program displays a toolbar with a tool for creating each shape, and a pointer tool.

When the user clicks inside the toolbar, the tool under the mouse becomes the selected one.

Render the tool bar as a vertical series of square black boxes, 20 pixels by 20 pixels, with no space between them. Each tool in the toolbar is rendered with a letter in its middle, as follows, and in this order: "p" for the pointer, "r" for the rectangle, and "c" for the circle. Whichever tool is currently selected should be displayed in "reverse video", with a black background and white foreground. There is always exactly one tool selected. When the program begins, the pointer ("p") is selected.

To be precise about the toolbar: If the user clicks between the top-left point (0, 0) and the point (20, 20), then the first tool will be selected (the pointer). If the user clicks between the point (0, 20) and (20, 40), the second tool will be selected (the rectangle tool). And so on. Don't worry about the boundary lines themselves: we won't test clicks on those exact points. If the user presses the mouse button within the toolbar, keeps it down, and drags the mouse anywhere, all those events (after the `"button-down"`) should be ignored, until the mouse button is released. That is, the tool selection does not change, and we do not modify the canvas in any way, until a user initiates a new mouse-down or drag gesture.

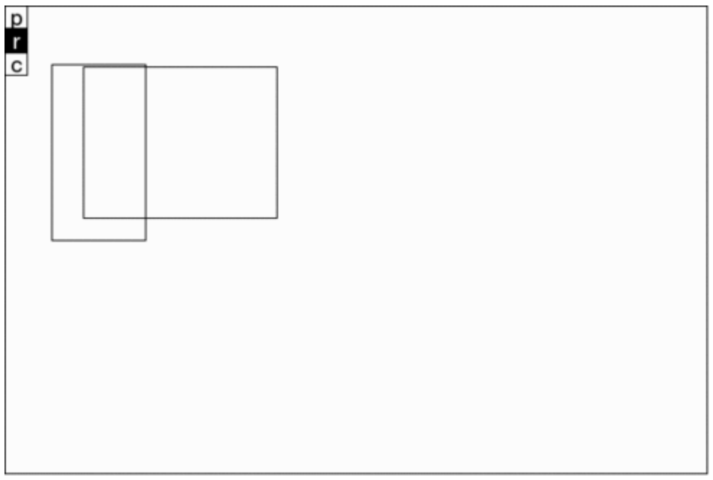
With a shape tool (rectangle or circle) selected, the user can click and drag to define the size and dimensions of the new shape. The way each shape is defined is different:

- For rectangles, the point where the user begins dragging becomes one corner, and the place where the drag completes becomes the opposite corner. The user can do this in any direction, so the corners might be the "northwest and southeast" corners, or the "northeast and southwest"
- For circles, the point where the drag gesture begins becomes the center of the circle and the end of the drag becomes a point on the circumference of the circle. The shape is always a true circle: an ellipse or an oval cannot be created.

While the user is creating a shape, render it as a solid red shape of 50% opacity.

For shapes that have already been created, render them as a black outline of 100% opacity.

Here is an example of what inserting a rectangle might look like.



With the pointer tool selected, there are two ways to manipulate a shape: moving it or resizing it. If the user begins a drag gesture in a "control region" of a shape, the shape will be resized during that gesture. If the user begins a drag gesture within the body of the shape, the shape will be moved during that gesture. **IMPORTANT**: *Any* shape whose body or control region is hit by the beginning of a drag gesture must be resized or moved, accordingly. A single drag gesture can move several shapes and resize several others.

While a shape is being moved, it maintains its size and dimensions but its position is offset according to the mouse movement. That means that if the user starts dragging at a point that is, for example, 10 pixels east and 5 pixels south of a circle's center (but still within its body), and drags 17 pixels eastward, the circle's position should be offset by 17 pixels eastward. We refer to this behavior as "smooth dragging." The circle should NOT move discontinuously, for example by snapping its center to the mouse position. (Contrast this with the behavior for resizing, below.)

The control regions (where a resize gesture begins) are different for each shape:

- For rectangles, the control regions are rectangular areas around the corners. Specifically, a `"button-down"` *within 5 pixels in both the x and y axes* of the true corner of the rectange must be considered a hit to the control region. Any of the four corners can act as the central point for such a region.

- For circles, the control regions are anywhere within 2 pixels of the circumference of the circle. That means if the radius is *r* and the user presses the mouse button down at a distance as little as *r-2* pixels from the circle's center, or up to *r+2* pixels, then the user has hit the control region.

The resizing behavior for each shape is as follows:

- When resizing a rectangle, the originally-selected corner moves while the opposite corner (called the "stationary corner") stays put. The corners can change their geographical relationship to one another while this is happening. For example, the user may begin by dragging the "southeast" corner of a rectange and drag it so far to the north that it becomes a "northeast" corner; the user can continue dragging until it becomes the "northwest" corner. The invariant is that there is always a corner at the spot where the mouse is (during the drag gesture) and there is always a corner at the spot where the stationary corner began.
- When resizing a circle, the center of the circle remains where it is, and the radius of the circle changes so that the mouse is always on the circumference.

When a shape tool is selected, and the user interacts with the canvas, creating a new shape, no other shapes are moved or resized.

IMPORTANT:

- When resizing, the actual mouse position dictates the exact corner position (for rectangles) or circumference point (for circles). That means that if the user first presses the mouse button at a point 3 pixels away from the rectangle's corner, the corner will snap to that position immediately. The position where the mouse button finally goes up becomes the new corner. That's in contrast to the way moves are handled, with "smooth dragging."
- The final effect of any drag gesture must ultimately be determined by the position where the `"button-up"` event occurs, not the final `"drag"` event. The shapes must, of course, be updated during the drag gesture as well.
- Don't assume anything about the sequence of events you will receive, except as follows:
  - A `"drag"` event will always be surrounded by a preceding `"button-down"` event and a following `"button-up"` event.
  - You can assume that the mouse pointer stays within the canvas window at all times. It does not matter what your program does if the user moves the mouse outside the window. (Racket's behavior is different on different platforms, like Mac, Windows and Linux.)

In particular, keep in mind these caveats:

- Do not assume anything about the x and y coordinates of the events you receive. Sequential events can differ by small or large amounts at any time; they may differ by nothing at all.

- A "button-up" event may come in at a different position from the preceding "drag" event, a "drag" event can be at a different position from a preceding "button-down" event, and likewise "move" events can differ from their neighbors. Or successive events can be at the same (x, y) positions.

What to provide:

All shape objects have a "bounding box", which is the smallest rectangular region that encloses the shape. Note that a bounding box is not at all the same thing as a "rectangle object" that is manipulable within this application.

```
; A BoundingBox is a (list Coordinate Coordinate Coordinate Coordinate)
; INTERPRETATION: (list left top right bottom).
; A BoundingBox represents a box whose left x-coordinate is at "left", whose
; top y-coordinate is at "top", whose right x-coordinate is at "right", and whose
; bottom y-coordinate is at "bottom".
```

The functions (and constants) you must provide are as follows:

```
; INITIAL-WORLD : World
; An initial world, with no Shape<%>s.

; handle-mouse : World Coordinate Coordinate MouseEvent -> World
; GIVEN: A World, mouse coordinates, and a MouseEvent
; RETURNS: A new World, like the given one, updated to reflect the action of
;    the mouse event, in the ways specified in the problem set.

; get-world-shapes : World -> ListOf<Shape<%>>
; GIVEN: A World,
; RETURNS: All the Shape<%>s which make up that world, i.e. all those that
;    have been created by the user through using the tools.

; create-circle : Posn Integer -> Shape<%>
; GIVEN: A center point and a radius
; RETURNS: A new Circle% object (implementing Shape<%>) with its center at
;    the given point and radius as given.

; create-rectangle : BoundingBox -> Shape<%>
; GIVEN: A bounding box,

; RETURNS: A new Rectangle% object (implementing Shape<%>) which is bounded
;    by the given BoundingBox.
```

Your shapes file must define and provide this interface:

```
(define Shape<%>
  (interface ()
    ; get-bounds : -> BoundingBox
    get-bounds

    ; handle-mouse : Coordinate Coordinate MouseEvent -> Shape<%>
    handle-mouse))
```

1.3 Video Demo

See the demo video below, which shows the behaviors the program should exhibit. In case there is any disagreement between the video and the requirements above, the requirements take precedence.

Drawing Demo Video