

# Working with images and scenes

## CS 5010 Program Design Paradigms “Bootcamp”



# Lesson Introduction

- Racket has a rich library for working with images and scenes.
- We will use this library extensively in this course.
- In this lesson, we will explore a few things from the library.
- Note: this lesson is mostly review from Lesson 0.4

# Learning Objectives

- At the end of this lesson, the student should be able to:
  - Create simple images and scenes
  - Combine simple images into more complex images and scenes
  - Determine the properties of an image
  - Test images and their properties

# Images are Scalar Data

- In Racket, images are scalar data
- Racket has:
  - Functions for creating images
  - Functions for combining images
  - Functions for finding properties of images
- In general, we build complex images by starting with simple images and then combining them using functions.

# Loading the image library

To load the image library, include the line

**(require 2htdp/image)**

in your program.

# Functions for Creating Images (1)

**bitmap : String -> Image**

GIVEN: the name of a file containing an image in .png or .jpg format

RETURNS: the same image as a Racket value.

# Functions for Creating Images (2)

**rectangle :**

**Width Height Mode Color -> Image**

GIVEN: a width and height (in pixels), a mode (either the string “solid” or the string “outline”), and a color

RETURNS: an image of that rectangle.

*[See the Help Desk for information on the representation of colors in Racket].*

# Functions for Creating Images (3)

**circle : Radius Mode Color -> Image**

Like rectangle, but takes a radius instead of a width and height.

There are lots of other functions for creating shapes, like **ellipse**, **triangle**, **star**, etc.



# Functions for Creating Images (4)

**text**

**: String Fontsize Color -> Image**

RETURNS: an image of the given text at the given font size and color.

# Combining Images

- The image library contains many functions for combining images into larger images.
- These functions generally align the images on their centers. This is usually what you want. If you really want to align images on their edges, there are functions in the library to do that, too. See the help desk, as usual.
- Let's look at the two most commonly-used image combining-functions: **beside** and **above**. Here's an example:

# beside and above


```
> (above (ellipse 70 20 "solid" "gray")  
          (ellipse 50 20 "solid" "darkgray")  
          (ellipse 30 20 "solid" "dimgray")  
          (ellipse 10 20 "solid" "black"))
```



```
> (beside (ellipse 70 20 "solid" "gray")  
          (ellipse 50 20 "solid" "darkgray")  
          (ellipse 30 20 "solid" "dimgray")  
          (ellipse 10 20 "solid" "black"))
```

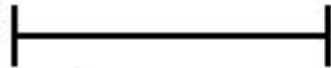


# Slightly more complicated images

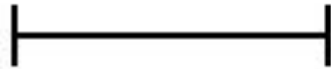
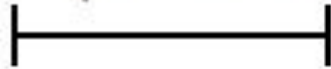
```
> (define vtick (rectangle 2 20 "solid" "black"))  
> (define hline (rectangle 100 2 "solid" "black"))  
> (define ruler1 (beside vtick hline vtick))  
> ruler1  
  
>
```

# Slightly more complicated images

```
> (define vtick (rectangle 2 20 "solid" "black"))  
> (define hline (rectangle 100 2 "solid" "black"))  
> (define ruler1 (beside vtick hline vtick))  
> ruler1
```



```
> (define vspace (rectangle 0 50 "solid" "black"))  
> (above ruler1 vspace ruler1 vspace ruler1)
```



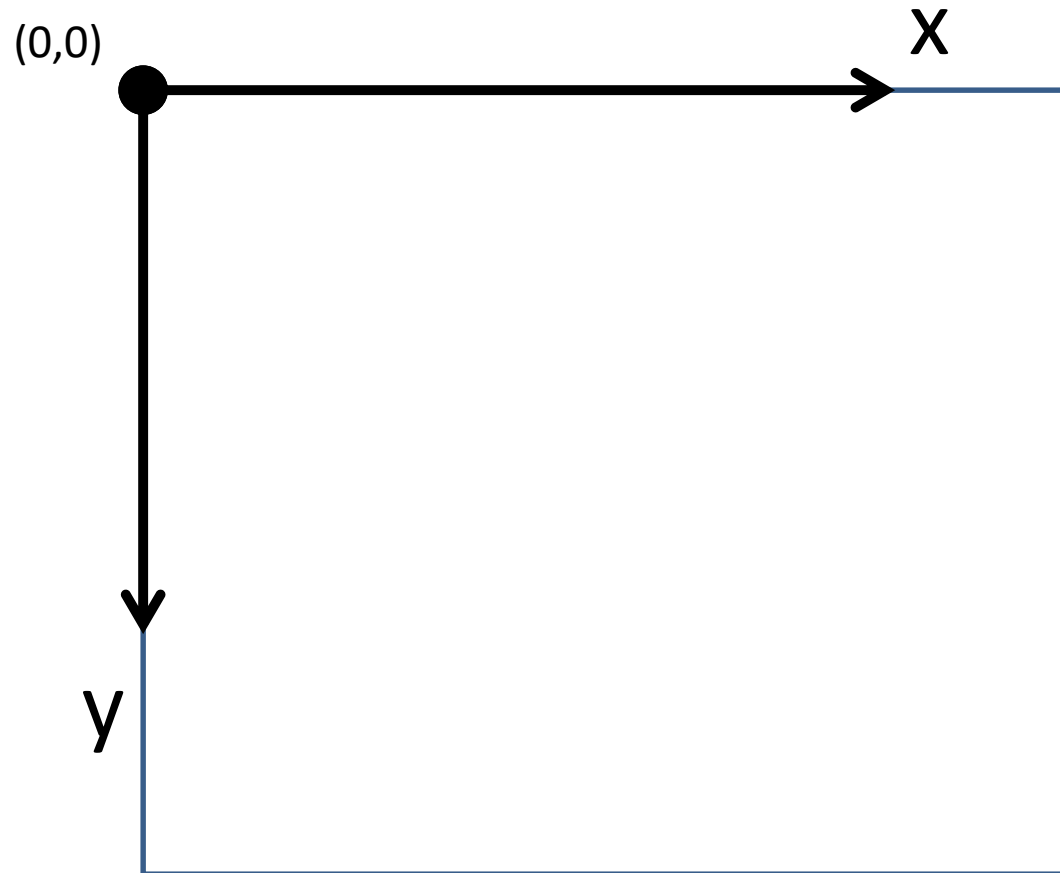
```
>
```

The rectangle has width 0, so it's invisible 😊

# Scenes

- A *scene* is an image that has a coordinate system.
- In a scene, the origin (0,0) is in the top left corner. The x-coordinate increases as we move to the right. The y-coordinate increases as we move down. These are sometimes called “computer-graphics coordinates”
- We use a scene when we need to combine images by placing them at specific locations.

# Scene Coordinates



# Creating Scenes

- **(empty-scene width height)**
  - returns an empty scene with the given dimensions.
- **(place-image img x y s)**
  - returns a scene just like **s**, except that image **img** is placed with its center at position **(x,y)** in **s**
  - resulting image is cropped to the dimensions of **s**.



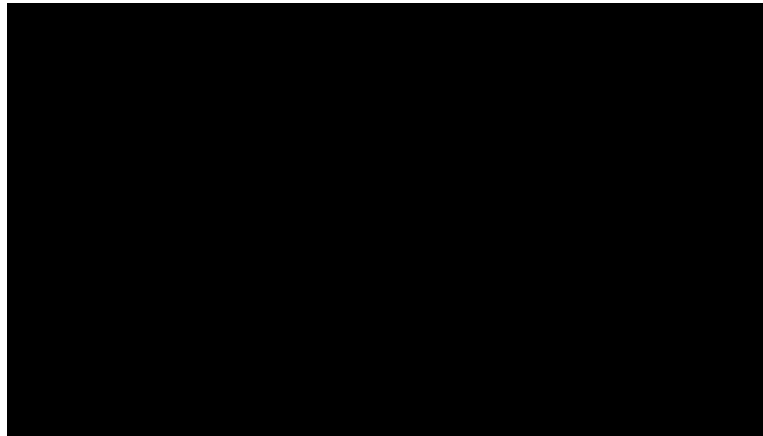
# **scene+line**

- **(scene+line s x1 y1 x2 y2 color)**
  - returns a scene just like the original **s**, but with a line drawn from **(x1,y1)** to **(x2,y2)** in the given color.
  - the resulting scene is cropped to the dimensions of **s**.

# Creating Scenes by Combining Simpler Functions

- Create scenes with images in them by combining them with functions.
- Start with an empty-scene, then paint images and lines on the scene by using **place-image** and **scene+line**.
- This is all functional: painting an image on a scene doesn't change the scene—it produces a new scene.

# Video Demonstration



# Measuring Images

- Racket also provides functions for determining image properties. Here the most important ones:

- **image-width** : **Image** -> **NonNegInt**
- **image-height** : **Image** -> **NonNegInt**
- **image?** : **Any** -> **Boolean**

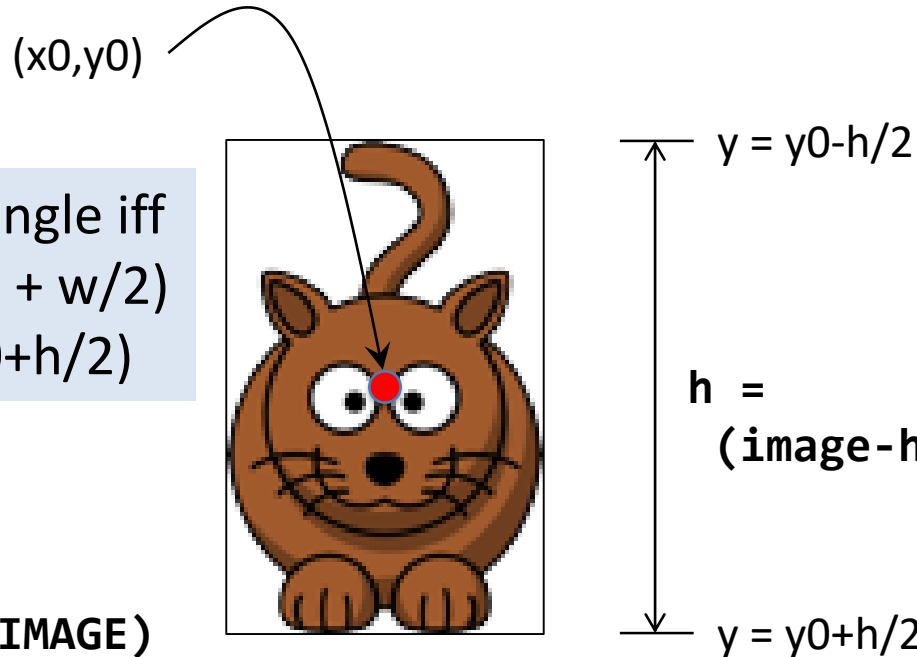
In pixels

# Bounding Box

- The *bounding box* of an image is the smallest rectangle that completely encloses the image.
- Its width will be the image-width of the image, and its height will be the image-height of the image.
- It is easy to determine whether an arbitrary point is inside the bounding box— let's look at an example.

# Bounding Box Example

$(x,y)$  is inside the rectangle iff  
 $(x_0 - w/2) \leq x \leq (x_0 + w/2)$   
and  $(y_0 - h/2) \leq y \leq (y_0 + h/2)$



$w = (\text{image-width CAT-IMAGE})$

# Images and the Design Recipe: Examples

- In your examples, describe the image in words.
- Consider a function that takes an image and doubles it.
- In your examples you might write:

```
(define red-circle1  
  (circle 20 "solid" "red"))
```

```
;; (double-image red-circle1)
```

```
;; = two red circles, side-by-side
```

# Images and the Design Recipe: Tests

## (1)

- First, construct the correct image. Do NOT use the function you are testing to construct the image.
- In your tests, you might write

```
(define two-red-circles  
  (beside red-circle1 red-circle1))
```

- Be sure to check it visually to see that it's correct
    - Alas, this step is not automatable.
- Then you can use **check-equal?** on the resulting images:

```
(check-equal?  
  (double-image red-circle1)  
  two-red-circles)
```



# Images and the Design Recipe: Tests (2)

- **check-equal?** is fairly clever, but not perfect.
- Which of the images below are visually equal?
- See which of them **check-equal?** accepts as equal.

```
(define vspace1 (rectangle 0 50 "solid" "black"))  
(define vspace2 (rectangle 0 50 "solid" "white"))  
(define vspace3 (rectangle 0 50 "solid" "red"))  
(define vspace4 (rectangle 0 50 "outline" "black"))  
(define vspace5 (rectangle 0 50 "outline" "white"))
```

# Summary

- Images are ordinary scalar values
- Create and combine them using functions
- Scenes are a kind of image
  - create with **empty-scene**
  - build with **place-image**
- **2htdp/image** has lots of functions for doing all this.
  - Go look at the help docs

# Next Steps

- If you have questions or comments about this lesson, post them on the discussion board.
- Go on to the next lesson.