# Lecture 8: Drawing Pretty Pictures

# Drawing

- DrScheme provides a simple drawing interface
- Load it now

# posn

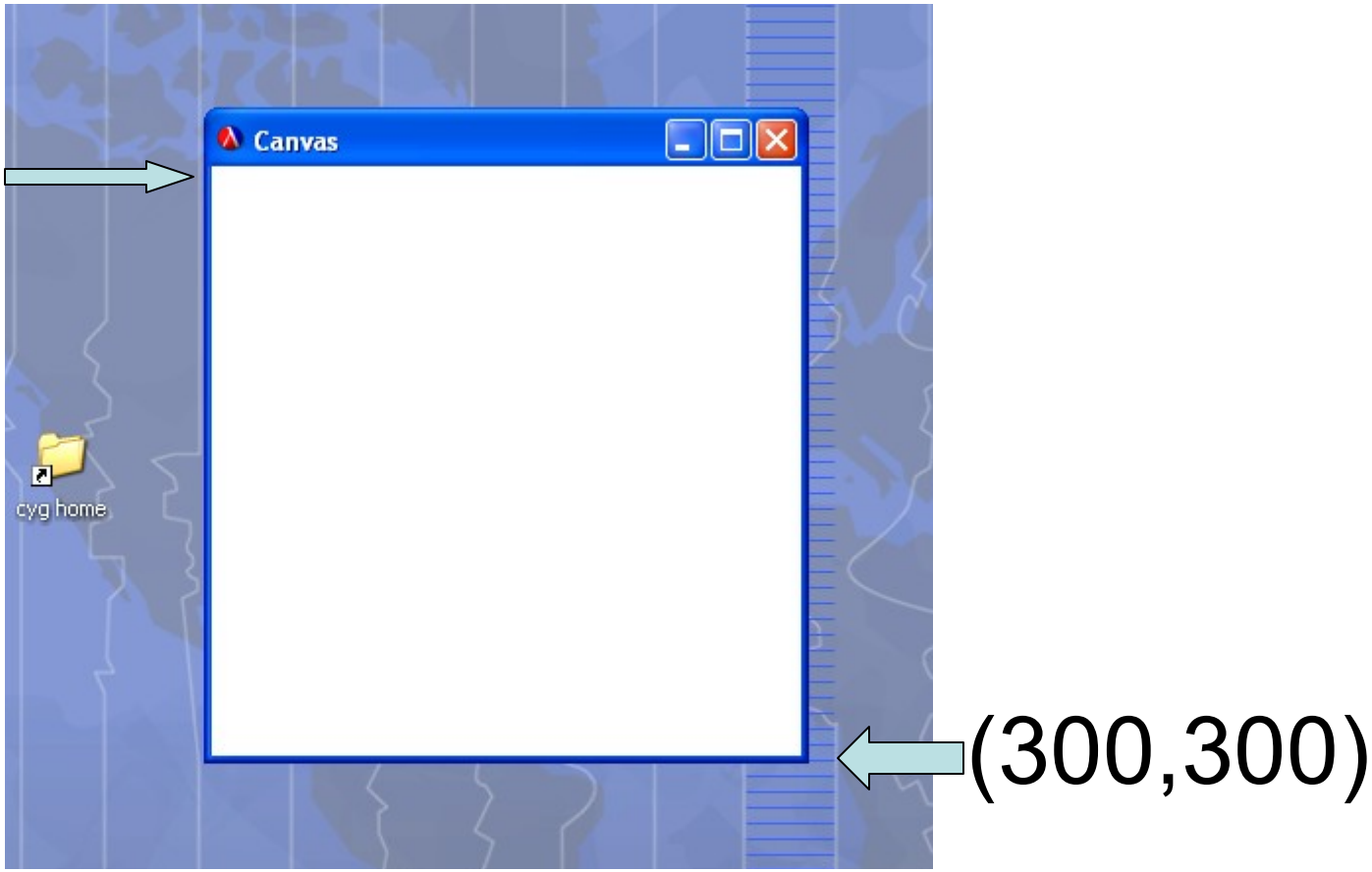
- The basic data abstraction for this package is a **posn** (short for position)
- These functions:
  - **posn?**
  - **(make-posn** *x y***)**
  - **posn-x**
  - **posn-y**

# Drawing

- To open a window for drawing you evaluate **(start** *win_size_x win_size_y***)**
  - *win_size* refers to the size of the drawing window in pixels
- Evaluate (start 300 300)

# Canvas

- The origin is at the upper left corner



# Drawing something

- Now you can draw in this canvas.
- To see the drawing primitives, look in the helpdesk
- We'll only use one primitive today
  - **(draw-solid-line** *p1 p2 color)*
  - *p1* and *p2* are posn's
  - color is denoted as 'red , 'green, 'blue, etc.

# Exercise

- Draw some lines on the canvas

# Exercise

- Write a procedure **draw-poly-line** that takes a list of posn's and a color and draws a line segments connecting the points
  - (**draw-poly-line** *pts color*)

- You'll need to use **begin**

# Abstractions

- We want to abstract the drawing process
- First we'll make abstractions for different kinds of shapes
- Use lists of points to define each of these shapes
  - Square
  - Triangle
  - pentagon?

# Operations

- We also want to define operations on these point lists
  - Translate
  - Scale
  - Rotate

# translate-pt

- Define a procedure (**translate-pt** *pt x y*) that returns a new posn where *x* and *y* have been added to the *x* and *y* coordinates of *pt*

# scale-pt

- Define a procedure (**scale-pt** *pt x-scale y-scale*) that returns a new posn where the *x* and *y* coordinates of *pt* have been multiplied by *x-scale* and *y-scale*

# transform-pts

- Write a procedure **(transform-pts** *transform pts***)** where
  - *transform* is a procedure that takes one posn as an argument and returns a “transformed” posn
  - *pts* is a list of points
- *Hint:* This is easy, but to make it *really* easy, look up **map** in the helpdesk

# Exercise

- Draw squares at different locations around the screen

# Adding sugar

- We want to make our graphics package useful
- Write (**make-translate** *x y*) that evaluates to a procedure that translates a posn by *x* and *y*
- Do the same for **make-scale**

# One more operation

- Define (**rotate-pt** *pt ang*) that takes an angle in radians and rotates the point around the origin.
- Equations
  - x’ = x*cos(ang) – y*sin(ang)
  - y’= y*cos(ang) + x*sin(ang)
- Also define (**make-rotation**)

# Making this work

- If I wanted to draw a square of width 10 that has been rotated by pi/4 at the center of the canvas, what are the operations that I will have to do?
- We'll need to *compose* multiple operations

# compose

- What does this code do?

```
(define (compose f g)
  (lambda (x) (f (g x))))
```

- How would I express the drawing operations using compose?

# Order of Operation

- Because the rotation is around the *origin*, we have to be careful about the order
- Try changing the order to see what happens

# The power of higher-order procedures

- We can define procedures for doing lots of convenient operations
  - (define (make-rotate-and-translate ang)
    (lambda (pt x y)
    (translate-pt (rotate-pt pt ang)
    x y)))))

# Exercise

- Write (**draw-star** *n inc*) that draws *n* squares each successively rotated by *inc* radians