6.090
Building Programming Experience

Lecture 8

1/23/2007

Outline

- Drawing Pretty Pictures

Drawing

- DrScheme provides a simple drawing interface
- Load it now:

```
(require (lib "draw.ss" "htdp"))
```

Drawing

- To open a window for drawing evaluate
  (**start** *size_x size_y*)
  Size is the number of pixels
- Evaluate (start 300 300)

Canvas

- Origin is at the upper left corner
- (0,0) is top left
- (300,300) is bottom right

Abstractions

- Basic data abstraction is a **posn**
- These functions are built in:
  - **posn?**
  - **(make-posn *x y*)**
  - **(posn-x *posn*)**
  - **(posn-y *posn*)**

Drawing primitive

- Only one drawing primitive we'll use today:
- (**draw-solid-line** *p1 p2 color***)**
- *p1* and *p2* are posn's
- *color* is a symbol: 'red 'green 'blue etc
- Exercise: Draw some lines on canvas

Exercise

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

You may want to use **begin**

Abstractions

- Lets add more abstractions
- First: make abstractions for different shapes
- Use lists of points to define each of these shapes:
  - Square
  - Triangle
  - Pentagon?

Operations

- Also define operations on these lists of points
  - Translate
  - Scale
  - Rotate

Translate-pt

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

Translate-pt

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

```
(define (translate-pt pt x y)
  (make-posn (+ (posn-x pt) x)
             (+ (posn-y pt) y)))
```

scale-pt

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

scale-pt

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

```
(define (scale-pt pt x y)
  (make-posn (* (posn-x pt) x)
             (* (posn-y pt) y)))
```

Transform-pts

- Write a procedure **(transform-pts** *transform pts*) where
  - *Transform* is a procedure that takes one posn as argument and returns a new posn
  - *Pts* is a list of points

Transform-pts

- Write a procedure **(transform-pts** *transform pts*) where
  - *Transform* is a procedure that takes one posn as argument and returns a new posn
  - *Pts* is a list of points

```
(define (transform-pts transform pts)
  (map transform pts))
```

Exercise

- Download graph.scm, evaluate it, and then draw squares at different locations around the canvas

Combining translations

- Write **(make-translate** *x y*) that returns a procedure that takes in a posn and translates it by *x* and *y*
- Type of make-translate:
  - Number,number → (posn → posn)
- Define a similar **make-scale**

Translation generators

```
(define (make-translate x y)
  (lambda (pt) (translate-pt pt x y)))

(define (make-scale x y)
  (lambda (pt) (scale-pt pt x y)))
```

One more operation

- (**rotate-pt** *pt angle*) takes a point and an angle in radians and rotates the point around the origin
- Equations:
  - x' = x*cos(ang) - y*sin(ang)
  - x' = y*cos(ang) + x*sin(ang)
- Also, **(make-rotation** *ang*)

Rotations

```
(define (rotate-point pt ang)
  (make-posn
   (-
    (* (posn-x pt) (cos ang))
    (* (posn-y pt) (sin ang)))
   (+
    (* (posn-y pt) (cos ang))
    (* (posn-x pt) (sin ang)))))

(define (make-rotation ang)
  (lambda (pt) (rotate-point pt ang)))
```

Putting these together

- Draw a square of width 10 at the center of the canvas that has been rotated by pi/4.
- Need to *compose* multiple operations

Putting these together

- Draw a square of width 10 at the center of the canvas that has been rotated by pi/4.
- Need to *compose* multiple operations

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

Look at pts1:

```
(define pts1
  (transform-pts
   (compose (make-translate 150 150)
            (compose (make-scale 10 10)
                     (make-rotation (/ pi 4))))
   square))
```

(draw-poly-line pts1 'red)

Order of operations

- Rotation is around the *origin* -- have to be careful about the order
- Try changing the order and see what happens

Power of higer order procedures

- Define procedures for doing lots of convenient operations

```
(define (make-translate-rotate-scale x y s)
  (lambda (ang)
    (compose (make-translate x y )
             (compose (make-scale s s)
                      (make-rotation ang) ))))
(define trans1
 (make-translate-rotate-scale 150
 150 100))
```

Putting it together

- **(draw-star** *n inc***)**
- Draws *n* squares each successively rotated by *inc* radians

Draw-star

```
(define (draw-star n inc)
  (if (>= n 0)
      (begin
        (draw-poly-line
         (transform-pts
          (trans1 (* n inc))
          square) 'red)
        (draw-star (- n 1) inc))))
```

Try it out

```
(let ((n 10))
 (draw-star (- n 1) (/ pi n)))
```

Last Exercise

- Draw something cool!