Project Ideas

Summary: You've already learned about a number of basic techniques that you can use in making images, including region-based color blends, string art, and even the basic GIMP tools. In this reading (and the corresponding lab), we will consider a few more techniques that you might find useful.

Color Trees, Revisited

In the reading on trees, we encountered an interesting way to use trees: We can represent an image with a *color tree*, a tree whose leaves are all colors. We can then render the tree systematically. In that first rendering algorithm, when we encountered a pair, we split the image (or subimage) horizontally and then rendered the left half of the tree in the left half of the image, and the right half of the tree in the right half of the image.

It can be more interesting (and more valuable) to render a color tree by alternately decomposing the image horizontally and vertically. The following alternate definition does just that. (You may find it valuable to study it for a moment.)

```
;;; Procedures:
;;;   image-render-color-tree!
;;; Parameters:
;;;   image, an image
;;;   ctree, a tree of colors
;;;   left, a real number
;;;   top, a real number
;;;   width, a real number
;;;   height, a real number
;;; Purpose:
;;;   Render the tree into the portion of the image bounded at
;;;   the left by left, at the top by top, and with the specified
;;;   width and height.
;;; Produces:
;;;   [Nothing; Called for the side effect.]
;;; Preconditions:
;;;   image is a valid image
;;;   ctree is a valid color tree
;;;   0 <= left < (image-width image)
;;;   0 <= top < (image-height image)
;;;   width < 0
;;;   height < 0
;;;   0 <= (+ left width) < (image-width image)
;;;   0 <= (+ top height) < (image-height image)
;;; Postconditions:
;;;   The tree has now been rendered.
(define image-render-color-tree!
  (lambda (image ctree left top width height)
    (let kernel ((ctree ctree)
                 (hsplit? #t)
                 (left left)
```

```
            (top top)
            (width width)
            (height height))
      (cond
        ; If it's too small, just stop.
        ((or (< width 1) (< height 1)))
        ; If it's a pair, and we're to split horizontally, do so
        ((and (pair? ctree) hsplit?)
         (kernel (car ctree) (not hsplit?)
                 left top
                 (/ width 2) height)
         (kernel (cdr ctree) (not hsplit?)
                 (+ left (/ width 2)) top
                 (/ width 2) height))
        ; If it's a pair, and we're to split vertically, do so
        ((pair? ctree) ; NOT hsplit?
         (kernel (car ctree) (not hsplit?)
                 left top
                 width (/ height 2))
         (kernel (cdr ctree) (not hsplit?)
                 left (+ top (/ height 2))
                 width (/ height 2)))
        ; Otherwise, it's just a single color, so render in the
        ; given space.
        (else
         (image-select-rectangle! image REPLACE
                                  left top (round width) (round height))
         (context-set-fgcolor! ctree)
         (image-fill-selection! image)
         (context-update-displays!)
         (image-select-nothing! image))))))
```

How can color trees be useful for the project? Well, if you're willing to put up with fairly "unpredictable" images, you can write a procedure that systematically builds a color tree from an integer. But how do we build an *interesting* color tree?

One strategy is to look at some characteristic of the number (e.g., the remainder after dividing by ten) and use that number to decide how one might split the number (e.g., if the remainder is 1 or 2, we build a color tree whose left subtree is built from some fraction of the number and whose right subtree is built from some other fraction of the number. If the remainder is 3 or 4, we build a color tree whose left subtree is a different fraction, and so on and so forth.

Here's an implementation of that technique.

```
;;; Procedure:
;;;   number->color-tree
;;; Parameters:
;;;   n, an exact integer
;;; Purpose:
;;;   Create an "interesting" color tree whose content depends only
;;;   on n.  Ideally, different values of n would give different
;;;   color trees.
;;; Produces:
;;;   ctree, a color tree
;;; Preconditions:
```

```
;;;   n >= 0
;;; Postconditions:
;;;   If x != y, then (number->color-tree x) is unlikely to be the same as
;;;   (number->color-tree y)
(define number->color-tree
  ; These are the colors used to build the tree
  (let* ((colors (vector (rgb-new 255 0 0) (rgb-new 204 0 0) (rgb-new 153 0 0)
                         (rgb-new 255 0 102) (rgb-new 204 0 102) (rgb-new 153 0 102)
                         (rgb-new 255 0 204) (rgb-new 204 0 204) (rgb-new 153 0 204)))
         (num-colors (vector-length colors)))
    (lambda (n)
      (let ((action (remainder n 10)))
        (cond
          ; For small numbers, we simply grab them from the color tree.
          ((< n num-colors)
           (vector-ref colors n))
          ((< action 2)
           (cons (number->color-tree (inexact->exact (round (* 0.4 n))))
                 (number->color-tree (inexact->exact (round (* 0.6 n))))))
          ((< action 4)
           (cons (number->color-tree (inexact->exact (round (* 0.6 n))))
                 (number->color-tree (inexact->exact (round (* 0.4 n))))))
          ((< action 6)
           (cons (number->color-tree (inexact->exact (round (* 0.25 n))))
                 (number->color-tree (inexact->exact (round (* 0.75 n))))))
          ((< action 8)
           (cons (number->color-tree (inexact->exact (round (* 0.75 n))))
                 (number->color-tree (inexact->exact (round (* 0.25 n))))))
          ((< action 9)
           (cons (vector-ref colors (remainder n num-colors))
                 (number->color-tree (inexact->exact (round (* 0.5 n))))))
          (else
           (cons (number->color-tree (inexact->exact (round (* 0.5 n))))
                 (vector-ref colors (remainder n num-colors)))))))))
```

Here's one way to incorporate that procedure into a series of images.

```
(define series-1
  (lambda (n width height)
    (let* ((ctree (number->color-tree n))
           (canvas (image-new width height)))
      (image-show canvas)
      (image-render-color-tree! canvas ctree 0 0 width height)
      canvas)))
```

You'll have a chance to explore these ideas a bit more in the lab.

Fractals

Color trees provide one way of decomposing and then rebuilding an image. But, at least as created using the techniques above, they create unpredictable images. As we explore in this section, one can more systematically build an image by decomposing it into smaller pieces.

A mathematically interesting kind of drawings is what is commonly called a *fractal*. Fractals are self-similar drawings. That is, each portion of the drawing bears some resemblance to the larger drawing. We normally draw fractals by breaking the larger drawing into equal portions and drawing each portion using the same technique.

For example, to draw an NxM rectangle, we might draw nine (N/3)x(M/3) rectangles in a grid. Similarly, to draw each of those nine rectangles, we might draw nine (N/9)x(M/9) rectangles, and so on and so forth. When do we stop? When we've recursed enough or when the rectangles are small enough.

We might express this technique in code as follows.

```
;;; Procedure:
;;;   fractal-rectangle!
;;; Parameters:
;;;   image, an image
;;;   color, the desired color of the rectangle
;;;   left, the left edge of the rectangle
;;;   top, the top edge of the rectangle
;;;   right, the right edge of the rectangle
;;;   bottom, the bottom edge of the rectangle
;;;   level, the level of recursion
;;; Purpose:
;;;   Draw a "fractal" version of the rectangle by
;;;   breaking the rectangle up into subrectangles,
;;;   and recursively drawing some of those rectangles
;;;   (potentially in different colors).  When does
;;;   recursion stop?  When the level of recursion is 0.
;;; Produces:
;;;   [Nothing; Called for the side effect]
(define fractal-rectangle!
  (lambda (image color left top right bottom level)
    (cond
      ; Base case: We're at a level in which we just draw the rectangle.
      ((= level 0)
       (context-set-fgcolor! color)
       (image-select-rectangle! image REPLACE
                                left top
                                (- right left)
                                (- bottom top))
       (image-fill-selection! image)
       (image-select-nothing! image)
       (context-update-displays!))
      ; Recursive case: Break the rectangle into a few parts and recurse
      ; on each.
      (else
       (let* ((midcol1 (round (+ left (/ (- right left) 3))))
              (midcol2 (round (- right (/ (- right left) 3))))
              (midrow1 (round (+ top (/ (- bottom top) 3))))
              (midrow2 (round (- bottom (/ (- bottom top) 3)))))
         ; First row of squares
         (fractal-rectangle! image
                             color
                             left top
                             midcol1 midrow1
                             (- level 1))
```

```
    (fractal-rectangle! image
                        color
                        midcol1 top
                        midcol2 midrow1
                        (- level 1))
    (fractal-rectangle! image
                        color
                        midcol2 top
                        right midrow1
                        (- level 1))
    ; Second row of squares
    (fractal-rectangle! image
                        color
                        left midrow1
                        midcol1 midrow2
                        (- level 1))
    (fractal-rectangle! image
                        color
                        midcol1 midrow1
                        midcol2 midrow2
                        (- level 1))
    (fractal-rectangle! image
                        color
                        midcol2 midrow1
                        right midrow2
                        (- level 1))
    ; Third row of squares
    (fractal-rectangle! image
                        color
                        left midrow2
                        midcol1 bottom
                        (- level 1))
    (fractal-rectangle! image
                        color
                        midcol1 midrow2
                        midcol2 bottom
                        (- level 1))
    (fractal-rectangle! image
                        color
                        midcol2 midrow2
                        right bottom
                        (- level 1))
    )))))
```

Why would we use such a technique, since all we end up with is the same rectangle? Well, things get a bit interesting when you make subtle changes (other than just the level of recursion) at each recursive call. Most typically, you might draw the different subrectangles in modified versions of the original color. Once you do that, you can also think about changing whether or not you use an even grid, or even whether or not you draw each sub-rectangle.

The technique sounds simple, but it can produce some very interesting images. More generally, fractals also let us provide interesting simulations of many natural objects, such as trees, mountains, and coastlines, that have some of the same self-similarity. We'll start our explorations with these rectangles.

What can we do? Well, instead of drawing all sub-rectangles the same way, we can vary them a bit. For example, we might draw some of the sub-rectangles in the complement of the color, in a lighter version of the color, or in a darker version of the color. We might break up the rectangle into a less even grid. We might use different levels of recursion for different sub-rectangles. We will explore these kinds of options in the corresponding lab.

Samuel A. Rebelsky, rebelsky@grinnell.edu