

## 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, spirographs, and even using 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 deep recursion, 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 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.)

```
(define new-render-color-tree!
  (lambda (ctree image 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 0.5) (<= height 0.5)))
        ((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))
        ((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)))
        (else
         (image-select-rectangle! image selection-replace
                                   left top (round width) (round height))
```

```

(context-set-fgcolor! ctree)
(image-fill! 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))))))
         (else
          (error "Unknown action: ~a" action)))))))

```

```

        (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)
      (new-render-color-tree! ctree canvas 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, we can more systematically build an image by decomposing it into smaller pieces.

One of the more mathematically interesting kinds 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  $N \times M$  rectangle, we might draw nine  $(N/3) \times (M/3)$  rectangles in a grid. Similarly, to draw each of those nine rectangles, we might draw nine  $(N/9) \times (M/9)$  rectangles, and so on and so forth. When do we stop? When we've recursed enough of 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
;;;   width, the width of the rectangle
;;;   height, the height 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 selection-replace
                                left top
                                (- right left)
                                (- bottom top))

       (image-fill! 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.

## Genetic Art

One of the more interesting applications of algorithms to images is so-called *genetic art*. In this technique, we build some form of image “DNA”, generate images from that DNA, rank the “fitness” of the images, and recombine the DNA, of images the DNA of more fit images more frequently in building the next generation. Electric Sheep are one of the more famous examples of genetic art.

Of course, the first part of genetic art is figuring out how to turn the “DNA” into an image. A spectacular variety of techniques exist for such transformations. Dr. Davis has one famous early example (done by hand, rather than by computer), in which someone used a phonebook as the DNA, and turned each odd digit into one color and each even digit into another color in a large grid.

We'll consider two versions of this repurposing of data: One in which we use this same grid technique to turn a file into a series of pixels (or blocks of colors or ...) and another one in which we turn a file into a series of instructions to a turtle.

We've already seen a bit of how we can interpret any file as a series of colors: We could read triplets of characters and treat each triplet as a new color. Here's one implementation of that technique, using `image-iterate!`, a variant of `image-compute-pixels!` that traverses the image systematically, row by row, from top-left to bottom right.

```
;;; Procedure:
;;; file-visualize
;;; Parameters:
;;;   filename, a string
;;;   width, an integer
;;;   height, an integer
;;; Purpose:
;;;   Create a new width-by-height image, filling in the pixels by
;;;   interpreting the contents of the file as colors.
;;; Produces:
;;;   image, a new image.
;;; Preconditions:
;;;   filename names a file.
;;;   That file contains at least width*height*3 characters.
;;; Postconditions:
;;;   image represents the contents of the file.
(define file-visualize
  (let ((color-read (lambda (port)
                      (let ((r (read-char port))
                            (g (read-char port))
                            (b (read-char port)))
                        (if (eof-object? b)
                            color-transparent
                            (rgb-new (modulo (char->integer r) 256)
                                     (modulo (char->integer g) 256)
                                     (modulo (char->integer b) 256)))))))
    (lambda (filename width height)
      (let ((image (image-new width height))
            (port (open-input-file filename)))
        (if (not port)
            (throw "Invalid file name: " filename))
        (image-show image)
        (image-iterate! image
                        (lambda (c r)
                          (color-read port)))
        (close-input-port port)
        image))))
```

One problem with that technique is that it provides a fairly wide variety of colors, giving no form at all to the image. It also doesn't scale appropriately. A simpler technique is to choose a smaller palette of related colors and use the characters as indices into that palette. That is, we pick sections of the image to draw (most simply, a grid of rectangles or ovals), and, for each section, read a character, convert the character to an integer, reduce that integer to the range of valid indices, and then get it from the palette.

```
;;; Procedure:
;;; image-file->grid!
;;; Parameters:
;;;   image, an image
;;;   filename, a string
;;;   palette, a vector of colors
```

```

;;; hcells, an integer
;;; vcells, an integer
;;; Purpose:
;;; Create a hcells-by-vcells grid of colors on image, computing the
;;; colors in the grid from palette and the given file.
;;; Produces:
;;; [Nothing; called for the side effects.]
;;; Preconditions:
;;; filename names a valid file.
;;; image is an open image
;;; hcells > 0
;;; vcells > 0
(define image-file->grid!
  (lambda (image filename palette hcells vcells)
    (let* ((source (open-input-file filename))
           (palette-size (vector-length palette))
           (width (image-width image))
           (height (image-height image))
           (hoffset (/ width hcells))
           (voffset (/ height vcells)))
      (if (not source)
          (throw "Invalid file name; " filename))
      (let kernel ((col 0)
                   (row 0))
        (cond
         ((or (eof-object? (peek-char source)) (>= row height))
          (close-input-port source)
          (image-select-nothing! image)
          (context-update-displays!)
          image)
         ((>= col width)
          (kernel 0 (+ row voffset)))
         (else
          (image-select-rectangle! image selection-replace
                                   col row hoffset voffset)
          (context-set-fgcolor! (vector-ref palette
                                             (remainder (char->integer (read-char source))
                                                         palette-size)))
          (image-fill! image)
          (kernel (+ col hoffset) row)))))))

```

We might use such a procedure to make a wide variety of images by selecting palettes, files, horizontal boxes, and vertical boxes from n.

```

(define series-2
  (let ((palettes (vector (vector color-black color-white color-grey)
                          (vector (rgb-new 255 0 0) (rgb-new 0 0 0) (rgb-new 51 0 0)
                                   (rgb-new 102 0 0) (rgb-new 153 0 0) (rgb-new 204 0))
                          (vector (rgb-new 0 0 0) (rgb-new 128 0 0) (rgb-new 255 0 0)
                                   (rgb-new 0 0 128) (rgb-new 128 0 128) (rgb-new 255 0 128)
                                   (rgb-new 0 0 255) (rgb-new 128 0 255) (rgb-new 255 0 255))
                          (vector color-black color-white color-grey color-red)
                          (vector (rgb-new 0 0 0) (rgb-new 51 51 0) (rgb-new 102 102 0)
                                   (rgb-new 153 153 0) (rgb-new 204 204 0) (rgb-new 255 255 0))
                          (vector (rgb-new 104 104 0) (rgb-new 104 104 51)
                                   (rgb-new 104 104 102) (rgb-new 104 104 153)
                                   (rgb-new 104 104 204) (rgb-new 104 104 255))
                          (vector (rgb-new 0 0 0) (rgb-new 104 104 104)
                                   (rgb-new 255 0 0) (rgb-new 104 0 0))

```

```

                                (rgb-new 153 0 0) (rgb-new 153 153 153))))
(files (vector "/usr/bin/vi"
              "/usr/bin/emacs"
              "/etc/aliases"
              "/bin/bash"
              "/etc/passwd"))))
(lambda (n width height)
  (let ((image (image-new width height)))
    (image-show image)
    (image-file->grid! image
                      (vector-ref files (modulo n (vector-length files)))
                      (vector-ref palettes (modulo n (vector-length palettes)))
                      (+ 2 (modulo n 9))
                      (+ 2 (modulo n 11)))
    image))))

```

While these grid-like visualizations of files are fun, they are certainly not the only thing we can do to interpret a file as the DNA of a drawing. As mentioned above, we can treat a file as a series of commands to a turtle.

```

;;; Procedure:
;;;   file-instruct-turtle
;;; Parameters:
;;;   file, a string that names a file
;;;   image, an image
;;;   n, the number of instructions to take from the file
;;; Purpose:
;;;   Interpret the file (loosely) as a sequence of instructions to
;;;   a turtle that is placed on the image.
;;; Produces:
;;;   [Nothing; called for the side effect]
;;; Preconditions:
;;;   file names a file available for reading.
;;;   turtle exists and is associated with some image.
;;;   n > 0.
;;;   file contains at least 2*n characters.
;;; Postconditions:
;;;   Um, something has happened to the image.
(define file-instruct-turtle
  (let* (; Choose the pens we want to use
        (pens (vector "Circle Fuzzy (03)" "Circle Fuzzy (05)" "Circle Fuzzy (07)"
                      "Circle Fuzzy (09)" "Circle Fuzzy (11)" "Circle Fuzzy (13)"
                      "Circle Fuzzy (15)" "Circle Fuzzy (17)"))
        (numpens (vector-length pens))
        ; Set some initial values
        (initial-color (rgb-new 0 0 255))
        (initial-pen (quotient numpens 2))
        ; Letters for different actions
        (advance-chars (list #\a #\b #\c #\d #\e #\space))
        (turn-chars (list #\f #\g #\h #\i #\j #\o #\u #\y #\. #\newline))
        (darker-chars (list #\l #\m))
        (lighter-chars (list #\n #\p))
        (smaller-chars (list #\q #\r))
        (larger-chars (list #\s))
        (teleport-chars (list #\t))
        ; A helpful function when we may have EOF
        (char2int (lambda (ch) (if (eof-object? ch) 0 (char->integer ch)))))

```

```

(lambda (filename image n)
  ; Open the file
  (let ((port (open-input-file filename)))
    (if (not port)
        (throw "No such file: " filename))
    ; Create the turtle and set some basic values
    (let ((turtle (turtle-new image))
          (midcol (quotient (image-width image) 2))
          (midrow (quotient (image-height image) 2)))
      ; Move the turtle somewhere easy to track and set it to it's initial
      ; values
      (turtle-teleport! turtle midcol midrow)
      (turtle-set-color! turtle initial-color)
      (turtle-set-brush! turtle (vector-ref pens initial-pen))
      ; Okay, let's read those instructions. We'll keep track of the remaining
      ; number of instructions, the current turtle color, and the current turtle
      ; pen.
      (let kernel ((remaining n)
                   (color initial-color)
                   (pen initial-pen))
        ; Read the next character.
        (let* ((ch (read-char port)))
          (display "[" (write ch) (display "]: ")
            (cond
              ; If we reached the end of the file, or run out of commands, stop.
              ((or (eof-object? ch) (= remaining 0))
               (close-input-port port)
               image)
              ; Sometimes we advance
              ((member ch advance-chars)
               (let ((distance (* 3 (quotient (char2int (read-char port)) 10))))
                 (display "Advancing ") (display distance) (newline)
                 (turtle-forward! turtle distance)
                 (kernel (- remaining 1) color pen)))
              ; Sometimes we turn
              ((member ch turn-chars)
               (let ((angle (* 30 (- (modulo (char2int (read-char port)) 9) 3))))
                 (display "Turning ") (display angle) (newline)
                 (turtle-turn! turtle angle)
                 (kernel (- remaining 1) color pen)))
              ; Sometimes we teleport back to the middle
              ((member ch teleport-chars)
               (display "Home!") (newline)
               (turtle-teleport! turtle midcol midrow)
               (kernel (- remaining 1) color pen))
              ; Sometimes we make the color darker
              ((member ch darker-chars)
               (let ((newcolor (rgb-darker color)))
                 (display "Darker") (newline)
                 (turtle-set-color! turtle newcolor)
                 (kernel (- remaining 1) newcolor pen)))
              ; Sometimes we make the color lighter
              ((member ch lighter-chars)
               (let ((newcolor (rgb-lighter color)))
                 (display "Lighter") (newline)
                 (turtle-set-color! turtle newcolor)
                 (kernel (- remaining 1) newcolor pen)))))))

```

```

; Sometimes we use a larger pen
((member ch larger-chars)
 (let ((newpen (min (+ pen 1) (- numpens 1))))
  (display "Bigger") (newline)
  (turtle-set-brush! turtle (vector-ref pens newpen))
  (kernel (- remaining 1) color newpen)))
; Sometimes we use a smaller pen
((member ch smaller-chars)
 (let ((newpen (max (- pen 1) 0)))
  (display "Smaller") (newline)
  (turtle-set-brush! turtle (vector-ref pens newpen))
  (kernel (- remaining 1) color newpen)))
; Sometimes nothing works.
(else
 (display "Skipping") (newline)
 (kernel (- remaining 1) color pen)))))))))

```

Unfortunately, this procedure is a bit less refined than most. Still, it may be worth exploring.

---

Copyright (c) 2007-8 Janet Davis, Matthew Kluber, and Samuel A. Rebelsky. (Selected materials copyright by John David Stone and Henry Walker and used by permission.)

This material is based upon work partially supported by the National Science Foundation under Grant No. CCLI-0633090. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.