

Algorithmic Art

Summary: In this laboratory, you will continue your exploration of Script-Fu by using more Scheme-like operations to automatically generate “art”. You will also explore techniques for modifying existing images.

- Exercises
 - Exercise 0: Preparation
 - Exercise 1: Grid-Based Drawing
 - Exercise 2: Your Own Procedure
 - Exercise 3: Alternative Mappings
 - Exercise 4: Loading Images

Exercises

Exercise 0: Preparation

- a. Start the GIMP.
- b. Open the Script-Fu console.
- c. Open the Script-Fu procedure reference.

Exercise 1: Grid-Based Drawing

We’ve seen a number of ways in which Scheme can be useful for creating images. But we haven’t yet used all the wonders of Scheme. For example, we have yet to take advantage of recursion or higher-order procedures. Here’s an interesting procedure that draws a full image by applying another procedure to a variety of points on the image.

```
;;; Procedure:
;;; map-image-grid
;;; Parameters:
;;; image, an image
;;; proc, a procedure
;;; hspace, an integer
;;; vspace, an integer
;;; Purpose:
;;; Applies proc to a grid of points on the image, where the points
;;; are separated horizontally by hspace and vertically by vspace.
;;; Produces:
;;; Nothing.
;;; Preconditions:
;;; image is a valid image id.
;;; proc takes four arguments: image, layer, x, and y.
;;; hspace and vspace are non-negative integers.
;;; Postconditions:
```

```

;;; The image may have changed.
;;; The brush and foreground and background colors may have changed.
(define map-image-grid
  (lambda (image proc hspace vspace)
    (let* ((layer (car (gimp-image-get-active-layer image)))
           (width (car (gimp-image-width image)))
           (height (car (gimp-image-height image))))
      ; We build a helper procedure that keeps track of the
      ; current x and y location on the grid.
      (letrec ((kernel
                 (lambda (x y)
                   (cond
                    ; If we've exceeded y, we're off the image,
                    ; so stop
                    ((>= y height)
                     ; If we've exceeded x, go on to the next line.
                     ((>= x width)
                      (kernel 0 (+ y vspace)))
                     ; Otherwise,
                     (#t
                      ; Apply the procedure
                      (proc image layer x y)
                      ; And continue with the next point
                      (kernel (+ x hspace) y)))))))
        ; Start the grid at 0,0
        (kernel 0 0))))))

```

What kinds of things can we do with `proc`? We might decide to draw a different color dot at each location, where the color of the dot depends on the location. Here are a few procedures that do just that.

```

;;; Procedure:
;;; blob
;;; Parameters:
;;; image, an image id
;;; layer, a layer id
;;; x, an integer
;;; y, an integer
;;; Purpose:
;;; Draws a blob on the image where the color of the blob depends
;;; on x and y.
;;; Produces:
;;; Nothing.
;;; Preconditions:
;;; The image and layer have been initialized.
;;; The point (x,y) is on the image.
;;; Postconditions:
;;; May have affected the image.
(define blob
  (lambda (image layer x y)
    ; Choose a color that depends on the position
    (gimp-palette-set-foreground (list (mod (* x 7) 256)
                                       (mod (* y 9) 256)
                                       (mod (* (+ x y) 5) 256)))
    ; And draw there using the current brush.
    (gimp-paintbrush image layer 0 2 (float-array x y))))

;;; Value:

```

```

;;; brushes
;;; Contents:
;;; A list of brush names
(define brushes
  (list "Circle (01)"
        "Circle (03)"
        "Circle (05)"
        "Circle (07)"
        "Circle (09)"
        "Circle (11)"
        "Circle (13)"
        "Circle (15)"
        "Circle (17)"
        "Circle (19)"
        "Circle Fuzzy (03)"
        "Circle Fuzzy (05)"
        "Circle Fuzzy (07)"
        "Circle Fuzzy (09)"
        "Circle Fuzzy (11)"
        "Circle Fuzzy (13)"
        "Circle Fuzzy (15)"
        "Circle Fuzzy (17)"
        "Circle Fuzzy (19)"
  ))

;;; Procedure:
;;; blot
;;; Parameters:
;;; image, an image id
;;; layer, a layer id
;;; x, an integer
;;; y, an integer
;;; Purpose:
;;; Draws something on the image at position (x,y) where
;;; the things drawn depends on x and y.
;;; Produces:
;;; Nothing.
;;; Preconditions:
;;; The image and layer have been initialized.
;;; The point (x,y) is on the image.
;;; Postconditions:
;;; May have affected the image.
(define blot
  (lambda (image layer x y)
    ; Pick a number for the brush.
    (let ((brushnum (mod (+ (* x 3) (* y 5)) (length brushes))))
      ; Select that brush
      (gimp-brushes-set-brush (list-ref brushes brushnum))
      ; And paint
      (gimp-paintbrush image layer 0 2 (float-array x y))))))

```

You can find all of these procedures in `gimpfun.ss`. I'd recommend that you make a copy of that file so that you can modify the procedures, but you can also load it with

```
(load "/home/rebelsky/Web/CS151/Examples/gimpfun.ss")
```

How can you use them? Here is one set of sample commands. Feel free to use variants.

a. Create, clear, and display an image with

```
(define image (car (samr-image 256 384)))
```

b. Select a brush with

```
(gimp-brushes-set-brush "Circle (11)")
```

c. Draw some colored circles with

```
(map-image-grid image blob 12 15)
```

d. Select a color (e.g., blue) with

```
(gimp-palette-set-foreground BLUE)
```

e. Draw some varying-shape circles with

```
(map-image-grid image blot 11 13)
```

f. Select another color (e.g. red) with

```
(gimp-palette-set-foreground RED)
```

g. Change the set of brushes with

```
(define brushes (caddr brushes))
```

h. Draw some more varying-shape circles with

```
(map-image-grid image blot 11 13)
```

Congratulations, you are now a master of *algorithmic art*.

Exercise 2: Your Own Procedure

Write your own procedure to use in a call to `map-image-grid`. Feel free to play with colors, brushes, shapes, and even positioning (after all, nothing says that you have to draw at (x,y)).

Exercise 3: Alternative Mappings

a. Write a procedure like `map-image-grid` that calls a drawing procedure on many points of an image, but that determines the points in some way other than using a grid. You might select points on a circle (provided you know your Math), using non-uniform spacing, or anything else you can come up with.

b. Use your procedure to create other interesting images.

Exercise 4: Loading Images

Things can get even more interesting when you work with an existing graphic. It's fairly easy to load and get information about an existing graphic. You can load an image file with

```
(define image (samr-load-image path))
```

For example,

```
(define image (samr-load-image "/home/rebelsky/Web/CS151/Images/NorthCampus.jpg"))
```

(Yes, there are also built-in commands for loading images. Feel free to look for them in the procedure browser. I find mine easier to use.)

You can even get information on the color at any point in the picture by using

```
(samr-get-pixel image layer x y)
```

Again, you can also use built-in commands.

Here's a fun little procedure that paints with the current brush using whatever color is at a designated location.

```
;;; Procedure:
;;; brush-point
;;; Parameters:
;;; image, an image id
;;; layer, a layer id
;;; x, an integer
;;; y, an integer
;;; Purpose:
;;; Paints with the current brush at (x,y) using whatever color is
;;; currently at x,y.
;;; Produces:
;;; Nothing.
;;; Preconditions:
;;; The image and layer have been initialized.
;;; The point (x,y) is on the image.
;;; Postconditions:
;;; May have affected the image.
(define brush-point
  (lambda (image layer x y)
    (let ((color (samr-get-pixel image layer x y)))
      (gimp-palette-set-foreground color)
      (gimp-paintbrush image layer 0 2 (float-array x y)))))
```

Here's a procedure that uses brush-point along with our earlier map-image-grid procedure.

```

;;; Procedure:
;;; munge-image
;;; Parameters:
;;; path, the path to an image file
;;; brush, the name of a paintbrush
;;; Purpose:
;;; Draw a "munged" version of the image.
;;; Produces:
;;; The id of the image.
;;; Preconditions:
;;; path names a valid file that contains a GIF or JPEG.
;;; brush is a valid paintbrush name.
;;; Postconditions:
;;; Shows some interesting variation of the image on
;;; the screen.
(define munge-image
  (lambda (path brush)
    (let ((image (samr-load-image path)))
      (gimp-display-new image)
      (gimp-brushes-set-brush brush)
      (map-image-grid image brush-point 5 9))))

```

Once again, you can find these procedures in `gimpfun.ss`.

a. Try them out using

```

(munge-image "/home/rebelsky/Web/CS151/Images/NorthCampus.jpg"
 "Calligraphic Brush #2")

```

b. Find other images and try different brushes and grid separations.