

Algorithmic Art

Summary: We consider ways in which we can use algorithmic techniques to explore design spaces.

Contents:

- Introduction
- Randomized Drawings
- Algorithmic Grids
- Anonymous Procedures
- Using the Console
- Procedures as Parameters, Revisited

Introduction

Randomized Drawings

A number of artists, from the founders of the Dada movement to Jackson Pollock and beyond, have reveled in the images that can be created by random or unpredictable processes. The Dadaists employed random selection to write poetry and subconscious drawing to create images. Pollock threw paint with an expectation that interesting patterns would result. (And yes, those are incredible simplifications of the philosophies and techniques of these artists.)

We can use a number of simple techniques to generate such randomized art. What can we randomize? We can certainly choose a random color (and relatively easily), simply by choosing random red, green, and blue components.

```
;;; Procedure:
;;; random-color
;;; Parameters:
;;; (none)
;;; Purpose:
;;; Selects and returns a random color.
;;; Produces:
;;; color, a color.
;;; Postconditions:
;;; It is difficult to predict color.
(define random-fgcolor
  (lambda ()
    (set-fgcolor (list (random 256) (random 256) (random 256)))))
```

Similarly, we can choose a random brush. The easiest way to choose a random brush is to select a random element of the brushes list.

```

;;; Procedure
;;;   random-brush
;;; Parameters:
;;;   (none)
;;; Purpose:
;;;   Select one of the brushes.
;;; Produces:
;;;   (nothing)
;;; Postconditions:
;;;   It is difficult to predict the brush.
(define random-brush
  (lambda ()
    (set-brush (list-ref (list-brushes) (random (length (list-brushes)))))))

```

However, that procedure can choose from perhaps too many brushes. We might create a similar procedure that selects from a particular list of brushes.

```

;;; Procedure:
;;;   randomly-select-brush
;;; Parameters:
;;;   brushes, a list of strings
;;; Purpose:
;;;   Select one of brushes.
;;; Produces:
;;;   (nothing)
;;; Preconditions:
;;;   All the strings in brushes name valid brushes.
;;; Postconditions:
;;;   The current brush is an element of brushes.
;;;   It is equally likely that each element of brushes is now the
;;;   active brush
(define randomly-select-brush
  (lambda (brushes)
    (set-brush (list-ref brushes (random (length brushes))))))

```

If we wanted to select one of the circle brushes, we might use

```
(randomly-select-brush (list "Circle (01)" "Circle (03)" "Circle (05)" "Circle (07)" "Circle (09)" "Circle (11)" "Circle (13)" "Circle (15)" "Circle (17)" "Circle (19)"))
```

In fact, we can rewrite `random-brush` using this procedure.

```

(define random-brush
  (lambda ()
    (randomly-select-brush (list-brushes))))

```

It is, of course, also possible that we'll want to randomly select from other lists (not just lists of brushes). For example, we might want randomly select a shade of brown by selecting an element of the list of blue.s Hence, we'll define a generalized random selection procedure.

```

;;; Procedure:
;;;   randomly-select
;;; Parameters:
;;;   values, a list
;;; Purpose:
;;;   Randomly select an element of values.
;;; Produces:

```

```

;;; value, a value
;;; Preconditions:
;;; values is nonempty.
;;; Postconditions:
;;; value is an element of values.
;;; value is equally likely to be any element of values.
(define randomly-select
  (lambda (values)
    (list-ref values (random (length values)))))

```

We can now select a random shade of blue with the following procedure.

```

;;; Procedure:
;;; random-blue
;;; Parameters:
;;; (none)
;;; Purpose:
;;; Set the foreground color to an unpredictable shade of blue.
;;; Produces:
;;; (nothing)
;;; Postconditions:
;;; The foreground color is a shade of blue.
;;; It is difficult to predict which shade of blue it is.
(define random-blue
  (let ((blues (list BLUE BLUE_VIOLET CADET_BLUE COBALT_BLUE
                     CORN_FLOWER_BLUE DARK_SLATE_BLUE
                     DEEP_MIDNIGHT_BLUE LIGHT_BLUE LIGHT_STEEL_BLUE
                     MEDIUM_BLUE MEDIUM_SLATE_BLUE MIDNIGHT_BLUE
                     NAVY_BLUE NEON_BLUE NEW_MIDNIGHT_BLUE OCEAN_BLUE
                     PALE_BLUE RICH_BLUE ROYAL_BLUE SKY_BLUE
                     SLATE_BLUE STEEL_BLUE)))
    (lambda ()
      (set-fgcolor (randomly-select blues)))))

```

We can also rewrite `randomly-select-brush` to use `randomly-select`. (The process of identifying common procedures and then rewriting code to use these common procedures is called *refactoring* and it's a good practice.)

```

(define randomly-select-brush
  (lambda (brushes)
    (set-brush (randomly-select brushes)))))

```

Okay. Where are we? We have a way to randomly select colors and to randomly select brushes. Now we need ways to randomly draw things. For lines, circles, and rectangles, we can simply randomly select all of the parameters (perhaps within certain limits, such as the width and height of the image). For example, here is a procedure to draw a line between two randomly selected points.

```

;;; Procedure:
;;; random-line
;;; Parameters:
;;; image, an image
;;; width, an integer
;;; height, an integer
;;; Purpose:
;;; Draw a random line in the image, assuming that its width

```

```

;;; and height are as specified.
;;; Produces:
;;; (nothing)
;;; Postconditions:
;;; A new line has been added to image, using the current color
;;; and brush.
(define random-line
  (lambda (image width height)
    (line image (random width) (random height)
          (random width) (random height))))

```

There are many interesting variants, such as ones that use a fixed starting point, a fixed line length, and so on and so forth.

Finally, we can write a procedure that puts it all together.

```

;;; Procedure:
;;; splat
;;; Parameters:
;;; image, an image
;;; width, an integer
;;; height, an integer
;;; Purpose:
;;; Draw a line between random points, using a random color and
;;; a random brush.
;;; Produces:
;;; (nothing)
;;; Postconditions:
;;; The foreground color may have changed.
;;; The brush may have changed.
;;; The image now contains another line.
(define splat
  (lambda (image width height)
    (set-fgcolor (random-color))
    (random-brush)
    (random-line image width height)))

```

Algorithmic Grids

Although randomness can provide interesting images and useful programming challenges it is, in the end, a bit too unpredictable for some. Another technique that some artists use to explore design spaces is to break the image up in to a grid and to draw different things in each grid space, with the choice of what to draw based on the position in the grid. For example, we might draw at each location with the same brush, but choose the color for the brush based on the location.

Here's one such strategy for choosing a color:

- Let the red component be 5 times the x coordinate (modulo 256).
- Let the green component be 255 times the absolute value of the sine of the y coordinate. (Since the sine ranges from -1 to 1, multiplying its absolute value by 255 gives us a range from 0 to 255.)
- Let the blue component be the product of the x and the y coordinate (again, modulo 256).

We might define procedures for each of these.

```
(define func1
  (lambda (x y)
    (modulo (* 5 x) 256)))
(define func2
  (lambda (x y)
    (trunc (* 255 (abs (sin y))))))
(define func3
  (lambda (x y)
    (modulo (+ x y) 256)))
```

Now, what can we do with these? If you install `grid.scm` in your GIMP scripts directory (instructions in the lab), you will see that the Script-Fu menu contains a Glimmer submenu with a Color Grid menu item. If you select that menu item, you can choose the size of the grid to build, the spacing between items in the grid, and the procedures for the red, green, and blue components. (Right now, the only valid procedures are `func1`, `func2`, and `func3`, but you can associate any one of the three with each component.)

The Color Grid item does little more than build an image and then recursively step through all the positions in the grid. For each position, it builds a new color (by applying the red procedure to the position, the green procedure to the position, and the blue procedure to the position, and then combining them into a color), and then paints a single “dot” using the current brush at the position.

For example, suppose we use `func1` for the red component, `func2` for the green component, and `func3` for the blue component. At the point (10,10), the color gets set to (50 138 20). At the point (50,10), the color gets set to (250 138 60). At that point (200,100), the color gets set to (232 129 44).

Anonymous Procedures

The color grid technique works well with the three procedures defined above, but clearly there are other ways to convert positions to colors. What should we do if we want a different color component procedure? For example, what if we want the color to depend on the square of the x component?

The most straightforward thing to do is, of course, to (1) create a new procedure (say `func4`) in a file, (2) load the file, and then (3) use that procedure in the dialog. For example, we might write

```
(define func4
  (lambda (x y)
    (modulo (* x x) 256)))
```

However, this is a bit inconvenient. If we just want to try out a new procedure, why do we have to go back to the editor, type it in, choose a name, go back to the GIMP, load the file, and so on and so forth?

In fact, Scheme programmers often ask a similar question: “Why do I have to name this procedure that I’m only going to use once?” The answer is, “You don’t!” So, how do we avoid naming these procedures? We might observe that names are just that, names. After the definition `(define grade 95)`, we know that whenever we use `grade`, Scheme plugs in the value 95. If you wanted to, you could just use 95 rather than `grade`.

It turns out the same thing happens when you define procedures. Whenever you write `func4`, for example, Scheme substitutes `(lambda (x y) (modulo (* x x) 256))`. Then, whenever Scheme tries to apply one of these *lambda forms*, it does what you'd expect: the Scheme interpreter substitutes the actual parameters for the formal parameters in the body, and then executes the body.

What does this mean to you? It means that you can write the lambda expressions directly. In particular, if we want to try new procedure you can write them in the *Red Component*, *Green Component*, or *Blue Component* fields. For example, we might fill in

- `(lambda (x y) (modulo (+ (* 4 x) (* 3 y)) 256))`
- `(lambda (x y) (modulo (* x y) 256))`
- `(lambda (x y) (modulo (abs (- x y)) 256))`

Since these procedures lack names, we call them *anonymous* procedures. You will, of course, have the opportunity to try using anonymous procedures in the Color Grid dialog box in the lab.

Using the Console

As you may recall, one of the reasons that we learned Script-Fu is so that we could enter commands in the console, rather than relying on dialog boxes. Can we draw color grids using the dialog box? Certainly. The `color-grid` procedure accepts seven parameters: the width of the image, the height of the image, the horizontal spacing, the vertical spacing, and the red, green, and blue procedures.

Wait a minute! Those last three parameters sound a bit odd, don't they? Normally, we only pass simple values (numbers, lists, strings, etc.) as parameters. However, in Scheme, you can also pass procedures as parameters to other procedures. For example, we might draw a variant of the original grid image (using `func1` for red, `func2` for green, and `func3` for blue) with the following.

```
(color-grid 100 100 12 13 func1 func2 func3)
```

Similarly, we might draw an interesting greyish image with

```
(color-grid 100 100 5 5 func2 func2 func2)
```

In fact, we can even use anonymous procedures as parameters here.

```
(color-grid 100 100 8 9 (lambda (x y) (modulo (* x y) 256)) (lambda (x y) (modulo (* x 5) 256)) (lambda (x y) (trunc (* 255 (abs (sin (* x y)))))))
```

Procedures as Parameters, Revisited

We've just seen that you can use procedures as parameters to some procedures. Can you write your own procedures that expect procedures as parameters? Certainly. You treat the procedure as you would any other parameter. For example, here's a procedure that takes a component procedure and draws a grey image using that component procedure for all three components.

```
(define grey-image  
  (lambda (proc)  
    (color-grid 100 100 5 5 proc proc proc)))
```

We could then draw images with commands like

```
(grey-image (lambda (x y) (+ x y)))  
(grey-image (lambda (x y) (trunc (abs (* 255 (sin (* x y)))))))
```

But we can do even more. We can *apply* the procedures that we get as parameters. For example, here's a procedure that takes three component procedures as parameters and sets the color to the color that should occur at (100,100).

```
(define standard-color  
  (lambda (redproc blueproc greenproc)  
    (set-fgcolor (list (redproc 100 100) (greenproc 100 100) (blueproc 100 100)))))
```

Copyright © 2006 Samuel A. Rebelsky. 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.