

## Higher-Order Graphics

**Summary:** We consider ways in which we can use higher-order techniques to explore design spaces. We look first at using higher-order techniques to generate images and then turn to the use of higher-order techniques to change existing images.

### Contents:

- Algorithmic Grids
  - Anonymous Procedures
  - Using the Console
  - Procedures as Parameters, Revisited
- Manipulating Existing Images with Script-Fu
  - Getting and Setting Colors
  - Modifying Images
  - Anonymous Color Transformers
  - Building Color Transformers
  - Sectioning
  - Combining Transformers
  - From Component Functions to Transformers

## Algorithmic Grids

One reason to use Script-Fu in the GIMP is to explore “design spaces”, groups of similar images that might inspire or initiate other design. One 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)))))
```

## Manipulating Existing Images with Script-Fu

In our initial explorations of the GIMP and Script-Fu, we have developed a number of techniques for creating new images. While this emphasis on creating images has been interesting and fun (or so we hope), it is only one kind of image manipulation that regularly happens in drawing programs.

In fact, many designers use the GIMP and Photoshop not just to create new images, but also to manipulate existing images. You already know some simple ways of manipulating images (well, you know how to draw on top of an image), but we should consider some other techniques. In this reading, we will consider one basic techniques: manipulating images by changing individual colors at individual locations.

## Getting and Setting Colors

As you've observed, the images we draw in the GIMP are, in effect, a simple grid of color values. At each (x,y) position, the image contains a particular color. In order to modify existing images, we need a way to get and set those colors.

The library `hog.scm` contains simple procedures for both activities. The `(get-color-at img x y)` procedure returns the color value for the position (x,y) in `img`. The `(set-color-at! img x y)` procedure changes the color value for position (x,y) in `img`. (Unfortunately, `set-color-at!` only works well for images that are not yet displayed.)

While you can use the color lists that you've learned about in previous readings, it turns out that the GIMP works slightly more efficiently if we use a different representation for colors. (`get-color-at` returns this alternate representation. If you're prefer a list, use `color->list` on the result.) We can extract the red, blue, and green components using the procedures `red`, `blue`, and `green`. For example,

```
> (define col (get-color-at img 20 10))  
> (color->list col)  
(32 128 64)  
> (red col)  
32  
> (green col)  
128  
> (blue col)  
64
```

We build one of these alternate kinds of colors with the `rgb` function.

```
(set-color-at! img 20 10 (rgb 128 128 128))
```

## Modifying Images

So, how do we use these procedures to manipulate an image? One straightforward strategy is to grab the color value at each position, manipulate it somehow, and then set the color at that position to the changed value. Of course, you don't want to bother writing the procedure to do that, so we'll just use one that has been written previously, `modify-image!`. The `modify-image!` procedure takes two parameters: a color transformer and an image to modify. The image is, as you might expect, an image. The color transformer is a procedure that takes a color as a parameter and returns a new color as a result.

For example, a common transformation is to reduce the amount of red by 10%. We might write this transformation as follows:

```
(define red-90
  (lambda (color)
    (rgb (* .9 (red color)) (green color) (blue color))))
```

We can apply this procedure everywhere in the image using the `modify-image` procedure:

```
> (define sam (load-image "Sam.jpg"))
> (modify-image! red-90 sam)
> (show-image sam)
```

In this example, we have scaled the red component by a certain amount. However, there is a debate in the image processing community as to whether it is better to scale components or to change them by a fixed amount. For example, we might reduce the red component by 32, no matter what that component is.

```
(define reduce-red
  (lambda (color)
    (rgb (- (red color) 32) (green color) (blue color))))
```

In the laboratory, you'll explore which of these techniques you prefer.

What other things might we do to an image? Another common transformation is to turn the image into greyscale, either by averaging the three color values or by doing a weighted average, based on the relative brightness of the three colors.

```
(define greyscale-simple
  (lambda (color)
    (let ((ave (/ (+ (red color) (green color) (blue color)) 3)))
      (rgb ave ave ave))))

(define greyscale-better
  (lambda (color)
    (let ((ave (+ (* 0.299 (red color)) (* 0.587 (green color)) (* 0.114 (blue color)))))
      (rgb ave ave ave))))
```

Another fun transformation is to rotate the three color components.

```
(define rotate-components
  (lambda (color)
    (rgb (green color) (blue color) (red color))))
```

In the laboratory, you will have the opportunity to use these procedures as well as other related procedures.

## Anonymous Color Transformers

One disadvantage of the `reduce-red` and `red-90` procedures above is that they have a fixed amount that we're changing the components. If we want a different change (say to reduce the red component by 16 rather than 32, or to scale the red component by 1.1 rather than 0.9), we either have to write a new procedure (which may be difficult to name) or we have to change these procedures (which will be awkward in the case of `red-90`). In either case, we'll then need to spend additional effort to tell Script-Fu that we've made changes.

Is there an alternative? Certainly. When we're just experimenting with possible values, it is much easier to use the anonymous procedures we've encountered previously. For example, suppose we want to scale the blue component by 110%. Rather than writing `blue-110`, we can simply do the following:

```
> (define sam (load-image "Sam.jpg"))
> (modify-image! (lambda (color) (rgb (red color) (* 1.1 (blue color)) (green color))) sam)
> (show-image sam)
```

That technique also makes it easy to combine simple transformations. For example, we might increment green and decrement red with

```
> (define sam (load-image "Sam.jpg"))
> (modify-image! (lambda (color) (rgb (- (red color) 32) (blue color) (+ (green color) 16))) sam)
> (show-image sam)
```

## Building Color Transformers

One difficulty that some people encounter with using the anonymous procedures above is that they get fairly long, which makes them hard to type in the Script-Fu console. Is there something better we can do? Yes, we can write procedures that generate transformers. What does that mean? It means that we'll write procedures that return other procedures as values.

Consider the problem of building a variety of procedures that add different amounts to the red component. Here are a few such procedures.

```

(define redder-32
  (lambda (color)
    (rgb (+ (red color) 32) (green color) (blue color))))
(define redder-16
  (lambda (color)
    (rgb (+ (red color) 16) (green color) (blue color))))
(define lessred-16
  (lambda (color)
    (rgb (+ (red color) -16) (green color) (blue color))))

```

As you might expect, these procedures are incredibly similar, differing only in the value that is added to the red component. Rather than writing all of these variants, we could make the value to be added a parameter to some procedure, say `change-red`, that also takes an image as a parameter.

```

;;; Procedure:
;;;   change-red
;;; Parameters:
;;;   amt, an integer in the range -255 to 255 [unverified]
;;;   color, the color to be transformed.
;;; Purpose:
;;;   Build a new color by adding amt to the red component of color
;;; Produces:
;;;   new-color, a color
;;; Preconditions:
;;;   (none)
;;; Postconditions:
;;;   For any color,
;;;     (red new-color) = (+ (red color) amt)
;;;     unless the sum is less than 0 or greater than 255, in
;;;     which case new-color is 0 or 255, respectively.
(define change-red
  (lambda (amt color)
    (rgb (+ (red color) amt) (green color) (blue color))))

```

Now we can define functions like `redder-32` in terms of `change-red`.

```

> (define grey (rgb 128 128 128))
> (define redder-32 (lambda (color) (change-red 32 color)))
> (red (redder-32 grey))
160
> (blue (redder-32 grey))
128
> (define lessred-32 (change-red -32))
> (red (lessred-32 grey))
96
> (green (lessred-32 grey))
128

```

Of course, there's no need for us to name these created procedures. Just as we can use anonymous procedures whenever we need a procedure, so can we use these built procedures. Here's one simple example

```
> (red ((lambda (color) (more-red 32 color)) grey))
192
```

More importantly, we can use these procedures as parameters to the `modify-image!` procedure.

```
> (define sam (load-image "Sam.jpg"))
> (modify-image! (lambda (color) (more-red 32 color)) sam)
> (show-image sam)
```

Is this less verbose than what we'd been writing before? It's certainly more convenient than defining `redder-32`, `redder-16`, and similar functions. It's also shorter than the following:

```
> (define sam (load-image "Sam.jpg"))
> (modify-image! (lambda (color) (+ 32 (red color)) (green color) (blue color)) sam)
> (show-image sam)
```

Can we do better? Certainly. One strategy is to *curry* the `change-red` procedure. When we curry a procedure, we put each parameter in a separate lambda.

```
;;; Procedure:
;;; redder
;;; Parameters (Curried):
;;; amt, an integer in the range -255 to 255 [unverified]
;;; color, the color to be transformed.
;;; Purpose:
;;; Build a new color by adding amt to the red component of color
;;; Produces:
;;; new-color, a color
;;; Preconditions:
;;; (none)
;;; Postconditions:
;;; For any color,
;;; (red new-color) = (+ (red color) amt)
;;; unless the sum is less than 0 or greater than 255, in
;;; which case new-color is 0 or 255, respectively.
(define redder
  (lambda (amt)
    (lambda (color)
      (change-red amt color))))
```

The structure of this procedure may seem a bit odd, because it has two *nested* lambdas, one for the amount and one for the color. In fact, we apply curried two-parameter procedures slightly differently than we apply normal two-parameter procedures. You may recall that we apply `change-red` by writing something like `(redder 32 grey)`. We apply `redder` by first applying it to the number and then applying the result to the color, as in `((redder 32) grey)`.

```
> (color->list ((redder 32) grey))
(160 128 12)
```

Does this new form help us? Not much, until we realize that there are two ways to think about procedures with nested lambdas: We can think about them as taking two parameters (as in the example above). Alternately, we can think of them as procedures that take one value as a parameter and *return a procedure* that takes the other value as a parameter.

```

;;; Procedure:
;;; redder
;;; Parameters:
;;; amt, an integer in the range -255 to 255 [unverified]
;;; Purpose:
;;; Build a color transformer.
;;; Produces:
;;; make-redder, a color transformer.
;;; Preconditions:
;;; (none)
;;; Postconditions:
;;; For any color,
;;; (red (make-redder color)) = (red color) + amt
;;; unless the sum is less than 0 or greater than 255, in
;;; which case the result is 0 or 255, respectively.
(define redder
  (lambda (amt)
    (lambda (color)
      (change-red amt color))))

```

In this case, we can think of the outer lambda as asking for the parameter to `redder`. The inner lambda describes the parameter to the procedure the `redder` returns.

For example,

```

> (define grey (rgb 128 128 128))
> (define redder-32 (redder 32))
> (color->list (redder-32 grey))
(160 128 128)
> (define lessred-32 (redder -32))
> (color-> list (lessred-32 grey))
(96 128 128)

```

The `hog.scm` library contains `redder`, `greener`, and `bluer`. In the corresponding laboratory, you will have the opportunity to write some similar procedures, such as `scale-red`.

## Sectioning

If you think about it, once we've defined `change-red`, `change-blue`, and `change-green`, the procedures `redder`, `bluer`, and `greener` are going to be very similar. Each takes a two-parameter procedure and a value as parameters, and creates a new procedure whose body fills in the first parameter of the two-parameter procedure.

In fact, It turns out that filling in the first parameter of a two-parameter procedure is a fairly common activity. For example, here is a procedure that, given an image as a parameter, converts the image to greyscale.

```

;;; Procedure
;;; convert-to-greyscale!
;;; Parameters:
;;; image, an image
;;; Purpose:
;;; Converts the image to greyscale.

```

```

;;; Produces:
;;; (nothing)
;;; Preconditions:
;;; (none)
;;; Postconditions:
;;; image is now in greyscale (that is, for each position, the red,
;;; green, and blue components are identical).
;;; The color at each position has approximately the same brightness
;;; as it did prior to the conversion.
(define convert-to-greyscale!
  (lambda (image)
    (modify-image! greyscale-better image)))

```

The increment procedure, which adds one to its parameter, is another instance of this pattern.

```

(define increment
  (lambda (val)
    (+ 1 val)))

```

In effect, increment turns the two-parameter + procedure into a one-parameter procedure by filling in the first parameter.

The process of filling in the first parameter of a two-parameter procedure is so common that it may be worth refactoring all of these procedures to create a common procedure. That common procedure is typically called left-section or l-s.

```

;;; Procedure:
;;; left-section
;;; l-s
;;; Parameters:
;;; binproc, a two-parameter procedure
;;; left, a value
;;; Purpose:
;;; Creates a one-parameter procedure by filling in the first parameter
;;; of binproc.
;;; Produces:
;;; unproc, a one-parameter procedure
;;; Preconditions:
;;; left is a valid first parameter for binproc.
;;; Postconditions:
;;; (unproc right) = (binproc left right)
(define left-section
  (lambda (binproc left) ; Parameters to left-section
    (lambda (right) ; Parameters to unproc
      (binproc left right))))
(define l-s left-section)

```

A fairly short procedure, but it has many implications. First, it simplifies many definitions. Consider some of the ones we've just written. We can rewrite them as follows

```

(define bluer (lambda (val) (l-s change-blue val)))
(define convert-to-greyscale! (l-s modify-image! greyscale-better))
(define increment (l-s + 1))

```

Note that `l-s` lets us define some procedures without even writing lambdas! It also makes it easier to do without `redder`, `bluer`, and `greener`. After all, `(redder amt)` is just `(l-s change-red amt)`. Since the latter is almost as short as the former, we might consider writing it instead.

```
> (define sam (load-image "Sam.jpg"))
> (modify-image! (l-s change-red 128) sam)
> (show-image sam)
i
```

## Combining Transformers

While `redder`, `greener`, `bluer`, and quite useful, they can be limiting. For example, suppose we want a procedure that decreases red by 32 and increases green by 16. What do we do? One possibility is to look at the context and to simply apply the two transformations in sequence.

```
> (define sam (load-image "Sam.jpg"))
> (modify-image! (l-s change-red -32) sam)
> (modify-image! (l-s change-green 16) sam)
> (show-image sam)
```

However, that is a bit inconvenient and a bit verbose. Is there a better strategy? Certainly. We can rely on a process that you may remember from your days in Mathematics, *composition*. The composition of two functions,  $f$  and  $g$  is a new function. The new function gives the same result as you'd get from first applying  $g$  to the parameter and then applying  $f$  to that result. For example, if you compose the square function with the function that increments by 1, you get a function that adds 1 and then squares. (That function, when applied to 3, gives 16; when applied to 7, it gives 64.)

We can define `compose` in Scheme using a literal translation of that description (well, of a modification of that description).

- “compose is”: `(define compose`
- “a function of two parameters,  $f$  and  $g$ ”: `(lambda (f g)`
- “that returns a new function of one parameter”: `(lambda (x) ...)`
- “that applies  $g$  and then applies  $f$ ”: `(f (g x))`

Putting it all together:

```
;;; Procedure:
;;; compose
;;; Parameters:
;;; f, a procedure
;;; g, a procedure
;;; Purpose:
;;; Compose f and g.
;;; Produces:
;;; fog, a procedure
;;; Preconditions:
;;; f can be applied to the results returned by g.
;;; Postconditions:
;;; (fog x) = (f (g x))
```

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

Now we can use this in a variety of ways. For example, here's an attempt to convert an image to greyscale and then make it a bit bluer.

```
> (define sam (load-image "Sam.jpg"))
> (modify-image! (compose (bluer 32) greyscale-better))
> (show-image sam)
```

We can also use composition in everyday computation.

```
> (define inc-and-square (compose square increment))
> (square-and-increment 3)
16
```

## From Component Functions to Transformers

You may have noted a slight disconnect between this approach to image manipulation and the approach you saw when dealing with color grids. In particular, when we worked with color grids, we wrote functions that computed each of the three color components separately. Can we do the same here? Certainly. In this case, the component functions will take colors, rather than points, as parameters.

```
;;; Procedure:
;;;   color-transformer
;;; Parameters:
;;;   redfunc, a function from colors to integers in the range 0..255.
;;;   greenfunc, a function from colors to integers in the range 0..255.
;;;   bluefunc, a function from colors to integers in the range 0..255.
;;; Purpose:
;;;   Combine the three functions into a color transformer.
;;; Produces:
;;;   transform, a function from colors to colors.
;;; Preconditions:
;;;   (none)
;;; Postconditions:
;;;   (transform color) =
;;;     (rgb (redfunc color) (greenfunc color) (bluefunc color)))
(define color-transformer
  (lambda (redfunc greenfunc bluefunc)
    (lambda (color)
      (rgb (redfunc color) (greenfunc color) (bluefunc color)))))
(define c-t color-transformer)
```

Why is this useful? It certainly simplifies the definition of rotate-components:

```
> (define rotate-components (c-t green blue red))
> (color->list (rotate-components (rgb 32 64 128)))
(64 128 32)
```

In fact, `color-transformer` provides an alternative to `compose` for building compound transformers. Consider the following way to change red by 32

and blue by -16:

```
> (define sam (load-image "Sam.jpg"))
> (modify-image! (c-t (compose (1-s + 32) red) (compose (1-s + -16) green) blue) sam)
> (show-image sam)
```