3 The Digital Darkroom

Computers work with numbers, not pictures. To get a computer to work on a picture we have to find a way to represent the image in numbers. Fortunately, the first thing that comes to mind turns out to be perfectly workable: think of the picture as a big array of dots and assign a value to each dot to represent its brightness. We will stick to black-and-white images here. For color images the same principles apply. We would use three brightness values per dot: one for each of the primary colors red, green, and blue. Two questions that remain are: How many dots (*pixels*) do we need, and how large a value should be used to define each dot?

From Pictures to Numbers

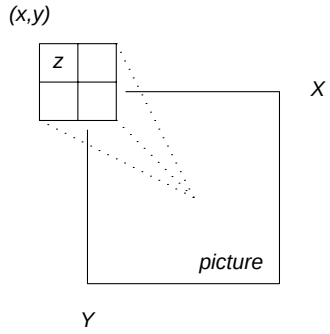
Let's start with the basics. Photographic film, say 35mm black-and-white film, has an exposure range in the order of 1:256 and a resolving power of roughly 4000 dots per inch. For photographic paper the numbers are somewhat lower, on the order of 1000 dots per inch resolution and a brightness range of 1:30, but for now let us use the quality of the film storing the original image as a reference. To store a brightness value between 0 and 255 we need an 8-bit number (2^8 equals 256), quite happily the size of a byte on most machines. So, one part of the problem is easily solved: each dot in the image to be digitized can be stored in one byte of memory. One frame on a 35mm film measures 24×36mm or roughly 0.9×1.4 inches. If we digitize the frame at 4000 dots per inch, each frame will produce 0.9×4000×1.4×4000 numbers, which corresponds to some 20 megabytes of data. But this is only part of the story.

The negative holds enough information to allow us to make enlargements on photographic paper at an acceptable resolution. If we enlarge the 24×36mm

frame five times, the effective resolution of the enlargement drops from 4000 dots per inch on the film to 800 dots per inch on the print. Another reference is the resolving power of the eye. After all, the image only needs to have a certain resolution to make the dots invisible to the eye. How many dots can we distinguish on a print? The resolving power of a human eye is approximately 1/60 of a degree, which means that even under optimal conditions we cannot resolve more than about 600 dots per inch at a viewing distance of 12 inches. An image digitized and reproduced at 600 dots per inch is indistinguishable from the original under normal viewing conditions. Most of the photos in this book were digitized at 750 dots per inch. Just for comparison, the resolution of a television image is less than 500 by 500 dots, which on a 15 inch diagonal screen corresponds to about 47 dots per inch.

Z Is for White

To make it easier to talk about digitized images and image transformations, we will introduce some shorthand. Let's stick to the analogy of a picture as a two-dimensional array of dots. Each dot then has three attributes: one number defining its brightness and two other numbers defining its location in the array (and in the image). Since we will use these three numbers quite extensively, I will give them names. There is nothing special about these names: they are just shorthand. The two numbers defining the location of a dot within a picture will be named x and y and the brightness of the dot can be called z.



The Sampling Grid

The dots are really "samples" of the brightness in the real image. The (x, y) grid is therefore sometimes referred to as the "sampling grid."

For a given picture the x, y, and z are of course not independent. Given the first two, the last one is determined: z is a function of x and y. The value of z at position (x, y) in some image named *picture* is written as

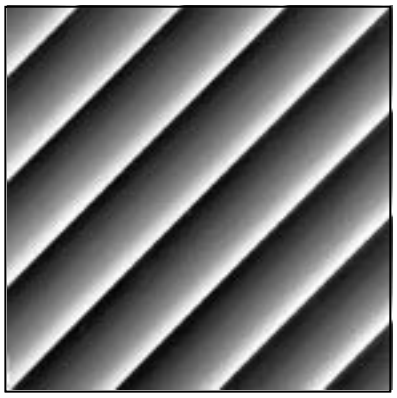
$$picture[x, y].$$

It is convenient to introduce symbols for the maximum values of the three numbers just named. The picture width in dots is named X. The x coordinate ranges in value from 0 to $X - 1$. Similarly, Y is the picture *height*. The y coordinate ranges from 0 to $Y - 1$. Z, finally, defines the maximum brightness value of the dots. If we use one byte per dot, Z will be 255, and z can have a

value between 0 and 255. X and Y may turn out to be 512 or 1024, depending on the size of the picture and the resolution at which it was scanned, but the precise values are largely irrelevant from this point on. A low value for z means a low brightness (dark) and a high value is a high brightness (light). A low value for x refers to the left side of the image and a low value for y refers to the top.

A Picture Transformation Language

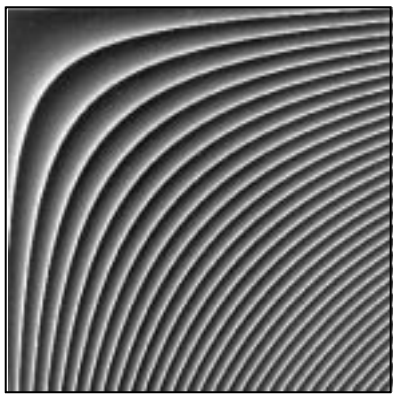
Given sufficient resolution, any image can be translated into numbers and those numbers can be translated back into an image. Our purpose here is to find a simple way to describe images and image transformations. To do that we are developing a little language that consists of symbols like the x, y, and z introduced above, and some transformation expressions. Two symbols that come in handy for defining transformations are *old* and *new*. We use the first to refer to the result of the last transformation performed. The second symbol, *new*, refers to the destination of the current transformation: the newly created image. We can also refer to specific dots in the old or new image by writing $old[x, y]$ and $new[x, y]$, where x and y are as defined above. So, using only the symbols introduced so far, we can *create* a picture by writing in this language:



$$new[x, y] = x + y \qquad (3.1)$$

The expression defines a brightness $x + y$ of the image *new* for each dot at position (x, y). It may confuse you that a location is translated into a brightness, but that's exactly what we are doing! The expression $x + y$ produces a number and all numbers together define an image. But there's a catch. The brightness value of the dots in the *new* image is restricted to values between 0 and Z, but the maximum value of $x + y$ can of course be much larger than Z. If dots are stored in bytes, the values assigned will wrap around the maximum value Z. If we want we can make this effect explicit by using *modulo* arithmetic. If $x + y \geq Z + 1$, we subtract $Z + 1$ as often as necessary from $x + y$ to get a value that fits the range. So, a brightness value Z remains Z,

but $Z+1$ becomes 0, and $3*Z+12$ becomes 9. The value of an expression E taken modulo-Z is usually written as $E\%Z$. As a variation on (3.1) we can try:



(3.2)

(3.3)

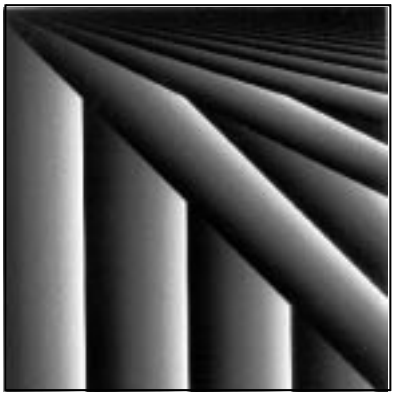
$$new[x,y] = (x*y)\%(Z+1) \qquad (3.2)$$

The modulo operator % will come back a few more times below. For now, just remember that all brightness values are by default modulo $Z+1$. The range of available brightness values can be matched precisely to the grid:

$$new[x,y] = (Z*x*y)/((X-1)*(Y-1)) \qquad (3.3)$$

In the upper left-hand corner both x and y are zero and thus the brightness z at $new[0,0]$ will be zero, or solidly black. In the lower right-hand corner $x*y=(X-1)*(Y-1)$ and z reaches its maximum value Z: white.

But why do only things that make sense? We just talked about the modulo operator %, so let's try something like

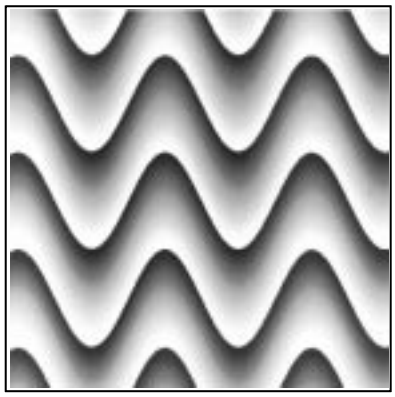


$$new[x,y] = x\%y \qquad (3.4)$$

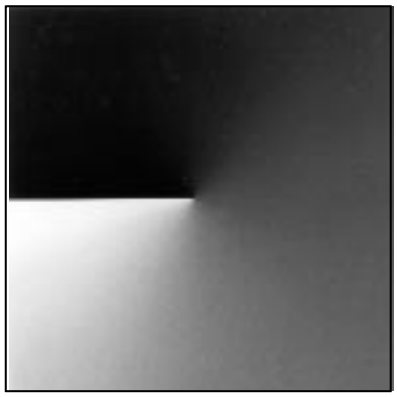
Where x is smaller than y (the lower left triangle) the picture could as well have been defined as $new[x, y] = x$, which, since dots are stored in bytes, is interpreted as $new[x, y] = x\%(Z + 1)$. Note that you can tell from the picture what the current values for X and Y are, given that $Z = 255$.

Trigonometry Made Pretty

If we also include trigonometric functions in our picture language, we can really start experimenting. Let $sin(a)$ be the sine function that returns a value between +1.0 and −1.0. Its argument a is an angle given in degrees. Try to explain the patterns defined by



(3.5)



(3.6)

$$new[x, y] = y + (sin(x) * Z)/2 \qquad (3.5)$$

and, with $atan(y, x)$ returning the arc-tangent of y/x in degrees,

$$new[x, y] = (atan(y - Y/2, x - X/2) * Z)/360 \qquad (3.6)$$

The possibilities for creating intricate patterns with random mathematical functions are endless. With some effort you can even find pictorial representations for interesting mathematical theorems.

Conditional Transformations

It is time to add a little more power to our expression language. We will use the notation

$$(condition)?yes : no$$

to mean that if the *condition* is true (or nonzero), the transformation is defined by expression *yes*, otherwise it is defined by *no*. So, trivially,

$$new[x, y] = (0)?Z : 0 \qquad (3.7)$$

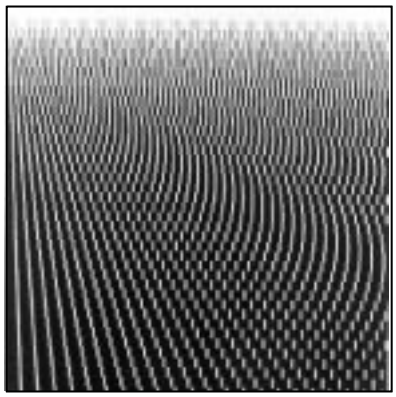
defines an all-black image, and

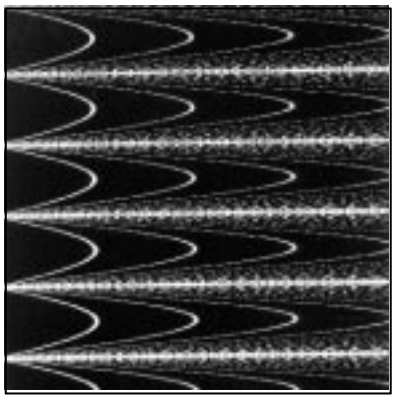
$$new[x, y] = (Z > 0)?old[x, y] : 0 \qquad (3.8)$$

has no effect whatsoever (the condition always holds). Note also that "transformation" (3.7) is independent of x and y coordinates: it defines the same new brightness value Z for every dot in the image, independent of its location.

Between the lines you have already been sold on the idea that an image (*old*) can be used in a transformation just as easily as an expression. We will explore this in more detail in the section titled *Geometric Transformations*.

Using the modulo operator and conditional transformations, we can define interesting patterns with one-liners such as the following two.





(3.9) *(3.10)*

$$new[x,y] = ((x\%(5+y/25)) > 5)?0:Z \qquad (3.9)$$

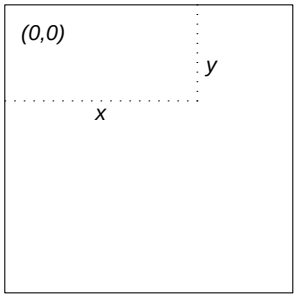
$$new[x,y] = (Z*abs(x\%sin(y)) > 10)?0:Z \qquad (3.10)$$

There is again an infinite number of variations on this theme. We can, for instance, try to make a composite of two photos, using some mathematical function, or even the brightness of a third photo in the conditional.

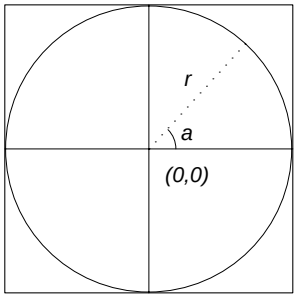
Polar Coordinates

Now let us slightly change the rules of this game. All expressions that we have invented so far used the symbols x and y. The x and the y were defined as Cartesian coordinates in the image array. We can also consider the image area as a simple two-dimensional plane with an arbitrary coordinate system for locating the individual dots.

We can, for instance, define a *polar* coordinate system, with the origin in the middle, and again two numbers to find the location of dots relative to the origin. We name the polar coordinates r and a. The radius r is the distance of a dot from the origin, and a is the angle between a line from the dot to the origin and a fixed, but otherwise arbitrary, line through the origin.



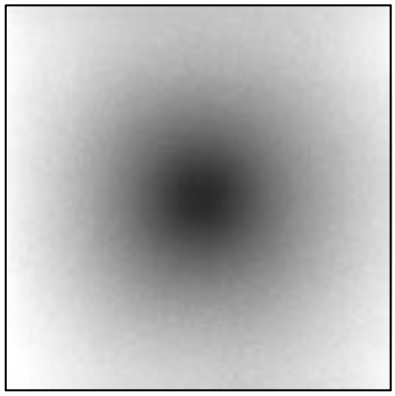
Cartesian Coordinates



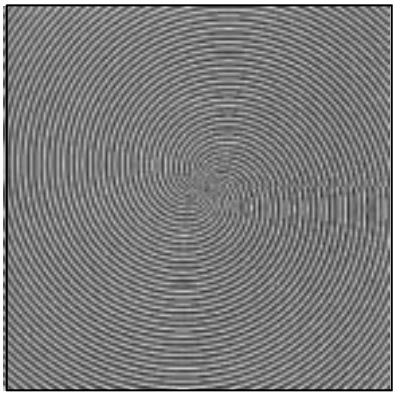
Polar Coordinates

We can again introduce a special shorthand R for the maximum radius and a shorthand A for the maximum angle (360^0). The origin of the polar coordinate system is $(X/2, Y/2)$ in Cartesian coordinates, and similarly, the origin of the Cartesian coordinate system is $(R, 3*A/8)$ in polar coordinates.

Now it is easy to make the picture



(3.11)



(3.12)

$$new[r, a] = (Z * r)/R \qquad (3.11)$$

or the more inspiring

$$new[r, a] = (((a + r)\%16) - 8) * Z/16 + Z/2 \qquad (3.12)$$

But enough said about these artificial images. The expressions that we used to calculate brightness values above can also be used to calculate a malicious deformation of an existing image: It's time to try our hand at some real image transformations.

Point Processes

Image transformations come in a number of flavors. We can, for instance, distinguish between point processes, area processes, and frame processes. A transformation that assigns new brightness values to individual dots, using only the old brightness value of a dot, is called a *point process*. A simple point process is

$$new[x,y] = Z - old[x,y] \qquad (3.13)$$

which makes a negative by subtracting each dot from the maximum brightness value Z. Another example is

$$new[x,y] = Z * log(old[x,y])/log(Z) \qquad (3.14)$$

This particular transformation can be used as part of a correction filter to cope with the "nonlinearity" of devices such as scanners and display monitors: the tendency to lose detail in the dark areas of the picture. The above transformation corrects for nonlinearity by boosting the values of the darker pixels with a logarithmic function.

We can also simulate the effect of photographic "solarization" with a point process. The solarization effect was discovered by Armand Sabattier in 1860. If, in a conventional darkroom, a partly developed image is exposed to raw light, the previously unexposed areas turn from positive to negative, but the previously exposed areas remain as they are. We can simulate this effect with the following transformation, applied here to a portrait of Jim McKie.

$$new[x,y] = (old[x,y] < Z/2)?Z - old[x,y]:old[x,y] \qquad (3.15)$$

We can also slowly fade in a solarization effect from left to right, for instance with:

(3.15)

(3.16)

$$new[x,y] = (old[x,y] > (Z * x)/(2 * X))?old[x,y]:Z - old[x,y] \qquad (3.16)$$

Or we could use the brightness values of one portrait to solarize another. A

point process can also be used to enhance or reduce contrast in a photo. Using conditional expressions we can also apply these changes to only specific parts of a picture.

Area Processes

If not just the old brightness value of the dot itself is used, but values of the dots in a small area around it, the point process becomes an *area process*. An image can be blurred a little by calculating the average brightness value of each dot and four of its neighbors.

(3.17)

(3.18)

$$new[x,y] = (old[x,y] + old[x-1,y] + old[x+1,y] + old[x,y-1] + old[x,y+1])/5 \qquad (3.17)$$

The blurring can be applied to a larger area surrounding each pixel (as shown in Chapter **6**) to enhance the effect. Or, using polar coordinates, the amount of blurring can be varied with the radius to simulate the soft-focus effect of an old lens.

If we can blur an image by *adding* neighboring dots to each pixel and normalizing, imagine what would happen if we *subtract* the brightness values of neighboring dots from each pixel. With the right normalization again we can write this as follows.

$$new[x,y] = 5*old[x,y] - old[x-1,y] - old[x+1,y] - old[x,y-1] - old[x,y+1] \qquad (3.18)$$

The effect of the transformation is a digital filtering that works as though it restored the focus in a blurry image. Formally, the process approximates the working of a *Laplacian* filter ∇^2.

Another example of an area process is this one to make a relief. The transformation is useful in more serious image processing applications as a fast edge detection filter, illustrated here with a portrait of Brian Redman.

$$new[x, y] = old[x, y] + (Z/2 - old[x + 2, y + 2]) \qquad (3.19)$$

Geometric Transformations

With a third type of transformation we can define geometric transformations to the sampling grid and change the coordinates of the dots in an image. The standard mapping defines a regular grid of dots. Geometric transformations are used to reshape that grid. If the portrait of Brian Redman is stored in an array named *ber*, we can write

(3.20)

(3.21)

$$new[x, y] = ber[x, y] \qquad (3.20)$$

which is the normal photo on the standard grid with each dot at location (x, y) in *ber* mapped to a new dot at precisely the same location in the *new* image. This is still a point process. But we can play more interesting games with the picture. For instance,

$$new[x, y] = ber[x, Y - y] \qquad (3.21)$$

turns the picture upside down, by reversing the y coordinate in the grid. And, only slightly more complicated,

(3.22)

(3.23)

$$new[x, y] = ber[y, X - x] \qquad (3.22)$$

rotates the image by 90^o clockwise. The x and y coordinates are swapped, and the order of the x is reversed. Reversing the order of y instead of x makes the image rotate counterclockwise.

We are still using all the dots in the old image to create the new one. We can also break that rule and try something like

$$new[x, y] = ber[x/2, y] \qquad (3.23)$$

to stretch the image horizontally by a factor of 2. This stretching operation can be made more interesting still by using arbitrary trigonometric functions to calculate the offset, or by stretching both the x and y coordinates. Note also that

$$new[x, y] = old[x*2, y*2] \qquad (3.24)$$

is a simple way to shrink an image. However, to avoid having the coordinates overflow their maxima and cause havoc, it is more prudent to write either

$$new[x, y] = old[(x*2)\%(X + 1), (y*2)\%(Y + 1)] \qquad (3.25)$$

or

$$new[x, y] = (x \le X/2 \&\& y \le Y/2)?old[x*2, y*2]: 0 \qquad (3.26)$$

The most rewarding geometric transformations on portraits are made with conditional expressions. We can, for example, make a perfect mirror composite of a portrait, once it is centered properly. These transformations

$$new[x, y] = (x >= X/2)?bwk[x, y]: bwk[X - x, y] \qquad (3.27)$$

(3.27)

(3.28)

$$new[x, y] = (x < X/2)?bwk[x, y]:bwk[X - x, y] \qquad (3.28)$$

are two different ways to mirror a portrait of Brian Kernighan vertically along its middle axis. Of course, even more startling effects can be produced by mirroring along a horizontal axis.

If we can do all this with a single image, imagine what could be done with two or more! Let's see how we could use the portraits of Rob Pike and Peter Weinberger. Here they are first shown in their original, unedited, version.

Rob Pike

Peter Weinberger

Frame Processes

Transformations that work on multiple images are called *frame processes*. Suppose we have the portraits of Rob Pike and Peter Weinberger stored in two image files named *rob* and *pjw*. An average of the two is quickly defined, though not very inspiring.

(3.29)

(3.30)

$$new[x,y] = (rob[x,y] + pjw[x,y])/2 \qquad (3.29)$$

All we have to do is add the pictures and divide by 2. We can also fade one picture slowly into the other, which makes for a more interesting picture. A first attempt might be a full linear fade.

$$new[x,y] = (x * rob[x,y] + (X - x) * pjw[x,y])/X \qquad (3.30)$$

But that doesn't really work out too well. If we restrict the fade to just the middle part of the image, it looks better. The transformation expression we need must have a different effect in three different areas of the image: left, middle, and right. We can use a conditional transformation again to accomplish this, but note that we need more than one condition this time. We can do that with a nested conditional as follows.

$$(left)?pjw:(right)?rob:fade$$

The last part

$$(right)?rob:fade$$

is treated as a separate transformation expression that takes effect only when the condition (*right*) of the first expression is false. Transformation (3.31) shows the details.

$$new[x,y] = (x < X/3)?pjw:(x > 2*X/3)?rob \\ :((x - X/3)*rob + (2*X/3 - x)*pjw)*3/X \qquad (3.31)$$

We can also do this transformation in a vertical plane, and use two, more carefully selected, portraits, to achieve the following effect (admittedly, the

resulting photo was touched up a little with a separate editor).

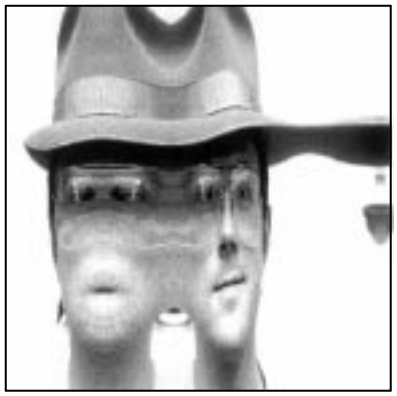
(3.31)

(3.32)

$$new[x,y] = (y < Y/3)?ber:(y > 2*Y/3)?skull \\ :((y-Y/3)*skull + (2*Y/3-y)*ber)*3/Y \qquad (3.32)$$

Again, nothing prevents us from experimenting at length with even less useful compositions. We can, for instance, use the brightness of an image in the conditional,

(3.33)



(3.34)

$$new[x,y] = (rob[x,y] > Z/2)?rob[x,y]:pjw[x,y] \qquad (3.33)$$

or play more involved tricks with the coordinates

$$new[x,y] = rob[x + (X*cos(((x-C)*A)*2/X))/6, y] \qquad (3.34)$$

where A is the maximum angle 360, and C is a constant. In this case X was 684 and C was 512.

And There's More

Well, we have now set the stage for more interesting work. What follows in Chapter **4** is a selection of the most startling image transformations we happened upon while playing with this picture language. Chapter **5** includes a discussion of some software that can be used to build an image editor to experiment further with these transformations on a home computer. Chapter **6** gives an overview of the image transformations that we have discussed.