PICO—A PICTURE EDITOR

Gerard J. Holzmann
[AT&T Technical Journal, Vol. 66, No. 2, March/April 1987, pp. 2-13]

Abstract

PICO is an interactive editor for digitized graphic images. Editing operations are defined in a simple expression language based on the C language. The editor treats images as an ordered set of pixel structures stored in two-dimensional arrays. PICO checks editing commands for syntax, translates them into programs, optimizes and then executes them, all within a few seconds of run time. The command structure is similar to that of conventional multifile text editors with options for reading, writing, and transforming digitized images.

INTRODUCTION

On the computer systems in our research center, several hundred megabytes worth of files are used to store pictures: synthesized, digitized, gray scale and color images. Some of these images, for instance, are used in the UNIX® system 9th edition to announce the arrival of computer mail on graphics terminals by showing a small picture of the sender. Each mail portrait is a 48-by-48-bit icon extracted from a digitized black-and-white photo.

Figure 1. Photo of Rob Pike
Referred to as `$rob` in equations.

Figure 2. Photo of Peter Weinberger
Referred to as `$pjw` in equations.

Figures 1 and 2 are unedited photos of Rob Pike and Peter Weinberger that will be used in the examples that follow. Several hundred similar photos have been digitized. Such a wealth of online image information, of course, invites further ventures into image processing, and creates a need for fairly general tools for editing, combining, and transforming images. This has lead, almost as a by-product, to the development of a surprisingly powerful general-purpose editor for digitized images that will be described in this paper.

Typical editing operations on digitized images include:

- Changing contrast or histograms
- Masking, filtering, or enhancing image areas

- Merging, fading, scaling, and rotating images.

It is not practical to write a new special-purpose program for each different type of transformation. Instead, we developed a graphics editor with a simple command language that is powerful enough to define almost any type of transformation. The editor parses user commands, translates them into optimized code, and executes image transformations as programs. The command language is simple, with defaults defining the most common choices. A transformation to make a negative of a picture with brightness values in the range 0...Z, for example, can be defined in one statement:

$$\texttt{new} = \texttt{Z} - \texttt{old} \tag{1}$$

This is independent of the size of the picture or whether it is in color or in black and white.

The expression above is expanded internally into a small program, which (unoptimized) could be as follows for a color image.

```
for (y = 0; y < Y; y++)
for (x = 0; x < X; x++)
{     newred[y*X+x] = 255 - oldred[y*X+x];
      newgrn[y*X+x] = 255 - oldgrn[y*X+x];
      newblu[y*X+x] = 255 - oldblu[y*X+x];
}
```

The upper left corner of the image is at the origin of the screen coordinate system, with positive `x` pointing to the right and positive `y` pointing down. `X` and `Y` are predefined constants that specify the current width and height of the image work buffer (often 512 pixels). Similarly, the `Z` we saw earlier is a predefined constant that defines the maximum brightness value (often 255).

Rotating the image by 90° is only slightly more complicated:

$$\texttt{new}[\texttt{x}, \texttt{y}] = \texttt{old}[\texttt{y}, \texttt{x}] \tag{2}$$

The editor again will fill in the details. Equation (2), however, not only rotates, but also mirrors the original. More accurately, therefore, we should say:

$$\texttt{new} = \texttt{old}[\texttt{y}, \texttt{X} - \texttt{x}] \tag{3}$$

for a rotation by 90° clockwise. To rotate counterclockwise, we write:

$$\texttt{new} = \texttt{old}[\texttt{Y} - \texttt{y}, \texttt{x}] \tag{4}$$

`X` and `Y` are predefined constants that specify the current width and height of the image work buffer (often 512 pixels).

Rotation by an arbitrary angle can be achieved by using built-in procedures for changing between polar and Cartesian coordinates. For example, a rotation by 30° clockwise can be written as follows using two global variables `r` and `a`:

$$\texttt{new} = \texttt{old}[\texttt{x\_cart}(\texttt{r} = \texttt{r\_polar}(\texttt{x}, \texttt{y}), \texttt{a} = (\texttt{a\_polar}(\texttt{x}, \texttt{y}) + 30)), \texttt{y\_cart}(\texttt{r}, \texttt{a})] \tag{5}$$

Normally, we have to insure that the array indexes fall within the bounds `X` and `Y`. In this case, the procedures `x_cart` and `y_cart` take care of that internally. Note also that the statement above relies on the order of evaluation of the arguments in the compiled code.

Not all operations have to refer to the `old` or `new` picture arrays. Any syntactically correct statement is by default executed once for each pixel in the image, with the appropriate settings of the coordinate variables `x` and `y`. To overrule this default, it is sufficient to enclose the statement in curly braces. The two global variables `r` and `a` used in Equation (5), for example, can be declared as follows:

```
{ global int r, a; }
```

To declare a global array of 256 integers initialized to zeros, we write:

```
{ global array histo[256]; }
```

The declaration prefix `global` extends the scope of array `histo` so that it can be referred to in subsequent procedures, programs, or expressions. Using the default loop and C syntax, we can now make a histogram of the current image by simply typing:

```
histo[old]++
```

Array `old` is by default indexed with the current values of `x` and `y` and, for a black and white image, returns a brightness value between 0 and 255. The brightness of a pixel in a color image is defined by three bytes, which in this case would first be averaged by PICO to obtain a single-byte index. (See **Structure**.) The value returned by `old` is used to select an element in array `histo` that is incremented. The statement (expanded into a loop over 512 by 512 pixels) is executed in eight seconds on a Digital Equipment Corporation VAX-750 computer. After the statement is executed, array `histo` can be used straightforwardly to change the brightness values of the original image (to enhance contrast, for example) or to improve the separation of the gray-scale values within the image (by histogram equalization).

Figure 3. Average of `$rob` and `$pjw` (Eq. 7).

THE PICO EDITOR

The editor that interprets expressions such as the above was named “PICO.” (Originally the name indicated its size; later it was more easily understood as an abbreviation of “picture composition.”)

To the editor, a picture is merely an ordered set of pixel structures stored in a two-dimensional array. Each pixel structure defines brightness values for up to three channels: one for each of the primary colors red, green, and blue. The work buffer of the editor is restricted to a maximum window in the picture being edited of 512 by 512 pixels. The editor stores up to three bytes per pixel, or up to 786,432 bytes for each image opened for editing.

The result of the last edit operation performed is accessible under the predefined name `old`. An edit operation is an assignment of new values to elements in the `old` pixel array. The result of the assignment is stored in a second pixel array named `new`. At the end of each edit operation, the pixel arrays pointed to by `old` and `new` are swapped.

The assignments may be based on pixel coordinates, variables, old pixel values, or arbitrary arithmetic combinations of these. Equations (1) to (5) are statements that describe more or less standard edit operations with a single expression. For transformations that can not easily be cast into simple arithmetic expressions, the user can define more explicit editing procedures and programs.

Consider Equation (2). Applied to all pixels, preserving the distinction between channels, this single assignment transposes the `old` matrix and assigns the result to pixel array `new`. For a three-channel image, it defines a parallel assignment for all `x` and `y`.

Figure 4. Conditional Expression. Figure 5. More complex transformation.

$$\begin{aligned} \texttt{new}[x, y].\texttt{red} &= \texttt{old}[y, x].\texttt{red} \\ \texttt{new}[x, y].\texttt{grn} &= \texttt{old}[y, x].\texttt{grn} \\ \texttt{new}[x, y].\texttt{blu} &= \texttt{old}[y, x].\texttt{blu} \end{aligned} \tag{6}$$

Names preceded by a dollar sign refer to picture fi les, with full path names abbreviated to base names (i.e., `"/usr/gerard/pico/pjw"` is referred to as `"$pjw"`). An average of the two pictures `$rob` and `$pjw` (Figures 1 and 2) can then be expressed as a matrix operation (Figure 3).

$$\texttt{new} = (\$\texttt{rob} + \$\texttt{pjw})/2 \tag{7}$$

Or similarly,

$$\texttt{new} = (\$\texttt{rob} * \$\texttt{pjw})/Z \tag{8}$$

Both of these are unconditional and apply to all pixels, again by default preserving the distinction between channels and matching pixels found at equal coordinates in the arrays.

Because each transformation expression is by default executed once for each pixel of the image, PICO contains an optimizing compiler that translates the expressions “on-the-fly” (interactively) into effi cient code for the VAX-750 computer.

Consider the following edit session in which “%” is a system prompt. Each number followed by a colon is PICO’s prompt for a new editing command.

```
%  pico
1:  a  "/tmp/image/rob"
2:  a  "/usr/gerard/pico/pjw"
3:  x  new = (x<256) ? ($rob+$pjw)/2 : Z-$pjw
4:  x  new = old[x, Y-y] }   (9)
5:  w  output
6:  q
%
```

Each command consists of a single leading character that identifi es the type of operation to be performed:

- `a` to attach (open) new image fi les
- `x` to execute transformations
- `w` to write a fi le
- `q` to quit the editor.

There can be zero or more arguments for each operation to fi ll in details. The range in the fi rst `x` command above [line 3 of Equation (9)] is defi ned in a C-style conditional expression of the form

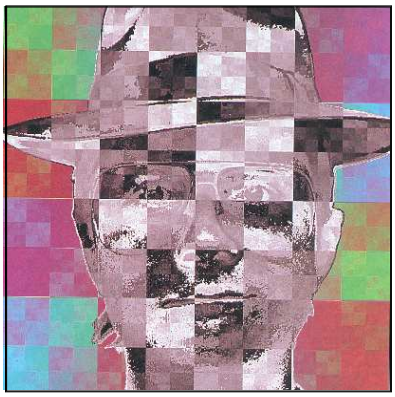

Figure 6. Equation 11.

Figure 7. Equation 13.

```
(condition) ? (iftrue) : (iffalse).
```

The expression defines an average of two images for all pixels with an x-coordinate less than 256 (i.e., the left-most 256 columns of the pixel array) and a negative of one of the pictures for all other pixels. (Compare with Figure 4.)

Line 4 simply turns the picture upside down; Y is the predefined constant for the height of the current image. Similarly, X gives the maximum x value, so we can fade `$rob` slowly into `$pjw` by writing a nested conditional (Figure 5):

$$\texttt{new} = (x < X/3)?\$\texttt{rob}: (x > X * 2/3)?\$\texttt{pjw}: 3 * ((x-X/3) * \$\texttt{pjw} + (X * 2/3-x) * \$\texttt{rob})/X \quad (10)$$

This expression language turns out to be a powerful editing tool. In PICO, all arithmetic and Boolean operators from C have been implemented, allowing seemingly nonsensical transformations such as those created by the following. [Equations (11), (13), and (14) produce the photos shown in Figures 6, 7, and 8 respectively. Some of these illustrations were colorized separately using PICO.]

$$\texttt{new} = x\ y\ \$\texttt{rob}. \quad (11)$$

$$\texttt{new} = (\$\texttt{rob} > 100)?\$\texttt{rob}:\$\texttt{pjw} \quad (12)$$

$$\texttt{new} = \$\texttt{rob}[x + (64-((x\&15)-8) * ((x\&15)-8))/8, y] \quad (13)$$

$$\texttt{new} = \$\texttt{pjw}[x + (64-(\$\texttt{rob}\&15) * *2)/8, y + (64-((\$\texttt{rob} >> 4)\&15) * *2)/8] \quad (14)$$

These transformations can produce startling effects. Equation (14), for example, distorts `pjw`'s portrait by somewhat arbitrarily smearing pixels, using a second portrait to calculate indices. (The operator $**$ in (14) is for exponentiation.)

STRUCTURE

Figure 9 shows how PICO is structured. Commands are either global operations that affect PICO's environment (such as attaching or deleting files) or transformation expressions. A parse tree for a transformation is passed to an optimizing compiler that generates machine code. The code is then executed, and typically completes within a few seconds. The result of the edit operation is shown on a monitor (if available), and the user is prompted for another command.

COLORS, COMPOSITES, AND TYPES

A pixel is not, in most cases, defined by a single brightness value but a composite of three colors. A composite is written in PICO as a comma-separated list of subexpressions enclosed in square brackets:

```
[red, grn, blu ]
```

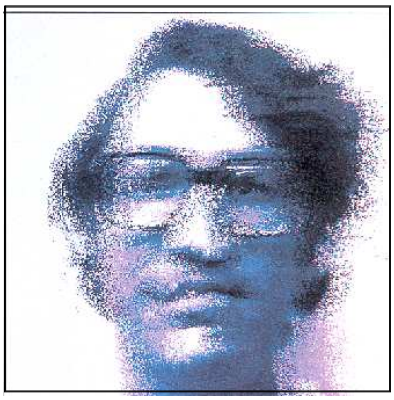

Figure 8. Equation 14.

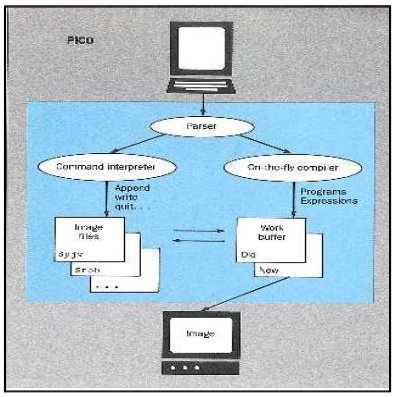

Figure 9. PICO Structure.

`Red`, `grn`, and `blu` are again predefined terms that can be used as suffixes to pixel array names or file names. For example:

```
new.red, old[x,y].red, $pjw.grn, ... etc.
```

Where necessary, PICO performs casting operations to match the types of the various parts in an assignment. For example, if the left-hand side of an expression is a value and the right-hand side a composite, the value is promoted to a composite before the assignment is performed.

With composites, we can readily rotate the colors in an image with a single assignment

$$\texttt{new} = [\texttt{old.blu}, \texttt{old.red}, \texttt{old.grn}] \tag{15}$$

The expression

$$\texttt{new.red} = \texttt{Z} - \texttt{old.blu} \tag{16}$$

assigns a negative of the old blue component in the picture to the red field of each pixel. Both sides of the assignment are noncomposite, so no type casting is performed. However, the expression

$$\texttt{new.red} = \texttt{old} \tag{17}$$

casts the composite `old` into a single value, the average of the three color components, so that the above expression is interpreted as

$$\texttt{new.red} = (\texttt{old.red} + \texttt{old.grn} + \texttt{old.blu})/3 \tag{18}$$

The reverse

$$\texttt{new} = \texttt{old.grn} \tag{19}$$

requires type promotion and is interpreted as

$$\texttt{new} = [\texttt{old.grn}, \texttt{old.grn}, \texttt{old.grn}] \tag{20}$$

PICO PROCEDURES

For nonstandard editing operations that can not easily be expressed in the language described above, there is a facility in PICO to declare named segments of code and use these as procedures. In the following edit script, a procedure that draws a circle of radius `r` is first declared and then called with argument 200.

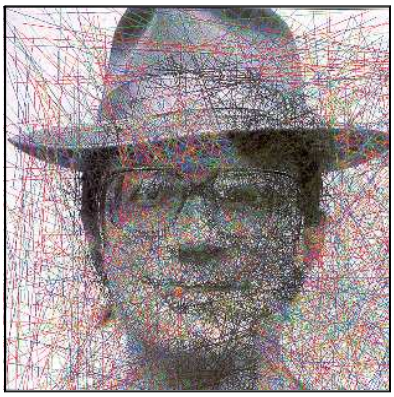

Figure 10. Random Lines.

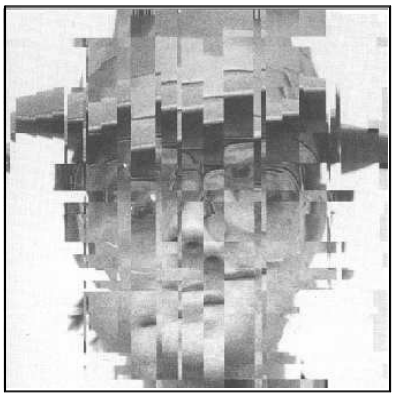

Figure 11. Random Slicing.

```
% pico
1: def circle(r)
{  int a

   for (a = 0; a < 360; a++)
     new[x_cart(r, a), y_cart(r, a)] = Z
}
3: x { circle(200); }
4:
```

There is one variable a declared local to the procedure `circle`. Statements are separated by new lines and/or semicolons. The function call in line 3 should only be executed once, and not by default once for every pixel. A statement, or a sequence of statements enclosed in curly braces, is taken to be an explicit PICO program defi ning its own control flow and is executed just once. Note that the code segment in curly braces that defi nes procedure `circle` is preceded by the keyword `def`. It is not executed, just compiled.

Procedures can be recursive. The following procedure, for example, approximates a straight line:

```
def line(x1, y1, x2, y2)
{   if (abs(x1 − x2) > 1 || abs(y1 − y2) > 1)
    {    line(x1, y1, (x1+x2)/2, (y1+y2)/2)
         line((x1+x2)/2, (y1+y2)/2, x2, y2)
    } else
         new[x1,y1] = 255
}
```

But we can do much better. The following set of procedures, written by Rob Pike of AT&T Bell Laboratories, made the cover of the November 1985 issue of *Computer Graphics*. The program draws *n* random lines on the display. Procedure `line` selects lines of pixels from the image of `pjw` randomly varying their intensity. Procedure `draw` calls library procedure `rand()` to select the parameters for `line.` (A color version of `draw` was run superimposed on `$rob`'s picture in Figure 10.)

Figure 12. Equation 21.

Figure 13. Edge Detection.

```
def xline(a0, b0, a1, b1)
{
    int a, b
    int r
    r=255-(rand()&127)
    for (a=a0; a<a1; a++)
    {
        b=b0+(b1-b0)*(a-a0)/(a1-a0)
        new[a, b] = $pjw[a, b]*r/255
    }
}

def yline(a0, b0, a1, b1)
{
    int a, b
    int r
    r=255-(rand()&127)
    for (b=b0; b<b1; b++)
    {
        a=a0+(a1-a0)*(b-b0)/(b1-b0)
        new[a, b] = $pjw[a, b]*r/255
    }
}
```

```
def line(a0, b0, a1, b1)
{
    if(abs(a1−a0)>abs(b1−b0))
    {   if(a0>a1)
            xline(a1, b1, a0, b0)
        else
            xline(a0, b0, a1, b1)
    } else
    {   if(b0>b1)
            yline(a1, b1, a0, b0)
        else
            yline(a0, b0, a1, b1)
    }
}

def draw(n)
{
    int a
    for (a = 0; a < n; a++)
        line(rand()&511, rand()&511, rand()&511, rand()&511)
}
```

Another inspiring procedure is shown below. (See results in Figure 11.)

```
def xy_slicer()
{   int r
    array yshift[512]
    for (y = 0; y < 512; y++)
    {
        if (rand() < 2000)
           r=(rand()&63)−32
        yshift[y] = r
    }
    for (y = 0; y<512; y++)
    {
        if (rand() < 2000)
           r=(rand()&63)−32
        for (x = 0; x<512; x++)
           new[x,y] = old[clamp(x+r), clamp(y+yshift[x])]
    }
}
```

`Clamp` is a built-in function defi ned as follows:

```
clamp(a)
{
   return ((a < 0)?0:((a>511)?511:a));
}
```

QUICK EDGE DETECTION

As a more serious application of the PICO picture editor, consider the following method to extract edges quickly and interactively from a black-and-white image. One method is to add positive and negative, with a small offset of perhaps two pixels in the x and the y direction (Figure 12):

$$\texttt{new} = \$\texttt{rob}[x, y] + (Z - \$\texttt{rob}[x + 2, y + 2]) \tag{21}$$

A slightly more sophisticated method is to smooth the image fi rst with a moving average. (We will

ignore here that the moving average can be calculated a factor of N^2 faster.)

```
def smooth(N)
{
    int a, b, c
    int N2

    N2 = 4*N*N
    for (y = N; y < 512−N; y++)
    for (x = N; x < 512−N; x++)
    {
        c = 0
        for (a = y−N; a < y+N; a++)
        for (b = x−N; b < x+N; b++)
            c += old[b, a]

        new[x,y] = c/N2
    }
}
```

This is followed by a quick approximation of the Laplacian ∇^2

```
def laplace()
{
    int a

    for (y = 0; y < 512; y++)
    for (x = 0; x < 512; x++)
    {
      a = 5*old[x,y]−old[x−1,y]−old[x+1,y] −old[x,y−1]−old[x,y+1]
      new[x,y] = (old[x,y] > a)?0:Z
    }
}
```

These two procedures can now be applied as follows (Figure 13):

```
% pico
1: x new=$rob
2: x { smooth(10); laplace(); }
3:
```

Figures 14 and 15 show the intriguing results of the following transformation applied to two portraits from our database (Al Aho and Doug McIlroy).

$$\texttt{new} = \texttt{old}[\texttt{x\_cart}(\texttt{r} = \texttt{sqrt}(256 * \texttt{r\_polar}(\texttt{x}, \texttt{y})), \texttt{a} = \texttt{a\_polar}(\texttt{x}, \texttt{y})), \texttt{y\_cart}(\texttt{r}, \texttt{a})] \quad (22)$$

PICO is most conveniently used interactively with a frame buffer display to show the result of each editing operation. Nevertheless, it can also be used without a display. The contents of the edit buffer can then, from time to time, be dumped into a picture file and, using halftoning, be displayed on a more standard graphics terminal such as the AT&T 5620 DMD (dot-mapped display terminal). At the time this article was written, two types of frame buffers were supported by PICO. The number is likely to increase when different types become available.

Figure 14. Equation 22.

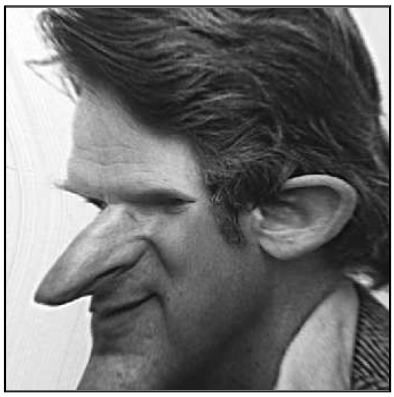

Figure 15. Equation 22.

Table I. Run Times

Transformation	Interpreter	Compiler	Int/Com
`new=128`	23.6s	5.3s	4.4
`new=$pjw`	43.3s	5.9s	7.3
`new=Z-old`	59.9s	6.5s	9.2
`new=($pjw+$rob)/2`	105.9s	9.3s	11.4
`new=(x<256)?$pjw:$rob`	107.7s	7.2s	14.9
`new=($pjw<128)?Z-$pjw : $pjw[X-x,y]`	304.5s	10.8s	28.2

SPEED

To make PICO truly useful as an interactive program, the execution speed of the transformations is, of course, a major concern. A statement such as `new = Z-old` for a color image (requiring 786,4321 values to be updated) takes 27.7s to execute. The same statement run for a black-and-white image takes 6.5s. By default, however, PICO will update the frame buffer display during the execution, which slows it down but is often psychologically more rewarding than the faster blind run.

The first version of PICO used an interpreter instead of a compiler. It is a small, portable program of only 312 lines of source text (Yacc, Lex, and C code), compared to the 5,761 lines of source for the latest version. The interpreter itself is 39 lines of C, simple, but without claims on efficiency. The differences in speed between this program and the version of PICO with the built-in optimizing compiler can be dramatic. Table I lists a few examples, giving user plus system times on a VAX-750 computer, for transformations of 512 by 512 black-and-white images.

Because each expression is evaluated once for every pixel, even small differences in run time per pixel are noticeable when the expression is evaluated 262,144 times over. Portable PICO (*popi*) has been ported to an AT&T PC 6300 PLUS personal computer and to a Cray X-MP/24 supercomputer. On any other system than a Cray, the portable version of PICO is clearly not very attractive as an interactive tool.

CONCLUSION

We have sketched the design of a language and an editor for transforming digitized images. The editor has proven to be a useful tool for quickly testing or exploring image transformation

techniques, providing a simple alternative to the often very costly commercial image processing systems.

PICO's expression language is largely derived from the C language. The most interesting aspects of the language are the defaults for control flow, array indexing, and type casting, that allow for a terse command structure. There are, of course, limitations. The language as implemented does not use floating-point operations. With an extension of the compiler, though, this restriction can be overcome. A more fundamental restriction is that the editor is an image manipulator, and is clumsy for image synthesis. PICO is an electronic darkroom, not a graphics camera.

ACKNOWLEDGEMENTS

The design of PICO's command language was decisively influenced by Rob Pike. The all-important built-in compiler that converts expressions and programs into VAX machine code, and thereby speeds up PICO's picture transformations by orders of magnitude, was written by Ken Thompson.

(Manuscript received May 28, 1986)

Author Biography

Gerard J. Holzmann is a member of the technical staff in the Computing Techniques Research Department of AT&T Bell Laboratories in Murray Hill, New Jersey. Mr. Holzmann joined AT&T in 1980. His current research is in distributed systems and formal methods for protocol analysis. He holds B.S. and M.S. degrees in electrical engineering from Delft University of Technology, The Netherlands, and a Ph.D. in technical sciences from Delft University of Technology.

FIGURE CAPTIONS

Figure 1. Photo of Rob Pike, unedited. Referred to as `$rob` in equations and text.

Figure 2. Photo of Peter J. Weinberger, unedited. Referred to as `$pjw`.

Figure 3. Photo created by average of `$rob` and `$pjw`. [See Equation (7).]

Figure 4. Photo resulting from simple conditional expression.

```
new = ($pjw<128) ? Z-$pjw : $pjw[X-x,y]
```

Figure 5. Photo showing more complex mapping and merging of two images.

```
def newy() { return clamp(4*y/3-75); }
{ global int R,L,yy;R = X/3;L = 2*X/3;}
  new = (x < R) ? $rob[x,yy=newy()]: \
        (x > L) ? 2*$pjw/3: \
        3*((x-R)*2*$pjw/3+(L-x)*$rob[x,yy])/X
```

[Compare with Equation (10).]

Figure 6. *Exclusive or* operation of `x, y,` and `$rob`. [See Equation (11).]

Figure 7. Transformation. [See Equation (13).]

Figure 8. Pixel smearing. [See Equation (14).]

Figure 9. Diagram of PICO's internal structure.

Figure 10. Photo with random lines, using `$rob` and `$pjw`.

Figure 11. Photo with random slicing, using `$rob`.

Figure 12. Photo with simple relief. [See Equation (21).]

Figure 13. Photo demonstrating edge detection, using `$rob`.

Figure 14. Caricature mapping using the following expression.

```
new=old[x_cart(r=sqrt(256*r_polar(x,y)), \
           a=a_polar(x,y)),y_cart(r,a)]
```

Figure 15. Caricature mapping.