Pico — A Language For Composing Digital Images

Gerard J. Holzmann

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Pico is a small expression language for picture compositions. It can be used either interactively with a display or stand alone as a picture file editor. The following *pico* script, for instance, turns an arbitrary digitized image stored in a file `in` upside down, rotates it by 90 degrees, and writes its negative into a file `out`:

```
$ pico
1: new = Z-$in[X-y,x]
2: w out
3: q
$
```

Numerous examples of *pico* transformations of pictures are included throughout this volume.

1. Black&White Images

The pictures that can be manipulated by *pico* are stored as regular files in the picture file format described in *picfile*(5). The picture editor is most conveniently used interactively with a Metheus frame buffer display. The result of picture transformations is then directly visible and can be used to correct or enhance the mistakes that one is bound to make.

New and Old

Assuming that you want to work interactively and have access to one of the Metheus frame buffers, e.g. on `pipe`, a session with *pico* can be started by typing

```
$ pico -mN
b&w, 512x512 pel, Metheus display
1:
```

where *N* is the device number of the frame buffer to be used. By default, *N* is zero and opens `/dev/om0`. Also by default the size of the workarea is 512x512 pixels. If you need to work with larger images, you can override the default by explicitly setting a different window width and height, for instance:

```
$ pico -m5 -w1024 -h1024
b&w, 1024x1024 pel, Metheus display
1:
```

The maximum size image that the Metheus frame buffer can display is 1280x1024 pixels.

The number followed by a colon in the example above is *pico*'s prompt for commands.

The result of the last edit operation (initially an all black image) is accessible under the predefined name `old`, and the destination of the image transformations is known as `new`. To quickly get a picture into the workbuffer, you can use the command `get` followed by the name of the file with the image.

```
1: get "pjw"
```

The most frequently used command in *pico* is `x`, short for `execute`. To make a black&white negative from the current picture the command would be:

```
2: x new=Z-old
```

where Z is a predefined constant with the value of maximum white (255). By default the transformation is applied to every pixel on the screen.

Assume we have two image files with portraits. We can open these files by using `get`, or we can specify them on the *pico* command line, as follows, using /dev/om5:

```
$ pico −m5 ./face/rob ./face/pjw
b&w, 512x512 pel, Metheus display
1:
```

We can create a new image, for instance, by averaging the two faces:

```
$ pico −m5 ./face/rob ./face/pjw
b&w, 512x512 pel, Metheus display
1: x new=($rob+$pjw)/2
2:
```

The transformation is written as an assignment of an expression to the destination new. Names preceded by a dollar sign refer to picture files, for instance as specified on the command line. A long name such as ./face/rob can be abbreviated to its base name rob (the part following the last slash). Not all file names have to be provided on the command line though. We can also append a new file doug, without reading it, by typing:

```
2: a "doug"
```

The double quotes are necessary. They avoid confusion when, for instance, / symbols are part of the filename. We can check which files are currently open by typing f:

```
3: f
$0: old color   resident
$1: rob  b&w resident
$2: pjw  b&w resident
$3: doug b&w absent
```

The numbers in the first column serve as a shorthand for the file names. Typing $1 therefore is equivalent to typing $rob. We will use both notations $1 and $rob below. The first line $0 is a shorthand for old and refers to the workarea (the screen in interactive usage).

We have a black and white image on the screen that is an average of the two files rob and pjw. To see rob separately we could type:

```
4: x new=$1
```

but that is hardly an inspiring procedure. Let's just take the left half or rob's face combined with the right half of peter's:

```
5: x new=(x<256)?$rob:$pjw
```

or to make a mirror image:

```
5: x new=(x<256)?$rob[x,y]:$rob[X−x,y]
```

The variable x used in the expression is predefined. Don't confuse it with the first x on the command line which identifies the type of command to be executed. The variable x gives the current x-coordinate of a pixel during transformations. Since in this case 512 pixels fit on one scan line, a pixel in the middle of the screen has an x-coordinate of 256. The maximum value of x is given by a predefined variable X. X/2, therefore, is a safer way to specify the middle of a scan line.

The expression above is a conditional of the form:

condition ? *iftrue* : *iffalse*

For every screen position where the condition holds the *iftrue* part of the expression applies and everywhere else the *iffalse* part applies. Another predefined variable of this type is y (the y-coordinate of the destination). The maximum y value is called Y. Since $0 refers to the screen we can turn the picture on the screen upside down by typing:

```
6: x new = $0[x, Y−y]
```

All pixels in the black&white picture are internally represented by a value in the range 0..255, where 0 means black and 255 means white. To reverse an image, therefore it would suffice to subtract the current value of each pixel from its maximum value 255, which is stored in constant Z. Getting very bold we can turn the picture on its side, and make it negative by saying:

```
8: x new = 255 − old[y,511−x]
```

or, slightly more abstract

```
9: x new = Z − old[y,Y−x]
```

Note that we swapped x and y to turn the picture on its side. Nothing can stop us now:

```
10: x new=(x<X/3)?$1:(x>X*2/3)?$2:
        3*((x−X/3)*$2+(X*2/3−x)*$1)/X
```

fades rob slowly into pjw. (We have used   to break the line into two pieces for lay-out purposes. When using *pico*, it should be typed as one complete line.) Actually this last transformation is easier to read as a little *pico* program. To see how this works, and what the defaults in the above expression are, the above expression could be typed as:

```
10: x {
        int L, R

        L = X/3; R = X*2/3

        for (y = 0; y < Y; y++)
        for (x = 0; x < X; x++)
        {       if (x < L)
                        new[x,y] = $1
                else if (x > R)
                        new[x,y] = $2
                else
                        new[x,y] = 3*((x-L)*
                                $2+(R-x)*$1)/X
        }
}
```

There fewer defaults here, though an assignment to `new` is still interpreted as a parallel assignment to all three color channels in the picture. You can override also these defaults by making the program still more explicit, for instance by using the suffixes `red`, `grn`, and `blu` to access color channels separately: `new[x,y].red`, `new[x,y].grn`, and `new[x,y].blu`. To see the effect you need to set the workbuffer to color mode first with the command `color`. (You go back to the default black&white mode with the command `nocolor`).

All normal arithmetic operators from C are available. The ^ operator, for instance, makes an *exclusive or* of its operands. Thus,

```
11: x new=x^y^$rob
```

is a particularly striking effect, and

```
13: x new = $rob +(Z - $rob[x+2, y+2])
```

is an attempt to make a relief.

There is no range checking on explicit or implicit array indexing. The use of `x+2` in the last expression is therefore risky and is best protected with a conditional:

```
13: x new=$rob+(Z-(x<509 && y<509)?
                $rob[x+2,y+2]:Z)
```

or more conveniently with the builtins `xclamp` and `yclamp`:

```
14: x new=$rob+(Z-$rob[xclamp(x+2),
                yclamp(y+2)])
```

Another promising attempt to make a core dump would be to type something like

```
14: x new=$rob[x*y, x/y]
```

2. Color Images

A complete picture specifies pixel values for each of three separate color channels: red, green, and blue. When the editor is used in black&white mode only the red channel is used. When a black&white picture is converted to color mode, all three channels are made equal. The omission of a channel suffix to `old`, `new` or a file name is similarly interpreted to mean that a transformation expression will apply equally to all three color channels. By specifying an explicit suffix `red`, `grn`, or `blu`, however, we can write each channel separately. So:

```
14: color
15: x new.red=$rob
16: x new.grn=$rob
17: x new.blu=255-$rob
```

will write `rob` on the red and green channels, and its negative on the blue channel. If `rob` is a black&white picture then typing `$rob` is, of course, equivalent to typing `$rob.red`. We could also have combined the first two lines in a chain assignment:

```
18: x new.red=new.grn=$rob
```

We can also write a separate value to each channel by using `composites`. A color composite is written as a comma separated list of three values, enclosed in square brackets:

```
19: x new.rgb=[$rob,$rob,Z-$rob ]
```

The channels are addressed by the three fields of the composite in the order: [red, green, blue]. Omitting to specify a composite when an `rgb` destination is used typically results in only the red channel being written. As expected,

```
20: x new.rgb=[old.grn,old.blu,old.red ]
```

rotates the colors of the picture. And, of course, you can freely combine the color suffixes with array indexing: `$rob.blu` is just a shorthand for `$rob[x, y].blu` where the variables `x`, and `y` can be replaced by just any monstrous C-style expression.

3. The Color Maps

When working interactively, the color map in the Metheus frame buffer display can be set with one of the commands `cmap` (all channels), `cmap.red`, or`cmap.grn`, `cmap.blu`. The color map is a mapping table that can arbitrarily map pixel brightness values in the range 0..255 to other brightness values, within the same range 0..255. The update, however, only happens on the screen and is not stored when the image file is written. The variable `i` is used to index the color map. For instance:

```
21: x cmap = Z-i
```

will very quickly make a negative, and

```
22: x cmap = i
```

turns the picture back to normal. To fake color in a black and white image you can try:

```
23: x cmap.red = (i<=85)?i:0
24: x cmap.grn = (i>85 && i<170)?i:0
25: x cmap.blu = (i>=170)?i:0
```

Remember that changing the color map only changes the appearance of the picture on the screen, not its definition in memory.

4. Read, Write, and Windows

The `append` command, to add files to the list of dollar arguments was discussed before. Using the command `get` instead of `a` will also put the file into $0, that is on the screen. To save the current state of the display in a file, use:

```
26: w filename
```

A raw black&white picture file, without the picture file header, can be written by using `w -` instead of `w` (for instance when dumping a file to be processed by software uneducated in picture file headers). The size of the file written conforms to the current window size of the editor (see also below). To close a no longer used file and free up some memory for others, say:

```
27: d doug
```

or

```
27: d $1
```

giving the file's base name or its dollar number. To restrict the updates to a window on the screen you can set:

```
30: window 10 100 200 300
```

which makes a window with origin at (x,y) = (10,100), 200 pixels wide and 300 pixels deep. And, if you really want to exit *pico* you can type a control-D or resort to the `quit` command:

```
32: q
```

5. Programs

As shown in one of the examples above, it is possible to write small *pico* programs for the more difficult transformations that cannot be handled by the defaults. The control structure for the *pico* programs is again stolen from C.

A *pico* program starts with a left curly brace `{` followed by zero or more declarations of (long) integers or arrays. For instance,

```
33: x {
        int a, b; array ken[100]

        ...
```

Note carefully that the left curly brace turns off the default control flow over all the pixels in a picture; the control flow in a program must be specified explicitly. The above program fragment declares two local integers `a` and `b` which by default will be initialized to zero, and an array of 100 (long) integers named `ken`, also initialized to zeros. To initialize it to another value, use constants:

```
int a = 9
```

Statements can either be separated by newlines or by explicit semicolons. Here then is a list of valid types of statements:

lvalue = *expr*
`if` (*expr*) *stmnt*
`if` (*expr*) *stmnt* `else` *stmnt*
`for` (*expr*; *expr*; *expr*) *stmnt*
`while` (*expr*) *stmnt*
`do` *stmnt* `while` (*expr*)
label: *stmnt*
`goto label`
{ *stmnt* }

An *lvalue* is an explicitly declared local variable or array element, one of the predefined variables `x`, or `y`, or a picture element. A picture element can again be a file name such as `$doug` with a default selection of the pixel inside it, `$doug[x,y]`, or it can be more elaborate as in `$doug[x/2, y<<1].red`. The destination of the transformation, is again referred to by the keyword `new`, but this time it needs an explicit array indexing. The equivalent of

```
34: x new=old[y,x]
```

to turn the picture on its side, can be written as a *pico* program:

```
35: x {
        for (y = 0; y < 512; y++)
        for (x = 0; x < 512; x++)
                new[x,y] = old[y,x]
}
```

Note that also the control flow must be explicit. Typing only

```
36: x { new[x,y] = old[y,x] }
```

would use the initial (zero) values of `x` and `y` and merely assign `new[0,0] = old[0,0]`.

6. Array Indexing and Control Flow Defaults

One more word about defaults. *Pico* tries to be smart about assigning types to values. When a single rvalue is needed and a color composite is available and average of the color channels is the default, for instance:

```
oldbecomes
(old[x,y].red+old[x,y].grn+old[x,y].blu)/3.
```

If on the other hand a value is available and a composite is needed the value will be replicated into a fake composite. To override the defaults assignments can of course always be made more explicit. Normal cases should work as expected, for instance, by default:

```
new = old
```

truly means

```
new.red=old.red;
new.grn=old.grn;
new.blu=old.blu
```

7. Procedures

There is a facility in *pico* to declare named segments of code and use these as functions or procedures. As an example, the following command declares a procedure `doit` that makes a histogram of a window of pixels on the screen. It is equivalent to the sequence:

```
37: window a b w d
38: x { global array histog[256]; }
39: x histo[old]++
40: window 0 0 512 512
```

In a procedure this is written as:

```
37: def doit(a, b, w, d) {
        global array histog[256]

        for (y = a; y < a+w; y++)
        for (x = b; x < b+d; x++)
                histog[old[x,y]]++
}
```

The declaration prefix `global` extends the scope of an array so that it can be referred to in subsequent procedures, programs or expressions. We can now call the procedure and use the histogram to arbitrarily change the color map:

```
38: x { doit(0,0,512,512); }
39: x cmap = histog[i]%256
```

Or more usefully, to calculate and apply a simple histogram equalization:

```
40: x {
        int ave, i, j, R, L, Hint;
        global array eqlz[256]

        for (i = ave = 0; i < 256; i++)
                ave += histog[i];
        ave /= 256

        for (i = R = Hint = 0; i < 256; i++)
        {       L = R
                Hint += histog[i]
                while (Hint > ave)
                {       Hint -= ave
                        R++
                }
                j = (L+R)/2
                eqlz[i] = (j>255)?255:j
        }
}
41: x new=eqlz[old]
```

When *pico* starts up it reads a small set of library procedures by simulating an `r` command (see Table 1). These definitions are stored in the file `/usr/lib/pico/defines`. The builtins listed in Table 2 can be used in *pico* programs and *pico* procedures.

8. Non-Interactive Use of *Pico*

When *pico* is used without having access to a frame buffer all commands will still work. To check the result of executing commands it can be convenient to write what would have been the current screen image into a file with the `w` command and view it in another *mux*-window on the 5620 (or 630) terminal with a command like *flicks*. For instance, in one window the session can be:

```
$ pico rob pjw
1: nocolor
2: x new=($rob*$pjw)/255
3: w - junk
4:
```

While in another window the file is displayed with *flicks*, as follows:

```
$ flicks -em junk
```

9. See Also

More examples on how to use *pico*, and on its internal structure, can be found in: |reference(pico %no_cite)|reference(digital darkroom %no_cite)

|reference_placement

COMMAND	
`a [x y w d] file`	append file with optional offset/dimensions
`d basename/$n`	delete file
`h file`	read header information from file(s)
`r file`	read (library) command file
`w [-] file`	write file or window with or without a header format: default = *pico* header, means no header
`nocolor`	update & display only 1 channel
`color`	update & display all 3 channels
`window x y w d`	restrict workarea to this window
`get [x y w d] file`	read file contents into `old`
`get $n`	refresh `old` with an already opened file
`f`	show mounted files
`show [name]`	show symbol information (values)
`functions`	show functions
`def name { pprog }`	define a function
`x expr`	execute expr in default loop
`x { pprog }`	execute *pico* program
`q`	quit

File names containing nonalphanumeric characters (period, slash) must be enclosed in double quotes.

Table 1. Command Summary

`printf(string, args)`	recognizes only: `%d`, `%s`, `\n`, `\t`
`x_cart(radius, angle)`	convert radius and angle (degrees: 0..360) into x_coordinate
`y_cart(radius, angle)`	convert radius and angle (degrees: 0..360) into y_coordinate
`X_cart(r,a),Y_cart(r,a)`	same as [*xy*]_*cart*, but expects *angle* in centidegrees
`r_polar(x,y)`	convert *x,y* coordinate into radius
`a_polar(x,y)`	angle returned is in degrees: 0..360, acuuracy=± 2 degrees
`A_polar(x,y)`	angle returned is in centidegrees: 0..36000
`putframe(nr)`	dump window into the file `"frame.%6d"`, *nr*
`getframe(nr)`	read from the file `"frame.%6d"`, *nr*
`setcmap(i, r,g,b)`	write *i*th value in colormap
`getcmap(i, r,g,b)`	read *i*th value in colormap
`redcmap(i, z)`	
`grncmap(), blucmap()`	
`sin(angle), cos(angle)`	returns 0..10,000, *angle* in degrees : 0..360
`Sin(angle), Cos(angle)`	same but expects *angle* in centidegrees: 0..36000
`atan(x, y)`	arc-tangent of *y/x*, returns angle in degrees: 0..360
`exp(a)`	as advertised
`log(a),log10(a)`	returns 1024*result
`sqrt(a)`	integer square root of *a*
`pow(a,b)`	*a* to the power *b*
`rand()`	returns a random integer *r*: 0f *r*<32768.

Table 2. Builtin Procedures