Pico Tutorial

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[Y−y,x]
2: w out
3: q
$
```

[Originally appeared as an AT&T Technical Memorandum, 8 October 1985, 14 pgs.]

1. Black&White Images

The pictures that can be manipulated by pico are either stored as UNIX® files or are held in a framebuffer memory. Pico is most conveniently used interactively with a framebuffer display. The result of picture transformations is then directly visible and can be used to correct or enhance the mistakes that the user is bound to make.

New and Old

Assuming that you want to work interactively and have access to one of the metheus framebuffers, e.g. on ‘pipe’, a session with pico can be started by typing

```
$ pico -mN
1:
```

Where N is the device number of the framebuffer to be used. By default, N is zero and opens /dev/om0.

The number followed by a colon and a space is pico’s prompt for commands. The result of the last edit operation (initially an all black image) is always accessible under the predefined name ‘old’, and the destination of all transformations to the picture is known as ‘new’. The most frequently used command in pico is ‘execute’ or ‘x’ for short. To make a black&white negative from the current picture the command would be:

```
1: 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, where pico keeps track of the simultaneous updating of x and y coordinates in both the new and the old image.

Assume we have two files with portraits of two colleagues. We can open these files by specifying them on the pico command line, and then create a new image by averaging the two faces:

```
$ pico -m2 ./face/rob ./face/pjw
1: x new=($rob+$pjw)/2
2:
```

The transformation is written as an assignment of an expression to the lvalue ‘new’. Names preceded by a dollar sign always refer to picture files, for instance as specified on the command line. A long name such as ‘/n/kwee/t0/face/512x512x8/rob’ is 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, for instance, append a new file ‘doug’ by typing:

```
2: a "/n/kwee/t0/face/512x512x8/doug"
```

(the double quotes are necessary) and we can check which files are currently open by typing ‘f’:

```
3: f
$0 old  color   resident
$1 rob  black/white resident
$2 pjw  black/white resident
$3 doug black/white 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 screen.

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 even to make a nice mirror image:

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

The variable ‘x’ used in the expression (not to be confused with the first ‘x’ on the command line which identifies the type of command to be executed) is predefined and gives the x-coordinate of the pixels on the screen. Since 512 pixels will fit on one scan line, a pixel in the middle of the screen has an x-coordinate of 256. We could also have used the predefined constant X, storing the maximum x value on a scan line (511) and use the expression ‘X/2’ to find the middle of a 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. Two other predefined variables of this type are ‘y’ (the y-coordinate of the destination) and ‘i’ (512*y+x). Since $0 refers to the screen we can turn the picture on the screen upside down by typing:

```
6: x new = $0[x, 511−y]
```

or equivalently:

```
7: x new = $0[512*512−i]
```

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,x]
```

or, slightly more abstract

```
9: x new = Z − old[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'. 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 really are, the above expression could also be typed as:

```
10: x {
        int left, right

        left = 512/3; right = 512*2/3

        for (y = 0; y < 512; y++)
        for (x = 0; x < 512; x++)
        {       if (x < left)
                        new[x,y] = $1
                else if (x > right)
                        new[x,y] = $2
                else
                        new[x,y] = 3*((x−left)*$2+(right−x)*$1)/512
        }
}
```

There far 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 the 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 though, using the command 'color'. You go back to the default black and white mode with the command 'nocolor' (see also below).

Of course, nothing tells us that we should only type things that make sense. 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,

```
12: x new = ($rob > $pjw) ? ($rob*$pjw)/Z : $doug
```

looks promising but is rather disappointing. But then,

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

is an attempt to make a relief that again works relatively well.

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 builtin 'clamp':

```
14: x new = $rob +(Z − $rob[clamp(x+2), clamp(y+2)])
```

A promising attempt to make a core dump would be to type something like: $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. For a black&white picture the three channels are equal. The omission of a channel suffix to 'old', 'new' or a file name is interpreted to mean that the 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. We could also have combined the first two lines in a chain single assignment:

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

We could also have written a separate value to each channel with the ‘rgb’ suffix and a ‘composite’ in square brackets:

```
19: x new.rgb = [ $rob, $rob, 255−$rob ]
```

The channels are addressed by the three fields of the composite in the order: [red, green, blue]. Omitting to specify a composite with an ‘rgb’ prefix 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: $rob.blu is just a shorthand for either ‘$rob[x, y].blu’ or ‘$rob[i].blu’ and the variables ‘x’, ‘y’, and ‘i’ in these can be replaced by just any monstrous C-style expression.

3. The Color Maps

When working interactively, the color map in the framebuffer can be set with one of the commands cmap (all channels), cmap.red, cmap.grn, or cmap.blu. The color map is a mapping table that assigns a pixel brightness in the range 0..255 to a pixel value as stored in the pixel array, also in the range 0..255. 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 a color picture 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
```

Note however that changing the color map only changes the appearance of the picture on the monitor, not its definition in the picture array.

4. Undo, Read, Write, and Windows

The undo command ‘u’ will restore the display (‘old’) to the state in which it was when pico was started.

The ‘append’ command, to add files to the list of dollar arguments was discussed before. ‘get’ instead of ‘a’ will 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 pico header, can be written by using ‘w −’ instead of ‘w’ (for instance when dumping a file to be viewed with an editor uneducated in picture file headers such as ‘flicks’). A file with an 1127 standard header (also known as the ‘td-header’) is obtained by typing ‘w +’ instead of ‘w’ (recommended for color pictures). 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. Pico Programs

As briefly shown in one of the examples above, it is possible to write small explicit 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. Data types, however, are largely absent.

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: in a program the control flow must be explicited. 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', 'y', and 'i', 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[i] †, or it can be more explicit as in $doug[x/2, y<<1].red. The destination of the transformation, is again referred to by the keyword 'new'. The equivalent of

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

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

† Note that is is the programmers responsibility to make sure that the variable i has the proper value. In particular, if only x and x are updated $doug will still default to $doug[i] and be equivalent all through the program to $doug[0,0] independent of x and y.

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

Note that the control flow must be explicited. 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. More On 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:

```
‘old’ becomes ‘(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. Pico 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 histogramme of a window of pixels on the screen. It is equivalent to the sequence:

```
37: window a b w d
38: x histo[old]++
39: window 0 0 512 512
```

For no solid reason "histo" is a predefined global array of 256 elements. 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 the array so that it can be referred to in subsequent procedures, programs or expressions. We can now call the procedure and use the histogramme to change the color map:

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

8. Non-Interactive Use of Pico

When pico is used without having access to a framebuffer 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 terminal with ‘flicks’. For instance, in one window the session can be:

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

While in another window ‘flicks’ is downloaded repeatedly as:

```
$ flicks junk
```

9. Appendix: Overview of Pico commands

Commands preceded by a star only have effect when the framebuffer display is open.

	COMMAND	DESCRIPTION
	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, - = raw, + = dump
	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,
*	faster,	update display 1 line iso 1 pel at a time
*	slower,	undo faster
	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.

Builtin Commands

printf(string, args)	- recognizes only: %d, %s, 0
x_cart(radius, angle)	- convert radius and angle into x_coordinate
y_cart(radius, angle)	- idem, y_coordinate, angle in degrees: 0..360
X_cart(r,a),Y_cart(r,a)	- same, but expects angle in hundreds : 0..36000
r_polar(x,y)	- convert x,y coordinate into radius
a_polar(x,y)	- angle returned is in degrees: 0..360 (+- 2 degrees)
A_polar(x,y)	- angle returned is in hundreds: 0..36000
putframe(nr)	- dump window into a file "frame.%6d", nr
getframe(nr)	- read from a file frame.%d (eg within a pico program)
setcmap(i, r,g,b), getcmap(i, r,g,b)	- write or read ith 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 hundreds: 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), pow(a,b)	- integer square root of a; a to the power b
rand()	- unix rand() function