The Icon Analyst

In-Depth Coverage of the Icon Programming Language

February 1999
Number 52

In this issue …

A Weaving Language — continued

May the warp be the white light of morning,
May the weft be the red light of evening,
May the fringes be the falling rain,
May the border be the standing rainbow.
Thus weave for us a garment of brightness.

– Native American Indian song [1]

In the first article on Painter's weaving language [2], we described nine of the 15 operators:

*	repeat
#	rotate
\	reverse
–>	extend
~	interleave
<	domain "upto"
>	domain "downto"
|	pattern palindrome
,	concatenate

Of the six remaining operators, the "block" operator, (operator symbol [ ]), probably is the most important in weaving. The left operand is a pattern. The right operand is a sequence of integers. Each character in the left operand is repeated individually by the corresponding integer in the right operand. For example,

```
1346[ ]9231
```

expands to

```
111111111334441
```

Integers greater than 9 can be specified by enclosing them in braces. For example,

```
1345[ ]12{10}3
```

expands to

```
1334444444444666
```

Here's a procedure that implements this operator:

```
procedure Block(p1, p2)
   local i, s, p3, counts

   counts := []

   p2 ? {
      while s := tab(upto('{')) do {
         every put(counts, !s)
         move(1)
         put(counts, tab(upto('}')))
         move(1)
         }
      every put(counts, !tab(0))
      }

   if *p1 < *counts then p1 := Extend(p1, *counts)
   else if *counts < *p1 then {              # extend list
      every i := seq() do {
         put(counts, counts[i])
         if *counts >= *p1 then break
         }
      }

   p3 := ""

   every i := 1 to *p1 do
      p3 ||:= repl(p1[i], counts[i])

   return p3

end
```

A list, counts, is used for holding the integers,

since they may be more than one digit long. If this were not the case, a string could be used with the same code structure.

Permutation, indicated by perm (no operator symbol) applies a permutation vector (right operand) to a pattern (left operand). The pattern is permuted in sections whose lengths are the lengths of the permutation vector. The permutation vector specifies the positions of the elements in a section. For example, 4123 puts the fourth character of the section first, the first second, the second third and the third fourth. Thus,

```
1346 perm 4123
```

expands to 6134.

In the case that the pattern is not the same length as the permutation vector, the pattern is extended to an *integer multiple* of the length of the vector.

Here's a procedure:

```
procedure Permute(p1, p2)
   local p3, chunk,  j

   j := *p1 % *p2
   if j ~= 0 then p1 := Extend(p1, *p1 + *p2 – j)

   p3 := ""

   p1 ? {
      while chunk := move(*p2) do
         every p3 ||:= chunk[!p2]
      }

   return p3

end
```

The pattern box operator, pbox (no operator symbol), like perm, has a left operand pattern and a right operand permutation vector. The permutation vector is extended to the length of the pattern and the permutation is applied. Here's a procedure:

```
procedure Pbox(p1, p2)
   local p3, i

   if *p2 ~= *p1 then p2 := Extend(p2, *p1)

   p3 := ""

   every i := !p1 do
      p3 ||:= p1[p2[i]]

   return p3

end
```

The template operator, :, provides for "sub-articulation" of a pattern (left operand) by a "texture pattern" (right operand). The first character (digit) in the template pattern is taken as the root. The remaining digits in the template pattern are taken with respect from their distance from the root. For example, in the template patterns 342 the root *r*, is 3 and the template is *r*, *r* + 1, *r* – 1. The template is applied to each character in the pattern with the character replacing the root. If this is unclear, an example may help:

```
12345678:121
```

has the template *r*, *r* + 1, *r* and expands to

```
12123234356567678 7818
```

Note that values wrap around on the domain, so that for the last character of the pattern, 8, *r* + 1 produces 1.

Here's a procedure for this operator:

```
procedure Template(p1, p2)
   local p3, dlist, i, j, k

   dlist := []

   every i := 1 to *p1 do
      put(dlist, p1[i] – p1[1])

   p3 := ""

   every j := 1 to *dlist do
      every i := 1 to *p2 do {
         k := p2[i] + dlist[j]
         if k > 8 then k –:= 8
         else if k < 1 then k +:= 8
         p3 ||:= k
         }

   return p3

end
```

The two remaining operators are related to "upto" and "downto", which were described in the last article [2].

The "updown"operator, <>, generates alternating ascending and descending domain runs. The first, ascending, run starts at the first character of the left operand and goes to the first character of the second operand. The second, descending, run starts from there and goes to the second character of the right operand, and so on, alternating between ascending and descending runs. For example,

```
1234<>5678
```

expands to

```
12345432345654345676545678
```

As in "upto", tick marks can be used to indicate domain cycles between runs.

Here's a procedure:

```
procedure UpDown(p1, p2)
   local p3, i, ticks

   ticks := ""

   p2 ?:= {
      ticks := tab(many('\''))
      tab(0)
      }

   if *p1 < *p2 then p1 := Extend(p1, *p2)
   else if *p2 <  *p1 then p2 := Extend(p2, *p1)

   p3 := p1[1]

   every i := 1 to *p1 do {
      p3 ||:= Upto(p1[i], ticks || p2[i])[2:0]
      p3 ||:= Downto(p2[i], ticks || p1[i + 1])[2:0]
      }

   return p3

end
```

The "downup" operator, ><, is like "updown" except that the order is descending, ascending, … .

Comments

That's it. It is interesting to note that weaving language, as rich as it is, does not have operations for some patterns that occur frequently in weaving. Two missing ones are true palindromes and the interleaving of more than two patterns.

Another thing to think about is other domains. Although some of Painter's built-in weaves use only four shafts and four treadles, and only the labels 1, 2, 3, and 4, there is no way to restrict domain runs to this subset. For example, 4<2 produces 4567812, not 412 as it would if the domain could be restricted. Of course, 5678 does nothing.

The restriction to 8 shafts and 8 treadles is more fundamental, since it limits the kinds of things that can be woven. Many looms have more than 8 shafts, and there usually are more treadles than shafts. As mentioned in the last article on weaving, more shafts and treadles could be handled by extending the domain to include more labeling characters. If the number of shafts and treadles is not the same, the situation becomes more complicated, especially for domain operations.

Next Time

In the next article on weaving, we'll look at ways of representing weaving specifications and how patterns and structures in them can be made evident. This will lead to weaving grammars.

References

1. *Song of the Sky Loom*, Tewa tribe.

2. "A Weaving Language" Icon Analyst 51, pp. 5-11.

Quiz — Structures

With this issue of the Analyst, we're starting a series of quizzes. We'll cover a variety of topics, mostly about the Icon language but also including some other material from past issues of the Analyst. We'll use ornate illuminated Qs like the one above to identify quizzes.

In general, we'll provide answers in the subsequent issue of the Analyst, but for those of you who don't want to wait, we'll also put the answers on the Web pages for the issues of the Analyst in which the quizzes appear.

This quiz relates to various aspects of structures. The letters L, R, S, and T are used to identify values of type list, record, set, and table, respectively.

1. True or false:L[0] references the last element of the list L.

2. True or false:L[i:j] produces a new list that is distinct from L.

3. What is the difference between

```
every write(!L)
```

and

```
every write(get(L))
```

4. What does put(L) do?

5. What do the following expressions do?

```
every !L do
  put(L, get(L))

every !L do
  push(L, get(L))

every !L do
  push(L, pull(L))

every !L do
  put(L, pull(L))
```

6. Write an expression that removes duplicate elements from L and puts the rest in sorted order.

7. True or false: All of these expressions reference the second element of L:

```
L[2]
L["2"]
L['2']
L[2.2]
```

8. True or false: A record declaration must have at least one field.

9. True or false: Two record declarations can have the same field name only if it is in the same position.

10. True or false:R.center and R["center"] are equivalent.

11. True or false:R.1 can be used to reference the first field of R.

12. copy(R) produces a record with the same type as R.

13. True or false: The expression

```
copy(R) === R
```

may succeed in some circumstances.

14. True or false:!S generates the elements of S in random order.

15. Suppose*S didn't work. How could you find out how many elements there are in S?

16. Write a procedure that removes all the strings in S.

17. set(L) produces a set consisting of the distinct values in L. What do set(R), set(S) and set(T) do? What do list(S) and table(S) do?

18. True or false: All the keys in a table are distinct.

19. True or false: All the values in a table are distinct.

20. Write a procedure that creates a set containing the keys of table T.

21. key(T) generates the keys in table T and !T generates the values. Why is key(T) more fundamental than !T?

22. True or falsekey(T) generates the keys in T in the same order as !T generates the values.

23. True or false: There is no limit to the number of elements that a table can contain.

Pattern Forms Revisited

Pattern is born when one reproduces the intuitively perceived essence. — Saetsu Yanaki [1]

When we introduced pattern forms as a way of describing the structure of character patterns [2], we started with only two pattern forms: repetition and reversal. Later we added collation [3]. With just a few pattern forms and no specific plans for others, it seemed reasonable to chose the syntax to make them easy to distinguish:

[s,i]	repeat
<s>	reverse
{s1,s2, …}	collate

Pairs of bracketing characters are needed to handle nesting. Square brackets, angular brackets, and braces are natural and visually appropriate. (You may recall we started with parentheses for repetition and then decided to save them for more conventional uses, such as grouping.)

The weaving language [4,5] opened the door to a host of other pattern forms that not only are useful in describing the structures of weaving specifications but also other kinds of character patterns. Of the 15 weaving operators, two, repetition and reversal, are part of our original set, and the weaving interleaving operation is a special case of collation. The explicit concatenation operator in the weaving language is covered by implicit concat-

enation in pattern forms. That leaves 11 operators that are candidates for pattern forms. Of these, five seem likely to be useful in other contexts:

–>	extend
[ ]	block
|	pattern palindrome
perm	permute
#	rotate

On the other hand, to develop weaving grammars in the fashion we developed versum delta grammars [6] and image grammars [7], we would need all 15 weaving operators cast as pattern forms. But even with only five more pattern forms, the syntax we used previously won't work — the ASCII character set has a paucity of bracketing characters or even characters that reasonably might be used as such. Given that the weaving language has suggested useful new pattern forms, it's also likely that other specialized applications might also. So we need a syntax that can accommodate many pattern forms and is easily extensible.

Here's what we came up with: one pair of bracketing characters for all pattern forms (square brackets) with different operator symbols between the operands to distinguish different pattern forms.

In the absence of a better set of operator symbols, we've used those from the weaving language, making only a few necessary changes. Here's the complete set (so far):

[s*i]	repeat
[s`]	reverse
[s#i]	rotate
[s1~s2~…]	collate
[s1–>s2]	extend
[s1+s2]	block
[s|]	pattern palindrome
[s1<s2]	upto
[s1>s2]	downto
[s1–s2]	upto or downto
[s1?s2}	permute
[s1%s2]	pbox
[s1<>s2]	updown
[s1><s2]	downup
[s1:s2]	template
[s1!s2]	true palindrome

In the case of pattern forms that have only one operand, it appears first. (These invite extension to two operands; we'll resist that temptation for now.) The collation pattern form allows multiple operands; if there are only two, it corresponds to the weaving interleave operation. The true palindrome pattern form was added to complement the pattern palindrome pattern form.

Note that the use of multi-character operators leaves the syntax open-ended with no limit to the number of possible operators.

Adding these pattern forms removes more characters from those available to name variables in character grammars. The full set of pattern-form meta-characters now is:

[ ] * ` ~ + – > < | ! ? % # :

The additional meta-characters are not a major loss for character grammars: They make poor variable names anyway.

The following procedure shows how pattern forms can be expanded into strings:

```
link strings
link weaving

procedure pfl2str(pattern)
   local result, expr1, expr2, op
   static symbols, optbl

   initial {
      symbols := '[]*`~–><+|#:!'

      optbl := table()

      optbl["*"]   :=  repl
      optbl["`"]   :=  reverse
      optbl["#"]   :=  rotate
```

```
      optbl["<"]   :=  Upto
      optbl[">"]   :=  Downto
      optbl["–"]   :=  UpDownto
      optbl["|"]   :=  PatternPalindrome
      optbl["!"]   :=  Palindrome
      optbl["+"]   :=  Block
      optbl["~"]   :=  Collate
      optbl["–>"]  :=  Extend
      optbl["~"]   :=  Interleave
      optbl[":"]   :=  Template
      optbl["?"]   :=  Permute
      optbl["%"]   :=  Pbox
      optbl["><"]  :=  UpDown
      optbl["<>"]  :=  DownUp
      }

   result := ""

   pattern ? {
      while result ||:= tab(upto('[')) do {
         move(1)
         expr1 := pfl2str(tab(bal(symbols, '[', ']'))) |
            error()
         op := tab(many(symbols)) | error()
         expr2 := pfl2str(tab(bal(']', '[', ']'))) | error()
         result ||:= \optbl[op](expr1, expr2) | error()
         move(1)
         }
      if not pos(0) then result ||:= tab(0)
      }

   return result

end
```

The table optbl contains the functions and procedures to be applied for the different pattern forms. As usual, recursion is used to handle nesting.

Unary suffix operators are treated as binary operators with an empty right operand. If the right operand is not empty, it is ignored except for Palindrome(), in which the second argument provides the middle.

Collate() handles multiple operands in an *ad hoc* fashion:

```
procedure Collate(s1, s2)
   local slist

   slist := [s1]              # s1 has been expanded

   s2 ? {
      while put(slist, pfl2str(tab(bal('~', '[', ']') | 0))) do
         move(1) | break
      }

   return multicoll(slist)

end
```

The procedure multicoll(), from the strings module in the Icon program library, collates a list of strings:

```
procedure multicoll(L)
   local result, i, j

   result := ""

   every i := 1 to *L[1] do
      every j := 1 to *L do
         result ||:= L[j][i]

   return result

end
```

Comments

It's hard to imagine that an application for analyzing character patterns, such as charpatt [8], would support searching procedures for all the pattern forms we now have. For synthesizing character patterns, however, they would provide valuable descriptive power. We've mentioned character-pattern synthesis before; it's still in our plans.

References

1. *Connections: The Geometric Bridge Between Art and Science*, Jay Kappraff, McGraw-Hill, 1991, p. 167.

2. "Character Patterns" Icon Analyst 48, pp. 3-7.

3. "Character Patterns" Icon Analyst 49, pp. 1-6.

4. "A Weaving Language" Icon Analyst 51, pp. 5-11.

5. "A Weaving Language" Icon Analyst 52, pp. 1-3.

6. "Versum Deltas" Icon Analyst 49, pp. 6-11.

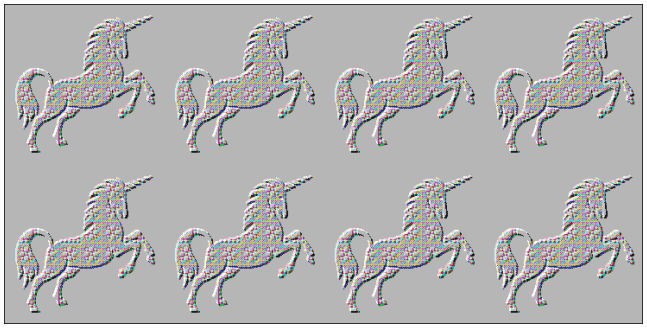
7. "Tricky Business — Image Grammars" Icon Analyst 50, pp. 14-19.

8. "Analyzing Character Patterns" Icon Analyst 50, pp. 1-7.

Downloading Icon Material

Implementations of Icon are available for downloading via FTP:

ftp.cs.arizona.edu (cd /icon)



Graphics Corner — Transparency

The unicorns were the most recognizable magic the fairies possessed, and they sent them to those worlds where belief in magic was in danger of failing altogether. After all there has to be some belief in magic — however small — for any world to survive. — Terry Brooks [1]

The term transparency, as it is used with respect to pixel-based images, is something of a misnomer. There is no such thing as a transparent pixel. Instead, so-called transparency is produced by omitting some pixels when an image is drawn, leaving what was on the canvas at those places intact — to "show through".

The GIF89a image format [2] allows one color to be designated as "transparent". When a GIF89a image is read, pixels of this color simply are not drawn.

Transparency is popular for Web graphics because it allows images to overlay a background without replacing it. Figure 1 shows an image drawn on top of another, first without transparency and then with.

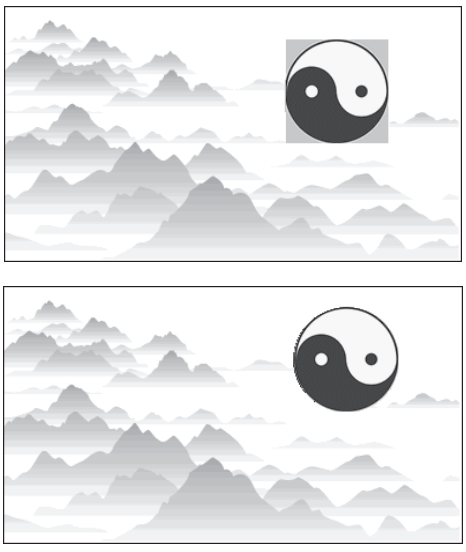


Figure 1. The Affect of Transparency

Icon can read GIF89a images, but it only can write GIF87a images, which do not support transparency. Many image processing programs, however, can convert from GIF87a to GIF89a and allow the user to specify the color for transparency.

Icon's image strings, on the other hand, do support transparency and they provide a way to use transparency without relying on pre-prepared GIF89a files. The characters "~" and "\377" designate transparent pixels, provided these characters do not designate colors in the palette used. **Pop Quiz 1:** What palettes do not support transparency? See the end of this article for the answer.

The real question is, of course, what uses are there for transparency aside from the one already mentioned? As in other articles on image strings, we won't attempt to give a comprehensive answer to this question. Instead, we'll give some examples to stimulate your imagination: Much graphics programming depends on cleverness and a goodly bag of gimmicks and tricks — or better, a meta-bag.

Masks

One use for transparency is masking — obscuring parts of an image and letting other parts show through. This is, of course, what a transparent GIF does to show an image over a background, but the concept has more general applicability.

Our familiar unicorn, taken from a GIF image as shown in Figure 2, provides an example.

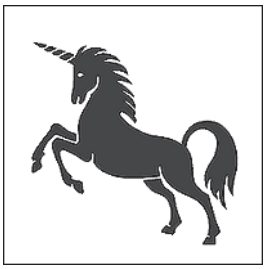


Figure 2. Unicorn Silhouette

Suppose you want to decorate the unicorn to make it more interesting, perhaps "painting" it with a numerical carpet [3].

The first thing to realize is that the unicorn is black on a white background. That makes it easy to separate the unicorn from the background. The next step is to get an image string for the unicorn. This can be done by reading the unicorn GIF image into a window and using Capture() from the Icon program library module gpxop to produce an image string.

The form of a call is Capture(p, x, y, w, h), where p is the desired palette and x, y, w, and h specify the rectangular area to capture. The palette defaults to c1 and the rectangle to the entire window. Since the unicorn image is bi-level grayscale ("black and white"), the appropriate palette is g2. The code to get the image string might look like this:

```
link graphics
link imrutils
                    ...
   WOpen("image=unicorn.gif") |
      stop("*** no unicorn")
   unicorn := imstoimr(Capture("g2"))
```

The module graphics includes gpxop as well as the procedures commonly needed in graphics programming — linking graphics is something you generally should do for programs involving graphics. The module imrutils contains the procedures for manipulating image strings as records [4].

In order to decorate the unicorn, we need to replace black pixels by the decoration and leave the white pixels alone. In the g2 palette, "0" stands for black (and "1" for white, although it's not necessary to know that). If you didn't know the character for a color, you can get it as in this example:

```
black := PaletteKey("g2", "black")
```

To create the unicorn mask, we change the black pixels "transparent" ones:

```
unicorn.pixels := map(unicorn.pixels, black, "~")
```

Next we want to replace the unicorn by the desired decoration:

```
ReadImage("carpet.gif")
```

We'll assume that the carpet image is the same size as the unicorn image. **Pop Quiz 2:** What happens if this is not the case?

The resulting canvas is shown in Figure 3.

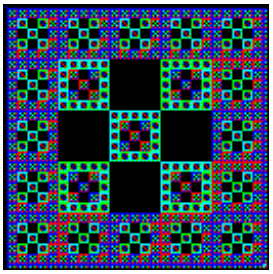


Figure 3. A Numerical Carpet

The last step is to draw the image string for the transparent unicorn:

```
drawimr(0, 0, unicorn)
```

The non-transparent white pixels in the image string replace corresponding pixels of the carpet. The result, a finely attired unicorn, is shown in Figure 4. To see the full-colored splendor of this beast, visit the Web site for this issue of the Analyst. Figure 5 shows the unicorn in two other garbs.

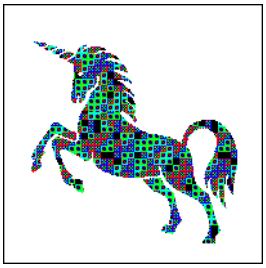


Figure 4. Unicorn Adorned with a Carpet

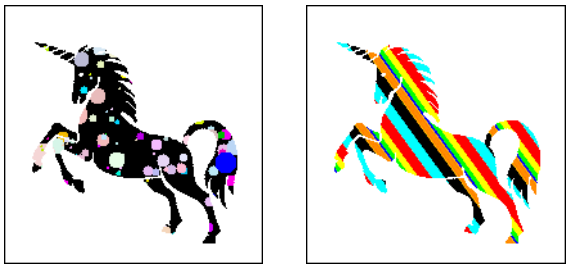


Figure 5. Other Unicorn Adornments

Pop Quiz 3: What happens if you draw the unicorn mask over a totally black window? Over a totally white one?

Image Transitions

A more sophisticated form of masking can be used to provide a gradual transition from one

Back Issues

Back issues of The Icon Analyst are available for $5 each. This price includes shipping in the United States, Canada, and Mexico. Add $2 per order for airmail postage to other countries.

image to another. The basic idea is to overlay a series of increasingly obscuring masks for the new image on top of the old image.

In the example shown here, a circular "shutter" of the new image gradually closes over the old image. The images for the masks can be made by drawing successively larger filled circles over the center of the new image. The color for the circle must be one not present in the new image, since the transparency is obtained by mapping pixels in the circle color to transparent pixels. In this case, white is not used in the image:

```
$define Increment 5

WOpen("image=carpet.gif", "fg=white") |
  stop("*** no carpet")

w := WAttrib("width") / 2
h := WAttrib("height") / 2

masks := []

every r := 0 to 2 * w  by Increment do {
  FillCircle(w, h, r)
  push(masks, imstoimr(Capture())
  }
```

Notice that the image strings are pushed onto the list, leaving the last one as the first on the list. The reason for this is that they will be applied in reverse order to their creation.

Figure 6 show the first six and last six of the images.

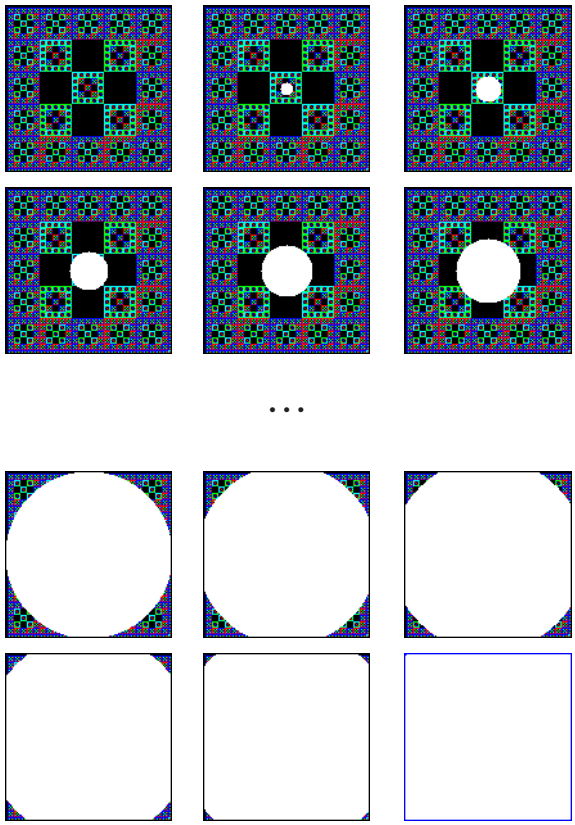


Figure 6. The Images for Masks

The masks then can be created by making the circles transparent:

```
white := PaletteKey("c1", "white")
```


The Icon Analyst

Ralph E. Griswold, Madge T. Griswold,
and Gregg M. Townsend
Editors

The Icon Analyst is published six times a year. A one-year subscription is $25 in the United States, Canada, and Mexico and $35 elsewhere. To subscribe, contact

Icon Project
Department of Computer Science
The University of Arizona
P.O. Box 210077
Tucson, Arizona 85721-0077
U.S.A.

voice: (520) 621-6613

fax: (520) 621-4246

Electronic mail may be sent to:

icon–project@cs.arizona.edu

THE UNIVERSITY OF
ARIZONA
TUCSON ARIZONA
and
Bright Forest Publishers
Tucson Arizona

```
every mask := !masks do
   map(mask.pixels, white, "~")
```

The image strings for the masks are successively overlaid on the new image:

```
every DrawImage(0, 0, !masks) do
   WDelay(Pause)
```

Figure 7 shows the first six and last six images in the transition from the unicorn to the carpet.

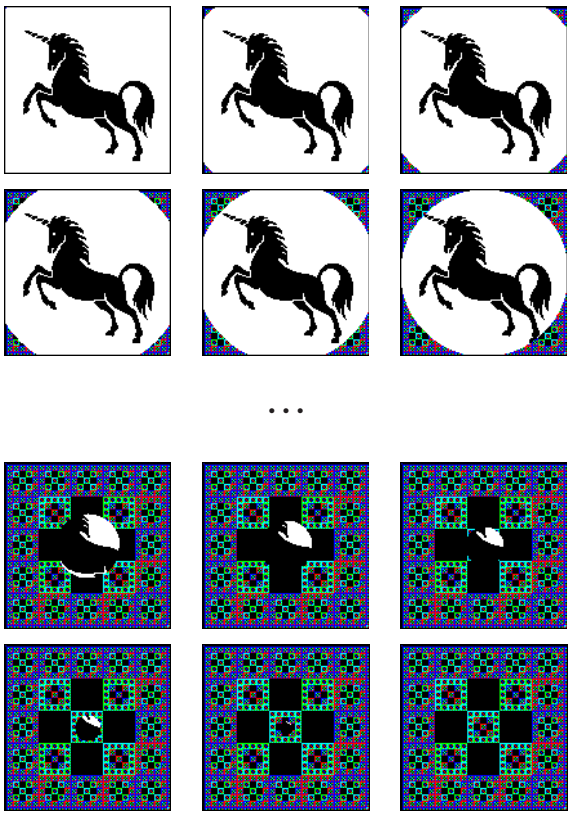


Figure 7. Shuttering the Unicorn

Practical Problems

Capture() is slow; it has to process every pixel in the rectangle to which it is applied. Capture(), in fact, is too slow to use in real time for any but tiny rectangles. Image strings for transparent effects generally need to be prepared in advance.

Image strings also are bulky. This suggests the use of compression techniques. For masks, a method of representing differences between successive image strings might be effective.

Incidentally, there are other methods for effecting transitions between images. If all the areas involved are rectangular, CopyArea() is much faster and easier to use than image strings and transparency.

Answers to Pop Quizzes

1. The only palette that does not support transparency is g256, which assigns all 256 characters to shades of gray. It's worth noting that "~" is used for colors in a number of the larger palettes. It therefore might be prudent to use "\377" for transparency, since it works for all palettes except g256.
Pop Quiz 4: How can you determine the transparency characters supported by a palette?

2. If the carpet image is smaller than the canvas, it is positioned at the upper-left corner of the window. If there are parts of the unicorn silhouette that fall beyond this, they are not changed. If the carpet image is larger than the canvas, parts that fall beyond the left and bottom edges of the canvas are clipped. The unicorn, however, is fully decorated.

In some situations, it may be necessary to make adjustments to the decorating image to get the desired effect.

3. If you draw the unicorn mask over a totally black canvas, you get the original unicorn image. If you draw it over a totally white canvas, you just get a totally white canvas.

4. Here's a procedure that generates the transparency characters, if any, in palette p:

```
procedure transchar(p)
   suspend !('~\377' -- PaletteChars(p))
end
```

Although this procedure is a generator, it can be used to get just one transparency character, as in

```
tchar := transchar(p) |
      stop("*** no transparency character")
```

References

1. *The Black Unicorn*, Terry Brooks, Turtleback, 1990.

2. *Encyclopedia of Graphics File Formats*, James D. Murray and William vanRyper, O'Reilly & Associates, 1994, pp. 321-329.

3. "Anatomy of a Program — Numerical Carpets", Icon Analyst 45, pp. 1-10.

4. "Graphics Corner — Fun With Image Strings", Icon Analyst 50, pp. 10-13.

Animation — Mutable Colors

Animation creates new apparent realities from sheer imagination. — Gary Chapman [1]

Some kinds of animation can be created not by changing shapes but by changing their colors. To do this effectively requires *mutable colors*.

A mutable color is one whose color value can be changed at will, say from green to red. When the color value changes, all canvas pixels of that mutable color change, virtually instantaneously. For example, if several circles are filled with a mutable color, the color of all of them can be changed without redrawing the circles.

The function NewColor(s) obtains a mutable color and returns a small negative integer that identifies it. If s is specified, the mutable color has that color value initially. NewColor() fails if a mutable color is not available; we'll have more to say about that later.

A mutable color can be used in place of an ordinary color specification. For example,

```
stoplight := NewColor("green") |
   stop("*** cannot get mutable color")
Fg(stoplight)
FillCircle(100, 100, 20)
```

gets a mutable color, changes the foreground color to it, and then fills a small circle, which is green initially.

The function Color(i, s) changes the mutable color identified by i to the color value given by s. For example,

```
Color(stoplight, "yellow")
```

changes the color of the circle from green to yellow.

The simplest way to use mutable colors to produce the illusion of motion is to design a path for the motion and draw shapes along the path, using a succession of mutable colors. Then the colors can be changed along the path in a way that appears to be motion.

A Marquee

An example is the familiar marquee, in which lights around the edge of a display are turned on and off in a sequence that makes it look like the lights are moving. Figure 1 illustrates the idea schematically.

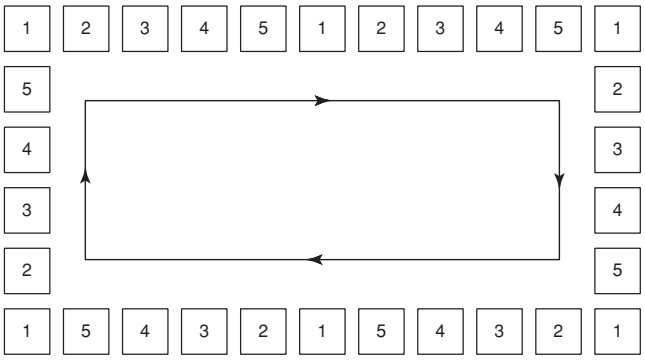


Figure 1. Marquee Layout

The squares represent lights and are numbered in rotation. At any time, all the lights with the same number are on or off together. Think of the lights with the same number as being wired in parallel. By starting with the lights in a "panel" in one state and then shifting this state one light clockwise, the lights appear to move clockwise around the marquee. The number of panels, the number of lights in a panel, the space between them, and so on are configuration parameters.

Here's a program that produces a marquee using mutable colors.

```
link graphics

# Parameters for the display

$define BackGround    "black"
$define LightColor    "yellow"

$define LightWidth    10      # width of light
$define LightHeight   10      # height of light
$define Lights         5      # lights in a panel
$define Offset         5      # space between lights
$define Hpanels        5      # horizontal panels
$define Vpanels        2      # vertical panels
$define Gap            1      # number lights off
```

Supplementary Material

Supplementary material for this issue of the Analyst — including color images, animations, program material, and Web links — is available on the Web. The URL is

http://www.cs.arizona.edu/icon/analyst/iasub/ia52/

```
$define Pause           50

$define Space  (LightWidth + Offset)
$define Wcount (Hpanels * Lights + 1)
$define Hcount (Vpanels * Lights + 1)
$define Width ((Wcount * Space) + Offset)
$define Height ((Hcount * Space) + Offset)

procedure main()
  local x, y, colors, color

  WOpen("size=" || Width || "," || Height,
    "bg=" || BackGround) |
      stop("*** cannot open window")

  #  Get mutable colors and make them the
  #  background color initially.

  colors := []

  every 1 to Lights do
    put(colors, NewColor(BackGround)) |
      stop("*** cannot not get mutable color")

  #  Assign mutable colors to lights, top row left to
  #  right, bottom row right to left.

  y := 0
  x := 0

  every 1 to Wcount do {
    Fg(color := get(colors))
    put(colors, color)
    FillRectangle(x + Offset, y + Offset,
      LightWidth, LightHeight)
    FillRectangle(Width – LightWidth – (x + Offset),
      Height – LightHeight – (y + Offset), LightWidth,
        LightHeight)
    x +:= Space
    }

  #  Right side top to bottom, left side bottom to top.

  x := Width – Offset – LightWidth
  y := Space + Offset
  every 1 to Hcount do {
    Fg(color := get(colors))
    put(colors, color)
    FillRectangle(x, y, LightWidth, LightHeight)
    FillRectangle(Offset + Width – LightHeight –
      (x + Offset), Height – LightWidth – y,
        LightHeight, LightWidth)
    y +:= Space
    }

  # Run the marquee.

  until WQuit() do {
    every Color(colors[1 to Lights – Gap], LightColor)
    every Color(colors[Lights – Gap to Lights],
      BackGround)
    WDelay(Pause)
    put(colors, get(colors))      # rotate the colors
    }

end
```

The many defined constants allow the marquee to be configured in different ways. The configuration here uses five lights per panel as in the layout shown in Figure 1, but there are five panels horizontally and two vertically. At any time, four lights in a panel are on and one off (Gap). Incidentally, if the distance between lights (Offset) is set to 0, individual lights are not evident and entire panels "flow" around the periphery of the canvas.

Figure 2 shows in miniature six stages of the marquee, the last being the same as the first. To get a better idea of the animation, visit the Web page for this issue of the Analyst.

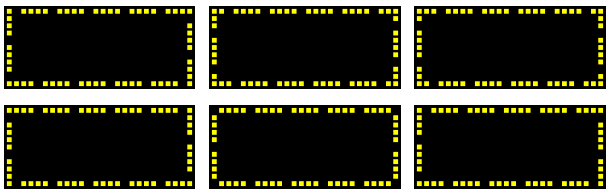


Figure 2. Successive Marquee Images

Try configuring the program in various ways to see how it affects the animation.

Worms

A path of lights can be laid out in many ways. Figure 3 shows "worms" that enter at the top left and exit at the bottom right, flowing around a path in the process.

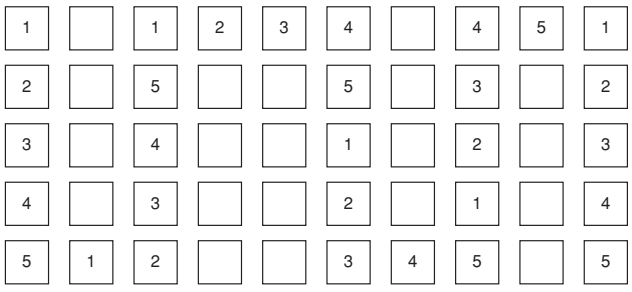


Figure 3. Path Layout

Here's a program for the worms.

```
link graphics

$define BackGround   "black"
$define LightColor   "yellow"
$define Side         15       # worm width
```

```
$define Length          5       # units per worm
$define Pause          50       # animation delay
$define Gap             1       # units between worms
$define Lights          5       # lights in segment

$define Width (10 * Side)       # width of window
$define Height (5 * Side)       # height of window

global colors

procedure main()
   local x, y, unit, loci, locus, i

   WOpen("size=" || Width || "," || Height,
      "bg=" || BackGround) |
         stop("*** cannot open window")

   #  Get mutable colors.

   colors := []

   every 1 to Lights do
      put(colors, NewColor(BackGround)) |
         stop("*** cannot get mutable color")

   # Array of locations.

   loci := [
      [[1, 1], [2, 5], [3, 1], [6, 3], [8, 4], [10, 1]],     # 1
      [[1, 2], [3, 5], [4, 1], [6, 4], [8, 3], [10, 2]],     # 2
      [[1, 3], [3, 4], [5, 1], [6, 5], [8, 2], [10, 3]],     # 3
      [[1, 4], [3, 3], [6, 1], [7, 5], [8, 1], [10, 4]],     # 4
      [[1, 5], [3, 2], [6, 2], [8, 5], [9, 1], [10, 5]]      # 5
      ]

   # Assign mutable colors to locations.

   every i := 1 to Length do
      every locus := !loci[i] do
         light ! push(locus, i)

   # Run the worms.

   until WQuit() do {
      every Color(colors[1 to Length –- Gap], LightColor)
       every Color(colors[Length – Gap + 1 to Length],
          BackGround)
       WDelay(Pause)
       put(colors, get(colors))      # rotate colors
       }
end

# Create grid.

procedure light(color, r, c)

   Fg(colors[color])

   FillRectangle(Side * (r – 1), Side * (c – 1),
      Side, Side)

   return

end
```

Figure 4 shows six frames of the animation, the last being the same as the first. It's difficult to visualize this animation from the snapshots. See the real thing on the Web page for this issue of the Analyst.

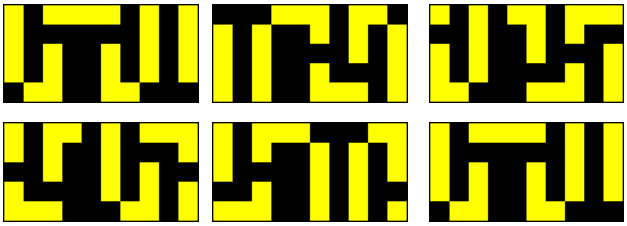


Figure 4. Snapshots of the Worm

The way the path is specified in this program deserves comment. The array loci has a row of coordinates for each mutable color. The coordinates, as lists, provide arguments for light(). The color itself is pushed onto locus before light() is invoked.

The problem with this representation of the path is that constructing it is tedious and error prone. This is the reason we did not give the worms more room to run, which would have produced a more interesting animation. What comes to mind then you think of paths? Turtle graphics [2], perhaps? We'll come back to this subject in a future Analyst.

Expanding Rings

A "path" for animation need not be along a line. For example, it can propogate away from a point, as in this program, which produces an expanding ring:

```
link graphics

$define Size   200
$define Incr     5

procedure main()
   local colors, radius, n, n_old, numbers, count

   WOpen("size=" || Size || "," || Size, "bg=black") |
      stop("*** cannot open window")

   colors := []

   #  Set up the rings.

   every radius := Size / 2 to 0 by –Incr do {
      push(colors, n := NewColor("black")) |
         stop("*** cannot get mutable color")
      Fg(n)
      FillCircle(Size / 2, Size / 2, radius)
      }
```

```
   number := *colors
   count := 0

   #  Animate.

   put(colors, n_old := get(colors))

   until WQuit() do {
      put(colors, n := get(colors))
      Color(n, "orange")
      Color(n_old, "black")
      n_old := n
      WDelay(100)
      count +:= 1
      if count % number = 0 then          # start over
         every Color(!colors, "black")
      }
end
```

Figure 5 shows an expanding ring.

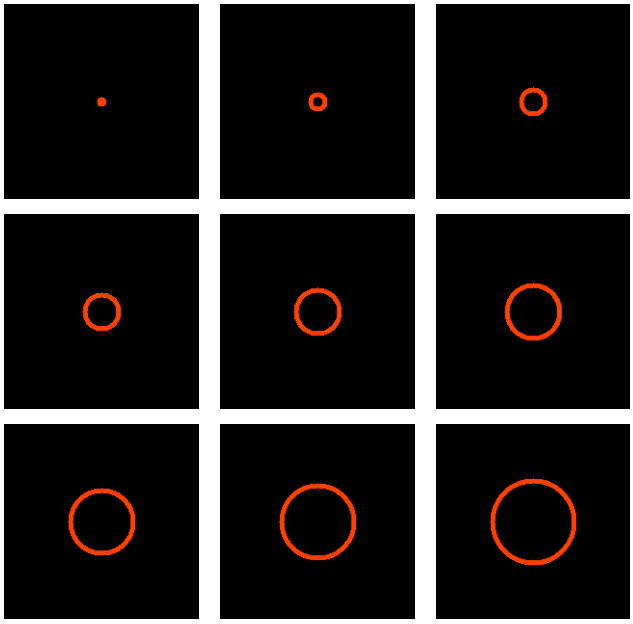


Figure 5. An Expanding Ring

Rotors

A path also can be circular, as illustrated by this program, which animates several randomly sized and placed rotors that revolve in unison.

```
link graphics

$define ForeGround    "black"
$define BackGround    "white"

$define Blades         12
$define Pause          40
$define Size          400
$define Height        400
$define Width         400
$define Blade (&pi / Blades)

global colors

procedure main()
   local color

   WOpen("size=" || Size || "," || Size, "fg=" ||
      ForeGround, "bg=" || BackGround) |
         stop("*** cannot open window")

   colors := []

   # Get mutable colors.

   every 1 to Blades do
      put(colors, NewColor(BackGround)) |
         stop("*** cannot get mutable color")

   # Create rotors

   every 1 to 15 do
      rotor(?Width, ?Height, ?(Size / 2))

   # Animate.

   color := colors[1]

   until WQuit() do {
      Color(color, BackGround)
      put(colors, color := get(colors))
      Color(color, ForeGround)
      WDelay(Pause)
      }
end

#  Create a rotor.

procedure rotor(x, y, r)
   local color, off

   off := ?0 * &pi

   every color := 1 to Blades do {
      Fg(colors[color])
      FillCircle(x, y, r, off + color * Blade, Blade)
      FillCircle(x, y, r, off + color * Blade + &pi, Blade)
      }

   Fg(ForeGround)
   DrawCircle(x, y, r)

   return
end
```

Each rotor has 12 blades. By using mutable colors for the blades, the appearance of motion is obtained by changing the colors of blades in opposition, two visible (black) and the rest "invisible" (white). Note that some rotors appear to be on top of others. This is because they were drawn after the others. The parts of the rotors in the background

that were not drawn over continue to revolve — the result is what you'd expect to see if they were real rotors, some in front of others in the line of sight, except that background rotors do not show through the "invisible" blades.

Figure 6 shows 12 frames of the animation. Note that revolution is clockwise.

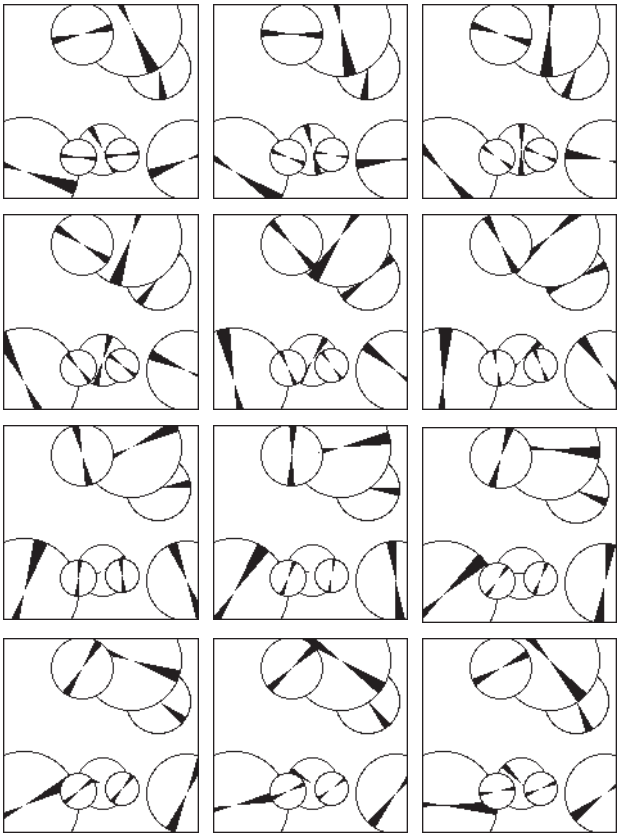


Figure 6. Revolving Rotors

See Reference 3 for another example of rotation using mutable colors.

Color Wheel

As a final example, here is a program that produces a rotating color wheel for a palette:

```
link graphics

$define Palette         "c1"
$define Pause           10

procedure main()
   local color, colors, mutants, alpha, theta, i

   WOpen("size=400,400", "bg=black") |
      stop("*** cannot open window")

   colors := []
   mutants := []

   chars := PaletteChars(Palette)

   # Get the mutable colors.

   every put(colors, color :=
      PaletteColor("c1", !PaletteChars(Palette))) do
         put(mutants, NewColor(color)) |
            stop("*** cannot get mutable color")

   alpha := 2 * &pi / *colors
   theta := 0.0

   #  Draw the wheel.

   every 1 to * colors do {
      put(mutants, color := get(mutants))
      Fg(color)
      FillCircle(200, 200, 200, theta, alpha)
      theta +:= alpha
      }

   #  Run the animation.

   until WQuit() do {
      put(colors, color := get(colors))
      every i := 1 to *colors do
         Color(mutants[i], colors[i])
      WDelay(Pause)
      }
end
```

Figure 7 shows six frames of the animation.

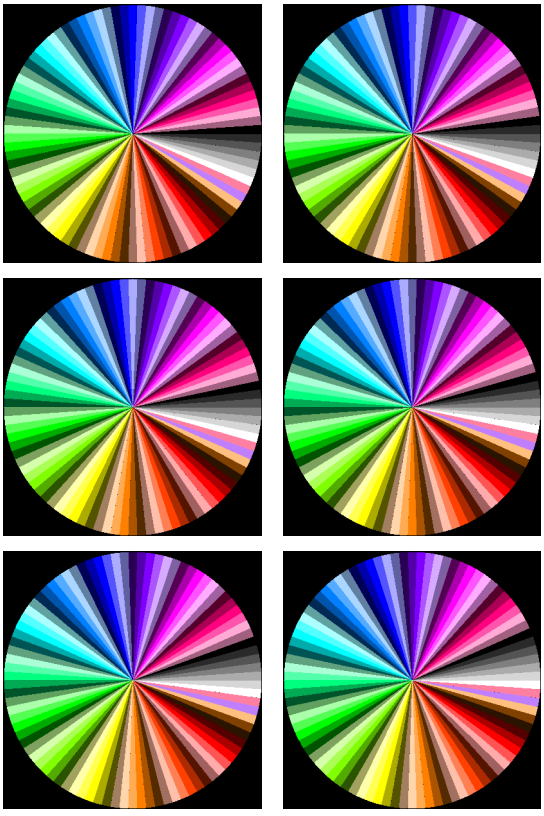


Figure 7. Revolving Color Wheel

Comments

The main value of mutable colors in animation is that the color value of all pixels in a mutable color — however many there are — can be changed at once and with no detectable delay. For example, the arrangement of pixels in the animated rotors is quite complex, but the animation works at the same speed regardless of the size or number of rotors — or even how big the canvas is. There is only one other method of doing such an animation with standard hardware that does not produce unacceptable visual artifacts. We'll describe that method in the next article on animation.

Doing a complicated animation using mutable colors requires significant planning and investment in setting up the canvas. And, of course, not all kinds of animation are possible using mutable colors.

Mutable colors can be used to provide a variety of effects besides the illusion of motion. In the next Analyst, we'll show a use of mutable colors that has nothing to do with animation.

The Limitations of Mutable Colors

Not all graphics systems support mutable colors, and when they do, they typically only work for monitors in 8-bit mode (256 colors) or less. Most UNIX systems support mutable colors with monitors in 8-bit mode, but mutable colors do not work properly in Windows Icon.

Incidentally, if you have a (non-Icon) application that requires setting your monitor to 8-bit mode, it's a good bet that the application uses mutable colors.

The number of mutable colors that are available is limited by colors used for other purposes. If no other colors are used, it's possible to get 254 mutable colors (black and white are reserved). In a VIB application, the maximum number of mutable colors that can be obtained is 252.

If a mutable color is not available, NewColor() fails; it is important to check for this to ensure an application does not malfunction.

It's wise to design applications with these limitations in mind and not to use mutable colors for applications that need to be portable to different kinds of platforms.

Exercises

Here are some things you might try using mutable colors.

1. Design marquees that are not rectangular — perhaps circular, oval, or polygonal.

2. Provide multi-colored lights for the marquee animation given in this article.

3. Modify the worm animation to provide multi-colored worms.

4. Modify the rotor animation so that the user can control the speed and direction of rotation.

5. Modify the rotor animation to have blades of several different colors.

6. Design a visual amusement in which the lights on a panel change colors randomly.

7. Modify the expanding rings animation to have multiple rings of different colors. Find colors that suggest an explosion.

8. Simulate a projectile shot from a gun traveling under the influence of gravity in a vacuum. Provide controls for a user to specify the angle of the gun and the initial velocity.

9. Write a program to trace a path along a plane curve. See the module curves in the Icon program library.

10. Design an interactive game in which the user tries to click on an erratically moving shape. Provide a scoring scheme and levels of difficulty.

11. Design a labyrinth and animate a light traversing it.

12. Design a maze and track the mouse position with a light as the user tries to find the way out.

13. Write a procedure similar to DrawImage() that works with mutable colors.

References

1. *Macromedia Animation Studio*, Gary Chapman, Random House, 1995.

2. "Turtle Graphics" The Icon Analyst 24, pp. 6-10.

3. *Graphics Programming in Icon*, Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend, Peer-to-Peer Communications, 1998, pp. 151-153.

From the Library — PostScript Graphics

I have never known any distress that an hour's reading did not relieve. — Montesquiou [1]

Image Quality

One of the problems with producing graphic images in windows is that windows are composed of finite-sized, discrete pixels. For this reason, it's not possible to draw a diagonal line or other non-rectangular shape precisely. The result of approximating such a shape is "jaggies", as illustrated in Figure 1.

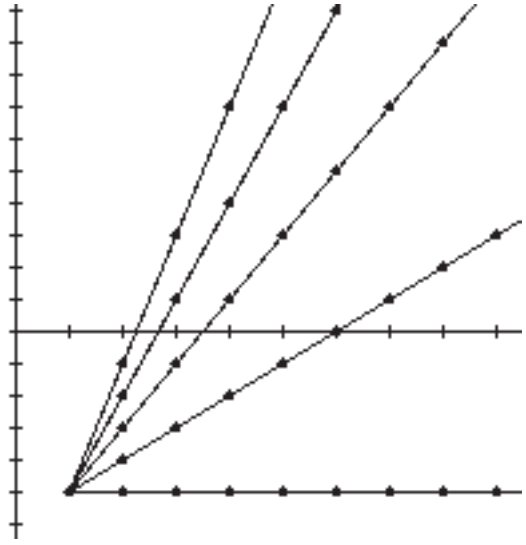


Figure 1. Screen Image

How good a non-rectangular shape looks depends on the angles involved and the pixel size. Figure 1 corresponds to pixels that are 1/72nd of an inch on a side. See the side-bar on **Resolution** on the next page.

Figure 2 shows a portion of the image with the pixels enlarged about eight times.

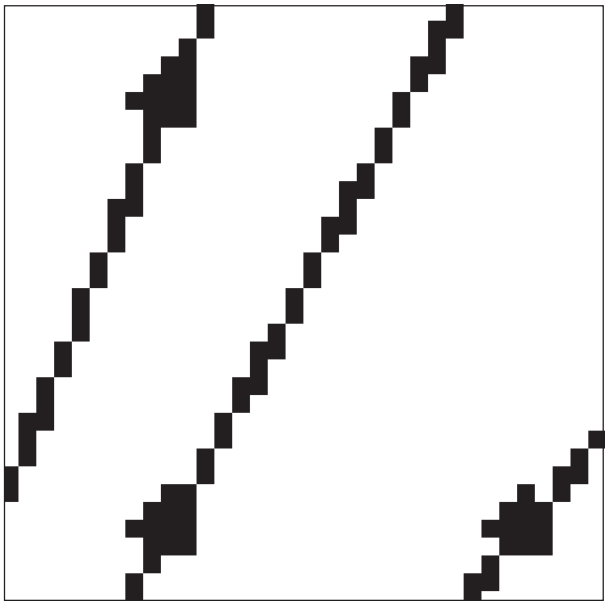


Figure 2. Enlarged Screen Image

The irregular "blobs" are the renditions of circles that are supposed to be four pixels in diameter. It's clear you can't come close to the desired shape, although as Figure 1 shows, if the pixels are small enough, the visual appearance is at least reasonable.

Although Figure 1 conveys the information it represents fairly well, it's crude and unacceptable for most forms of publication. Now compare Figures 1 and 2 with Figures 3 and 4, which are rendered in PostScript [2-4].

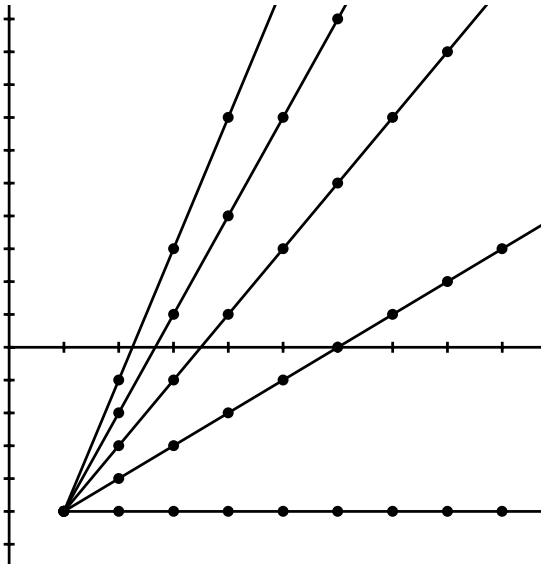


Figure 3. PostScript Image

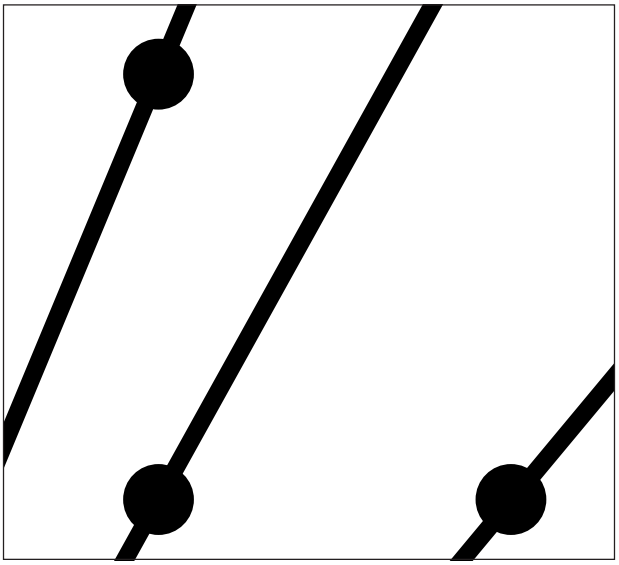


Figure 4. Enlarged PostScript

PostScript represents shapes by their geometrical description and renders them using "vector" graphics. PostScript graphics can be scaled by arbitrary amounts without loosing quality. The practical limit is the resolution of the device on which they are rendered. The Analyst is printed on a 600 dpi laser printer, which gives adequate quality for a publication of its type.

Most PostScript graphics are produced by applications such as Adobe Illustrator <1> and Macromedia Freehand <2>. Since PostScript is a language (a page-description language, to be precise), it's possible to create graphics by writing directly in PostScript, but doing that is so tedious that few do it — and with the tools available, there is no need for it.

Creating PostScript Graphics

This article is about creating PostScript graphics in Icon, not by writing in PostScript but rather by using a package in the Icon program library that emulates Icon's graphic operations. This package automatically creates PostScript that corresponds to graphics function calls. This does not affect what appears on the screen; it is the same with or without PostScript emulation.

All that's necessary to use this facility is linking psrecord from the library and adding procedure calls to control the emulation. At its simplest, it might look like this:

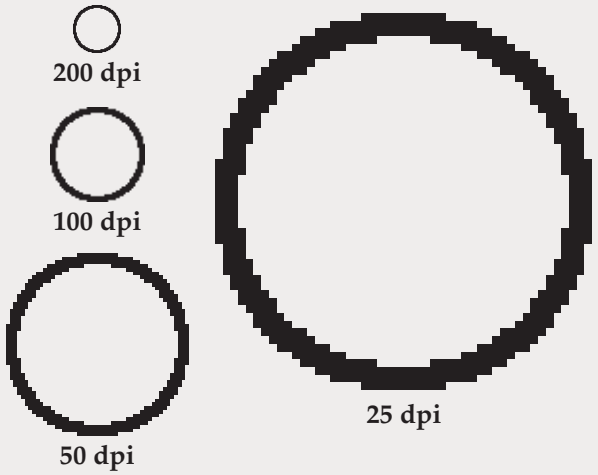
Resolution

There is considerable confusion about the meaning of the word resolution as it applies to computer-related devices and images. This is due in part to misuse of the term.

The most notable misuse is to describe monitor resolution in terms of the number of pixels it displays, as in a 1024×768 screen. Resolution, rather, should be measured in the size of the pixels, as in 75 pixels per inch (ppi). (Since pixels are so small, ppi is easier to deal with than, say, 0.01333".) This is the way that printer resolution is given, as in 300 dots per inch (dpi) — you'll not see *printer* resolution listed as, say, a 4975×6375 page.

Another confusion results from describing the size of an image by its pixel dimensions, as in a 750×375 image. How much area an image covers when displayed on a monitor depends on the size of the pixels. At 75 ppi, such an image covers an area of 10" by 5". At 150 ppi, it covers 5" by 2.5".

Adding to the confusion is the fact that image resolution can be set for the purpoes of printing. For example, the resolution of an image might be set to 150 dpi, 300 dpi, or 600 dpi. This does not affect the number of pixels it contains. Instead, the higher the dpi, the smaller the image is when printed. Such settings do not affect the printer but only the interpretation of pixels. Within the limitations of a printer, specifying higher resolution for pixel-based images not only produces smaller images but also "crisper" looking ones. Here, for example, is the same image printed at four different resolutions:



```
# open a window…
PSEnable()            # start PostScript emulation
    …
# create graphics image
    …
PSDone()              # terminate the emulation
```

The enabling procedure has two optional arguments: PSEnable(window, file). The window defaults to &window (a window must be opened before PSEnable() is called), and the file to which PostScript is written defaults to xlog.ps. There are other procedures to provide more control; see psrecord.icn.

The output is "encapsulated" PostScript and can be imported without modification into most word processing and desktop publishing programs, which is what we did to produce the PostScript graphics shown in this article.

Here's a complete example:

```
link graphics
link psrecord

procedure main(args)
   local i, j, k, angle, incr, xpoint, ypoint
   local size, radius, xc, yc

   i := integer(args[1]) | 20

   size := 300
   radius := size / 4
   xc := yc := size / 2

   WOpen("label=design", "width=" || size, "height=" ||
      size) | stop("*** cannot open window")

   PSEnable("design.ps")        # enable PostScript

   angle := 0.0
   incr := 2 * &pi / i

   every j := 1 to i do {
      spokes(xc + radius * cos(angle),
         yc + radius * sin(angle), radius, i, angle)
      angle +:= incr
      }

   WriteImage("design.gif")     # produce pixel image

   PSDone()                     # terminate PostScript

end

procedure spokes(x, y, r, i, angle)
   local incr, j

   incr := 2 * &pi / i

   every j := 1 to i do {
      DrawLine(x, y, x + r * cos(angle),
         y + r * sin(angle))
      angle +:= incr
      }

   return

end
```

Figures 5 and 6 show the GIF and PostScript graphics produced by this program.

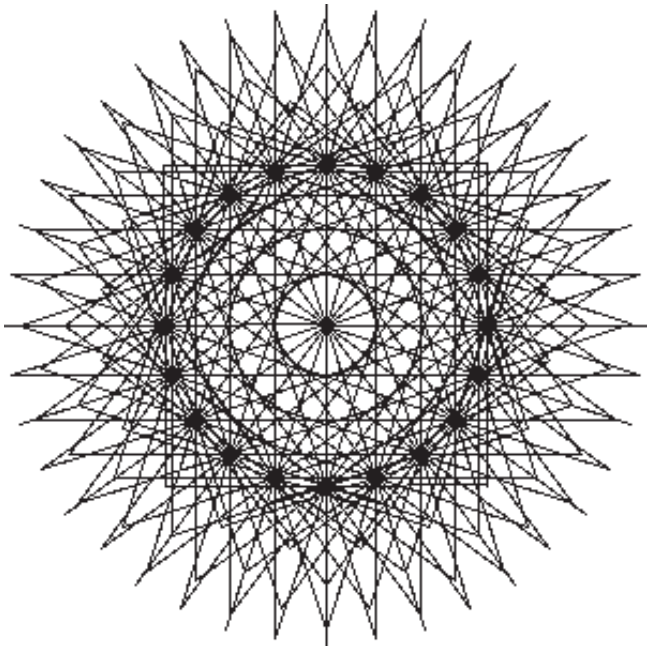


Figure 5. Pixel Rendering

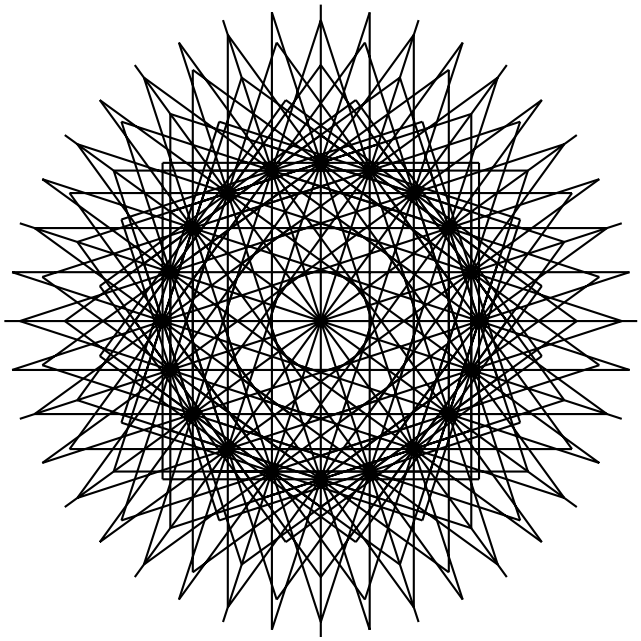


Figure 6. PostScript Rendering

Although both forms of rendering produce

attractive results, the difference in quality should be apparent. And the PostScript version could be scaled to cover the side of a barn without loss of quality.

Caveats

The emulation of Icon's graphics facilities by psrecord is imperfect. Not all graphics facilities are supported (CopyArea() is an example) and some produce crude approximations (for example, DrawCurve() produces straight line segments in PostScript). For details on such matters, see the program documentation and attend to its admonition "… psrecord works best for programs designed with it in mind".

Despite its limitations, psrecord can be used to produce high-quality graphics from Icon programs, as illustrated by the examples in this article. Give it a try.

References

1. *Pensées Diverses*, Comte Robert de Montesquiou-Fezensac, 1899.

2. *PostScript Language Tutorial and Cookbook*, Adobe Systems, Addison-Wesley, 1985.

3. *PostScript Language Reference, aka The Red Book*, 3rd edition, Adobe Systems, Addison-Wesley, 1999.

4. *PostScript Language Program Design*, Adobe Systems, Addison-Wesley, 1988.

Links

1. Adobe Illustrator (Macintosh and Windows):

http://www.adobe.com/prodindex/illustrator/main.html

2. Macromedia Freehand (Macintosh and Windows):

http://www.macromedia.com/software/freehand/

Icon on the Web

Information about Icon is available on the World Wide Web at

http://www.cs.arizona.edu/icon/

What's Coming Up

Computers are useless. They can only give you answers.

— Pablo Picasso

In the next Analyst, we'll continue the series of articles on weaving, focusing on how weaving specifications can be represented.

We'll have answers to the quiz in this issue and follow up with a quiz on expression evaluation. We also plan to have results from a programming problem posted to icon-group.

For the **Graphics Corner**, we'll describe an application that uses mutable colors to allow the users to change the colors in an image.

We'll also be starting a series of articles on generators and sequences — one that may occupy pages of the Analyst for some time.

There are a few other things we've been holding onto, including a somewhat whimsical article on digit patterns in large prime numbers. If space permits, we'll include it.