by Jeffreys Copeland and Haemer

"Every house is a fixer-upper."
– Arthur D. Hlavaty

"A house is a machine for keeping your books dry."
– Tom Shippey, paraphrasing Le Corbusier

Escher, Penrose, Foyer

We were recently reading Simon Singh's entertaining book, *Fermat's Enigma* (published by Walker & Co., 1997, ISBN 0-8027-1331-9), about the quest to prove Fermat's Last Theorem. In the process of discussing modular forms in complex hyperbolic space and *reductio ad absurdum* proofs, Singh touches on tessellations and tilings. Tessellation, or tiling a plane with regular figures, was made famous by Dutch artist M.C. Escher, and written about extensively by Douglas R. Hofstadter in his wonderful book *Gödel, Escher, Bach* (published by Basic Books, 1979, ISBN 0-465-02685-0).

Before that, though…

We Get Letters…

In our February column, "Back to Basic(s)" (Page 42, `http://sw.expert.com/C9/SE.C9.FEB.00.pdf`), we passed on a problem from one of our Windows-bound friends, Michael Mendelson. Michael had a C program to randomize the lines of a text file. "I'm sure," he said, "you could write this one in half the lines from the shell." We tossed that, in turn, to you, Gentle Reader, and you responded with a variety of interesting solutions. We don't have space to show them all, but they fell into a few categories.

First, we had some shell-like solutions. Erik Jacobsen and Geoff Clare provided ones that use `awk` to prepend random numbers to each line, and then `sed` or `cut` to remove them after sorting.

Marty McGowan provided us with a shell-only solution. It provides a series of one-line shell functions, triggered by an invocation of the first one. This has the fascinating feature of allowing you to invoke the script on itself without damage.

We had some Perl-based solutions, like this one from O'Shaughnessy Evans:

```
perl -api -0777 -F'\n' \
   -e 's/^.*?$/splice
(@F,rand ($#F+1),1)/meg;
print STDERR
"randomized $ARGV\n"'
```

Our favorite submission was a pathological entry from Peter Kernan:

```
perl -pe 'splice @a, rand $.,
0, $_} for(@a) {'
```

Note the curly braces; they're correct. Because we've used the `-p` flag, the statement on the command line is wrapped in

```
while (<>) { ... }
```

so the braces match up. To see this, use the flags `-MO=Deparse`, which will show you what the Perl compiler thinks this code means.

Thank you one and all for your entertaining bits of code. We had fun looking at them.

Back to Tiles

One of the interesting features of tessellations is that they're often both rotationally and translationally symmetric. In other words, you can turn them, or move the origin of your grid without changing the layout of the pattern. But Singh reminds us of some work British mathematician Roger Penrose did in the 1970s with two shapes that can equally well tile a plane. The kite and dart (see Figures 1 and 2) can be combined into a number of patterns that are completely nonsymmetrical.

Oddly enough, we read this at about the same time Copeland realized he needed to retile the entryway of his house. "Wouldn't it be interesting," we said, "if we could get tiles of kites and darts in interesting colors, and then redo the entryway in a Penrose pattern?" We thought we'd see just how nonsymmetrical we could make the patterns. We printed out a couple of sheets of kites and darts on brightly colored paper, and brought them home. It's not as easy as it looks, and every time the cats climbed up on the table to help, we lost all our work.

Figure 1. Kite

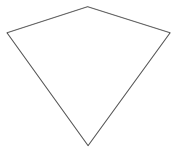
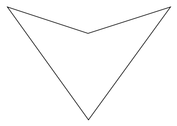


Figure 2. Dart



"This," we thought to ourselves, "is a job for graphics software." We have a phenomenal graphical engine in just about every printer around us: all we have to do is write the tiling software primitives in PostScript.

Those are nice shapes, you're saying to yourself, but how do I draw them? It's pretty simple. We begin with some definitions and parameters, which we incorporate into a PostScript prolog:

```
/tan { dup sin exch cos div } def
/dela 54 def
/delb 18 def
/delk delb def
/theta 72 def
/phi 144 def
/side 100 def
/sideb side dela cos mul delb cos div def
/span 2 side mul cos dela mul def
/kiteht side def
/dartht side theta 2 div cos mul
 sideb phi 2 div cos mul sub def
```

This defines the apex angles of the dart and kite (`theta` at the "bottom," `phi` at the "top"), the angles of the shoulders (`del`) and the length of the long and short sides (`side` and `sideb`, respectively). We also define the height from point to point (`dartht` and `kiteht`), and the distance between the shoulders (`span`).

Given those parameters, drawing a dart is fairly easy. We just give PostScript some driving directions:

```
/dartpath {
 currentpoint translate
 newpath 0 0 moveto
 theta 2 div neg rotate 0 side rlineto
 phi rotate 0 sideb rlineto
 180 phi sub neg rotate 0 sideb rlineto
 closepath
} def
```

In the event that's not obvious, let's give you a dramatic reading of what we just did. We began by adjusting the origin of our coordinate system to the current point. Beginning there, we started a new path, turned to the right `theta/2` degrees, moved forward `side`, turned to the right `phi` degrees and moved forward by `sideb`. At that point, we'd travelled the path for the right side of the dart. We finished by making the oblique angle turn at the dart's top, moved forward by another `sideb` and closed the path by returning to the origin.

Notice that we haven't drawn the path at all, merely described what it is. This means that we can write two complementary routines to use this one. The first draws the outline of the dart, and the second takes a color off the stack and draws a filled-in dart:

```
/dart {
 gsave
  dartpath
  stroke
 grestore
} def

/dartcolor {
 gsave
  dartpath
  setrgbcolor
  fill
 grestore
} def
```

As we said, the first routine, `dart`, draws the dart's outline, standing on its point at the current position. The second, `dartcolor`, draws a dart shaded in the color specified by the red/green/blue (RGB) triple on the stack. Notice that we're carefully saving and restoring the graphic state each time we draw the dart. This means that when we finish the dart using either routine, even though `dartpath` has done violence to our coordinate system, we have the same origin and rotation that we did when we started.

Our definition for kites is remarkably similar to the definition for darts. To wit:

```
/kitepath {
 currentpoint translate
 newpath 0 0 moveto
 theta 2 div neg rotate 0 side rlineto
 180 dela delb add sub rotate 0 sideb rlineto
 180 phi sub rotate 0 sideb rlineto
 closepath
} def

/kite {
  gsave
 kitepath
 stroke
  grestore
} def
```

```
/kitecolor {
  gsave
 kitepath
 setrgbcolor
 fill
  grestore
} def
```

Interlocked Tiles

That was easy. Now we have to write some routines to place tiles relative to one another. We begin with a handful of routines that give us motion relative to the size of a tile. For example, if we want to rotate a tile to the left or right, we need:

```
/plustheta { theta rotate } def
/minustheta { theta neg rotate } def
```

This means we can say something like:

```
dart plustheta kite
```

Similarly, we'll need fractional rotations:

```
/plushalf { theta 2 div rotate } def
/minushalf { theta 2 div neg rotate } def
```

Also, we'll want some translations along the breadths and heights of the tiles:

```
/pluskite { 0 kiteht translate } def
/plusdart { 0 dartht translate } def
/plusside { 0 side translate } def
```

We need two additional primitive motions and a service routine: "turn completely around," "make sure I'm positioned here" (this is useful just before a tile) and "print a mark for debugging purposes." To wit:

```
/aboutface { 180 rotate } def
/here { 0 0 moveto } def
/mark { 0 0 moveto (v) show } def
```

One other interesting primitive is "pick a random color." PostScript provides us with an integer random number generator. Do you remember what its range is? Neither do we, except that it gives us an integer, so we take a modulo of the low-order bits and scale into the range [0,1). We repeat three times to give us our RGB triple. This allows us to provide primitives for randomly colored darts and kites, `Dart` and `Kite`. We also seed the random number generator:

```
/color { 0 1 2
 { pop rand 256 mod 256 div } for
} def
/Dart { color dartcolor } def
/Kite { color kitecolor } def
realtime srand
```

(An aside: Special mention in a future column for the first reader to give us an attribution for the following: “The generation of random numbers is too important to be left to chance.” We tripped over the quotation not so long ago, but our source had no idea where it came from. We thought it might be from Volume 2 of Knuth’s *The Art of Computer Programming*, but we can’t find it there.)

Now we can build some very interesting composite motions. For example, if we want to add a tile to the left or right we could use the following:

```
/addleft { plustheta here } def
/addright { minustheta here } def
dart addleft kite addright addright dart
```

(Notice that we can compound these.) Similarly, we may want to add a tile upside down from the current one, as you can see in Figure 3, where the added tile is in color:

```
/addleftinvert {
 plushalf plusside aboutface
 minushalf here
} def

/addrightinvert {
 minushalf plusside aboutface
 plushalf here
} def
here kite addrightinvert Dart
```

In Figure 4, we put our tiles nose-to-nose, and again, the second tile is in color. Why do we always move the height of a dart plus a kite? Because it doesn’t matter which order they’re in; we can’t have tiles of the same kind facing each other like this:

```
/addtop {
 0 kiteht dartht add
 translate aboutface here
} def
here kite addtop Dart
```

Last, we have a series of composite motions that allow us to stack the tiles nose-to-nose, but offset (see Figure 5):

```
/addtopleftdart {
 plushalf 0 side dartht add
 translate aboutface here
```

Figure 3

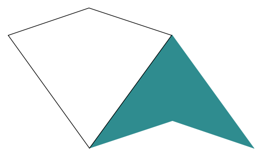
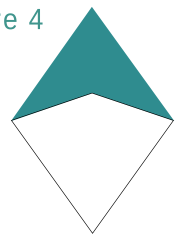


Figure 4



```
} def
/addtoprightdart {
 minushalf 0 side dartht add
 translate aboutface here
} def

/addtopleftkite {
 addtop minushalf here
} def

/addtoprightkite {
 addtop plushalf here
} def
here dart addtopleftkite Kite
```

Some of these might be easier to read if we were adherents of the "EveryWordCapitalized" school of variable naming, or even the "underscore_between_words" school. However, whatever these composite drawing functions are named, we can make interesting patterns.

For example, Figure 6 has a tile pattern that's translationally symmetric. We drew it with the following:

```
here
kite addrightinvert dart addleftinvert
kite addrightinvert dart
addtop kite addleftinvert dart
addrightinvert kite addleftinvert dart
```

However, we can make much more interesting patterns. See Figure 7, which was drawn using the following:

```
here kite addleft kite addleft kite addleft
kite addleft kite addtopleftdart dart
addleftinvert dart addright kite addrightinvert
dart addleftinvert dart addright kite
addrightinvert dart addleftinvert dart addright
dart addrightinvert dart addleftinvert kite
addright kite addrightinvert dart addleftinvert
dart addright kite addtop dart addleftinvert
kite addrightinvert kite addleftinvert kite
addright kite addrightinvert dart
```

Notice that this is almost, but not quite, rotationally symmetric. It's the sort of pattern we were looking for in the first place for Copeland's entryway.

Challenge

We can manually lay out as many tiles as we want using these primitives. It's a lot more stable than the cat-endangered cutouts on the dining room table. On the other hand, it's still trial and error.

The next logical step is one we don't know how to do, so here's the part where we turn to you, our helpful and energetic readers: What's the algorithm we need at this point to lay tiles out automatically in a random pattern? This is a second opportunity to share your knowledge with us, and the rest of our readers, in exchange for having your name made famous in these pages.

Until then, happy trails.

Jeffrey Copeland *(*`copeland@alumni.caltech.edu`*) is currently living in the Pacific Northwest, where he spends his time writing UNIX software in a large development organization and fighting damp rot.*

Jeffrey S. Haemer *(*`jsh@usenix.org`*) works at QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm and did a lot of other things, like everyone else in the software industry.*

Note: The software from this and past Work columns is available at `http://alumni.caltech.edu/~copeland/work` *or alternately at* `ftp://ftp.expert.com/pub/Work`*.*

Figure 5

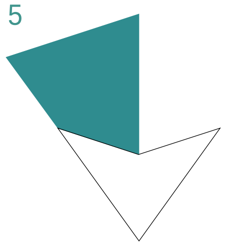


Figure 6

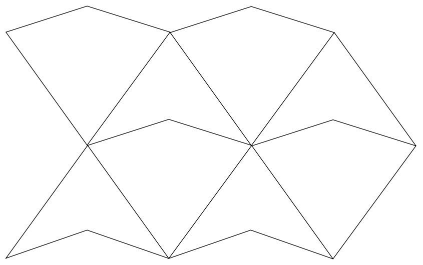


Figure 7