Swhirling Squares: A Simple Math Flip-Book Animation

Douglas McKenna • Mathemaesthetics, Inc.
P.O. Box 298 • Boulder, Colorado • 80306

Abstract

The PostScript[1] programming language is integrated with a mathematically elegant, 2-D drawing model. Using a very simple PostScript template file, one can quickly introduce students to programming, mathematical, and æsthetic concepts, requiring only a simple text editor and a standard graphic file previewing application. In particular, geometries parameterized by time lend themselves to creating algorithmic art in the form of an animated PDF "flip book." A simple project, based on little more than generalizing the drawing of graph paper to make squares rotate relative to themselves in various ways, proved visually and intellectually captivating to students.

Introduction. A small experiential school in my community [5] requires some of its high-school students to participate in an intensive, week-long internship with a local expert on some subject. The student then gives a public presentation of what he or she learned. As a longtime programmer with a bent towards mathematical illustration and algorithmic art, I found myself tasked with teaching and demonstrating to a young person how one might program a computer to draw some literally moving mathematical drawings.

A recurring experience of those of us who use math to make art is that many people, whether children or adult, are surprised to hear the words *math* and *art* uttered in the same sentence. The 16-year-old I was tutoring was no exception. Furthermore, he had no programming experience. All the two of us had to work with were a pair of Apple Macintosh laptop computers. With little experience teaching kids, and no pedagogical infrastructure to back me up, my challenge was to find a *really simple* way to achieve some meaningful and memorable visual results as quickly and interactively as possible in a very short time. My desire was also to avoid, at least as much as possible, getting bogged down in the myriad, time-consuming and invariably frustrating details of programming languages, specialized software interfaces, development environments, libraries, plug-ins, the cryptic world of Unix command lines, downloads, installations, debuggers, object-oriented this or that, etc., etc.

Python, then PostScript. The initial plan was to use the Python programming language [3], thinking that its ease-of-use, clear syntax, and interactive design, as well as its availability on both our machines, would suffice. But because Python isn't graphics-ready, the moment the time to start drawing arrived, all the vaunted ease-of-use disintegrated. Example code found on the web wouldn't compile. Required libraries could not be easily found. Error messages were inapposite. Version numbers inexplicably got out of sync. An entire GUI toolkit appeared to be necessary. Complicated dependencies reared their ugly heads—the usual story. In short, faced with having to explain way too much rigmarole to a neophyte in too little time, I gave up.

So after spending the first day building a simple, text-based, input/output guessing game [4] using non-graphic, computer language concepts in Python, my student and I switched over to programming directly in PostScript, in which many of the same beginning programming concepts occur, albeit using a very different syntax.[2] The text editor we used was nothing more than the minimal TextEdit application. The "com-

[1] PostScript is a trademark of Adobe Systems, Inc.

[2] Having two different ways of doing the same thing, however, is a powerful teaching tool.

piler"/"interpreter" was just the Preview application. Both of these applications are currently shipped with every Mac computer.[3]

The Preview application instantly draws a variety of graphic file types in a window. In particular, it converts PostScript files (with suffix "`.ps`") to PDF and displays the result, including a thumbnail list of pages. Although printer drivers usually output very complex and unreadable versions of these files, one can create very simple PostScript programs by hand in any text editor. Preview then instantly converts the PostScript file into a picture, or better yet, a series of pictures that, when scrolled through quickly by simply pressing the down-arrow key, create an animation. No special software or commands are needed.

These simple tools worked, albeit with the caveat that I have been programming in raw PostScript for a long time, so I've come to know what to do when things go awry, as invariably occurs while coding in any computer language.[4] The initial five-day pedagogical plan was as follows: (1) write a simple text-only game in Python; (2) introduce graphics fundamentals using raw PostScript; (3) create a simple and/or "cool" animation of something mathematical, and play with it; (4) animate graphical representations of $(a+b)^2 = a^2 + 2ab + b^2$ and the Pythagorean theorem $a^2 + b^2 = c^2$; and (5) animate a rolling Reuleaux triangle [1]. The remainder of this paper concerns the third day's project, in which we made something fanciful for no other reason than to have fun. By virtue of easy experimentation, our flipbook animation captivated the interest not only of me and my pupil, but also of many of his classmates.

A Flip Book Template. A flip book is a bound set of pages, each with a slightly changing drawing on it, that one can riffle through in order to animate the series of images into a very short cartoon. Flip books are easy to make by hand, but it is important that their page size not be too large. The same is true when making a PDF flip book. It's annoying to view a PDF file only to have the first displayed page so large that some portion of the drawing is invisible or requires adjusting. So the very first thing to type (or cut and paste) into the PostScript template file is the following seven lines of admittedly arcane-looking code.[5] This installs a custom page size that always fits on the screen. It also prepares a default typeface with which we can label our flipbook frames (if desired).

```
%!PS                                                % "This is a PostScript file"
/pgWidth 5 def                                      % Set a page width in inches
/pgHeight pgWidth def                               % Make height same as width
1 dict dup                                          % Push 2 refs to a dictionary
/PageSize [ pgWidth 72 mul pgHeight 72 mul ] put % Set custom page size in pts
setpagedevice                                       % Install new PageSize entry
/Helvetica findfont 1.5 scalefont setfont           % Use small font to show text
```

Each animation frame will be on a separate page. So we must next create a custom command, called `preparepage`, that (re-)installs our desired coordinate system at the start of each new page (for those unfamiliar with reading "postfix" PostScript code, if a command needs arguments, they precede it in the input):

[3]Instead of Preview, Adobe's free Acrobat Reader application will work also, but requires downloading. On Unix or Linux systems, one can use previewing applications based on "ghostscript" (or `gs` command), which may also require installation.

[4]The only real complication was that if a bug in our handwritten code caused the PostScript interpreter to issue an error message, Preview doesn't pass that message on to the user directly. Instead, Preview sends the error message to the normally hidden system console window. But any such error messages can be easily accessed using the Console application (in the Applications→Utilities folder). PostScript interpreter error messages are even more minimal than Python's, which can be a problem for neophytes working without an experienced PostScript programmer nearby.

[5]When you copy this code into your template file (ending in `.ps`), ensure that it's all in the same order as presented here. And as long as all later definitions are made prior to invoking the final `makeflipbook` command (described below), it should work.

```
/preparepage {                      % Call this to set up coordinate system
  72 72 scale                       % Change units from points to inches
  pgWidth 2 div pgHeight 2 div      % Compute the center of the page and ...
  translate                         %  ... move the drawing origin to it
  .1 .1 scale                       % Ensure largest drawing fits on the page
  .01 setlinewidth                  % Use fairly thin outlines of figures
  } def                             % Install new command for later invocation
```

The simple flipbook animation loop is next. It will output 361 pages, one frame for each angle from 0° up to and including 360°. Within each iteration, a global variable `angle` is defined for other routines to use. The variable `angle` increases by 1° with each frame. Similarly, we declare a `time` variable that varies from $0.0 \rightarrow 1.0$ as `angle` varies from $0° \rightarrow 360°$. These routines (some of which we have yet to define below) set up the frame's initial coordinate system, label the frame, adjust the coordinate system and draw whatever we want, parameterized by `angle` or `time`. They finish up by outputting the current page:

```
/makeflipbook {                 % Call this routine to create entire flipbook
  /nFrames 360 def              % Define the number of frames (pages) minus 1
  0 1 nFrames {                 % Repeat nFrames+1 times, from 0 to nFrames
    /angle exch def             % Save loop index in a variable named angle
    /time angle nFrames div def % The animation "time" varies from 0.0 to 1.0
    preparepage                 % Install coord system, origin at page center
    labelframe                  % Draw any fixed label prior to setpointofview
    setpointofview              % Place or scale or rotate drawing w/r/t frame
    drawframe                   % Once placed, draw slightly changed contents
    showpage                    % Done -- "print" the page, and on to next one
    } for                       % Continue iterating everything in braces above
} def                           % Install this new command for later invocation

/labelframe { .5 setgray 17 -19 moveto angle(      )cvs show } def
```

`labelframe` prints in gray the frame's `angle`. We call it before calling `setpointofview`, so that the label's position remains fixed in each frame. `labelframe` isn't necessary, but it helps show why special patterns occur for certain angles. (If you cut and paste this code, ensure there are at least 3 spaces in the parentheses.)

That is essentially all there is to the generic PostScript flip-book template program. But we still have to customize it by defining what to draw, and where, for each value of either `angle` or `time`.

Drawing Graph Paper from Unit Squares. To reinforce concepts of coordinate systems, and to leverage off of the student's experience with simple graph paper, the next task was to draw an array of squares. But not in the usual way of drawing evenly spaced, cross-hatched lines. In PostScript, as well as many graphic libraries such as OpenGL, it is more useful to write commands that represent mathematically defined shapes in a canonical coordinate system. Then, you can manipulate and easily duplicate those shapes using composable affine transformations: `translate`, `rotate`, and `scale`. Path shape definitions also make it easier to understand introductory graphic concepts, such as stroking paths or filling closed paths, line weight, colors, etc. And the use of PostScript's composable mappings leads into the usefulness of a graphics state stack (`gsave` and `grestore`) that allows nested drawing contexts. So the next task was to define a command `drawfigure` that creates—but does not yet draw—a path representing a simple figure. Initially this was just a unit square: move the computer's virtual "pen" to the square's origin at $(0, 0)$, then draw a line to $(1, 0)$, then to $(1, 1)$, then to $(0, 1)$, and finish by looping back to wherever the sequence of lines started:

```
/drawfigure { 0 0 moveto 1 0 lineto 1 1 lineto 0 1 lineto closepath } def
```

Using this simple command, we draw the same figure over and over, except each time we're going to shift, stretch, and rotate our virtual *paper* underneath the "pen". So we define a `gridsize`, and then a routine to draw an initially horizontal line of `gridsize` unit squares. Each black-stroked square is first filled with a gray (and later, color) tone that varies from dark to light as squares are drawn "logically" from left to right:

```
/gridsize 12 def                % Eventually a 12 x 12 array at origin
/zeroAng 0 def                  % Rotating by zeroAng does nothing (no-op)

/rowofsquares {                 % This draws a rotated row of rotated squares
  zeroAng rotate                % Rotate entire line some angle
  gsave                         % Prepare to restore coordinate origin below
  1 1 gridsize {                % Push integer index, from 1 up to gridsize
    setcolor                    % Install a color based on current loop index
    zeroAng rotate              % Rotate w/r/t last square by some angle
    drawfigure fill             % Fill in a unit square with current color
    0 setgray drawfigure stroke % Outline same shape again, but in black
    1 0 translate               % Move origin right one unit square width
    } for                       % Draw it again, on "paper" shifted by (1,0)
  grestore                      % Restore graphics state and coordinates
  } def
```

To draw the frame, use `rowofsquares`[6] in another loop to draw the entire grid of squares, nominally in a vertical stack of rows of length `gridsize`.

```
/drawframe { gridsize { rowofsquares 0 1 translate } repeat } def
```

And finally, we set a fill color based on the loop index that is a hidden stack argument available at the start of each iteration, and we allow our point of view to change with time: the origin will rotate by some angle, initially `zeroAng`:

```
/setcolor { gridsize div setgray } def     % Set gray level stacked loop index
/setpointofview { zeroAng rotate } def     % Rotate "paper" by some angle

makeflipbook
```

The very last line of the program, `makeflipbook`, invokes our previously defined command, which sets all the above generic and specific flip-book apparatus "in motion". It creates a 361-page document in which each frame contains the image on the left side of Figure 1.

Swhirling Squares. So far, these routines draw nothing more than a square array of squares, looking like graph paper with a gradient tone from dark to light, left to right, in the upper right quadrant of each page. But this is where the animated fun begins (only a little of which is representable here in printed form).

[6]There are more general ways to provide arguments to subroutines in PostScript. Like Forth, PostScript has a postfix or "reverse Polish" syntax that relies heavily on a hidden run-time stack to push and pop arguments. But stack variables represent invisible state, which can be confusing, especially to a beginner. So for simplicity, this code avoids passing arguments via the stack.

In the foregoing code, there are three instances of the no-op command "`zeroAng rotate`", one in `setpointofview` and the other two in `rowofsquares`. To begin the animation, in `setpointofview` simply change the constant `zeroAng` to the variable `angle`. Now, as one quickly scrolls through the frames, the square grid, with its lower left corner at the origin, rigidly rotates a full circle about the origin. This is why we originally set the origin in the page's center.

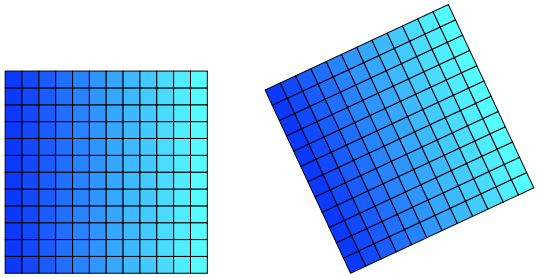

Figure 1: *Left: Frame* 0 *shows the* 12×12 *grid of squares, with gradient fill form left to right. Right: After changing the first* `zeroAng` *to* `angle`*, the* 25*th frame shows the grid rotated by* 25°*.*

Next, make the same change to the first `rotate` command on the first line of `rowofsquares`. Individual rows of squares will begin rotating with respect to previous rows, in addition to the rotation of the overall drawing. Figure 2 shows some frames very near angles that are multiples of integral divisors of 360°.

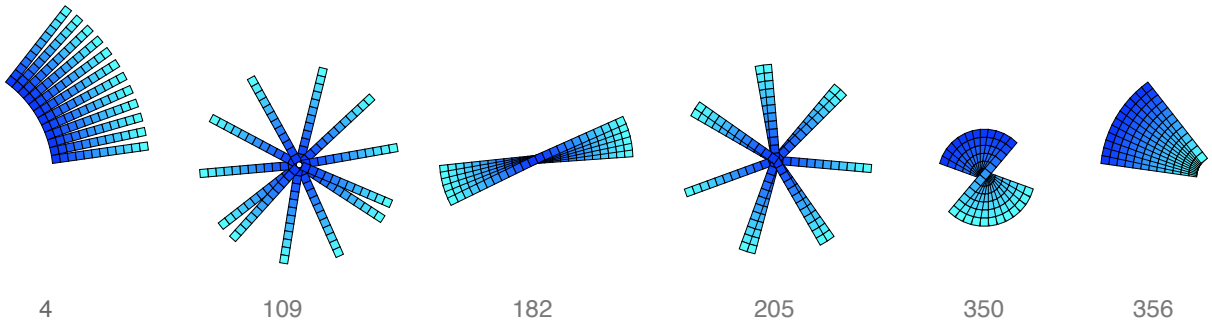

Figure 2: *Six frames for various angles for two added relative rotations.*

Finally, make the identical change to the second `rotate` command inside the loop of `rowofsquares`. This makes each individual square rotate with respect to its previous neighbor in the row, while still touching at a corner. And each row is rotating further away from its previous neighboring row. The addition of this third "degree" of rotation gets all the squares really "swhirling". Figure 3 shows some snapshots from the animation. Notice that for certain values of `angle`—those that are multiples of integral divisors of 360°—all 144 squares collapse into simple patterns comprising only a few squares, overlapping close to the origin (indeed, at 90°, *all* the squares coincide). Because of this, it is useful to change `setpointofview` to add a time-dependent scaling factor that zooms the "camera's" point of view in while `time` $< 1/2$, and zooms back out for `time` $> 1/2$. A time-based scale factor `f` $= 1.0 + 15 \times (0.5 - |$`time` $- .5|)$ works fairly well:

```
/setpointofview {
  angle rotate
  .5 time .5 sub abs sub 15 mul 1.0 add dup scale          %  f f scale
  } def
```

This slowly magnifies each frame's pattern so that at `angle` 180°, each square is 16 times its original size, roughly keeping drawings approximately the same size throughout the animation. It also makes it easier to see the kaleidoscopic patterns that almost magically coalesce and disappear over time. In reality, of course, the patterns demonstrate commensurate divisibility, fractional parts of 360°, polygons, envelopes, etc.

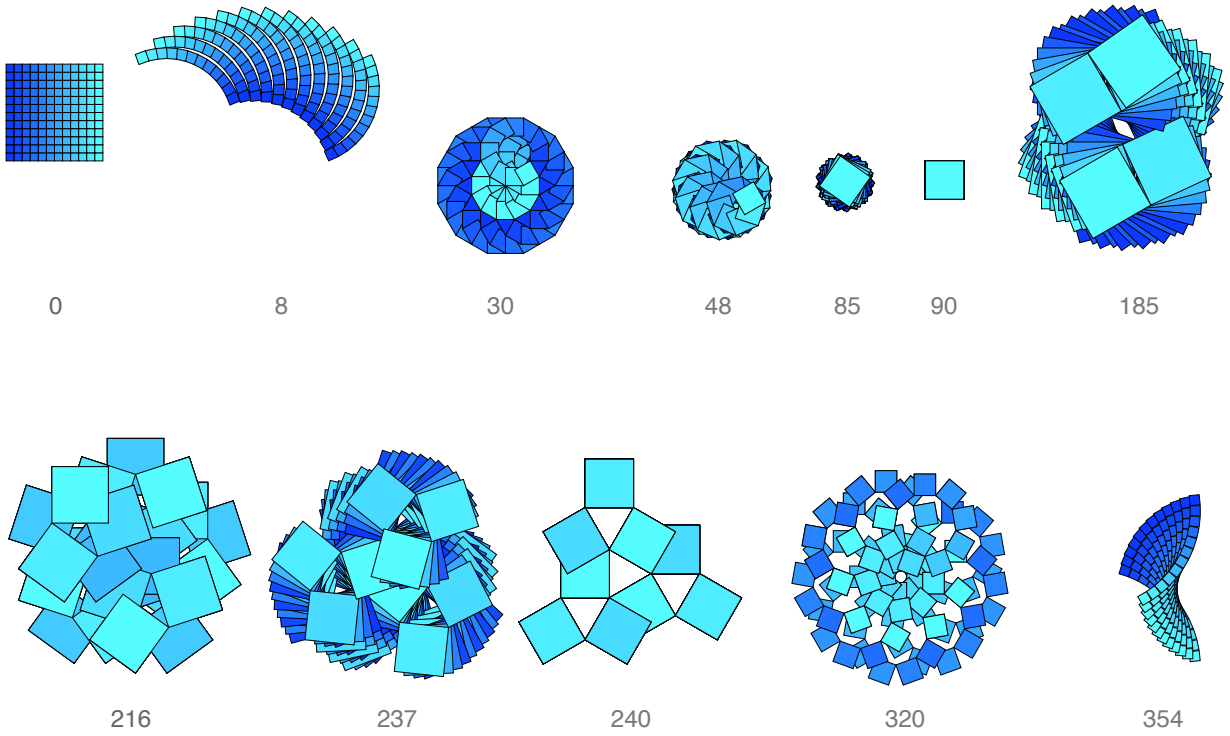

Figure 3 : *Twelve sample frames, with scaling towards 180°, after installing three positive rotations.*

Further Fun. At this point, we began experimenting with the animation or graphic parameters. First, we changed `setcolor` to use color rather than gray scale gradations (see the CD version of this paper):

```
/setcolor { gridsize div dup .3 mul exch 1 setrgbcolor } def % Nice set of blues
```

We then tried reversing one or two (but not all) of the rotations, by negating the value of `angle`. This was accomplished by substituting "`angle neg`" for "`angle`" (the PostScript `neg` command negates whatever precedes it). This led to one frame shown in Figure 4, which shows a flowing design with quite elegant form and an almost three dimensional feel due to the shading and order of drawing. It exhibits a nice balance of symmetry and asymmetry, a hallmark of worthwhile mathematical art. The question *Why is this beautiful?* leads into an interesting discussion about repetition, reference, mathematical form, curves, and æsthetics.

Another experiment was to make the angular rotation of the inner loop depend on the loop index, by changing the first two lines of `rowofsquares`' loop to:

```
dup setcolor          % Make a copy of loop index; set color using first copy
angle mul rotate      % Rotate by angle multiplied by other copy of the index
```

Figures 5 and 6 each show a frame with an interesting pattern that arose when `angle` was multiplied by the distance from the left that each individual square originally appears in. Very complex motions arise, with arms of squares expanding and collapsing, not unlike certain amusement park rides, or whirling dervishes.

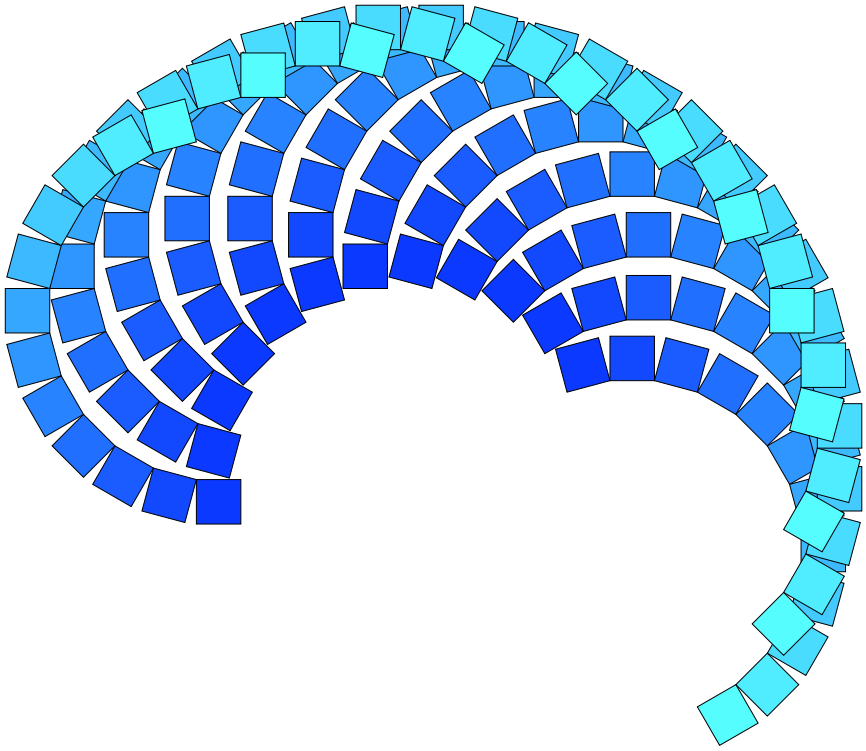

Figure 4 : *A particularly æsthetically satisfying frame, for* `angle` = $15°$*, using all three rotations, with the rotation in the inner loop reversed.*

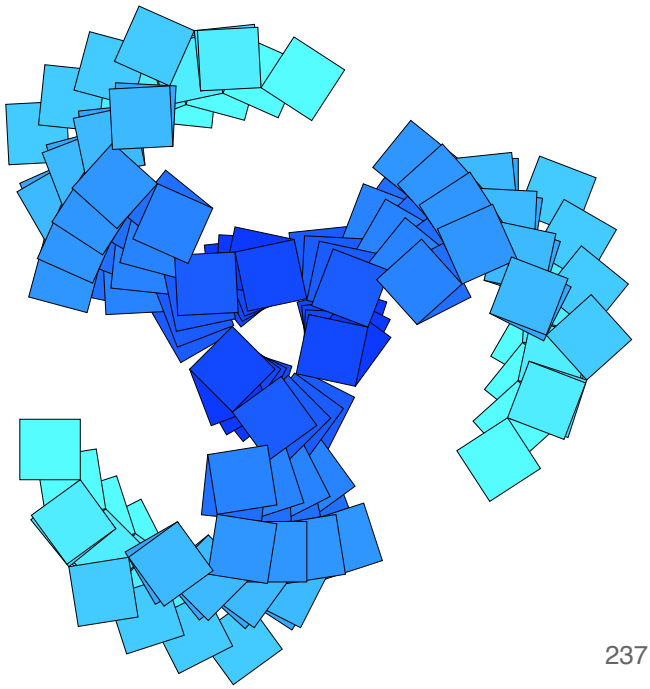

Figure 5 : *Here, the rotation in the inner loop has been multiplied by the count of squares from the left. Many squares will be drawn directly on top of each other at frame 240, three frames later.*

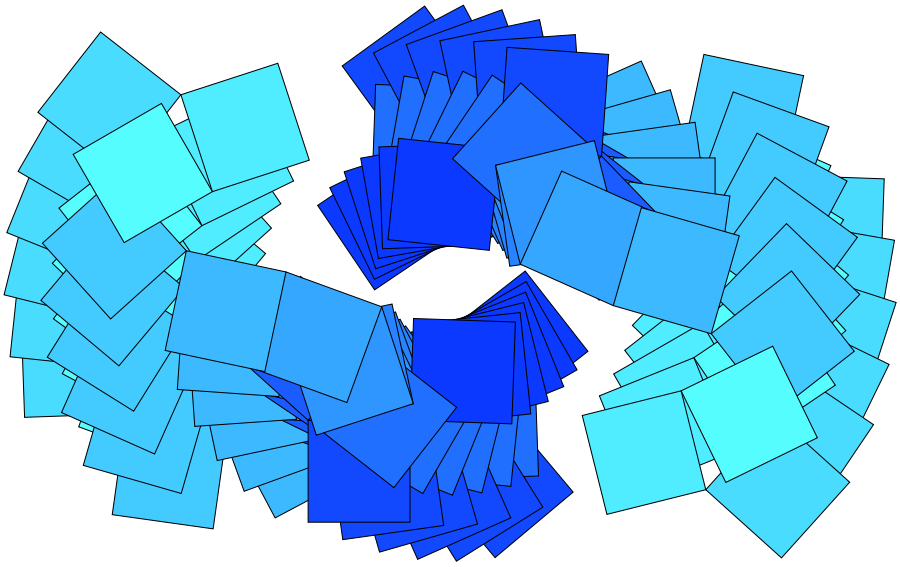

Figure 6 : *Frame 176 in the same continuous animation as Figure 5.*

Conclusion. One can create simple PDF flip-books by hand and view them as animations on any computer that supports a multi-page, PostScript file previewing application. PostScript is a fairly low level tool for creating precisely drawn mathematical figures, and as such is not ideal for beginners—there are no doubt better, higher-level, sophisticated animation tools. On the other hand, the PostScript language is mathematically sophisticated and available on nearly all personal computers, making it easy to do quick interactive experiments, using only a very small number of the language's commands.

An animated flip-book lets you create and view parameterized geometries that change over time. With a basic and short initial template program one can easily create a rich set of images (frames) that capture the "animagination" as the designs morph from one to another over time. For students, PDF flip-books provide a visual hook into further pedagogical discussions of important mathematical, design, or æsthetic principles. And they provide a platform for experimentation, playfulness, and for demonstrating how math and algorithms can be combined to create art that communicates less-obviously-mathematical ideas. With the ubiquity of laptop computers among students who are already familiar with computer-animated movies, mathematical flip-books—whether in PostScript or some other language or system—should be part of the toolkit of every mathematics teacher, and of anyone interested in mathematical art.

References

[1] Gardner, M., "Curves of Constant Width", ch. 18, *The Unexpected Hanging and Other Mathematical Diversions*, Simon & Schuster (1969), pp. 212–221.

[2] Adobe Systems, Inc., *PostScript Language Reference Manual, Second Ed.*, Addison Wesley (1990) (see also any of numerous tutorials published on the internet).

[3] Lutz, Mark, *Programming Python*, O'Reilly (2001); see also `http://www.python.org`

[4] Sweigart, Al, *Invent Your Own Computer Games with Python* (2008), a self-published PDF book, freely available at `http://inventwithpython.com` (as of Feb. 1, 2010).

[5] The Watershed School, Boulder, Colorado, `http://www.watershedschool.org` (as of Feb. 1, 2010).