Mathematics 308 — Fall 98

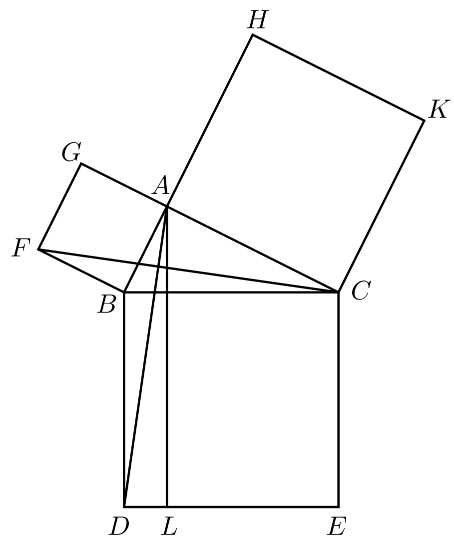
An animation exercise

Animation can often make mathematical proofs enormously easier to understand. In this note we shall see how it works with Euclid's proof of Pythagoras' Theorem.

Euclid's version

This from Book I of *The Elements of Geometry*:

Proposition 47. *In right-angled triangles, the square on the side subtending the right angle is isequal to the squares on the sides containing the right angle.*



Proof. *Since each of the angles BAC and BAG is right, it follows that with a straight line BA, and at the point A on it, the two straight lines AC and AG not lying on the same side make the adjacent angles equal to two right angles, therefore CA is in a straight line with AG.*

For the same reason BA is also in a straight line with AH.

Since the angle DBC equals the angle FBA, for each is right, add the angle ABC to each, therefore the whole angle DBA equals the whole angle FBC.

Since DB equals BC, and FB equals BA, the two sides AB and BD equal the two sides FB and BC respectively, and the angle ABD equals the angle FBC, therefore the base AD equals the base FC, and the triangle ABD equals the triangle FBC.

Now the parallelogram BL is double the triangle ABD, for they have the same base BD and are in the same parallels BD and AL. And the square GB is double the triangle FBC, for they again have the same base FB and are in the same parallels FB and GC.

Therefore the parallelogram BL also equals the square GB.

Similarly, if AE and BK are joined, the parallelogram CL can also be proved equal to the square HC. Therefore the whole square $BDEC$ equals the sum of the two squares GB and HC.

And the square $BDEC$ is described on BC, and the squares GB and HC on BA and AC.

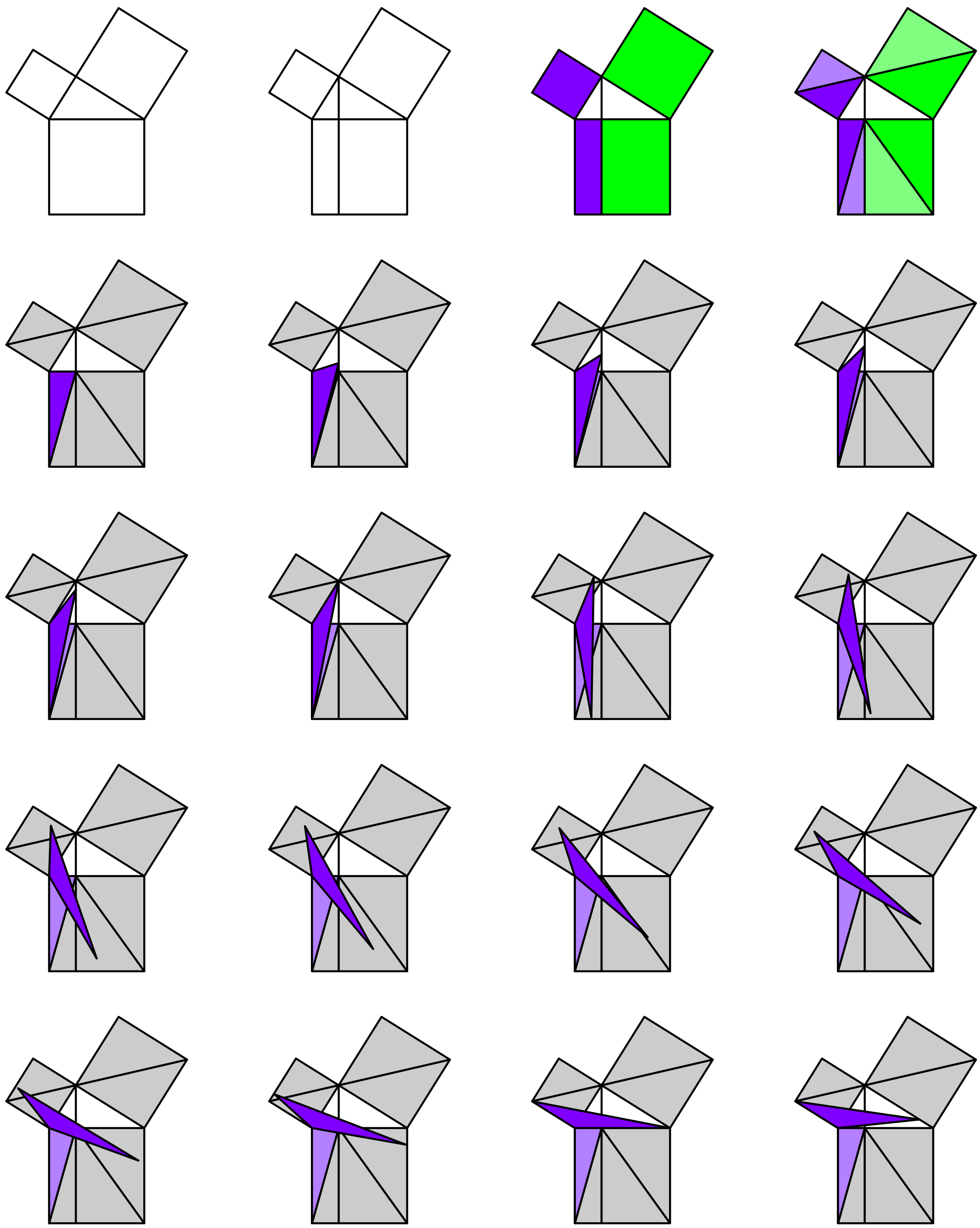
Therefore the square on BC equals the sum of the squares on BA and AC.

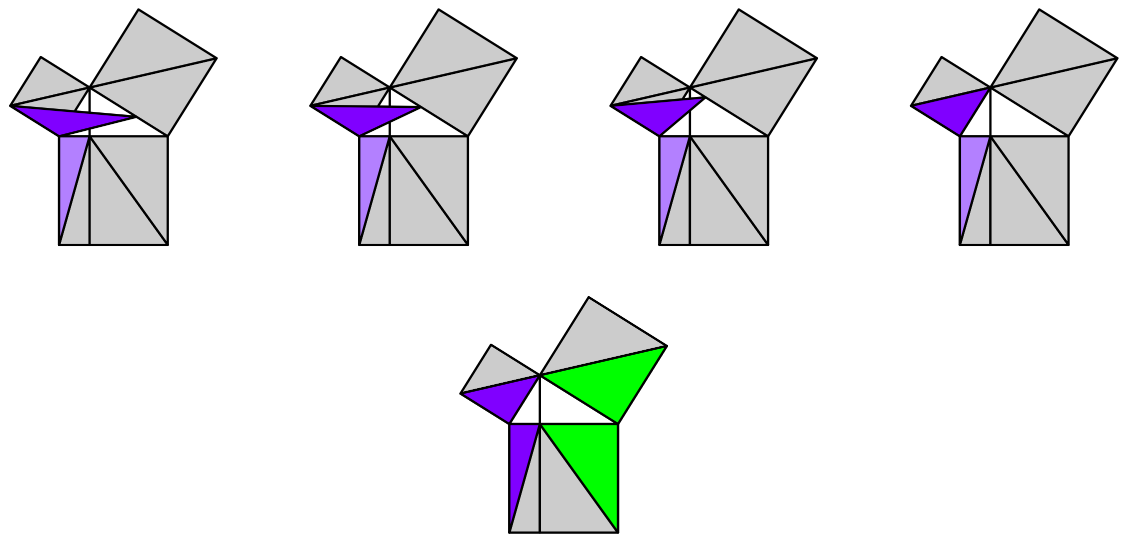
Therefore in right-angled triangles the square on the side opposite the right angle equals the sum of the squares on the sides containing the right angle.

Q. E. D.

Animating it

At first sight, Euclid's proof doesn't look appealing. The German philospher Schiller is said to have called it a 'proof walking on stilts'. But in fact the lack of appeal is a matter of cosmetics, and the underlying ideas are rather elegant. The elegance can be brought out by turning his argument into an animated sequence of pictures.





How it's done

Doing this is entirely possible with the tools you already have at your disposal, but a few new ones will make things more efficient.

To start with, something very simple. What are the coordinates of the point A? From an argument about similar triangles we have

$$\frac{x}{a} = \frac{a}{c}$$
$$x = \frac{a^2}{c}$$
$$\frac{y}{a} = \frac{b}{c}$$
$$y = \frac{ab}{c}$$

So we put into our program

```
/c a a mul b b mul add sqrt def
/x a a mul c div def
/y b a mul c div def
```

The major new tool will be **arrays**. The figures above are made up of a collection of shapes, mostly triangles and squares, which are drawn over and over again. Of course we already know how to make up a procedure to draw each one of these. For example, we can define

```
/asquare {
  0 0 moveto
  x y lineto
  x y sub x y add lineto
  y neg x lineto
} def
```

to draw the left hand square in Euclid's picture, if (x, y) is the point A. We can even allow procedures to have parameters to allow us to handle animation easily. But things will be far more efficient if we can refer to ponts as single objects, rather than by referring to theior coordinates separately. The way to do this is to define a point to be an array of two numbers. For example

```
/A [x y] def
```

defines the point A to be the array with coordinates (x, y). Arrays can be of any length. How are their entries accessed? The items in an array are indexed starting from from 0. This the first element of the array has index 0 and the last one has index $n - 1$ if there are n items altogether. If A is an array then `A 0 get` returns its first element and `A i ge` returns its i-th. Furthermore, `A length` returns the number of items in it. For example, the sequence

```
A 0 get A 1 get moveto
B 0 get B 1 get lineto
```

will construct the line from A to B.

We can also think of shapes themselves as arrays—i.e. as an array of points. Therefore we might represent the square on a as the array `[A G F B]`. And we can define a procedure which has sucj an array as a single argument and constructs the path it determines:

```
% A polygon = array of at least one point
% This procedure has an array P of points [x y]
% as argument and builds the path P[0] ...  P[n-1]

/mkpolygon {
2 dict begin
/poly exch def
/n poly length def
poly 0 get
aload pop moveto
1 1 n 1 sub {
  % i on stack
  poly exch get
  aload pop lineto
} for
end
} def
```

Note that the path is not closed up, so we have to do that ourselves after calling it, we want a closed path. Here I have used the commnad `aload` which has the following effect: if a is an array then `a aload` unpacks the array a, puts its contents on the stack, and also a copy of a itself. Thus

```
[x y] aload pop
```

puts on the stack

```
x y [x y]
```

Therefore `[x y] aload pop` just puts `x y` on the stack.

Using arrays for points can also make animation easier. In animating, we frequently want to construct a series of evenly spaced points starting at one point P and ending up at another point Q. This is called **linearly interpolating** the points P and Q. For this we need to know that the point t of the way from P to Q is given by the formula

$$(1 - t)P + tQ = P + t(Q - P)\ .$$

Here is a PostScript procedure which implements this formula:

```
/interpolate {
4 dict begin
/t exch def
/s 1 t sub def
/Q exch def
/P exch def
[
```

```
  P 0 get s mul Q 0 get t mul add
  P 1 get s mul Q 1 get t mul add
]
end
} def
```

Finally, it will help if we have our own procedures to rotate points, instaed of relying on PostScript's `rotate` command.

```
% P t => P rotated by t

/rotate-2d {
4 dict begin
/t exch def
/P exch def
/c t cos def
/s t sin def
[
  P 0 get c mul P 1 get s mul sub
  P 0 get s mul P 1 get c mul add
]
end
} def
```

We can even rotate whole polygons.

```
% polygon t

/rotate-polygon {
4 dict begin
/t exch def
/poly exch def
/n poly length def
[
  0 1 n 1 sub {
    poly exch get t rotate-2d
  } for
]
end
} def
```

We can now see how all these work together. In the course of drawing the figures above, we show a certain triangle rotating. We do this by

```
newpath
[B D A] t rotate-polygon mkpolygon
closepath
gsave
blue
fill
grestore
stroke
```

We can also shear:

```
newpath
[B D M A t interpolate] mkpolygon
closepath
gsave
blue
```

```
fill
grestore
stroke
```

But is it really Euclid?

The sequence of figures does not follow Euclid's argument exactly, but nonetheless almost everything Euclid mentions plays a role in the graphics. And of course the pictures themselves do not provide a real proof, but only suggest one. To get it all, one has to add details explaining why the figures work. Sample questions one could ask are: *Where does Euclid's initial argument about straight lines come in? What does it take to justify the rotation sequence?* But it is undeniable that the pictures not only suggest answers to these questions, and even suggest others, but also interest one in a second look at Euclid's argument.