A Program to Interpolate (and Extrapolate) Between Turtle Programs

Ken Kahn
London Knowledge Lab
Institute of Education, University of London
23-29 Emerald Street
London, WC1N 3QS, UK
kenkahn@toontalk.com

Abstract

People have been creating geometric figures with computer programs consisting of turtle commands such as forward and right since the late 1960s [1]. Here I describe a program that takes in two such programs and produces a new program capable of producing both figures and all the intermediate figures. It can produce a figure that is one third circle and two thirds triangle or one that is half star and half pentagon. The program produced by interpolating, say, a square and a circle program takes in a number between zero and one and produces a figure between a square and a circle. If, however, it is given a number greater than one, or a negative number, it will produce an extrapolation between a square and circle.

Interpolated programs can be the basis of playful aesthetic explorations. The intermediate forms can be drawn on the same image. Or animations can be generated where the figures morph into (and beyond) each other. Colours and other attributes of the turtle pen can also be interpolated. Unlike conventional morphing programs, we are interpolating between computational processes rather than static images.

Interpolating and Extrapolating Geometric Figures

Imagine one had a program that could draw a blue square, a red triangle, and any intermediate shape. One could use it to produce the following image by calling the program with inputs 0.0, 0.01, 0.02, …, 0.98. 0.99, and 1.0:

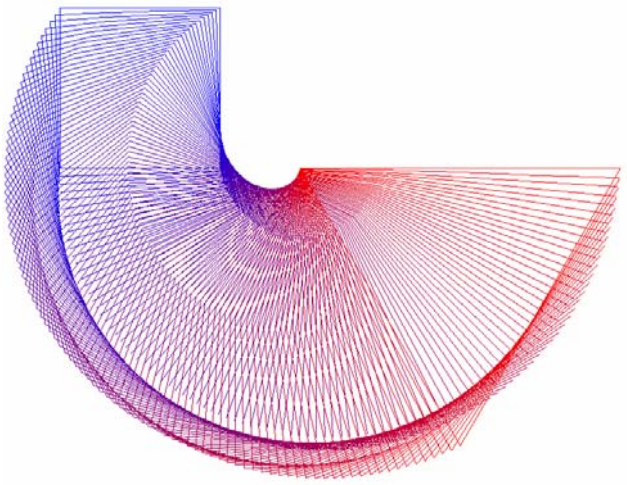

Figure 1: *Interpolation between a square and triangle*

If the input to the procedure is 0 it draws a square, if it is 1 it draws a triangle, and if it is 0.5 then it draws something that is half square and half triangle (an *interpolation*). But what if the input is 2? Can it draw

something that is beyond triangle when starting from a square? Or if it is -1 can it draw something that is before square in the transition to triangle? Such figures are *extrapolations* between a square and triangle.

Here are 100 figures extrapolating between -0.5 to 1.5 for a circle (really a 360-side polygon) and a triangle. The original circle is at 9 o'clock in the image and the triangle is at 3 o'clock.

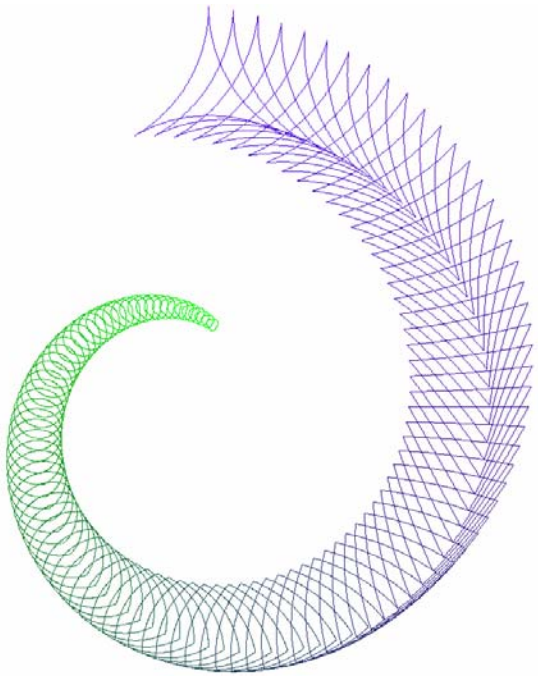

Figure 2: *Circle to triangle from -0.5 to 1.5 in 0.02 increments*

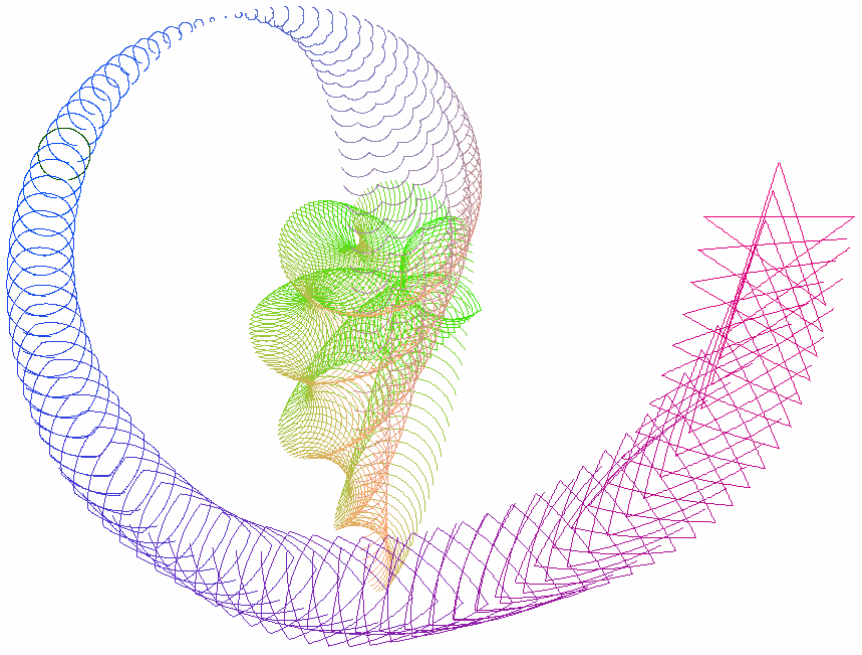

Figure 3: *A circle to a five-pointed star from -2 to 1 incremented by 0.02*

More extrapolations are illustrated in Figures 4 to 8.

How to interpolate turtle programs

Rather than building a special program for each pair of figures I wanted extrapolations, I built a program in Logo that takes as input two turtle Logo programs [1] and produces a new program. Consider the very simple example where the two programs draw lines:

```
to short
 forward 40
end
to long
 forward 100
end
```

The resulting interpolated program produced by my program is:

```
to short_to_long :x
 forward interpolate :x 40 100
end
```

Where *interpolate* is defined as:

```
to interpolate :x :a :b
 output :a + :x * (:b - :a)
end
```

When *short_to_long* is called with 0 it draws a line 40 units long and with 1, it is 100. When called with 0.5 the line is 70 units long, the average value. When called with 2 it is 160 units long, an extrapolation.

A challenging task is to produce an interpolation between a triangle drawn by **`repeat 3 [forward 100 right 120]`** and a square drawn by **`repeat 4 [forward 100 right 90]`**. The two programs need to be placed in a *canonical form* before the interpolation can be generated. My program does this by first computing the least common multiple of the repeat counts. The triangle program can be transformed to make 4 calls to *forward* repeated 3 times while the square program can be transformed to make 3 calls to *forward* repeated 4 times. A single call to *forward* is automatically transformed into many calls to *forward* by going forward a fraction of the original distance each time. However, for interpolation we need a standard form so the two programs "line" up. My program interpolator uses sequences of calls to *forward* and *right*. The triangle program is transformed to

```
repeat 3 [forward 100/4 right 0
          forward 100/4 right 0
          forward 100/4 right 0
          forward 100/4 right 120]
```

And the square program becomes

```
repeat 4 [forward 100/3 right 0
          forward 100/3 right 0
          forward 100/3 right 90]
```

Clearly the drawings produced by these programs will be identical to the originals. The next step is to "open code" the *repeat* statements so that we can interpolate corresponding commands. In the following table corresponding commands from the triangle and square programs are interpolated. When the values are the same there is no need to generate interpolation code since the value will be constant.

`to triangle`	`to square`	`to triangle_to_square :x`
`forward 100/4`	`forward 100/3`	`forward interpolate :x 100/4 100/3`
`right 0`	`right 0`	`right 0`
`forward 100/4`	`forward 100/3`	`forward interpolate :x 100/4 100/3`

`right 0`	`right 0`	`right 0`
`forward 100/4`	`forward 100/3`	`forward interpolate :x 100/4 100/3`
`right 0`	`right 90`	`right interpolate :x 0 90`
`forward 100/4`	`forward 100/3`	`forward interpolate :x 100/4 100/3`
`right 120`	`right 0`	`right interpolate :x 120 0`
`forward 100/4`	`forward 100/3`	`forward interpolate :x 100/4 100/3`
`right 0`	`right 0`	`right 0`
`forward 100/4`	`forward 100/3`	`forward interpolate :x 100/4 100/3`
`right 0`	`right 90`	`right interpolate :x 0 90`
`forward 100/4`	`forward 100/3`	`forward interpolate :x 100/4 100/3`
`right 0`	`right 0`	`right 0`
`forward 100/4`	`forward 100/3`	`forward interpolate :x 100/4 100/3`
`right 120`	`right 0`	`right interpolate :x 120 0`
`forward 100/4`	`forward 100/3`	`forward interpolate :x 100/4 100/3`
`right 0`	`right 90`	`right interpolate :x 0 90`
`forward 100/4`	`forward 100/3`	`forward interpolate :x 100/4 100/3`
`right 0`	`right 0`	`right 0`
`forward 100/4`	`forward 100/3`	`forward interpolate :x 100/4 100/3`
`right 0`	`right 0`	`right 0`
`forward 100/4`	`forward 100/3`	`forward interpolate :x 100/4 100/3`
`right 120`	`right 90`	`right interpolate :x 120 90`
`end`	`end`	`end`

Table 1: *The interpolation of two expanded turtle programs*

After adding a few commands to the triangle and circle programs to set the pen colour and initial position Figure 1 was produced by 100 calls to *triangle_to_square* with arguments ranging from 0 to 1.

The following figures were generated by extrapolating between a 9-pointed star, **`repeat 9 [forward 100 right 80],`** and a doubly-drawn polygon with 360 sides, **`repeat 360 [forward 1 right 2]`**. One obtains very different (but also appealing) results if a normal circle program is used instead. Notice how "nine-ness" pervades all the figures. Both figures are drawn with two alternating colours.

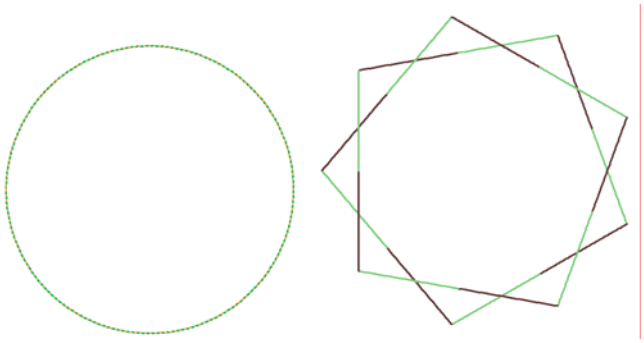

Figure 4: *The double circle (x = 0.0) and 9-pointed star (x = 1.0)*

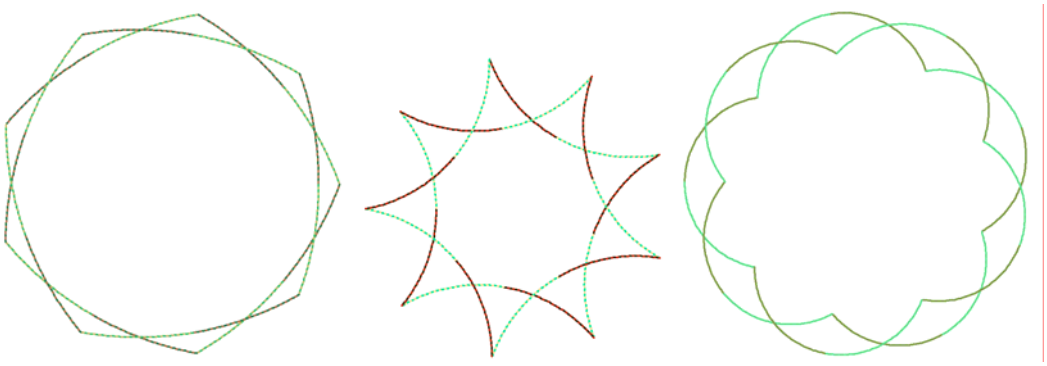

Figure 5: *Circle to Star Extrapolations, x = 0.5, x = 2, x = -1*

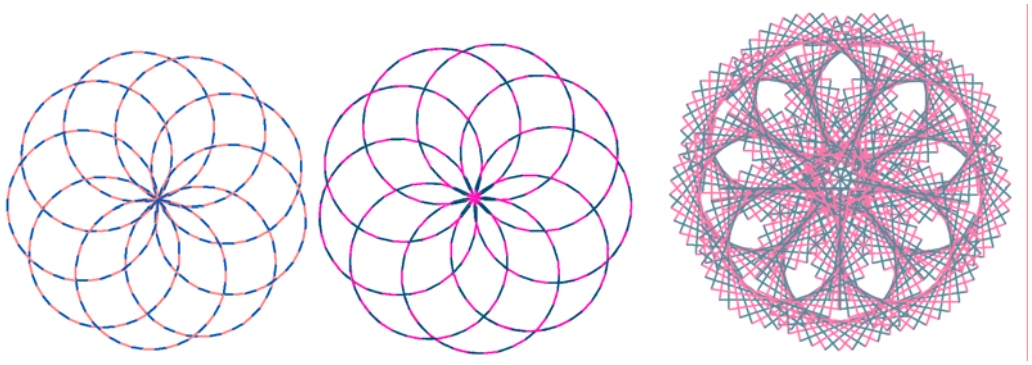

Figure 6: *Circle to Star Extrapolations, x = 10, x = -10, x = 100*

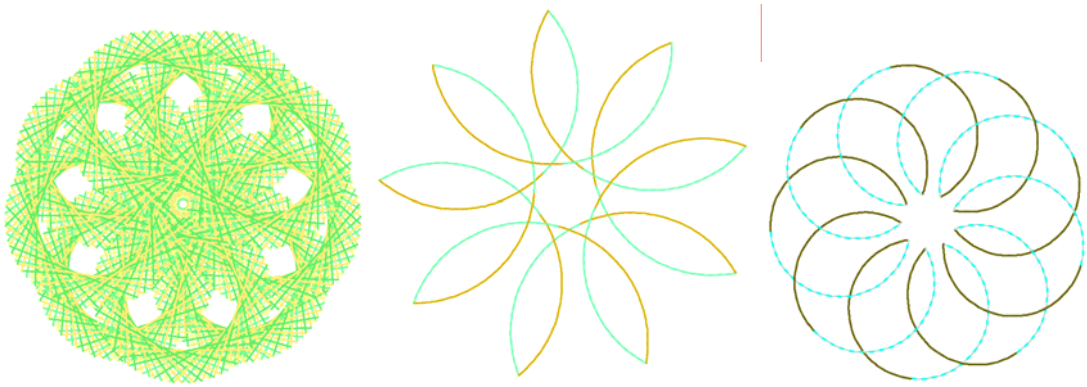

Figure 7: *Circle to Star Extrapolations, x = -100, x = 3.14159, x = -3.14159*

Isn't this "just" morphing?

Films and music videos have been morphing images and shapes to produce intermediate images and shapes for decades. How is this different? My program interpolates between other *programs* – not images or shapes. It automates the process of finding corresponding elements needed in morphing.

A program for a shape contains more information than the shape itself. A circle drawn once or twice looks the same but interpolates differently; as does a circle drawn clockwise instead of counter-clockwise. The *processes* for drawing two figures are what is being interpolated – not the figures themselves.

What does it really mean to extrapolate programs?

It is hard to get an intuition for program extrapolation in general. Some simple cases are intuitive. A positive extrapolation from a short line to a long line will be lines that are even longer. A negative extrapolation will be lines that are even shorter until a zero-length line is reached. Further negative extrapolation produces lines that start at the same location but grow in the other direction. The extrapolation from a small figure to a similar larger figure will be the figure even larger. (By "similar" I mean the sense in geometry that the two figures have identical angles and the same ratio of lengths of sides.) The extrapolation between a straight line and two lines that form a right angle will be angles greater than 90 degrees. What is hard to think about, for example, is the extrapolation of a circle to a triangle. How can something be more of a triangle than a triangle?

Are there always cycles in the extrapolations?

Corresponding turtle commands are matched with an interpolation function between the two values. We can see that each interpolated *right* command will cycle since for every unit increase in the interpolation/extrapolation parameter the angle is increased by the difference between the two angles in the original programs. So the values will cycle with a period of the greatest common divisor of the angle difference and 360. The full cycle is the least common multiple of the cycle length of each interpolated call to *right*. Similarly, the red, green, and blue colour components will cycle since they each have 256 distinct values.

Hence the sequence of the extrapolation between any two turtle programs whose sides have the same ratios will cycle except for the scale of the figures. If the canonicalised form of the two programs has identical calls to *forward* then the scale will remain constant.

Can all turtle programs be interpolated?

My program interpolator currently only works with programs built using the following primitives: *forward*, *right*, *repeat*, *setPenColour*, *penUp*, and *penDown*. This list could be greatly expanded. It is an open research question as to how to deal with conditionals, recursion, state variables, and calls to *repeat* with a non-constant repeat count. Note, however, that the current set of primitives is adequate for expressing all *fixed instruction turtle programs*. Only figures produced by infinite processes (e.g., fractals or spirals) are missing.

The history and future of the program interpolator

I first built the program interpolator in 1978. I used it an "improvisational" computer animated film called *Two Nights on a Computer* that was shown in the Boston area and in some film festivals. I recreated the program interpolator in 2006 in Imagine Logo. I have no plans to develop it further (though I may just for its recreational value). Anyone who wants to explore it further (either aesthetically or computationally) is welcome to the source code.

References

[1] Harold Abelson and Andrea A. diSessa. *Turtle Geometry, the Computer as a Medium for Exploring Mathematics*. The MIT Press, Cambridge, MA, 1981.