Complex Drawing with jit.lcd

The jit.lcd object is a blank canvas. You can draw on it (using a set of commands derived from the old QuickDraw technology) and the results appear in a jit.matrix ready for display or further processing. Jit.lcd is almost identical to lcd[1]. The major difference is that jit,lcd is a object box in the patch, not a user interface object.

Some Conventions

You can think of the jit.lcd (or the plain lcd) as a chunk of screen consisting of a rectangular array of pixels. Pixels are numbered from the upper left corner, across and down[2]. The top left one is 0, 0. Technically, the numbers refer to points, which is the spot at the upper left of a pixel, but that will seldom matter.

Many commands require the specification of rectangles or rects. A rect is defined by two points, left top and right bottom. So 10 20 40 50 would define a square that started 11 pixels from the left and 21 down from the top and 30 pixels across and down. Note that pixel 40 50 is not in this rectangle.

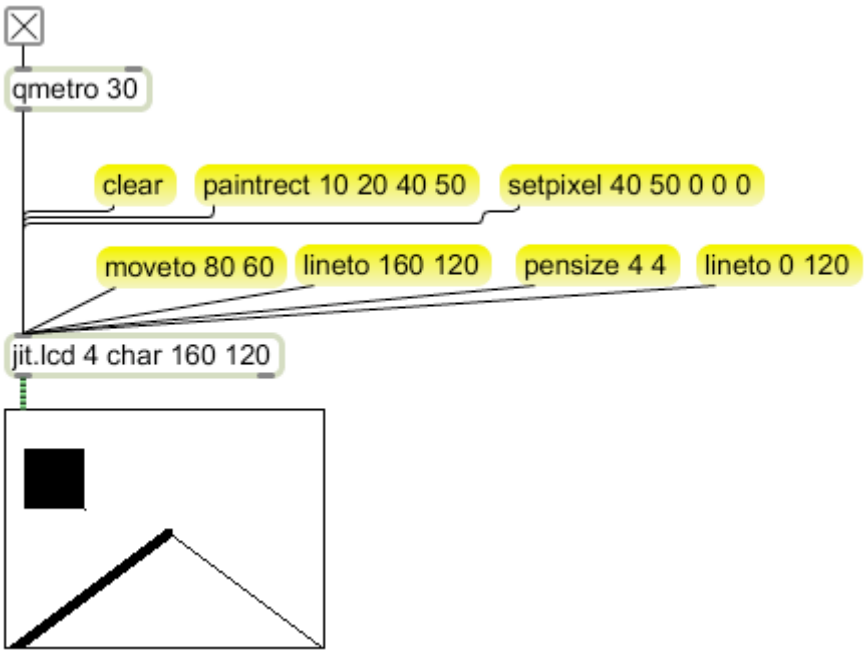


Figure 1.

Figure 1 illustrates the easy way to use jit.lcd. We supply bangs to jit.lcd to show the drawing in a jit.pwindow. Figure 1 also illustrates some commands:

[1] See “Max & Graphics” for a detailed description of lcd.

[2] Apple has been counting pixels this way since the original Macintosh. They decided to change with OS X, which counts from the left bottom, PC style. The transition should be interesting.

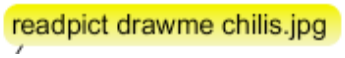
- Clear[3] fills the entire lcd with the background color
- Paintrect draws a colored rectangle in the rect defined.
- Setpixel draws a single pixel.
- Moveto puts the drawing pen on the pixel specified. This will be the origin of line commands and text (ascii) commands.
- Lineto draws to the point specified.
- Pensize sets the width of lines.

Other Commands, which do pretty much what you'd expect:

- Framerect draws the stroke of a rectangle
- Paintoval draws an ellipse in the specified rectangle
- Frameoval draws the stroke
- Paintarc draws part of an ellipse two more arguments specify start angle (0 is the top) and degrees of arc.
- Framearc draws a stroke
- Linesegment draws from a point to another point.
- paintroundrect draws a rectangle with round corners two extra arguments control horizontal rounding and vertical rounding.
- Frameroundrect
- Paintpoly draws a shape connecting a series of points. The last connects to the first.
- Framepoly outlines a shape connecting a series of points. If the first and last are different, there will be a gap.

Images

There are two ways to put an image into jit.lcd. One is to load a matrix, which will become the background for further drawing. The other is to create a pict object. The message to do this is readpict.



The message drawpict will place the image in the jit.lcd. The arguments (despite what you may read in the documentation) are

- Left
- Top
- Horizontal size of drawing
- Vertical size of drawing
- Left origin in source (the source is rescaled to the jit.lcd size)
- Top origin in source
- Right origin in source
- Bottom origin in source

[3] Commands in jit.lcd are all lowercase. Microsoft objects to that here.

LCD colors

You will notice that setpixel has more argument numbers than needed to specify which pixel we are talking about. The extra three set the color in RGB format. 0 0 0 indicates black. You can follow most commands with an RGB triad to specify color, or a single number that refers to an indexed color. The indexed colors are shown in figure 2:

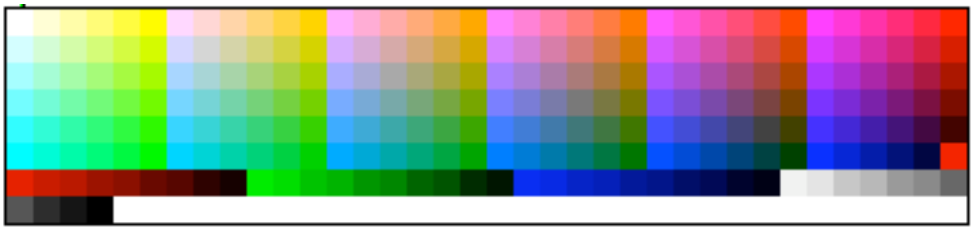


Figure 2. Index colors 0 to 255

This array is 36 across by 8 down. The colors are reasonably well behaved, stepping B, G and R values by 51. At 215 you get a set of pure reds, then greens at 225, blues at 235 and grays at 245. The last color, 255 is black. The color n message sets the foreground color.

A background color can be specified by the brgb command, which takes 3 arguments. The default foreground color can be set by frgb. Foreground will also be changed by RGB arguments to the paint or frame commands. To make matters further confusing, lineto does not accept RGB color arguments and setpixel requires them.

Text

You can write in jit.lcd. First use moveto to define where printing starts. This is the origin of the first letter, (lower left unless the letter descends below the line).

Set the font with a font message. Fonts can specified by number, but using the name is better. The syntax is [font name size]. The size is in points.

The style is set with the textface message. Options are normal, bold, italic, underline, outline, shadow, condense, extend.

Printing the text can be done with the write message, which takes symbols, or the acsii message, which works directly with ascii values. (see Max and ASCII)

Basic drawing

I do 99% of my drawing in the jit.lcd one pixel at a time using paintrect. I use paintrect instead of setpixel because I sometimes paint with larger squares as they project better. (There's no obvious difference in speed.) I simplify the process with a couple of my Lobjects. The basic patch is shown in figure 3. On each qmetro tick:

- A bang is sent somewhere to generate the image data. (Most of the other readers are about how this is done.)
- The data returns to newpoint as a list of X and Y.
- A multiplier may be used to spread the points out.
- X and Y are unpacked and repacked with color data in the format X,Y,X,Y,R,G,B. (Don't forget to set the swatch to output 0-255 values.)
- An offset is added to move the image to the origin (generally the middle of the screen.) The second X and Y values get the origin and rectangle width added.
- The paintrect command is prepended and sent to the jit.lcd
- A bang arrives from qmetro to output the matrix.

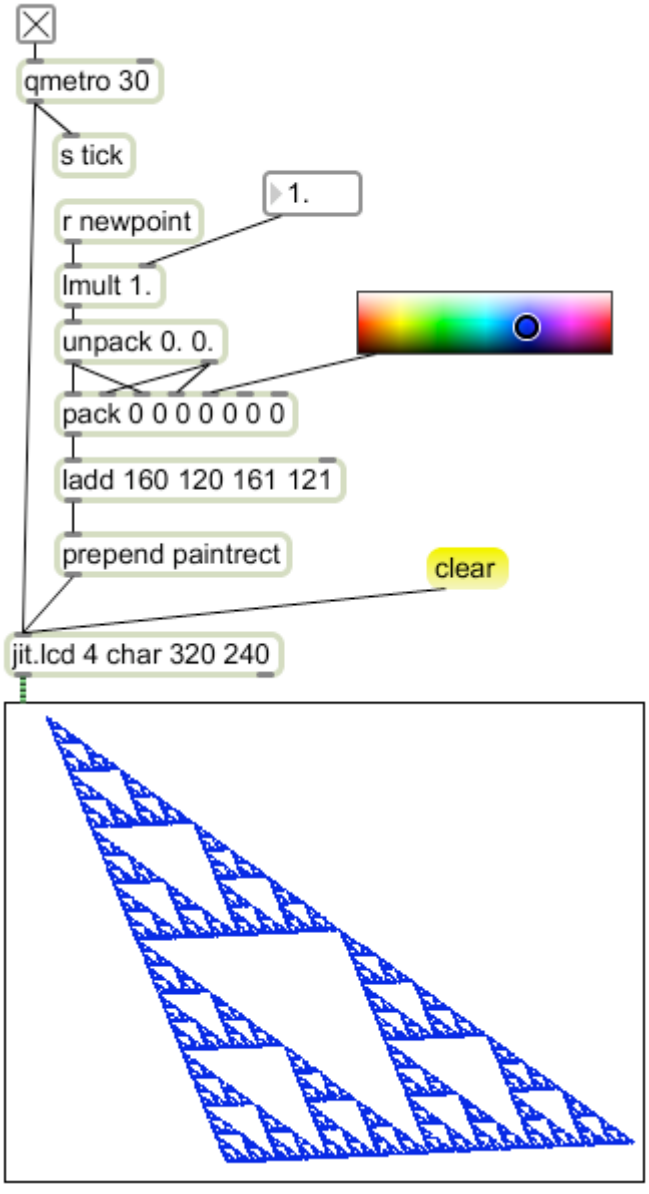


Figure 3.

This drawing is cumulative, meaning all of the points remain until I clear it by hand. If I'm doing an animation, the jit.lcd must be cleared before any drawing is done. I add a trigger object to the qmetro to ensure the proper order of things.

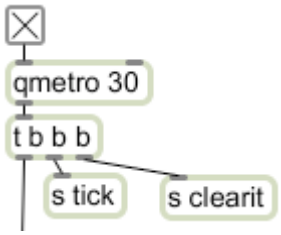


Figure 3A

Drawing with framepoly

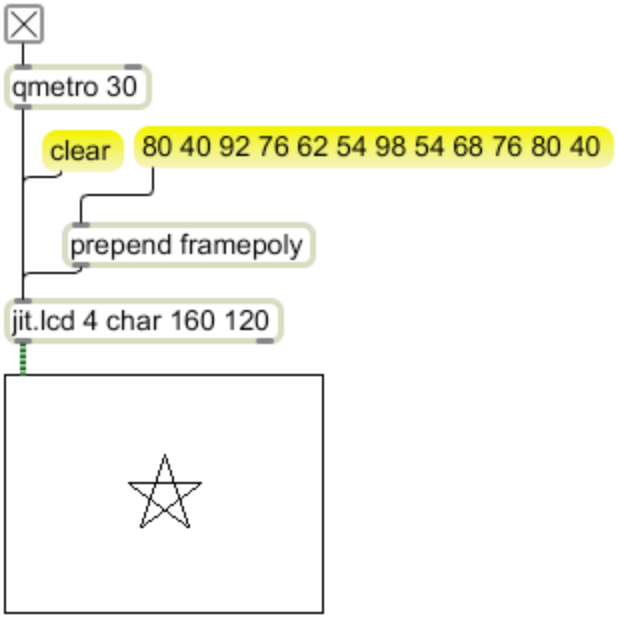


Figure 4.

Figure 4 shows one way to use framepoly (or paintpoly) commands to draw a simple geometric shape. You create a drawing script of points and prepend the command. Several point lists can easily be contained in a coll. In this case the points are given relative to the origin in good quickdraw style. It is a bit more awkward but ultimately more useful to draw relative to the center of the display. Figure 5 shows how.

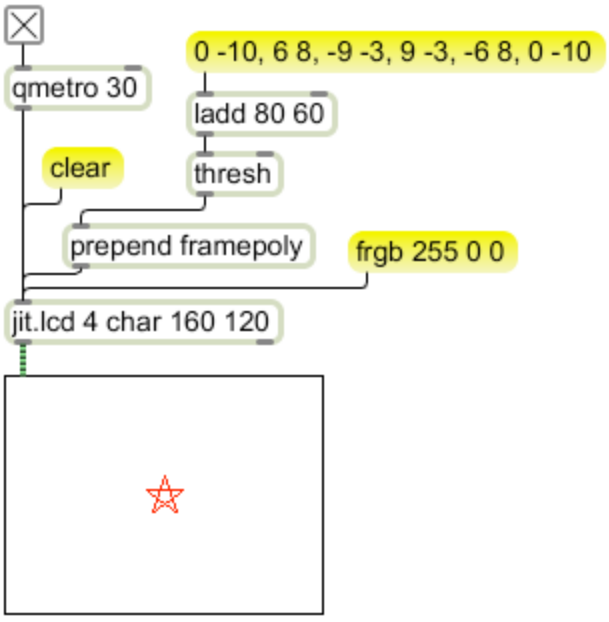


Figure 5.
The list of points is built up by adding the point at the center of the display to each point. Negative points appear in the upper left of the drawing. The results are then threshed to form the arguments to framepoly. The advantage of working with relative shapes is that it is easy to modify the image.

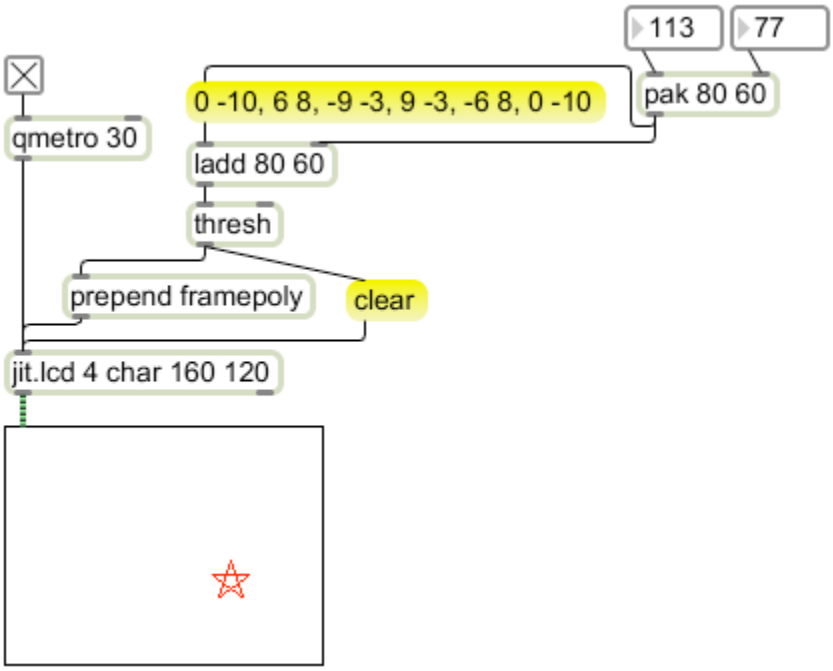


Figure 6.

In figure 6, I change the origin to move the shape around the display. Note that the jit.lcd is cleared immediately before drawing. There's a bit of an art to picking the point in the

patch to trigger the clear. If too much calculation happens between the clear and the draw command, the image will flicker. If the clear happens too late, the display will be blank.

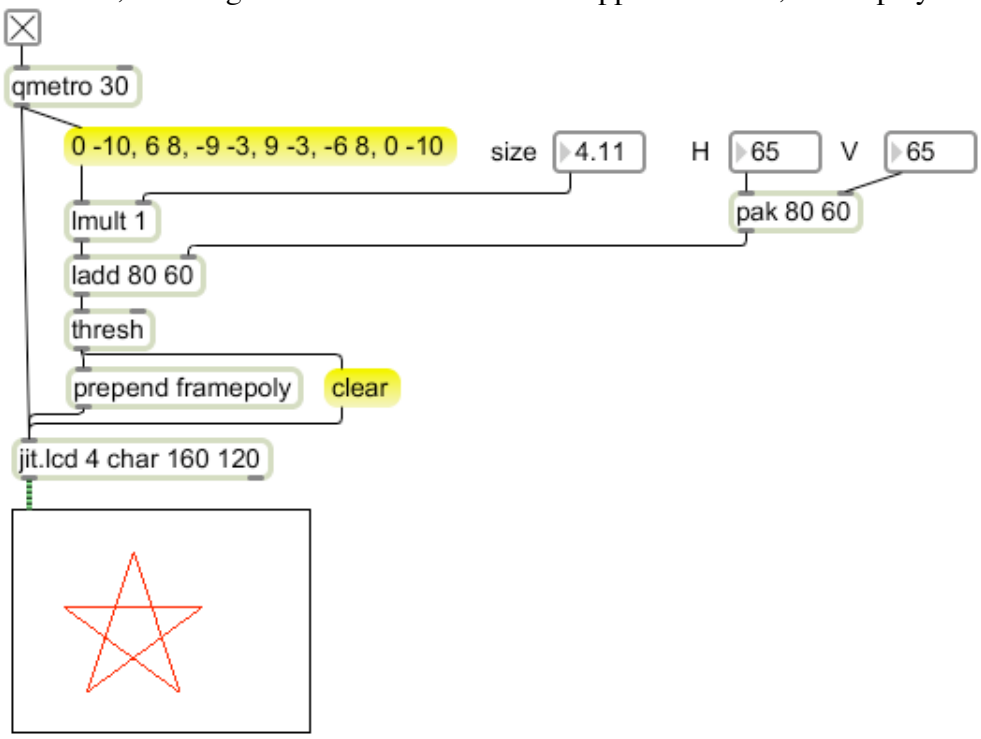


Figure 7.

Figure 7 has independent controls for position and size. Figure 8. draws two stars.

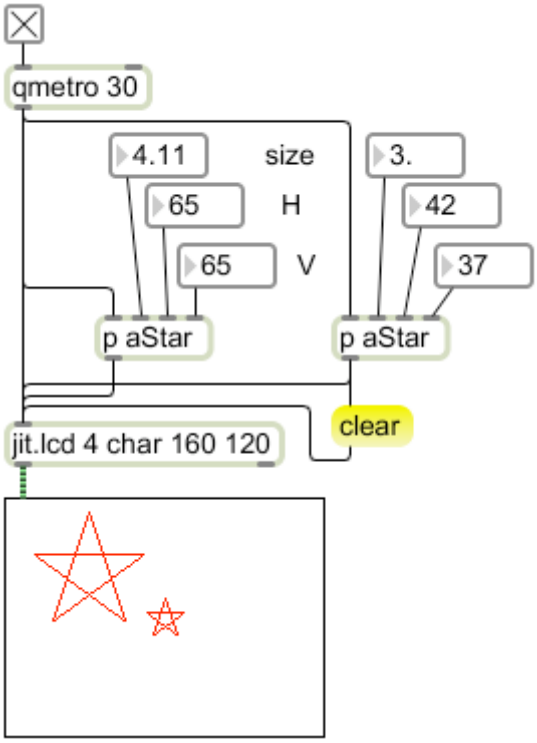


Figure 8.

Again, notice the treatment of clear. Clearing happens immediately before the first draw, which is usually the rightmost subpatch. The aStar subpatch is shown in figure 9.

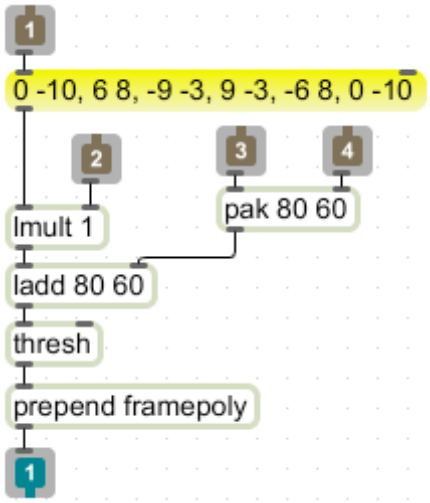


Figure 9. aStar

I can build a library of simple shapes in subpatchers and combine them any way I like. These can easily be moved around by counters and other mechanisms:

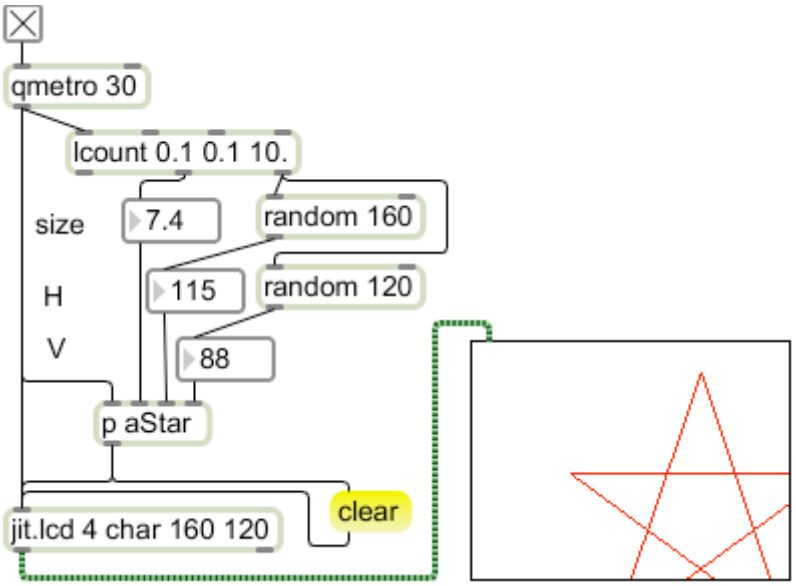


Figure 10.

In Figure 10 a star appears at a random spot and grows. Lcount is very useful for this kind of thing because the fractional counting increment gives fine control of the drawing rate. Of course you can accomplish the same thing with counter and a divide. Processing the count can have other benefits- try making this modification to the patch:

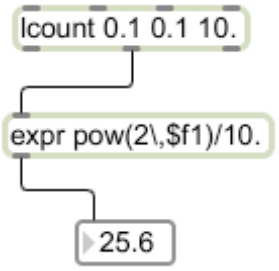


Figure 11.

The expr object in figure 11 will convert the growth to an exponential form, which is what you get as you move closer to an object. Encapsulating all of that into xStar gives me multiple objects

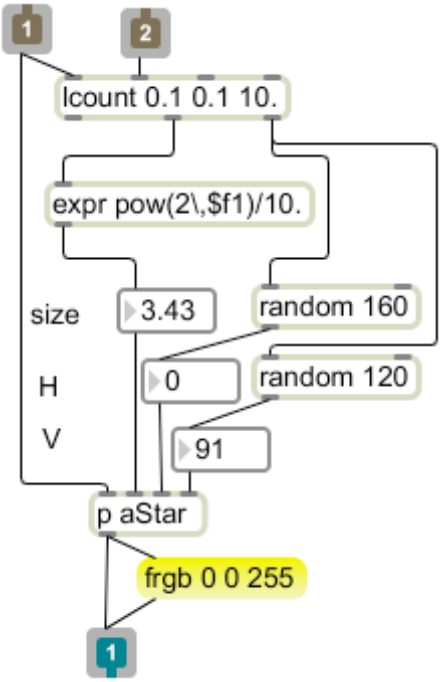


Figure 12.

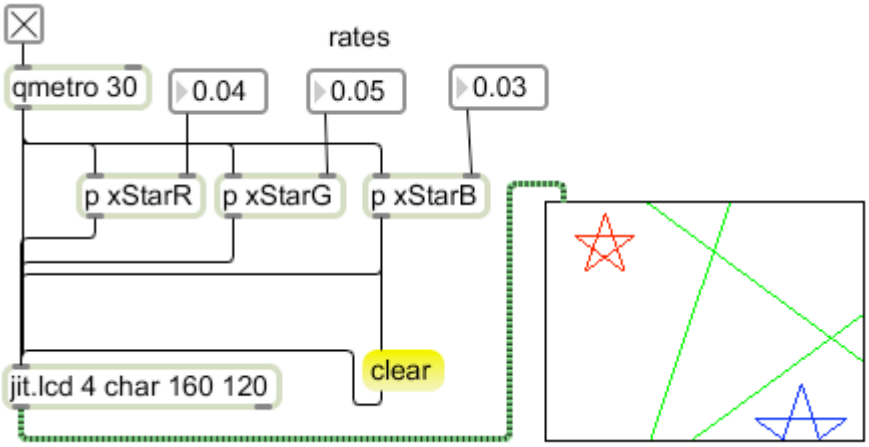


Figure 13.

Each xStar sets its own color (like the clear, the color message should be just before the drawing command) and has independent adjustment of rate.

Drawing with lines

A fair amount of drawing is done with lineto. You can only get straight lines this way, but if the data available is connected points, lineto is the best method. If the points are sufficiently close together, you will see curves. Often the drawing must begin with a moveto to prevent extra lines across the window.
Here's another way to draw the star:

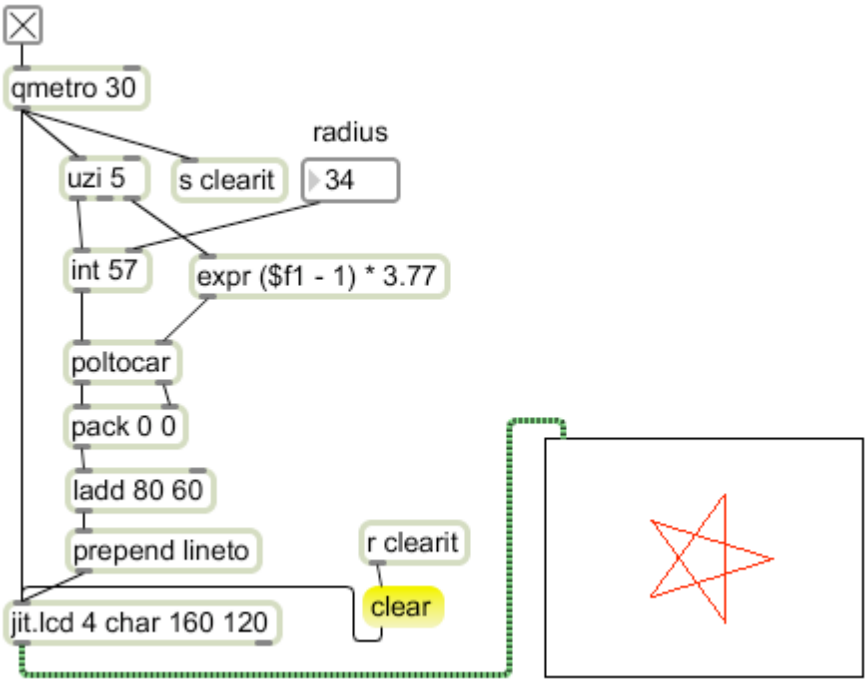


Figure 14.

Figure 14 is built around the poltocar object, which converts polar coordinates to Cartesian coordinates. For the star of figure 4, I worked out the points on graph paper. Here, the computer is figuring out the points for me. This involves a bit more computation per frame, but the ultimate reward is more flexibility in the image. For instance, I can reorient the star by adjusting the angle:

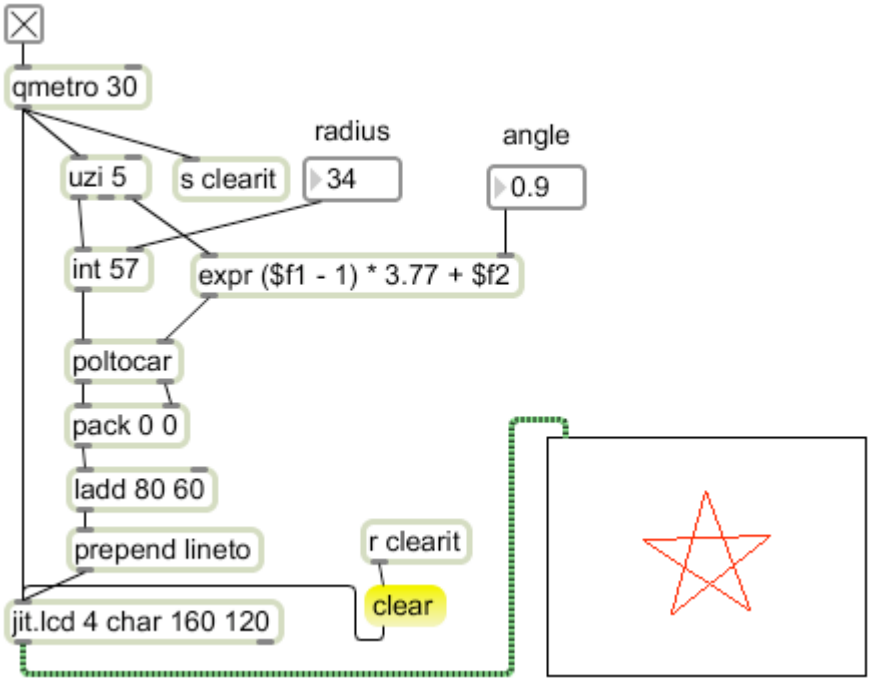


Figure 15.

The math in figure 15 may need a bit of explanation. Poltocar requires a radius in pixels and an angle in radians[4]. The angles are supplied by the expr object, multiplying the index output of uzi (parenthetically converted to 0 based count) by 3/5s of the circle. A addition of a small offset to the angle will rotate the figure. You should keep in mind that the 0 angle in poltocar is the 3:00 position. It's not hard to get more points:

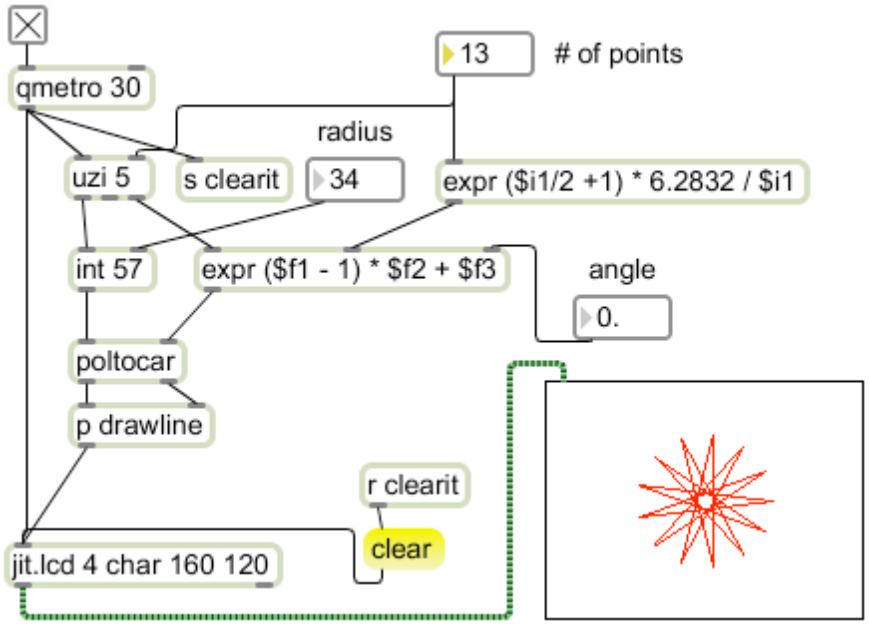


Figure 16.

[4] Remember, there are 2 pi or 6.283185307 radians in a circle.

The contents of the drawline subpatcher are simply the pack, ladd and prepend.

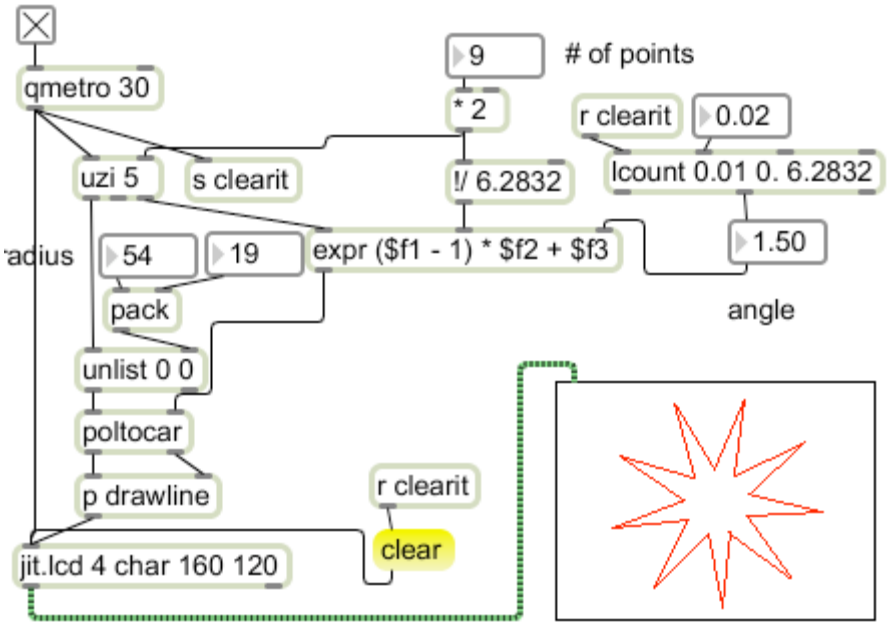


Figure 17

In figure 17, a simpler image is derived by drawing from an inner circle to the outer one. Adding a counter to change the angle will make the image rotate.

If I simplify the patch, it will just draw a circle.

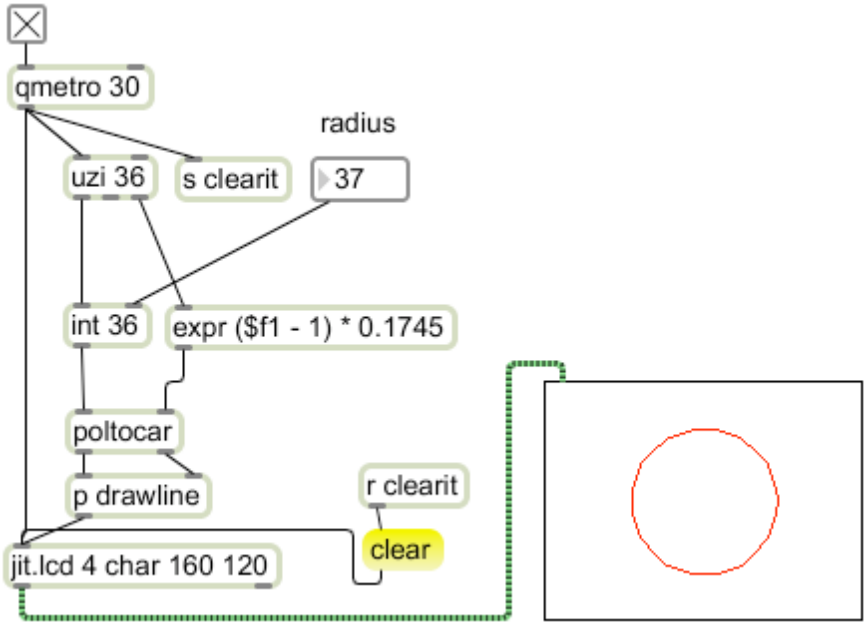


Figure 18.

The radius of the circle is the only variable. Let's replace that with a bit of math.

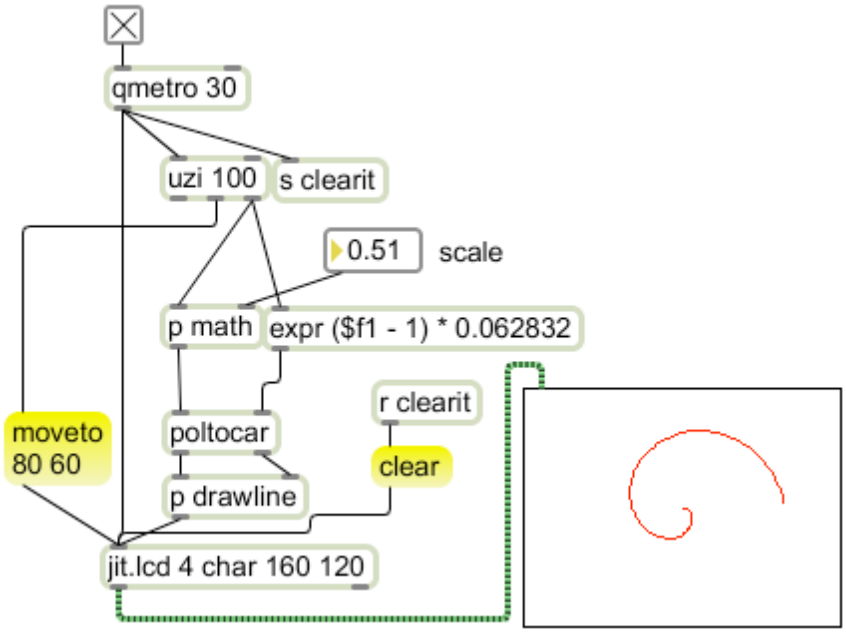


Figure 19.

If the radius changes with the angle, we get a spiral. The math here is a scaling value times the uzi count. You have to add a moveto command for any figure that does not return to its start point. Trigger this from the middle outlet of the uzi. Since there are 100 steps in this figure the index is multiplied by 1/100th of a circle. If you increase the number of bangs from the Uzi, the spiral will wrap around.

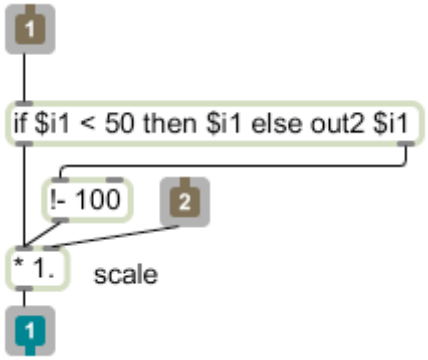


Figure 20A. Math subpatch.

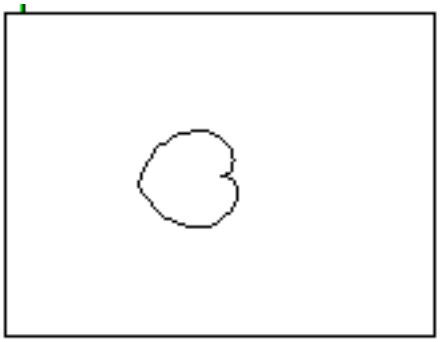


Figure 20B.

The math in figure 20A results in the shape in figure 20B. This can be rotated around the notch by adding an offset to the angle as in figure 15. Here you can see how scaling is done. Since it's in a subpatch, I use a receive object to set the value. This is easily sent from an external control or automated by a counter.

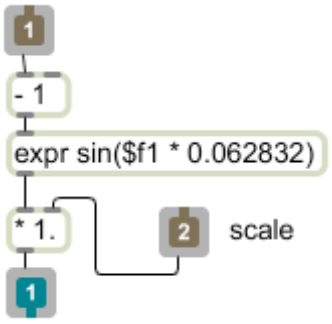


Figure 21A

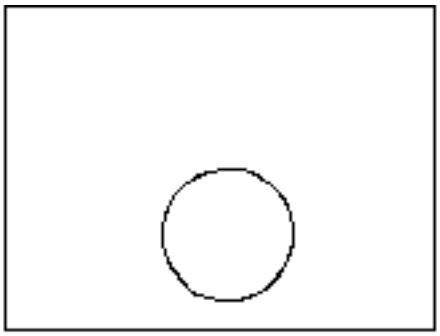


Figure 21B

The math in figure 21A produces another circle, but this one is drawn with the edge at the origin. If you change the scale (it's 50 here), the edge will remain centered in the display.

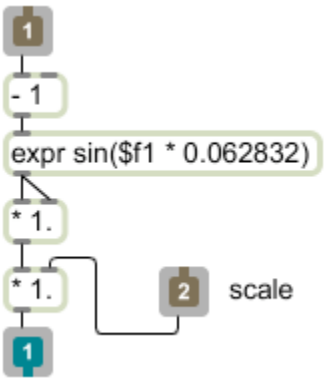


Figure 22A

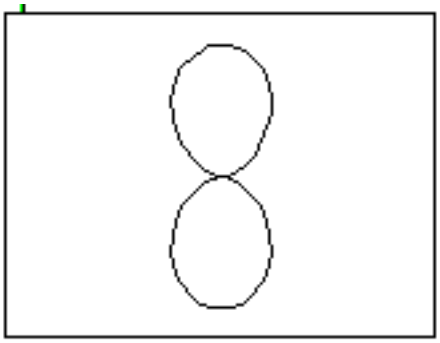


Figure 22B.

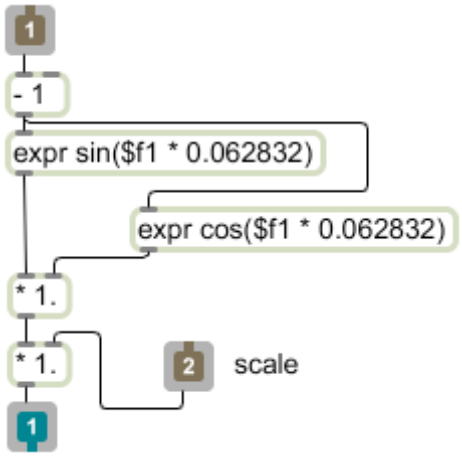


Figure 23A

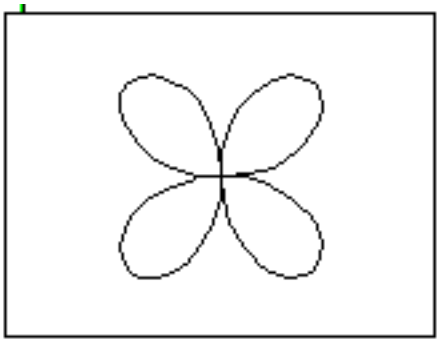


Figure 23B.

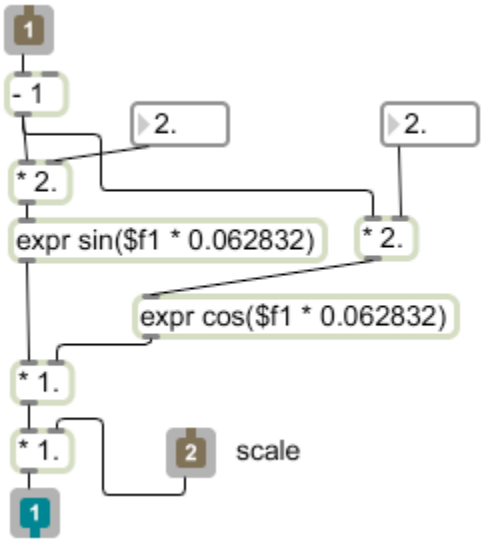


Figure 24A

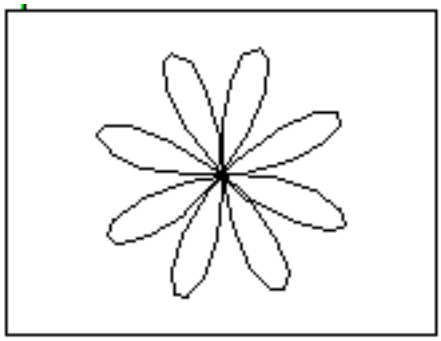


Figure 24B

Figures 22-24 show some more adventures in trigonometry. You get a particularly interesting effect as you adjust the number boxes in 24A. These are great candidates for external control and automation, giving a sinuous family of knots.

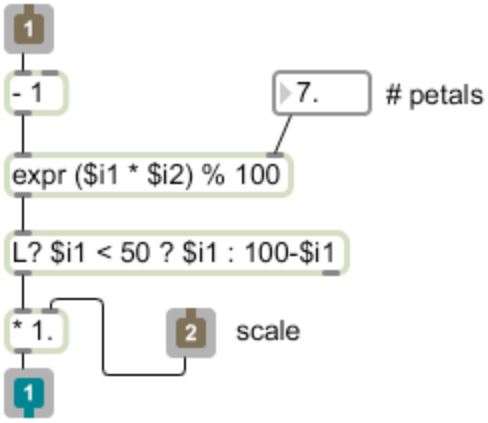


Figure 25A

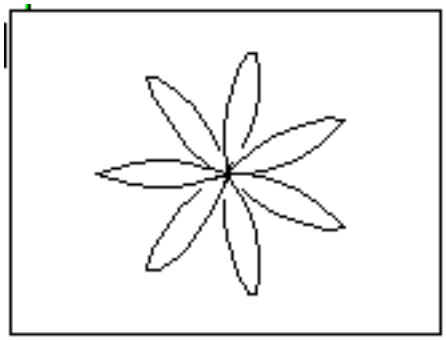


Figure 25B

Figure 25 combines two techniques. This starts as a spiral as in figure 19. The expr object breaks the count into segments with large steps. The number box sets how many segments there are. The L? object is exactly like the if statment in figure 20. This folds the counts from 0 – 100 into steps to 0-50 and back. This generates multiple sections of symmetrical spirals.

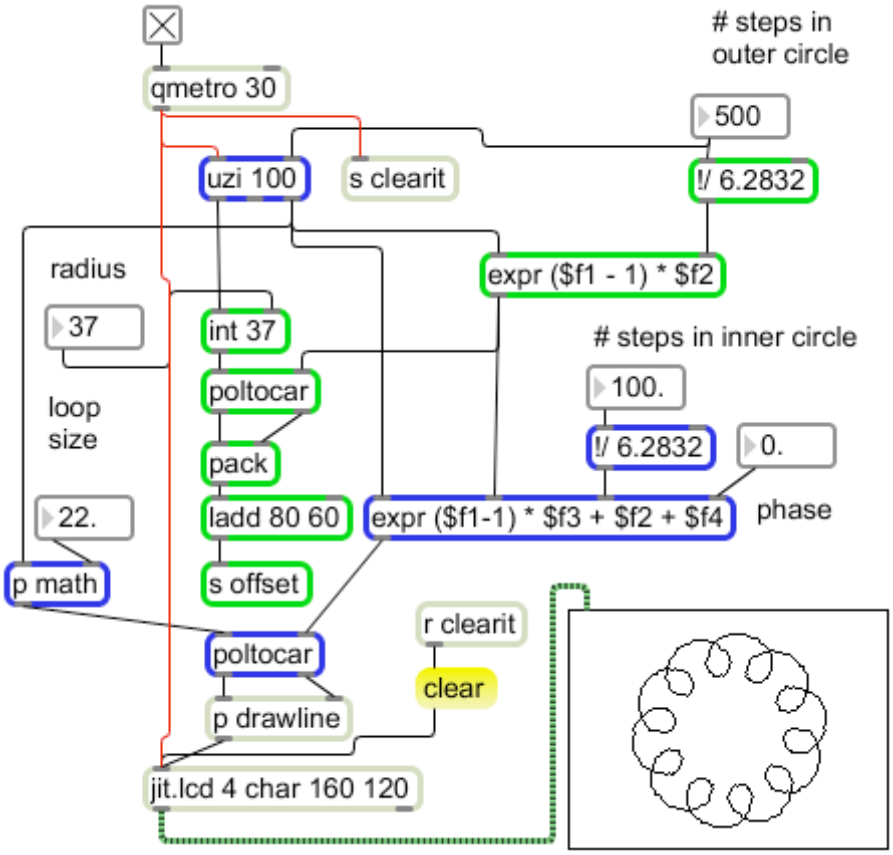


Figure 26.

Figure 26 shows a combination of several techniques. I am drawing two kinds of circle from the uzi index. One is relatively large, and since its number of steps is linked to the number of bangs from the uzi, it will always be a single complete circle. The points of this circle are supplied to the drawline subpatcher via the send offset object:

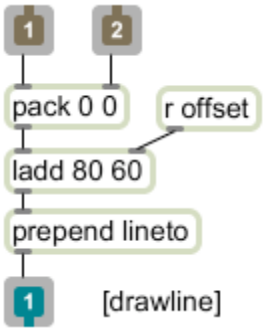


Figure 26B. Drawline

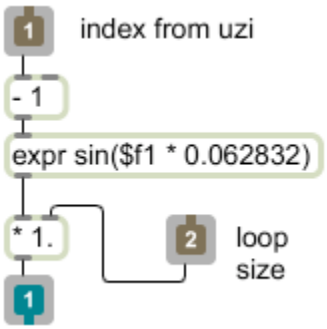


Figure 26C. Math

The inner circles are drawn using these points as origins. The math for the inner circle is that of figure 21, which draws a circle tangent to the origin. Since the origin is constantly changing, the inner circles are stretched into a spiral shape. To keep the inner circles properly oriented, the angle of the outer circle is added to the angle for the inner circles. A further tweak to the phase of the inner angle will modify the shape

The shapes resulting are determined by the ratio between the number of steps in the outer and inner circles, as well as their radii. Here's a gallery of various settings:

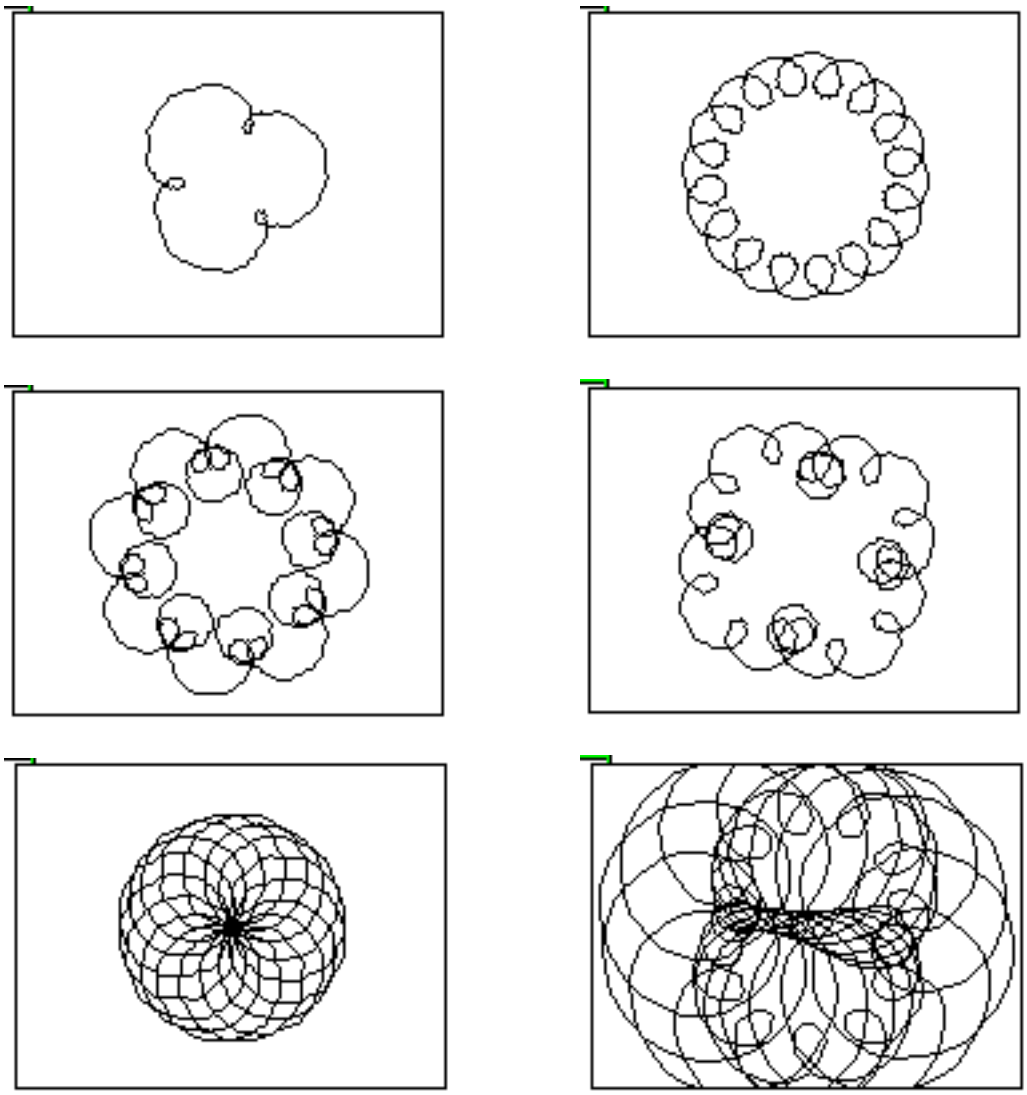


Figure 28 gallery.

Animating the controls produces continuous transformations from figure to figure. Animating phase from 0. to 6.28318 by steps of 0.1 is particularly interesting. Here are some high rez shots of extremely high ratios, which are nearly chaotic:

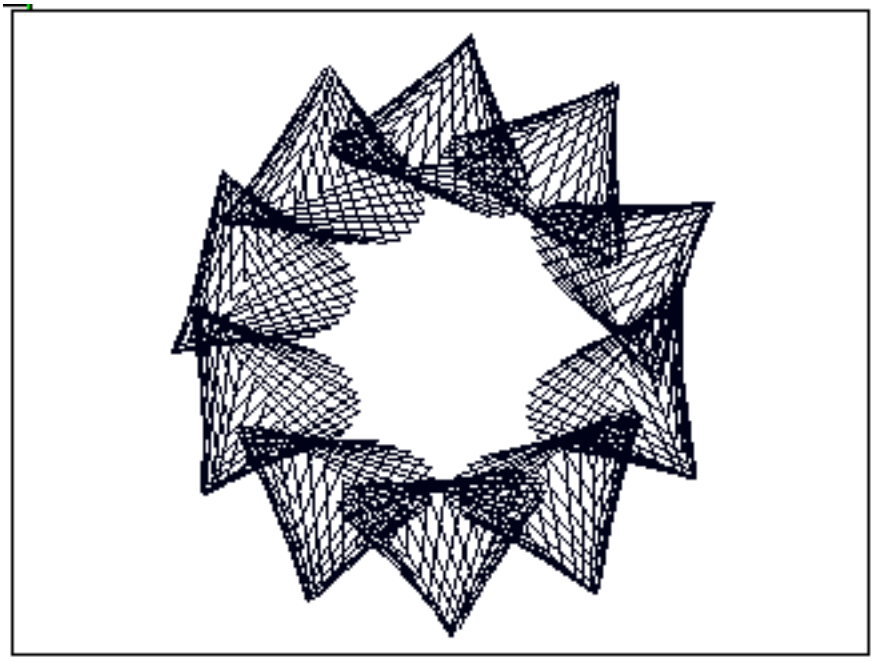


Figure 29 500:1.5

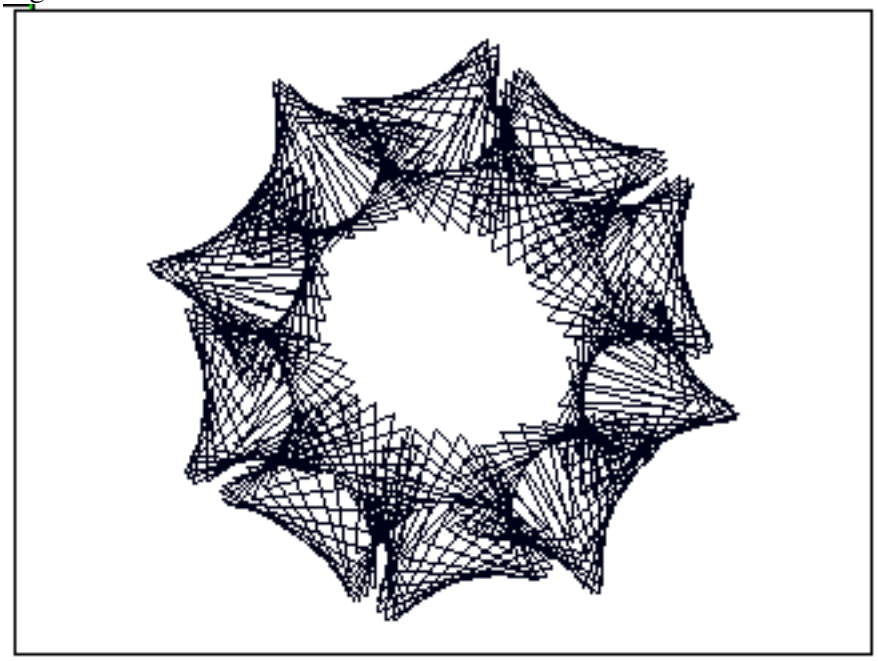


Figure 30 500: 1.52

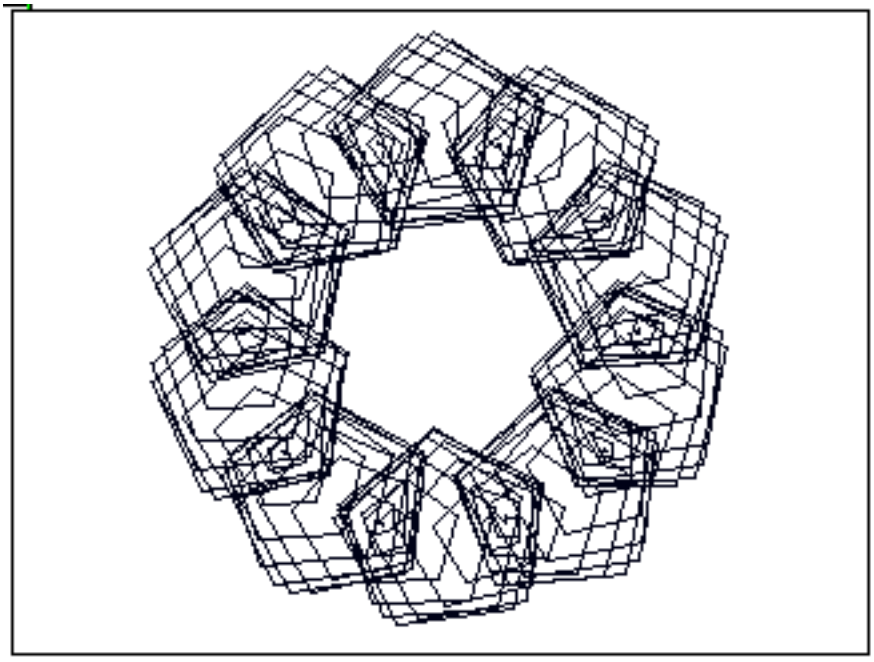


Figure 31 500: 1.25

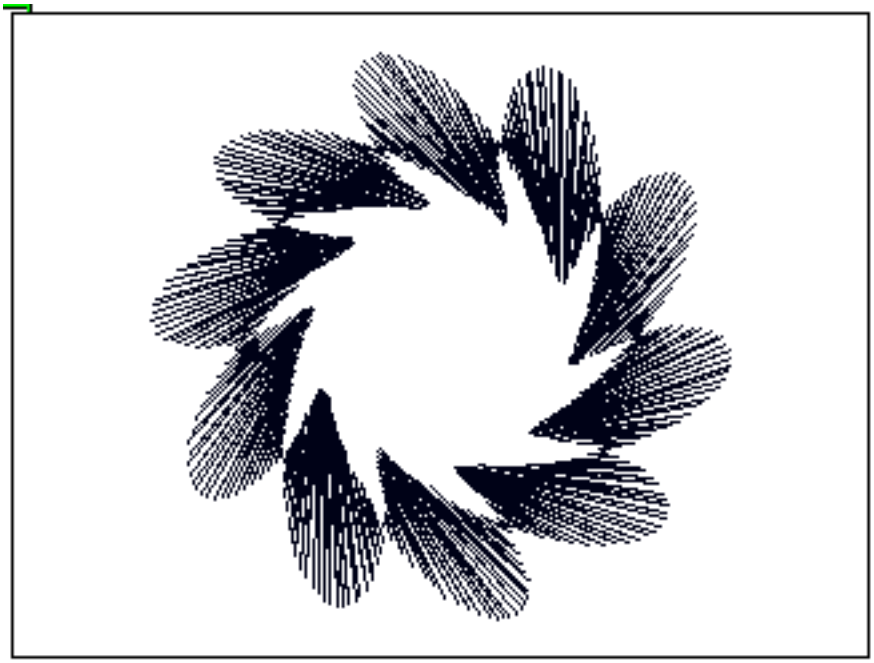


Figure 32 500: 2.0

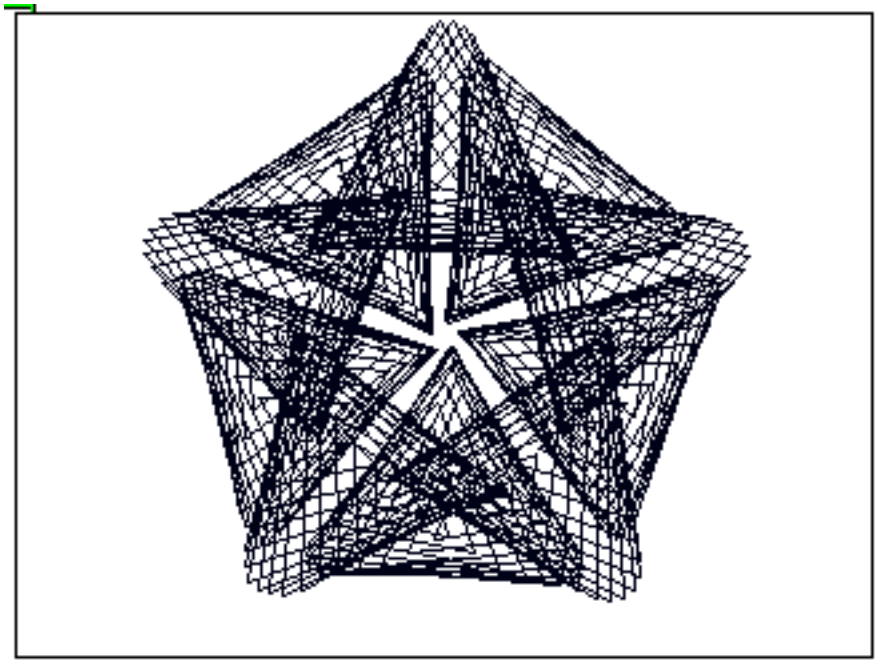


Figure 33 500: 1.5 with increased inner radius.