Fractals in Max and Jitter

Simple iterative process

Fractal geometry is the study of objects that have a property known as self-similarity – They are made up of smaller copies of the overall shape. One of the most popular is called the Sierpinski triangle:

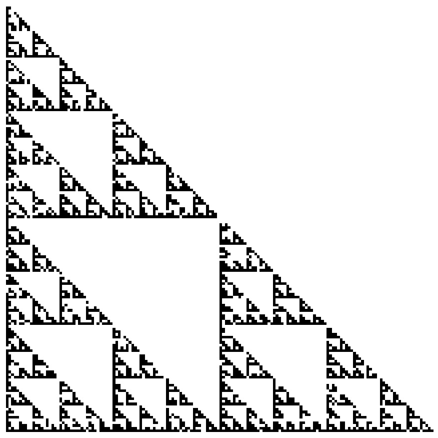


Figure 1.

It is traditionally made by starting with a solid triangle and removing smaller and smaller triangles. This is awkward to program, but there's another way to generate them that is simple and fun to watch.

The process goes like this:
1. Plot the three corners of the triangle.
2. Pick an arbitrary starting point (not one of the above)
3. Randomly choose one of the corners.
4. Plot a new point halfway between the previous point and the corner.
5. Repeat steps 3 and 4 a couple of thousand times.

What's going on? Well, we are finding points halfway between the last point and random corners of the triangle. So if you start outside the triangle, the points will be inexorably pulled into the triangle, and once inside, can't get out. If you start from a point inside the triangle, you will never land in the central hole, because all points in there would have to be derived from a point outside the triangle. If you look at the process backwards for a moment, you'll see what I mean. If there can be no points in the central hole, well, anywhere halfway between the hole and the corners is excluded as well.

With other divisions of the line, you get related but different pictures, like Figure 2

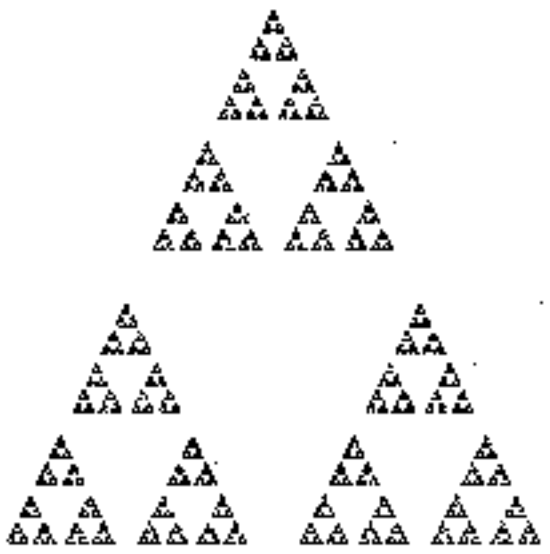


Figure 2.
The triangle begins to break apart because the exclusion zone is larger. These figures can also be drawn with more corners as shown in figure 3.

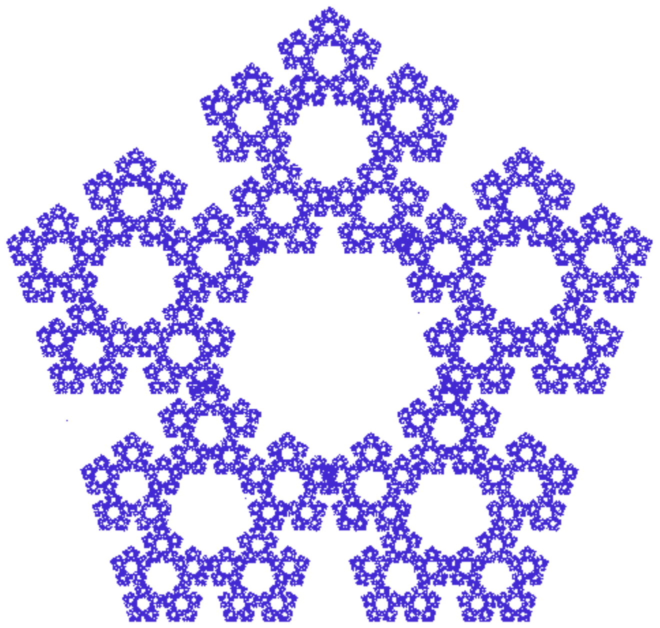


Figure 3.

What is the area of something like this? Well, in the ideal version, unlimited by printing resolution, every pentagon is missing a pentagon of 1/6th its area, in an infinitely diminishing series. That means its area approaches zero. It's really a very complicated line. What is the length of the line? Following all infinitely small twists and turns, the length approaches infinity. Since it has no area, it is not a two dimensional object, but it has length, so it is more than one dimension. Its dimension is some fraction between one and two, which is why it is called a fractal.

Generating the triangle

We can generate a Sierpinski triangle in the jit.lcd object. For every point we plot, we'll call paintrect to draw a square one pixel on a side. Figure 4 is a basic patch that will draw any manner of forms, depending on what is in the subpatch just below the metro. Points produced by the subpatch will be defined as a list of X,Y,X,Y,R,G,B. The X and Y coordinates are repeated so the outer patch can set the origin and size of the squares.

The subpatch iwill use X =0 and Y = 0 as the center of the image. The Ladd object will center the image in the display and set the rectangle size to 1. The paintrect command does the drawing, and the qmetro sends matrices out for display.

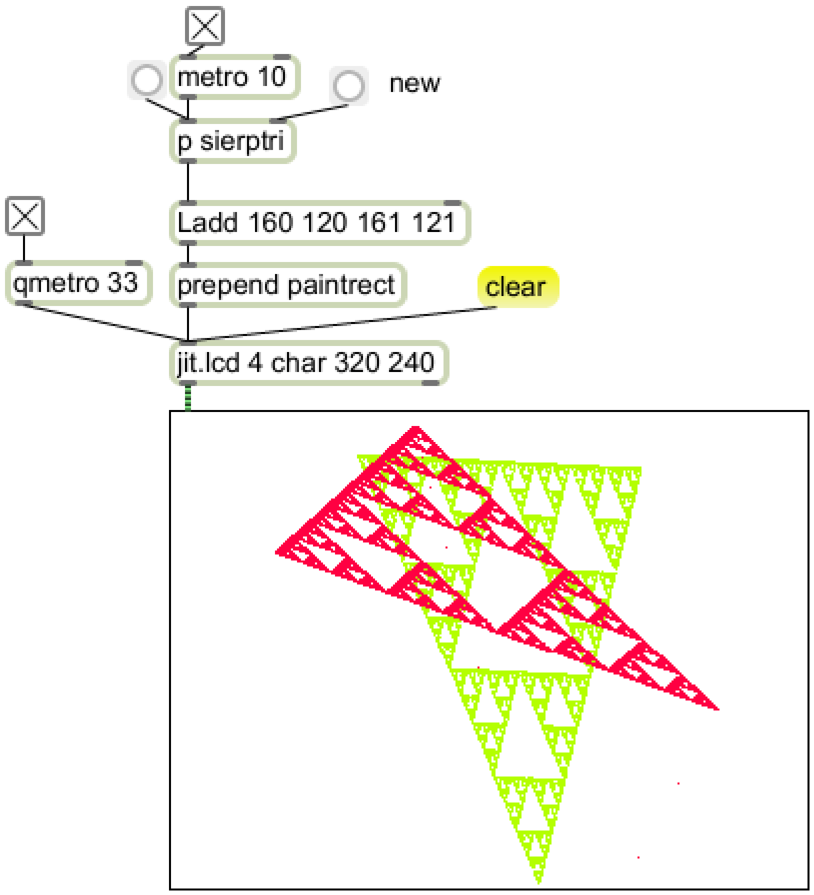


Figure 4.
The fractal generation is in the sierptri subpatch shown in figure 5. All points are managed as lists of the X and Y coordinates. The corners of the triangle are created randomly when the patch is loaded or the new button clicked. When the left inlet is banged, the current point is loaded into the halfwaythere subpatcher. Then one of the corners is sent to the left inlet for the calculation. The calculation appears in figure 7.

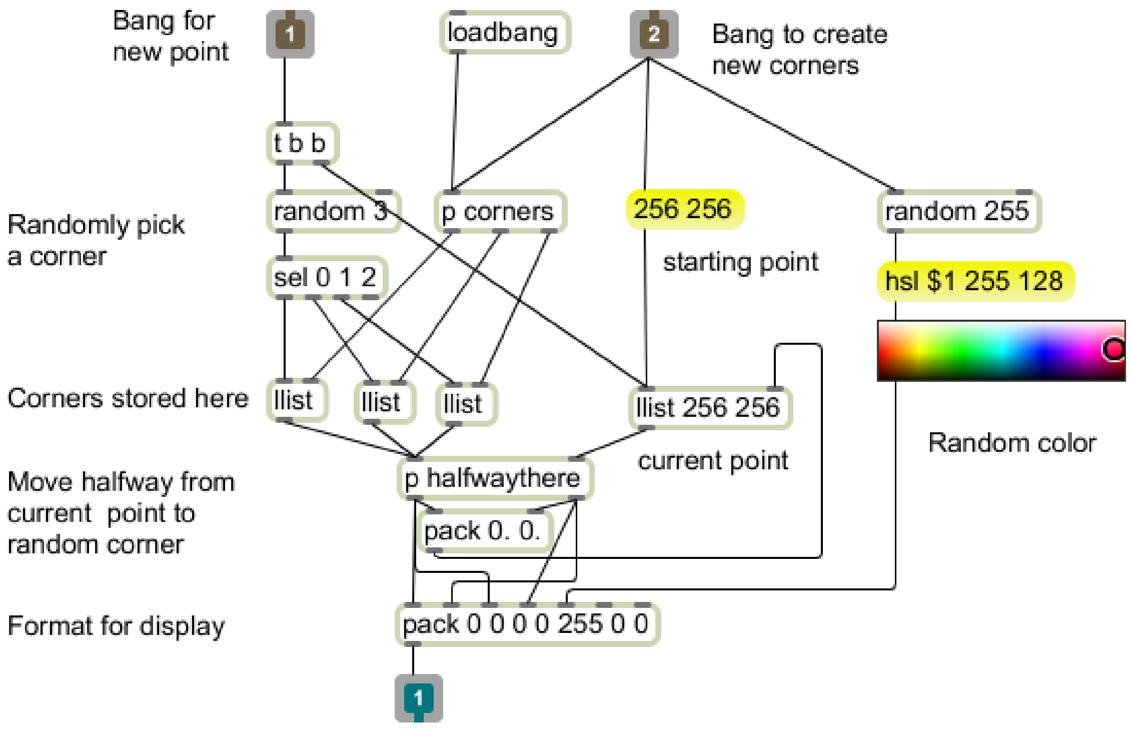


Figure 5. The sierptri subpatch.

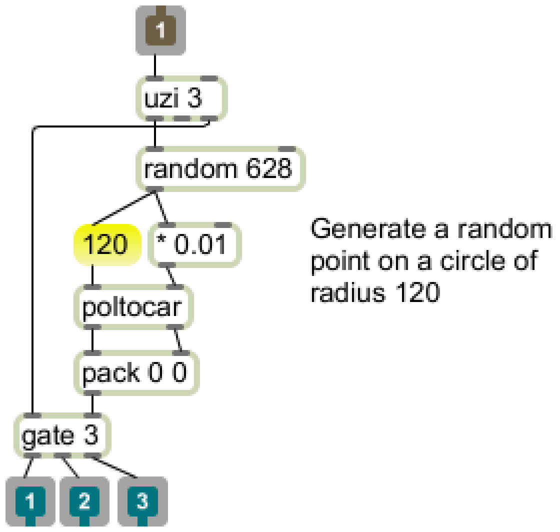


Figure 6. The corners subpatch

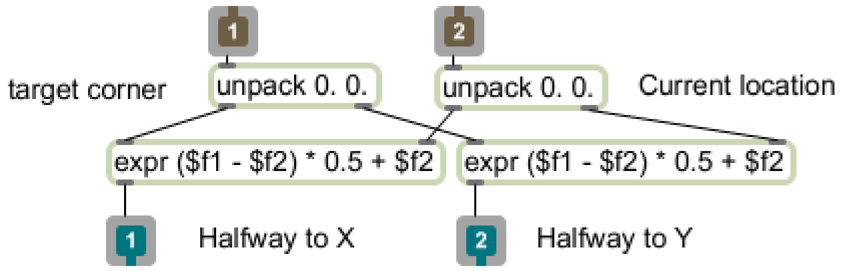


Figure 7. The halfwaythere subpatch.

To find a point P in the middle of a line from A to B you simply move half of the difference in X and half of the difference of Y.

This patch can be generalized to make symmetrical figures of n corners. The corners will be specified in polar coordinates evenly distributed around a circle of fixed radius. Figure 8 shows the inner workings. (The outer patch is much the same as figure 4.)

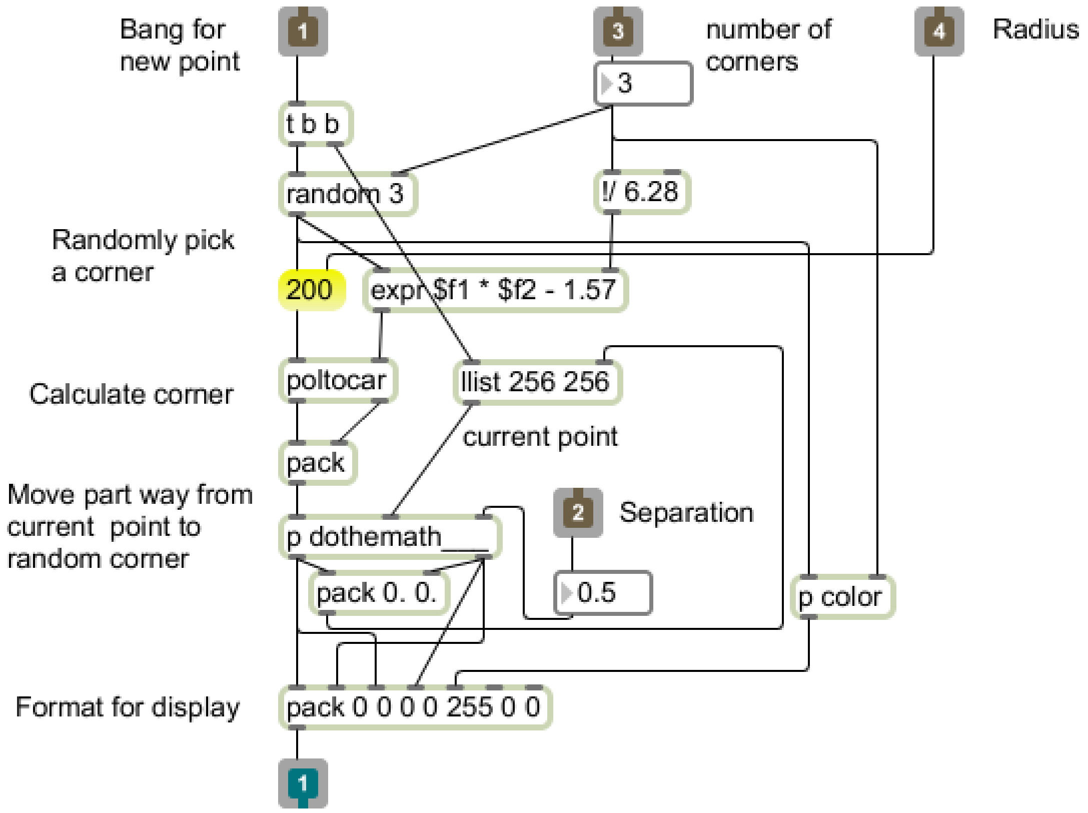


Figure 8.
When a bang arrives, the random number is multiplied by an angle equal to 2π divided by the number of corners. (I subtract 1.57 from this to place the 0 angle at the top.) This is dropped into a subpatch with one difference from figure 6. The multiplier is adjustable.

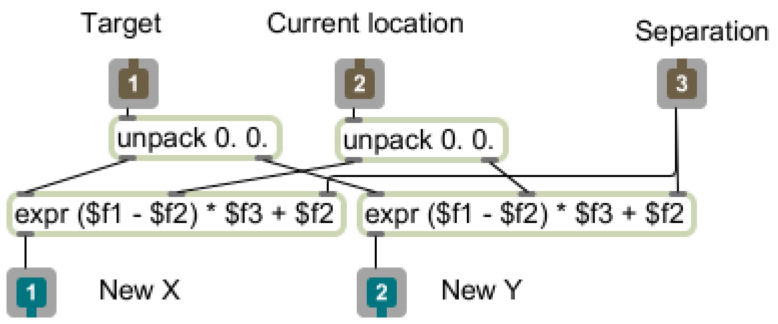
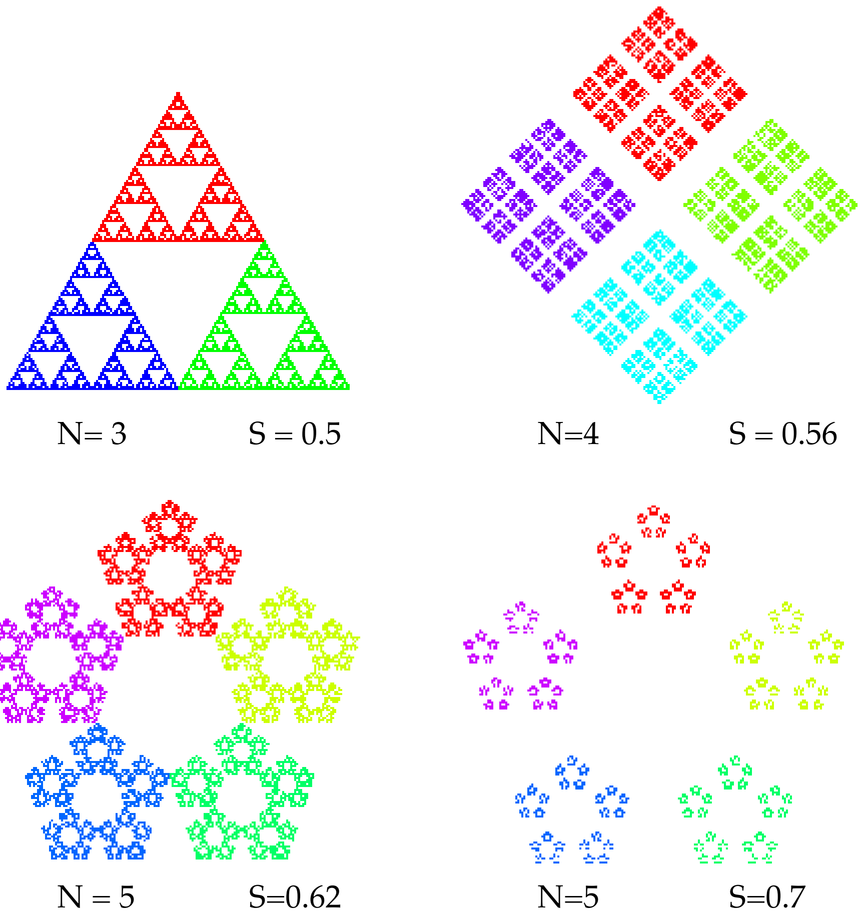
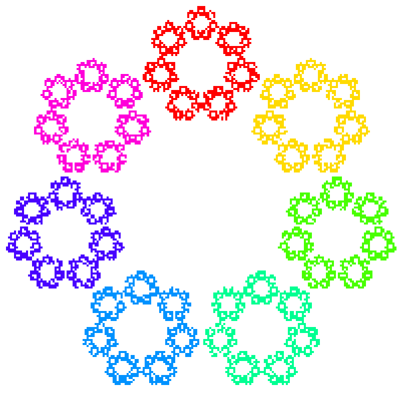
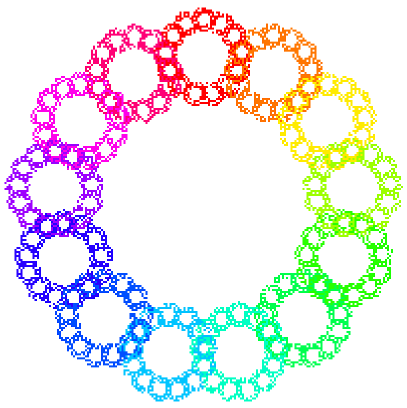


Figure 9.

Figure 10 shows some shapes derived from this patch.

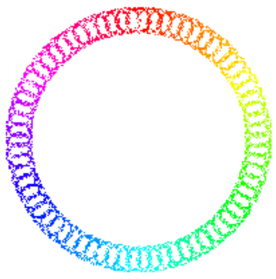
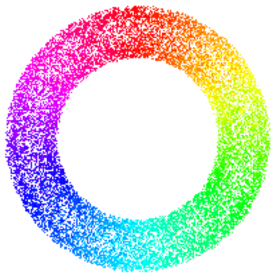
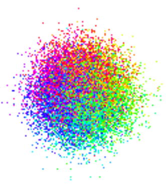


N= 6 S = 0.7 N= 13 S= 0.75

Figure 10. Symmetrical fractals.

You can see that as the number of corners increases, the shape quickly becomes circular. It is also necessary to increase the separation coefficient to keep the shape clear. If the coefficient is low, the result is a chaotic doughnut or splotch as shown in figure 11.

N=70 S= 0.9 N=70 S=0.5 N=70 S=0.1

Figure 11.

The Koch Snowflake

The Koch curve is another popular fractal shape. It is made by taking a line, dividing that line into thirds, then adding another segment to make a peak in the shape. Once this is done, repeat for each segment. Figure 12 illustrates.

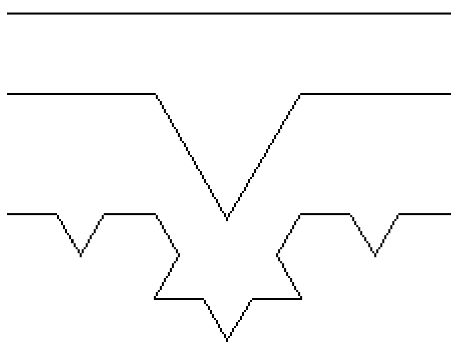


Figure 12. The Koch curve.

If you start with an equilateral triangle, you get the Koch Snowflake:

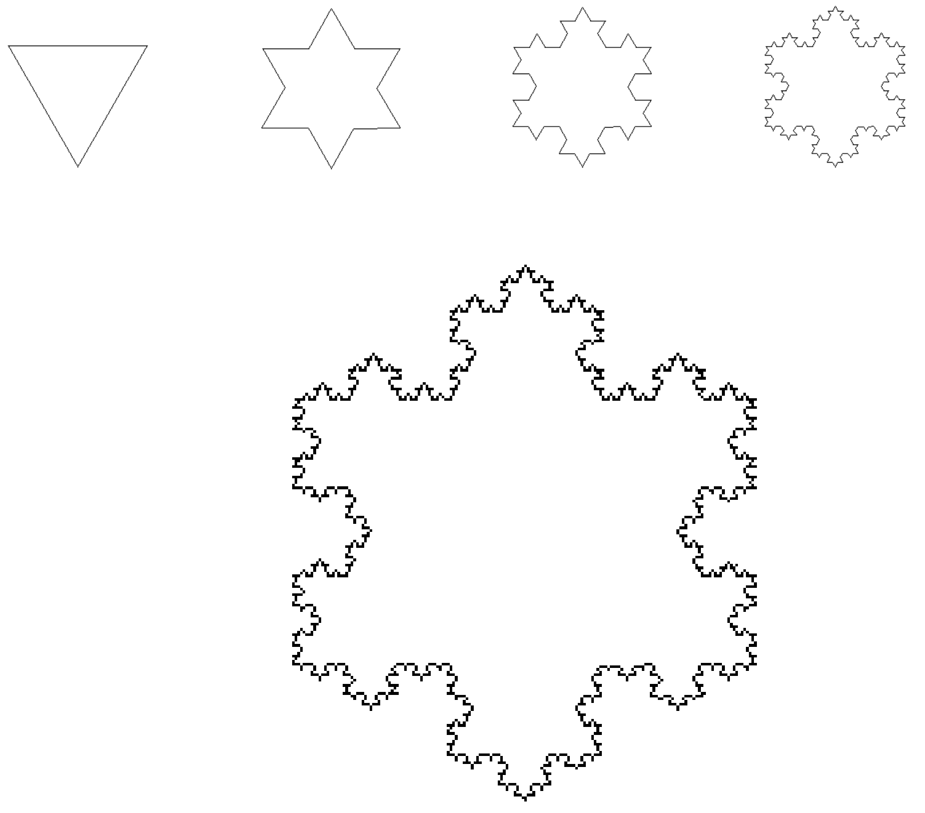


Figure 13. Five iterations of the Koch Snowflake

After five iterations, the shape begins to challenge the resolution of the window. Figure 14 shows a patch that creates Koch Snowflakes.

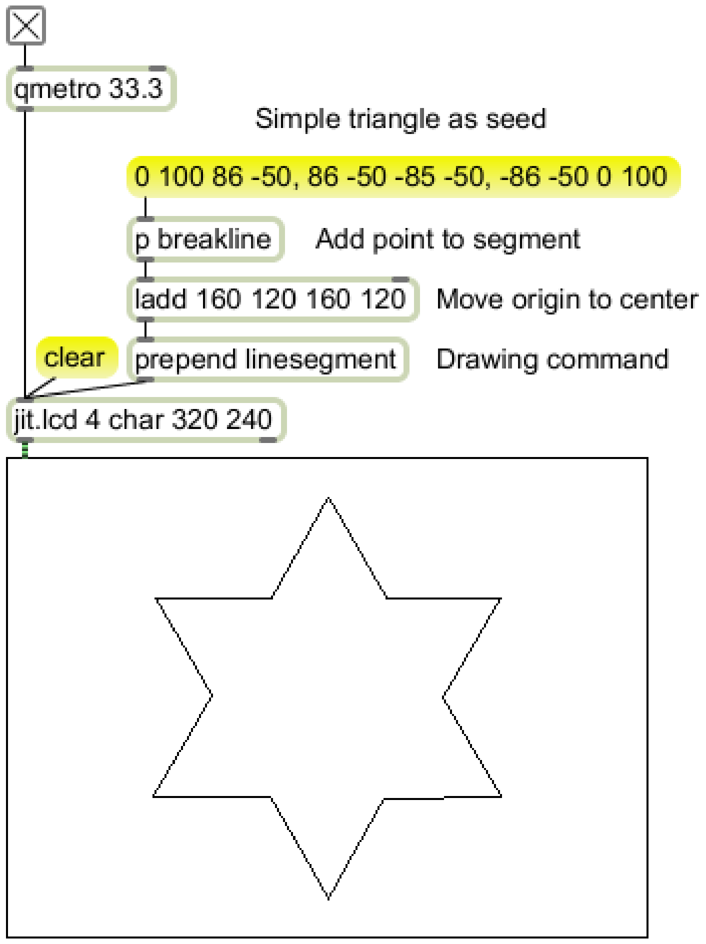


Figure 14. Koch patch

This is a basic lcd drawing patch that uses linesegment as the drawing command. Linesegment requires four arguments to represent the beginning and end points. The seed list contains three sets of arguments separated by commas. This produces three messages to start things out. The action happens in the breakline subpatch which produces four segments for each segment fed in.

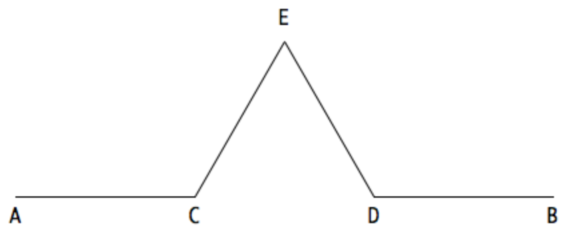


Figure 15. The line breaking challenge.

Figure 15 illustrates the problem. The input is a segment AB. Point C is one third the distance from A to B, and point D is two thirds the distance. The math for finding these two points is simple vector algebra:

Cx = 0.33*(Bx - Ax) + Ax;
Cy = 0.33*(By - Ay) + Ay;

Dx = 0.66*(Bx - Ax) + Ax;
Dy = 0.66*(By - Ay) + Ay;

Subtracting Ax from Bx is an example of translation. We momentarily shift the origin of our Cartesian grid to be at A. Then the math works properly no matter which way the line actually points. (We must remember to add A back in when the math is done.) Figure 16 shows how the breakline subpatch does this.

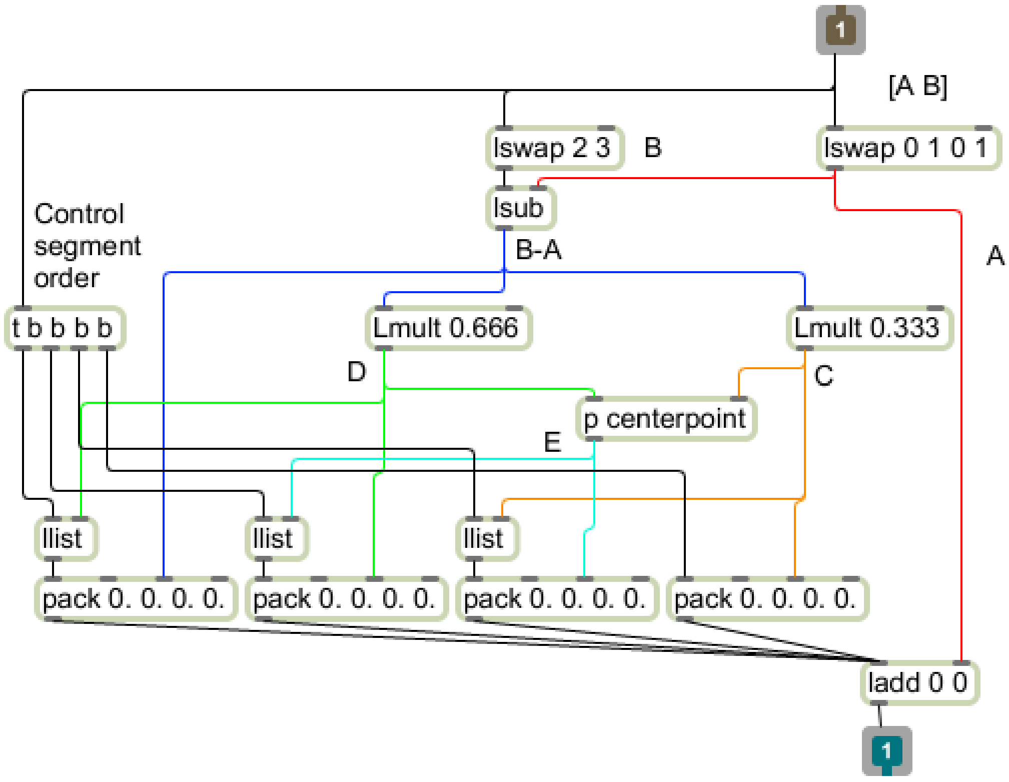


Figure 16. Inside breakline.

The only thing tricky about this patch is ensuring that the segments are output in the correct order[1]. The lline objects capture each result and hold it until the trigger object lets them go. With an input of AB, the required output is AC, CE, ED and DB.

The Lobjects allow us to process the X and Y coordinates of each point in a single object, which keeps the patch cord count down. (I've colored them to clarify the action.) The first step is to load the point A (red) into the Ladd object at the outlet-- this will undo the translation. Note that the coordinates of A will be added to both points of each segment. Next, [Lswap 2 3] and the following Lsub produce the A-B term that is in all of the calculations. This is distributed via the blue lines. An Lmult creates point C (orange) which is packed into the first

[1] That doesn't actually matter here, but it will when we use this subpatch in openGL.

segment. Since A is added to both points in the segment, the first two values in the pack must be 0. Point C will also be used in the second segment. Point D (green) is constructed and distributed to the third and fourth segments in a similar manner. B-A will complete the fourth segment. The only thing unaccounted for is point E. This has its own subpatch, shown in figure 17.

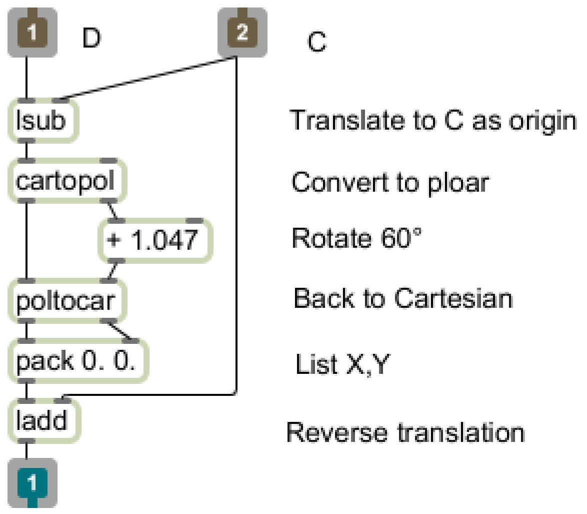


Figure 17. Constructing point E.

The centerpoint subpatch does another translation of origin, this time to the coordinates of C. Then the coordinates of D relative to C are converted to polar form, rotated 60°, converted back to Cartesian, and translated back to the original coordinate system[2]. Note that the rotation is done by an addition. We will play with this a little later.

Back in figure 15, the output of centerpoint is point E (teal), needed for the second and third segments. Once this is in place, the trigger object sends everything out to the master patch. Figure 14 shows the output from a single pass through the breakline subpatch.

The shapes of figure 13 are created by adding more copies of breakline. It would be possible to build a system to run the results of breakline through the same object several times, that would be more complex that just copying the object. This system only needs five or six iterations at the most. If you like, you can add a gate to control the number of times the process is applied. This is illustrated in figure 18.

[2] Which you will remember is relative to A. I know this seems convoluted, but it is far simpler than any other approach.

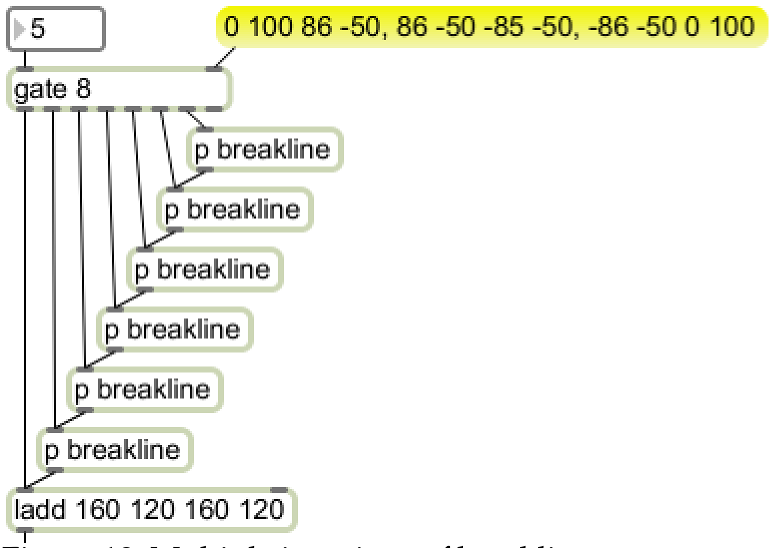


Figure 18. Multiple iterations of breakline.

Variations on the snowflake can be created by changing the rotation performed in finding point E on each segment. If the angle addition is changed to a subtraction, the points will break into the body instead of out. This creates a completely different set of images.

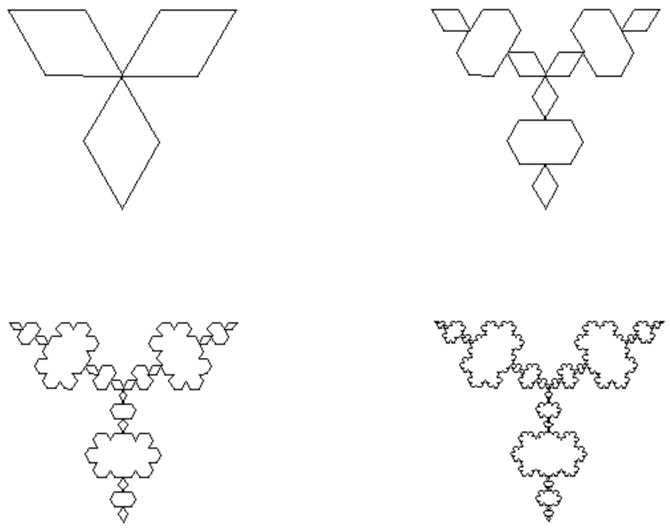


Figure 19.

Alternating direction produces even more variations:

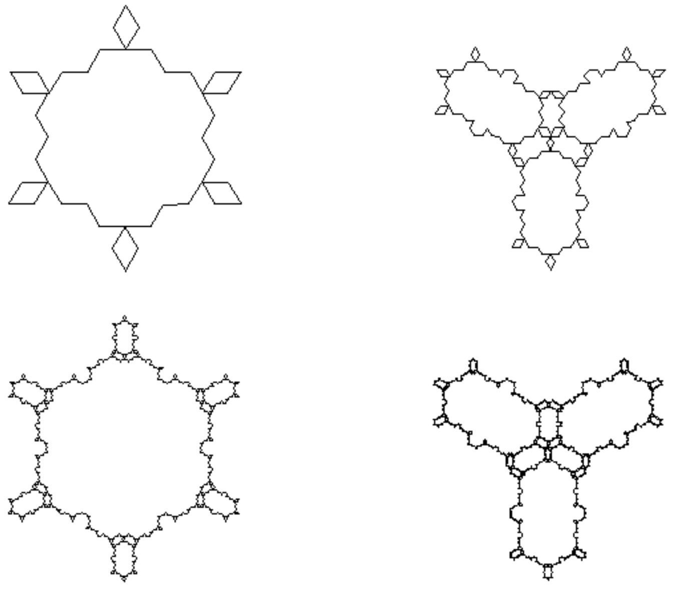


Figure 20.

As will other combinations and overlays without clearing:

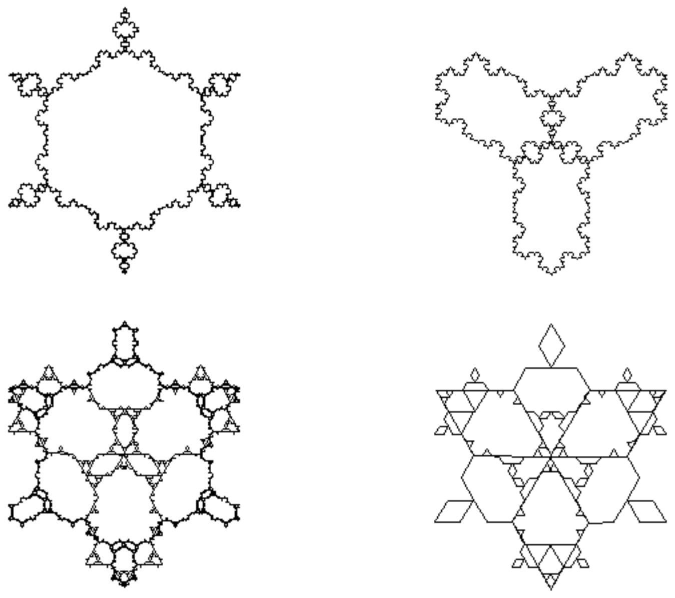


Figure 21.

Sierpenski Carpet

Sierpenski's carpet is similar to his triangle.

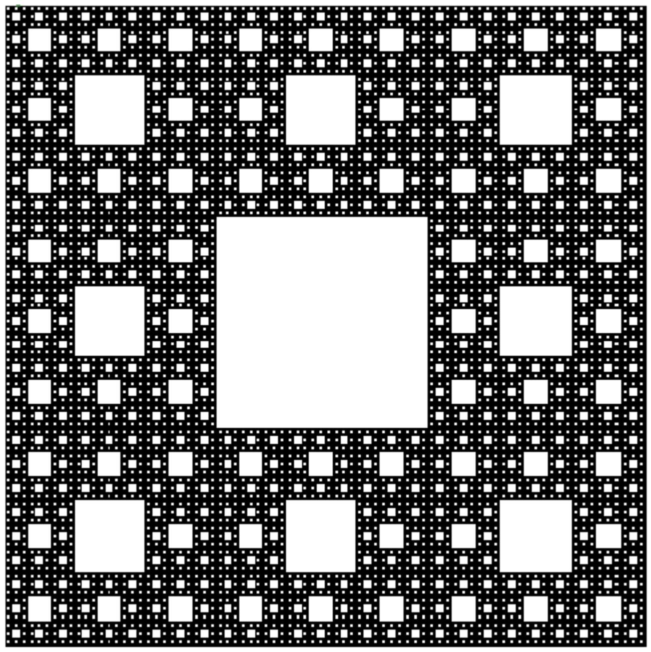


Figure 22.

The result is similar, but the algorithm is quite different.

- Start with a black square.
- Punch a 1/3rd size square hole in the middle.
- Add holes surrounding the big hole, again reduced by 1/3rd.
- Surround those holes with...

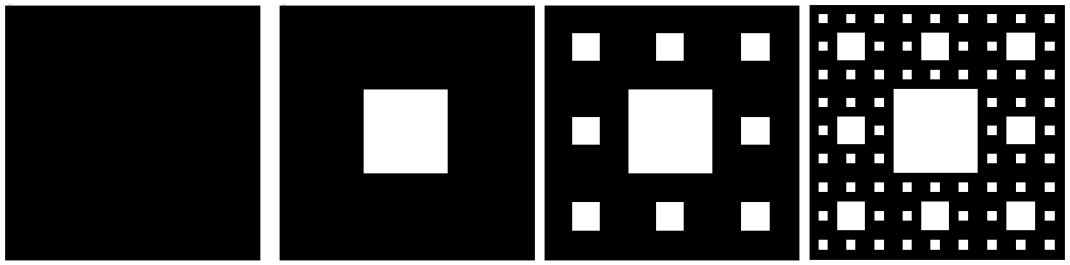


Figure 23.
As with the triangle, the area is nearly 0 and the length of the edge is nearly infinite. Figure 24 shows how this is done with a Max patch:

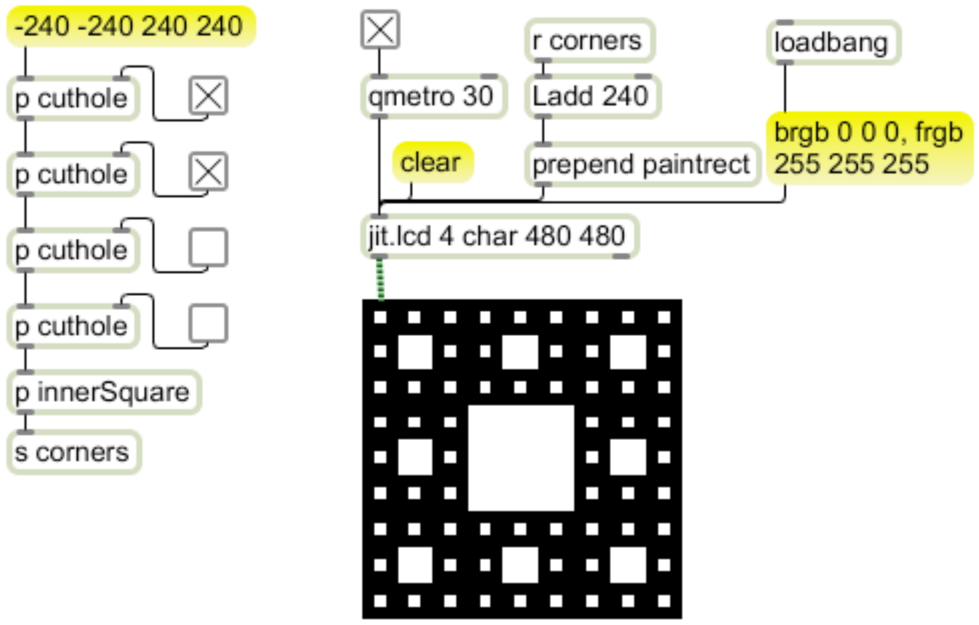


Figure 24.

The right side of this patch shows a basic drawing mechanism—if a list of left, top, right and bottom coordinates is received in the **r(eceive) corners** object a white rectangle will be painted with the coordinates relative to the center of the window. The actual calculation of the coordinates is done in the various cuthole subpatches figure 25. (Note the list of coordinates at the top of figure 24 —this starts the process with the size of the window.)

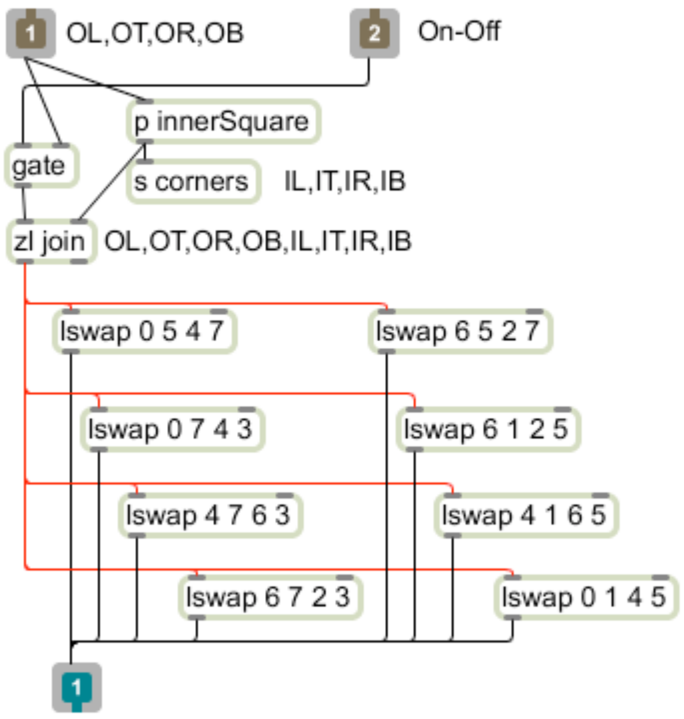


Figure 25.

The first thing that happens in the cuthole subpatch is in another subpatch, innerSquare (figure 26.). Innersquare is fed the list of coordinates that represent the corners of the window, or area available for drawing. The expr objects calculate a new set of coordinates set in by 1/3rd from each corner of the area input. Note that some of the expr object have four inlets, but only use three of them. The results of innerSquare are sent to the receive corners object to paint the inner square. The results are also joined to the original input to create a master list of outer_Left , outer_Top, outer_Right, outer_Bottom, inner_Left, inner_Top, inner_Right, and inner_Bottom. These are fed to a set of Lswap objects. Lswap reorders lists according to the arguments. The first with arguments 0 5 4 7 defines a square consisting of outer_Left, inner_Top, inner_Left and inner_Bottom. That defines a region above and to the left of the inner square. There will be eight such regions output from this subpatch.

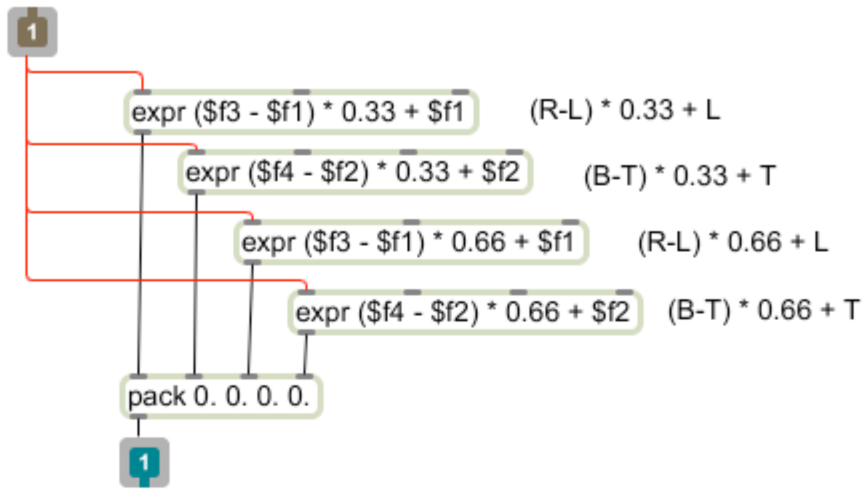

1
expr ($f3 - $f1) * 0.33 + $f1
(R-L) * 0.33 + L
expr ($f4 - $f2) * 0.33 + $f2
(B-T) * 0.33 + T
expr ($f3 - $f1) * 0.66 + $f1
(R-L) * 0.66 + L
expr ($f4 - $f2) * 0.66 + $f2
(B-T) * 0.66 + T
pack 0. 0. 0. 0.
1

Figure 26.

What happens to these new regions? They are applied to an identical cuthole subpatch which paints a white square for each. If the exprs are enabled (note the gate object connected to an external toggle), a further 8 lists are created for each list in. That is 64 lists at this point. The process can continue until the rectangles are only one pixel across. After all that is needed is the innerSquare subpatch to paint the 4096 tiniest squares.

Since this system just generates corner points, it can easily be adapted to draw anything in the holes. For instance, figure 27 uses picts. The modifications required to paint a pict called drawme are shown in figure 28.

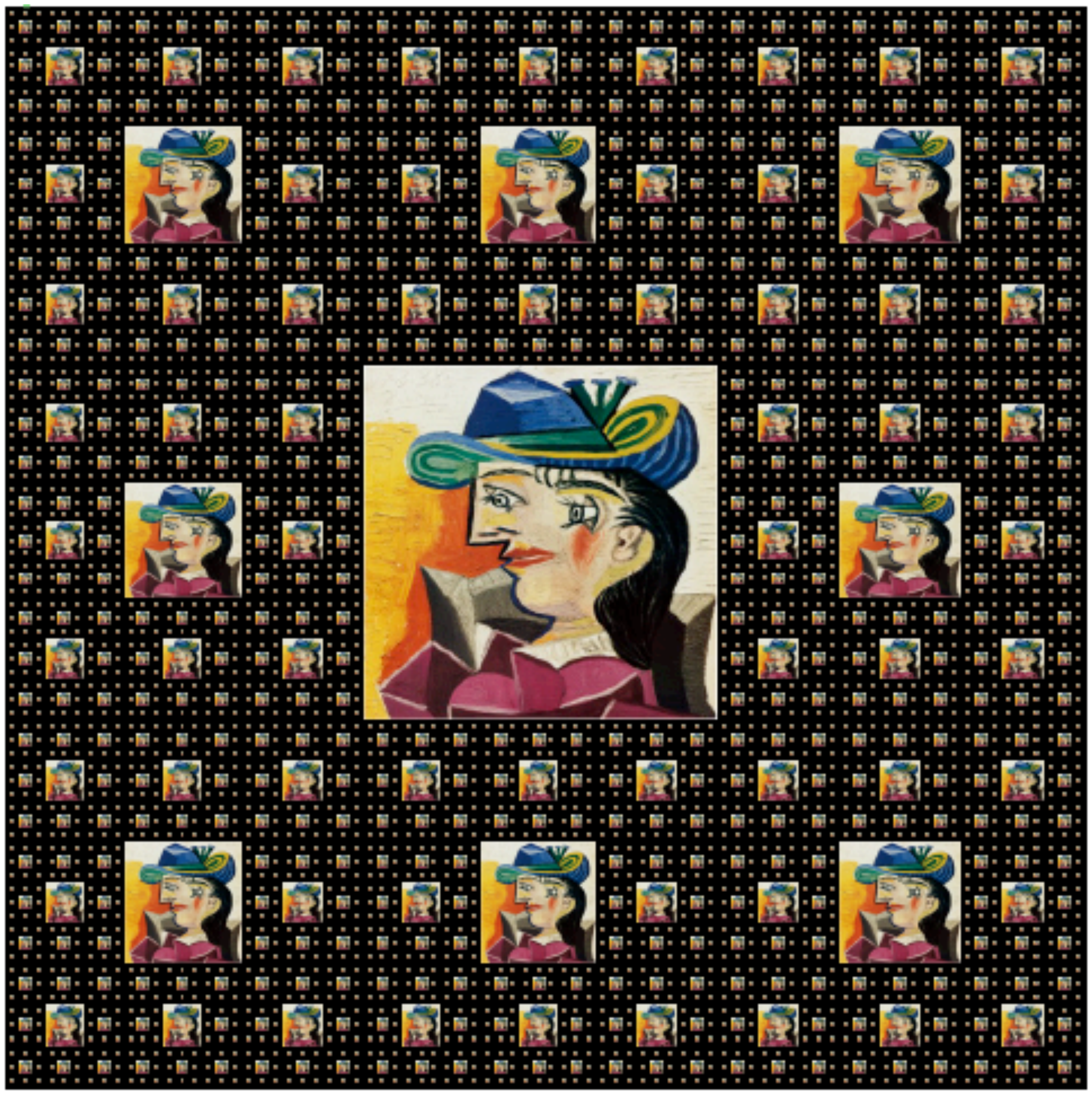
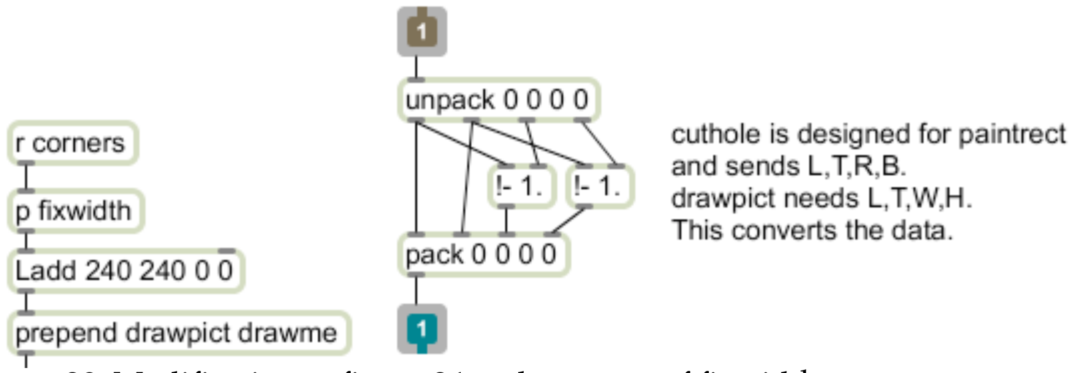


Figure 27.



cuthole is designed for paintrect
and sends L,T,R,B.
drawpict needs L,T,W,H.
This converts the data.

Figure 28. Modification to figure 24 and contents of fixwidth.