Visualization of Audio

Visualization is a type of video synthesis that generates images from audio input. The effects of this can be tedious or sublime.

In a typical approach, audio information is mapped to visual attributes of simple shapes. This is possible with a few simple mechanisms that can be applied to a wide range of images. It is possible to derive the following types of information from an audio signal:

- Amplitude. This is related to the loudness of the sound. To match perception of the listener, amplitude must be averaged over a brief period. Slightly differing averaging times can produce quite different effects.
- Frequency. This is related to the pitch of the music. Accurate pitch extraction is only possible with simple sounds, but some degree of error is tolerable in most pitch to image mappings.
- Waveform. This is one way of representing timbre. Waveforms may be drawn directly on the screen, but aside from a general association between the size of the waveform loops and loudness, it is difficult to link sound quality with a particular waveform.
- Spectrum. Another representation of timbre, spectrum produces a set of values that can control many visual parameters or objects. Spectrum displays can convey timbre with some accuracy (after practice), and if detailed enough can suggest pitch.

The following image attributes can be controlled by simple values:

- Size.
- Color.
- Shape.
- Position.

Our visualization toolkit will use any of these types of audio information to control any of the image attributes.

Amplitude analysis

Amplitude is easy to detect with the average~ object, which calculates the mean sample value over a period determined by the argument (in ms). The update period should be adjusted to fit the input signal. Low frequency material may fool average~ if the update rate is too fast. 20 hz has a period of 50 ms. There's a sort of reverse Nyquist going on here-- in order to accurately average out low frequency tones, you must capture the entire thing, so low update rates are most accurate. Of course too low a rate will miss some detail of transients. In any case, the screen refresh rate of 60 hz makes measuring any quicker than every 30 ms pointless. In practice, updating 5 times a second is plenty .

Average~ has three modes:

- Bipolar just gives the mean—for most signals, this is 0, as they run both negative and positive.
- Absolute is the mean of the absolute values. For a sine wave with amplitude of 1.0 this will be 0.67.
- RMS is the root mean squared of the values, which follows the sensitivity of the ear reasonably well. For a sine wave of amplitude 1.0 this will be 0.707.

Avg~ is a simpler version of average that only produces an output when banged. It is limited to absolute response, but may be easier to synchronize with a video generator since its output is a float rather than a signal.

Sometimes we want to convert the output of average~ to dB. In Max there are both atodb and atodb~ objects. Atodb~ works at signal rate and is appropriate for building dsp processors such as compressors. Atodb will work for video synthesis, since we only do a few calculations per frame. Both implement the famous dB formula, which in an expr object is

20*log10($f1)

There is one problem with this expression: if the input is zero, the value returned by log10 is not defined. When that happens, atodb will output -inf, which may mess up processing somewhere down the patch. If we add 0.00001 to the output of average~ this problem is avoided, and we have the extra bonus of a predictable bottom to the range, in this case -100 dB. Using 0.0001 as the fudge factor will give -80 as the low point, which may be more useful.

Figure 1 shows a simple patch that uses the amplitude of a signal to control the size of a picture. Jit.lcd will be the heart of most drawing, since it behaves much like a typical bitmap display. Later we will look into the OpenGL environment. Jit.lcd is almost identical to the plain lcd. The only significant difference is the lack of sprite support, but since sprites just stored drawing commands, they won't be missed much.

Figure 1. uses the drawpict feature of jit.lcd.. First, an image file is loaded into a matrix named "face" with the importmovie command. This could be any type of image recognized by QuickTime.

The number supplied by avg~ is used to set the arguments for drawpict. These arguments indicate the horizontal coordinate of the top left corner, the vertical coordinate of the top left corner, the width of the drawing and the height of the drawing.

The alpha layer is ignored by drawpict. With line drawings similar to the one shown, you can make composite images by setting the penmode to 1 (or), but usually you will have one jit.lcd per drawing and combine the outputs with other jitter techniques.

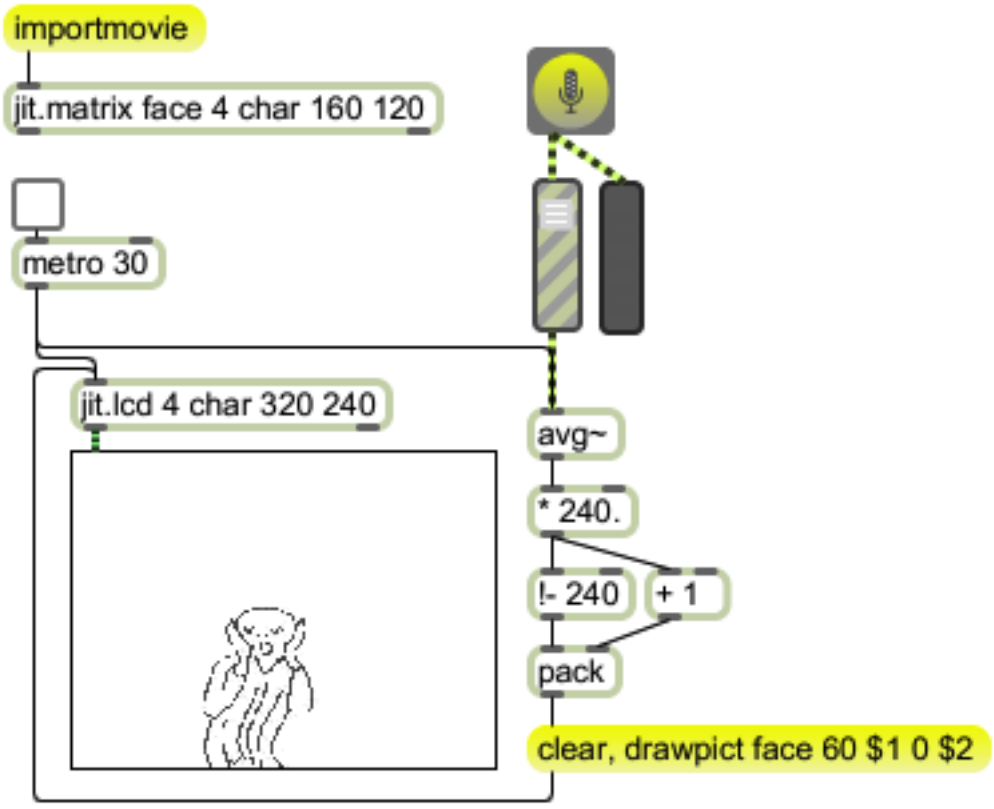


Figure 1. Amplitude controlling image size

Drawing directly to jit.lcd

Figure 2 does more complex drawing. It's 64 circles drawn touching at the center of the image. Any image that is radially symmetrical like this one is best drawn with polar coordinates. The metro drives the whole process, starting with the amplitude measurement and clearing the lcd.

The uzi generates 64 numbers, which are translated into angles (in radians) for the poltocar object. The amplitude (scaled from the original values of > 1.0) is banged into the radius input of the poltocar, which produces an origin in Cartesian coordinates for each circle.

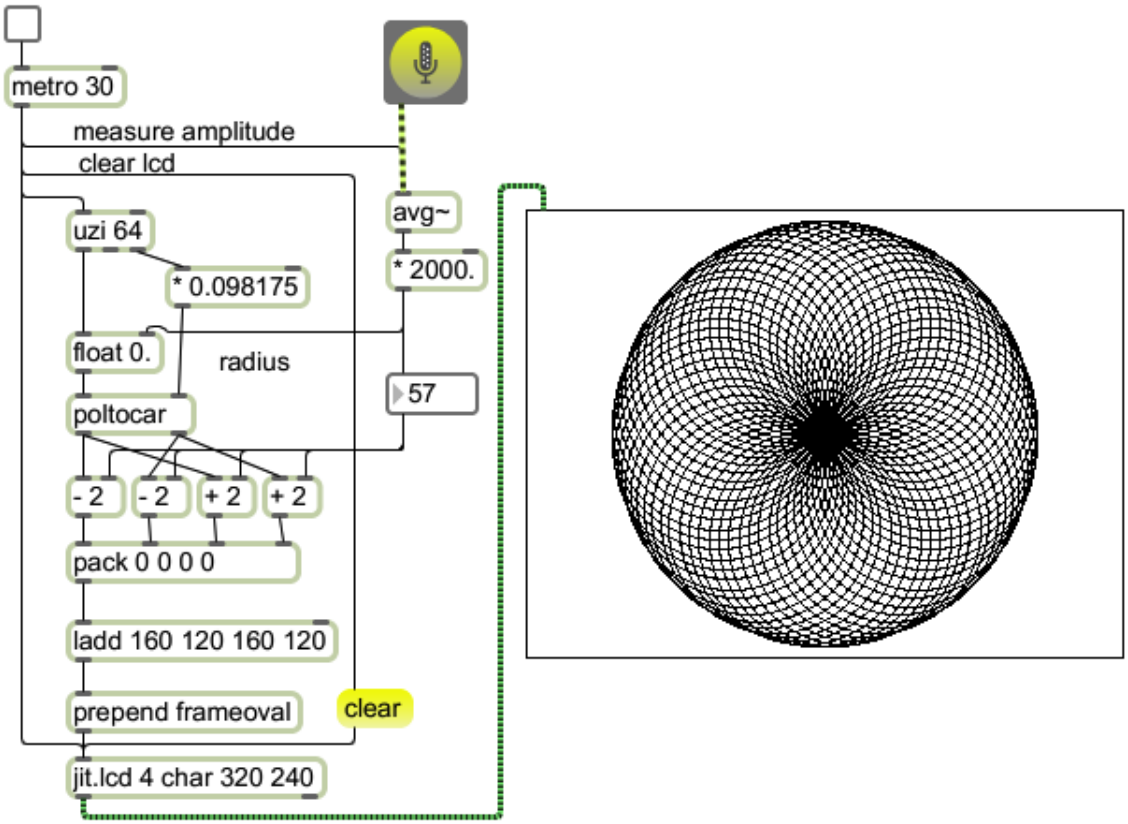


Figure 2.

The arguments to frameoval are the rectangle that encloses the circle. To get these from the origin and move to the center of the lcd, the arguments are calculated thus:

- Left is origin X + center X – radius
- Top is origin Y + center Y – radius
- Bottom is origin X + center X + radius
- Right is origin Y + center Y + radius

The dynamics of this patch are pretty simple. The image expands as the music gets loud, pretty much pulsing to a beat[1]. A more subtle effect is produced by keeping the circles constant in size and moving the origins. Here's a modification to the upper part of the patch of figure 2:

[1] This can actually make some people sick if not used carefully.

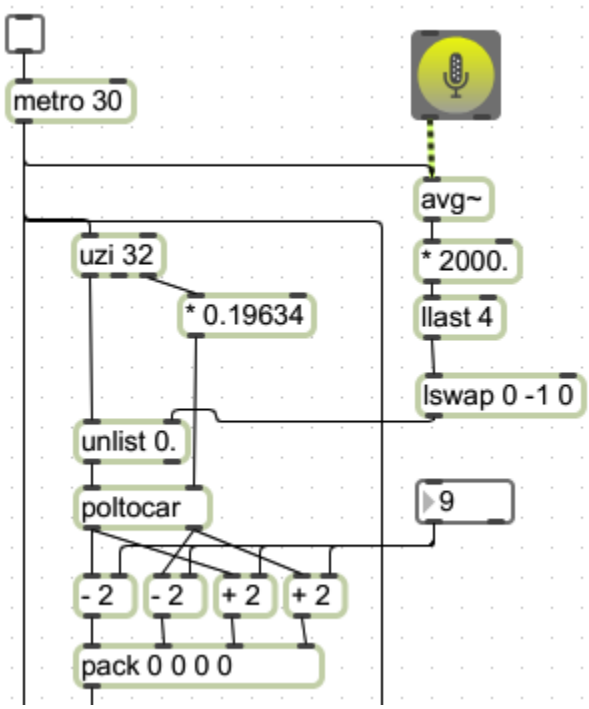
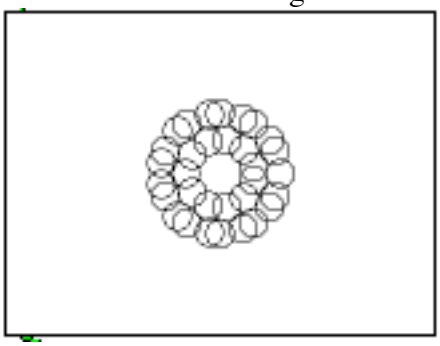
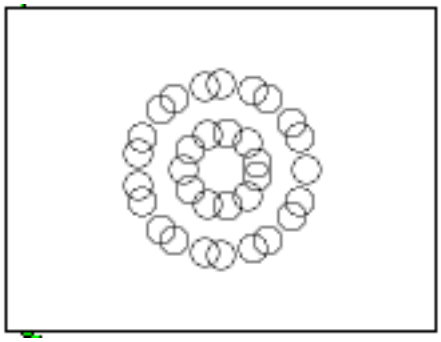


Figure 3.

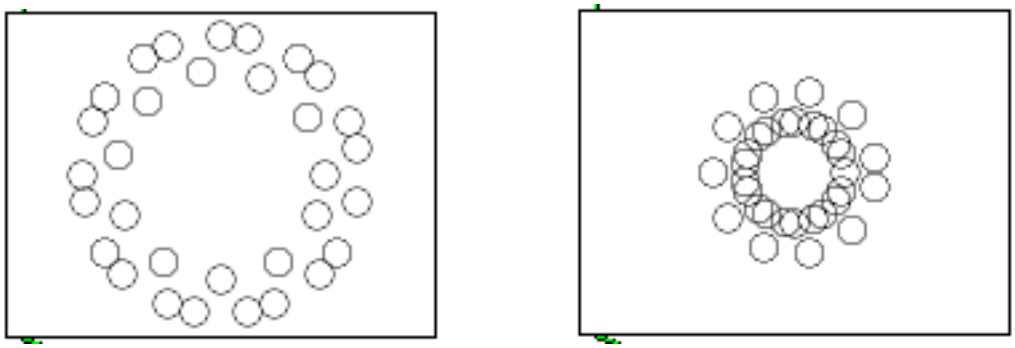
The automatic radius control has been removed and replaced by a user control.
The number of circles has been cut to 32. The amplitude measurements are now gathered into a list, expanded into a retrograde (by lswap, giving an ordering of 0 1 2 3 2 1 0) and via unlist used to give varied origins for the circles.
Results are shown in figure 4.



A



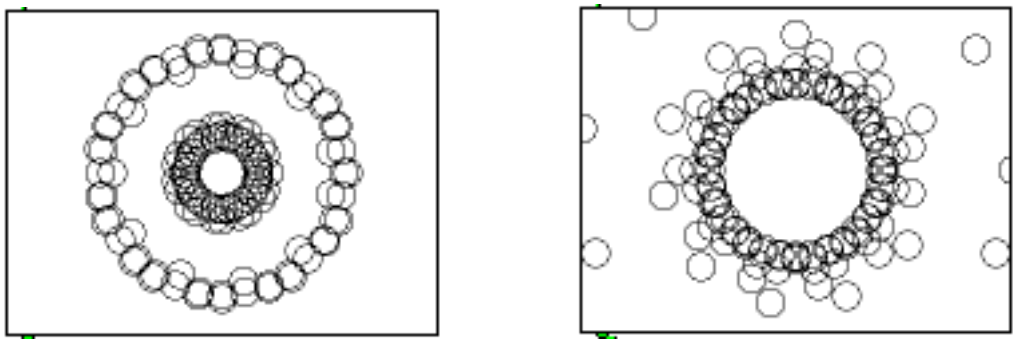
B



C D

Figure 4.
These also appear to spin, because 7 measurements are used to produce the 32 circles. If you look at 5B, you will notice a lone circle at the right side, where all the other outer circle are in pairs. This anomaly precesses around the image.

We can add even more complexity by only clearing the jit.lcd every fourth time.



A B

Figure 5.

This persistence effect can also be achieved with feedback. To find more about drawing in LCD, look at the Max & Graphics tutorial.

Color considerations in jit.lcd

We can add color to the frameoval message just by tacking rgb values to the arguments. Figure 6 shows the principle of synchronizing the color change with the uzi so each circle has a unique but stable color.

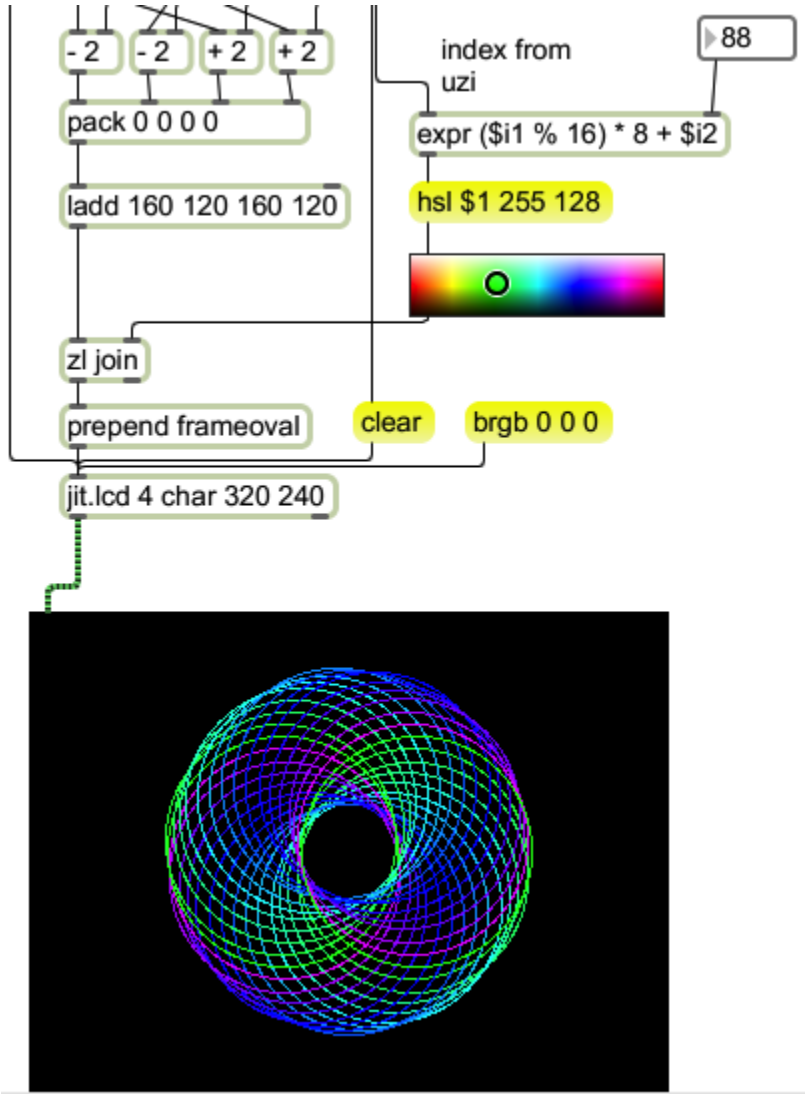


Figure 6. Adding color to jit.lcd drawing

The patch above the pack is the same as figure 2. The index value from the uzi has been used to peek into a swatch object for a cheap conversion from hue to rgb color values. The expression forces the index to repeat 0 to 15 four times and multiplies this by 8 to increase the range of color change. That produces colors that balance through the image. The number box allows tweaking the color range. Note that the background of jit.lcd has been set to black with the message brgb 0 0 0, and the swatch has been set to "old style output".

Pitch Driven Drawings

Pitch extraction is tricky business. There are a couple of third party pitch extractors like fiddle~, but the best graphics results will happen when pitch is derived from Midi data. Once pitch is known, mapping to graphic elements is very straightforward. Here is a patcher that triggers a short animation with each note.

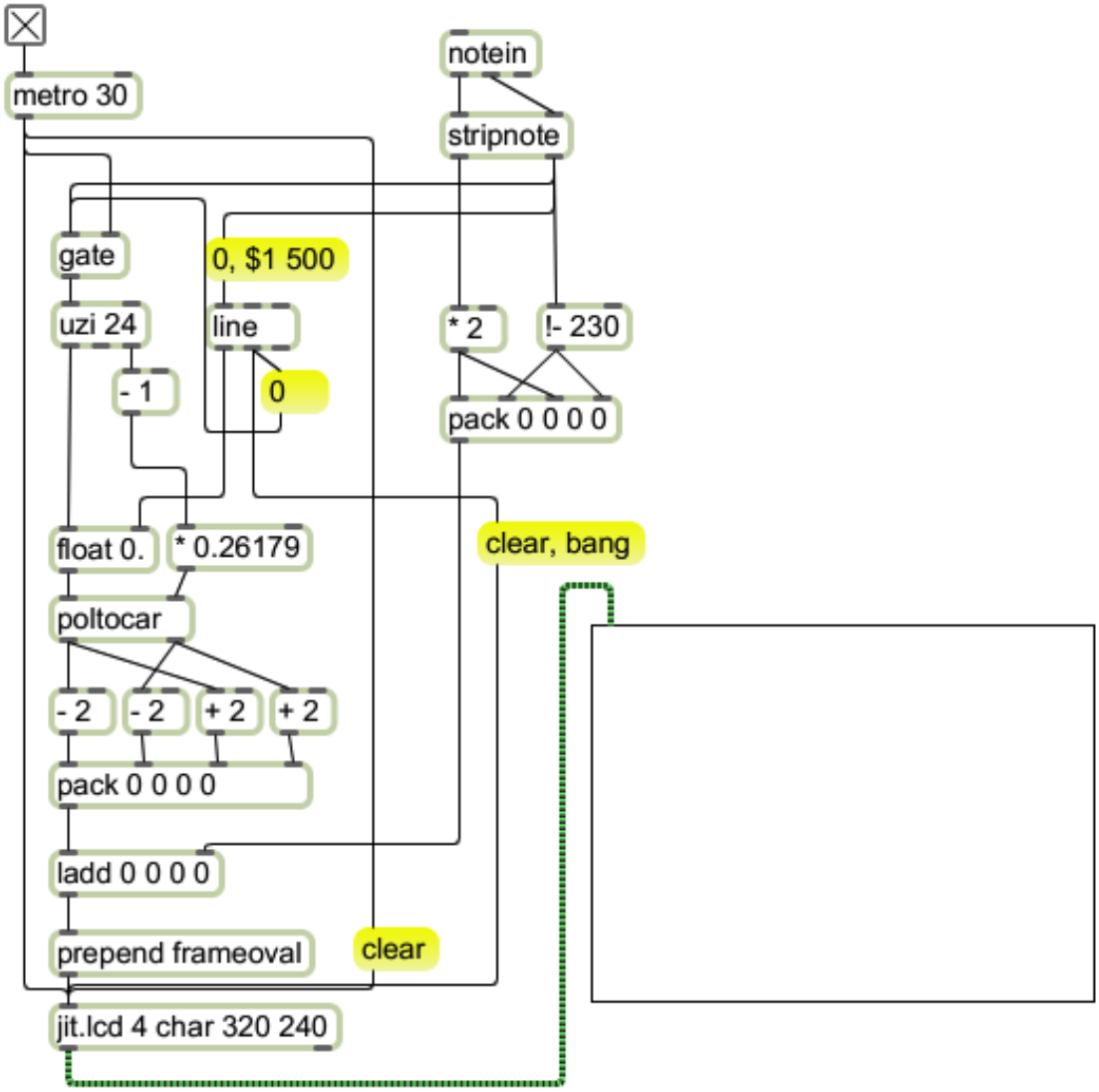


Figure 7.

This has many elements from figure 2. Drawing is triggered by the notein object, when a velocity opens the gate and lets bangs reach the uzi. The pitch and velocity are used to set the origin for the figure, with pitch mapped left to right and velocity to height. Velocity also sets the target for a line, in a manner that should be familiar from synthesis in MSP. The line sets the radius of the circle, which is drawn by uzi and poltocar as before. The effect is a little explosion of dots that will be wider on high velocity notes. Note that the uzi is and turned off when the line hits its end. The line end also sends a clear and bang message to the lcd. This makes the lcd go blank after the explosion is finished.

To draw multiple notes at once, we use poly~.
The drawing mechanism is converted into a subpatch as in figure 8.

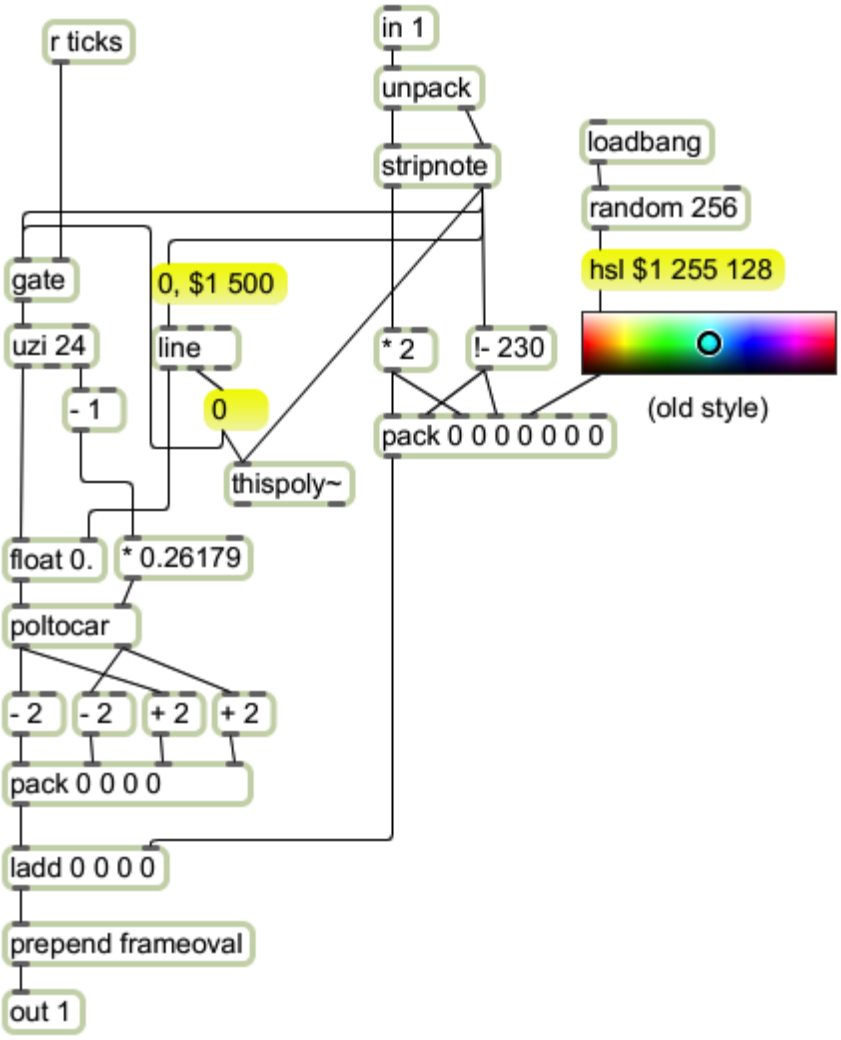


Figure 8.

Here the note data is treated exactly the same way, with the addition of a random color that will be determined when the patcher is opened. The bangs from the metro in the main patch are brought in via the ticks receive object. Note that the values used to gate the ticks to uzi can also serve to provide busy status to thispoly~. The patch is saved as boom_poly.

Figure 9 shows the enclosing patch. There's very little change here. The midinote message to poly~ will trigger drawing from the first available subpatch.

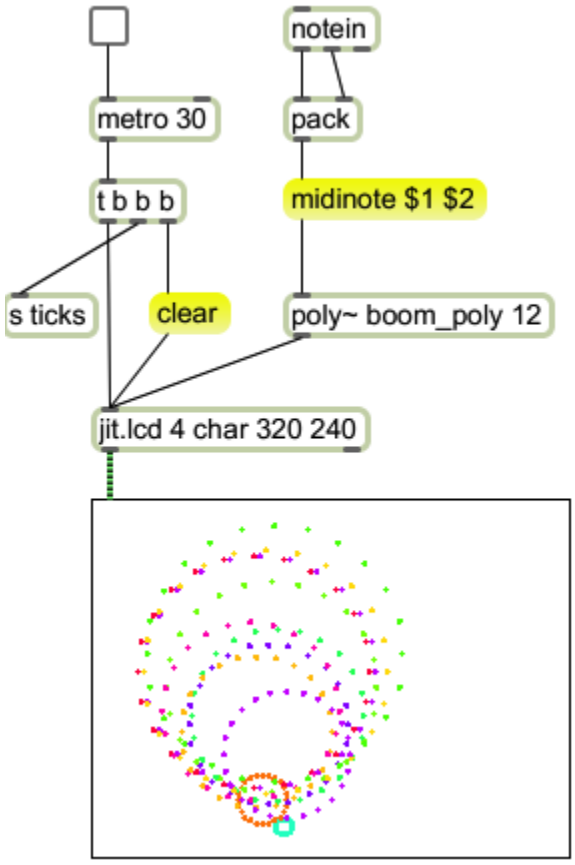

notein
metro 30
pack
t b b b
midinote $1 $2
s ticks
clear
poly~ boom_poly 12
jit.lcd 4 char 320 240

Figure 9.

Drawing waveforms: the easy way.

Since version 1.6, Jitter has offered a pair of objects that greatly simplify the display of waveforms. Here's how to use jit.catch~ and jit.graph to make an oscilloscope:

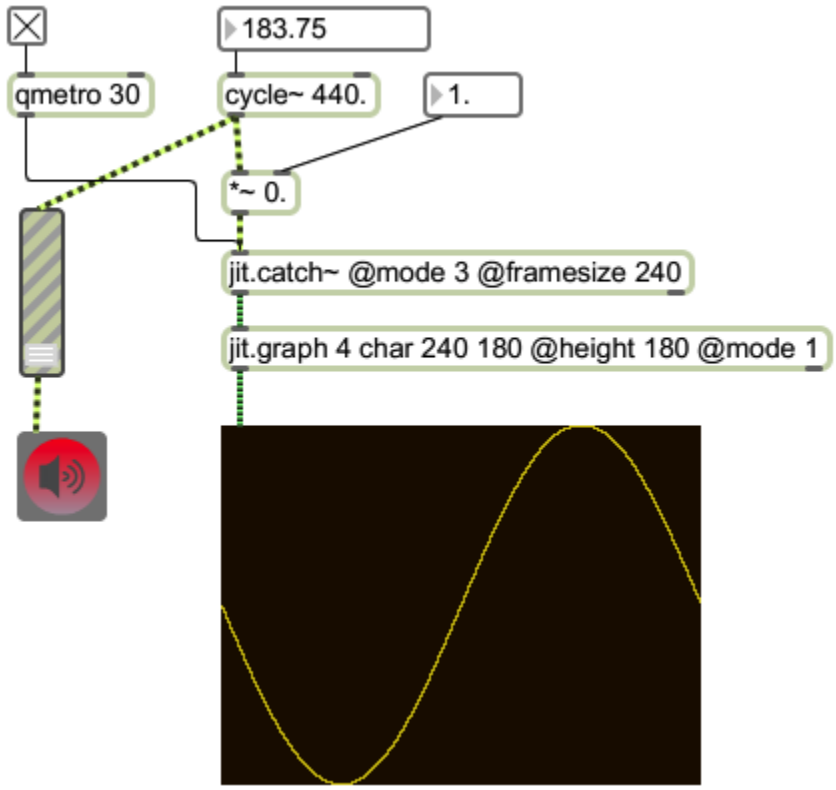


Figure 10.

There are several factors that are important in getting a clean display. Most are related to attributes of the jit.catch and jit.graph objects.

The framesize attribute of jit.catch determines how much wave will be displayed across the window. Specifically it's the number of samples handed to jit.graph at a time. The best setting is the same as the width of the display window. Then jit.graph will not need to resample the curve and the result will be nice and smooth. Otherwise, there will be some bumps in the curve.

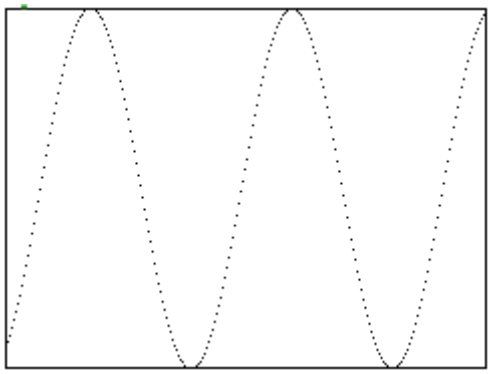
The height attribute of jit.graph defines the number of pixels to use for a full amplitude signal (-1.0 to 1.0). This should match the height of the display window. The effect of a mismatch is gaps in the curve. Just for good measure, I set the matrix dimensions for jit.graph to match the window.

Jit.catch has several modes of operation:
Mode 0 dumps everything since last output in one long matrix. Jit.graph will resample this to squeeze the entire signal into the display. Since there is no synchronization between the bangs that cause output and the signal, the display will dance around.

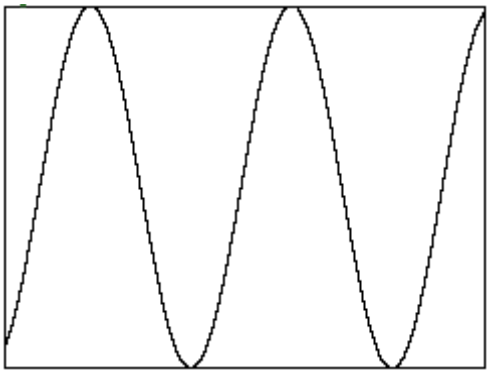
Mode 1 dumps everything since the last output as a matrix of framesize by the number of rows it takes to hold the whole thing. Jit.graph will display the first frame, and others are available via jit.unpack. This might be useful in displaying successive frames of an fft in a waterfall type spectrogram. This mode is not synchronized either.

Mode 2 skips to the most recent frame and outputs only that. It produces snapshots of the waveform at each bang. It is not synched.

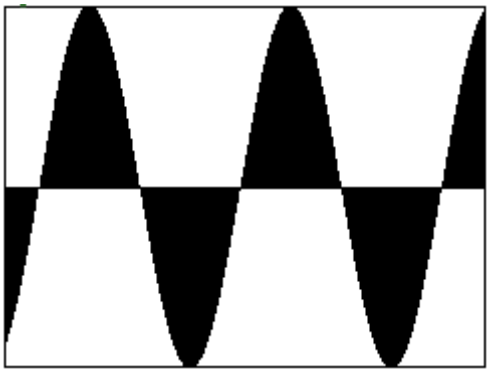
Mode 3 outputs the most recent frame but always starts at the point where the input matches a trigger threshold. This keeps the image stable in the display. Interestingly, the value defined by trigthresh is shown in the center of the display rather than the left edge, as is typical of oscilloscopes. If trigthresh is outside the range of signal values, the display will not sync.



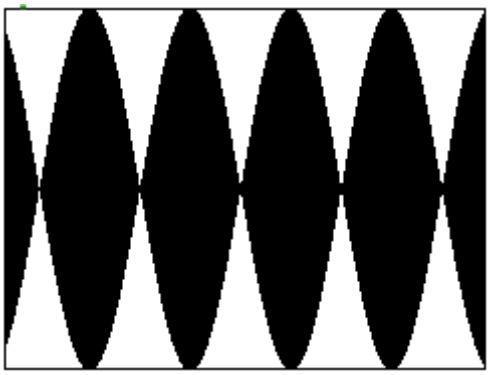
Mode 0



Mode 1



Mode 2



Mode 3

Figure 11

Jit.graph also has modes. These determine how the data from jit.catch are displayed. Figure 11 illustrates the options. The colors of the display are set by attributes of jit.graph. Brgb determines the background color, with 255 255 255 specifying white. Frgb determines the drawing color: that requires four arguments, the first setting alpha.

Drawing Waveforms the Old Way

Jit.graph is great object, but there is still some benefit from displaying waveforms using jit.lcd. Figure 12 shows the display section, which will also generate some synchronization signals.

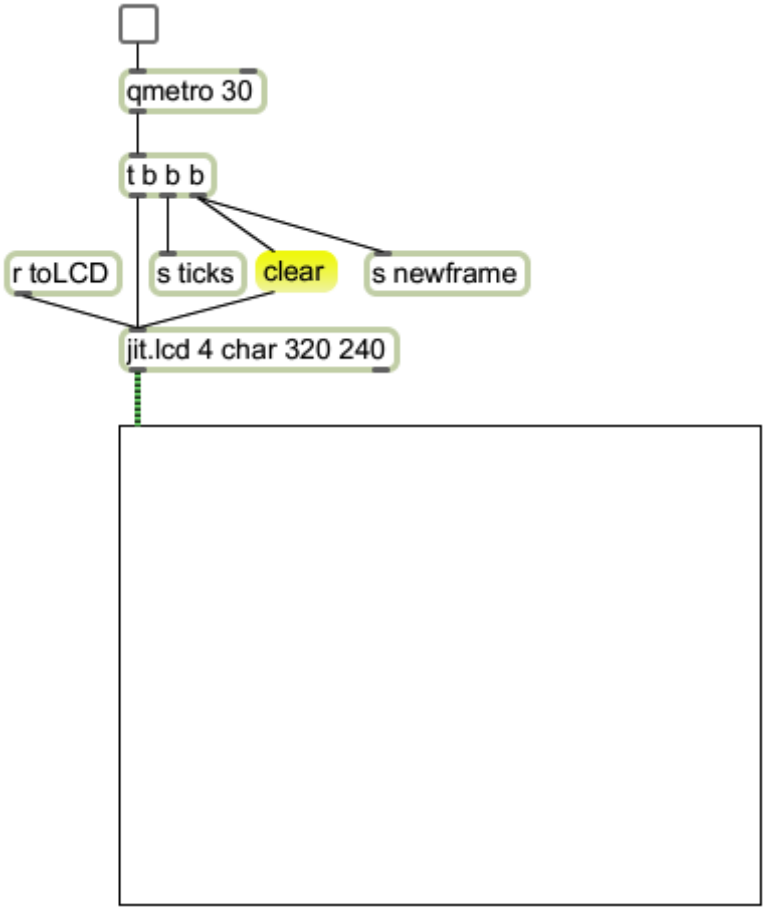


Figure 12.

Note that distinct bangs are sent to newframe and well as ticks. Newframe will initialize drawing routines and ticks will trigger them. Separating the functions this way keeps the image on the screen as long as possible and reduces flicker. Figure 13 shows the signal capture section,

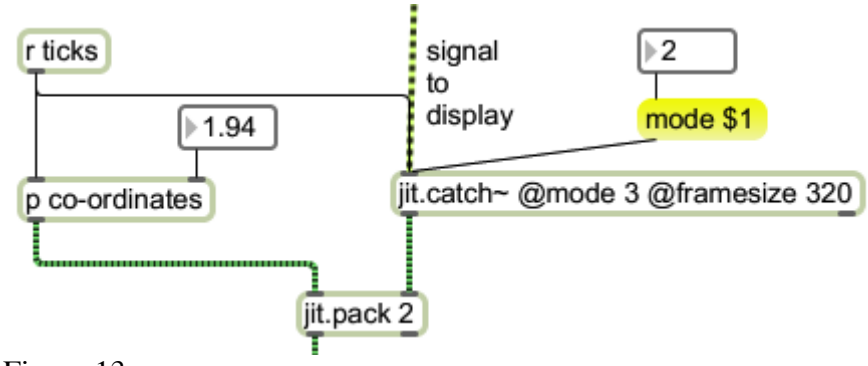


Figure 13.

Catch creates a 1-row matrix with 320 sample values. The receipt of a bang from the qmetro via ticks sends this matrix to a jit.pack. The jit.pack object will create a matrix that is 2 rows, each 320 values long. The purpose of the co-ordinates subpatcher is to create the X values for graphing the waveform. The internal works are shown in figure 14.

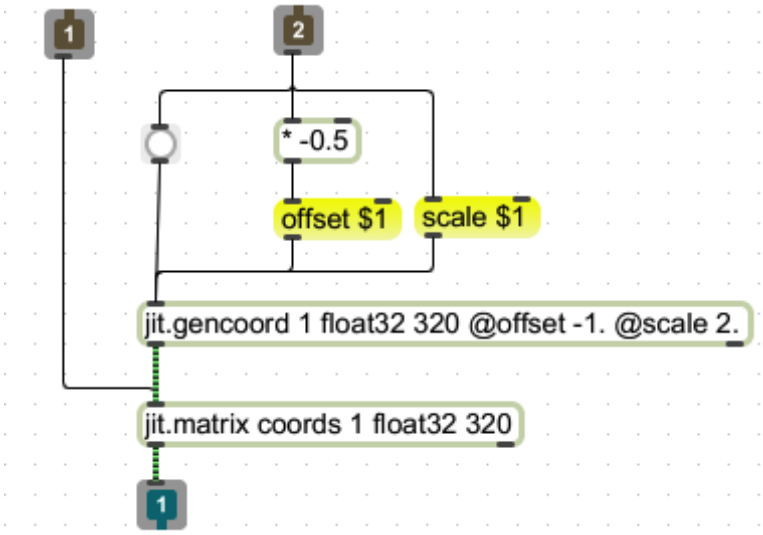


Figure 14. (This should include a loadbang to gencoord for initialization.)

Jit.gencoord produces a matrix that is filled with a float version of the index of each cell. Float indexing assigns a value of 1.0 to the last member of a data array, 0.0 to the first and appropriate intermediate values in between. That way an index of 0.5 points to the center regardless of the size of the array. We have seen this in buffer~ and related objects. In jit.gencoord, the scale and offset attributes modify the index in a familiar way: all values are multiplied by scale, then offset is subtracted. I use them here because I want to match the positive and negative signal values with positive and negative coordinates. That way the image will be centered in the LCD when I manipulate the size. Notice in figure 14 the result of the gencoord operation is stored in a matrix and banged out when needed. There's no point in recalculating indices if they are not changed.

The matrix from jit.pack has the X coordinates in the first row and the signal values in the second row. Figure 15 shows how these are processed.

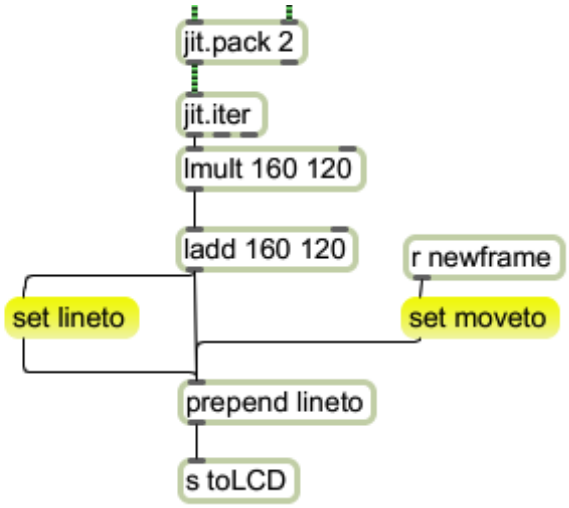


Figure 15.
Jit.iter breaks the matrix into lists of X-Y pairs. These are manipulated individually and used to draw short lines in the jit.LCD. Lmult expands the values to the size of the display, and ladd offsets the drawing to the center of the screen. The moveto and lineto commands for LCD manipulate the drawing pen. Moveto locates the pen without drawing, then lineto draws from wherever the pen is to the desired destination. One moveto at the beginning starts drawing at the left edge. It may seem inefficient to set the prepend contents to lineto for every segment, but any code that determined it was unnecessary would take more time to execute than the set command. The result of this drawing is shown in figure 16.

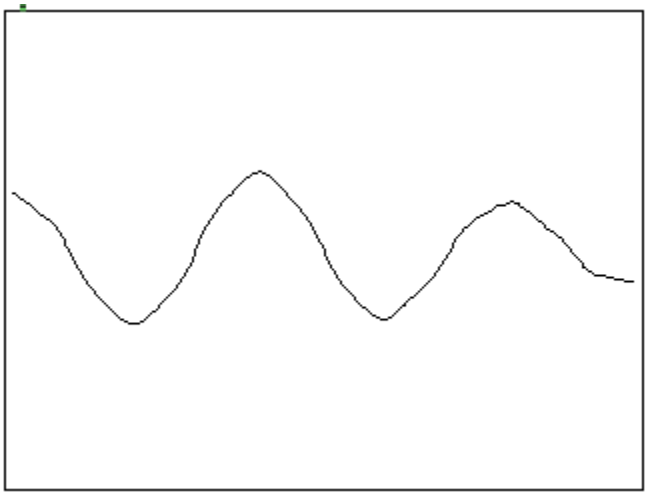


Figure 16.
Jit.graph and jot.plot are presumably more efficient in generating this image, but now that we have access to all of the calculations, we can modify the image in ways not afforded by jit.graph. For instance, we can produce a mirrored waveshape by duplicating the drawing routine, with one minor modification. See figures 17 and 18

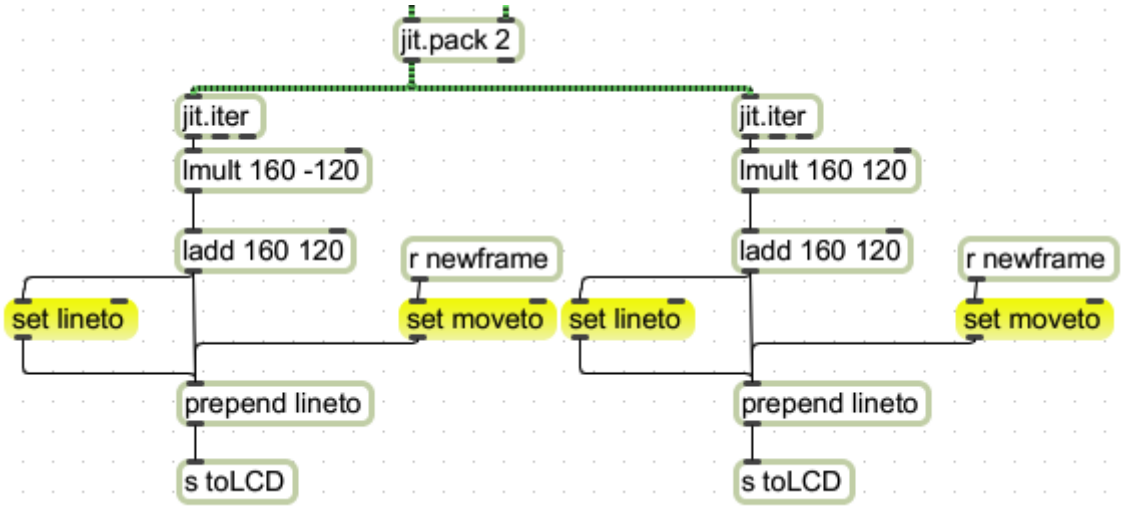


Figure 17.

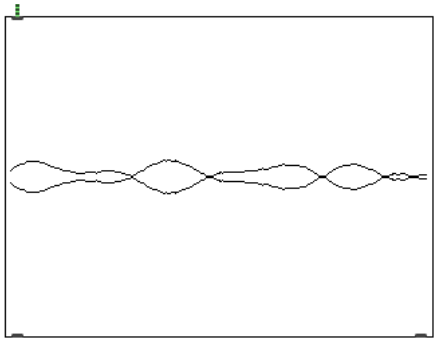


Figure 18.

We can generate filled patterns with the scheme in figure 19.

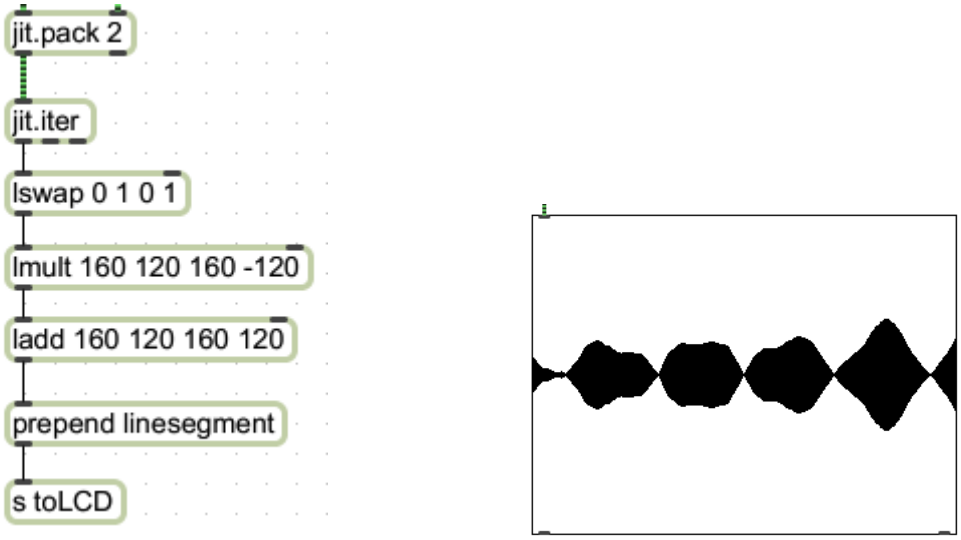


Figure 19.

The line segment command takes two points for arguments; a start and stop. Drawing from the wave to the inversion results in the block patterns shown. This is similar to

jit.graph in mode 2, But these blocks can be made more colorful by the modification in figure 20.

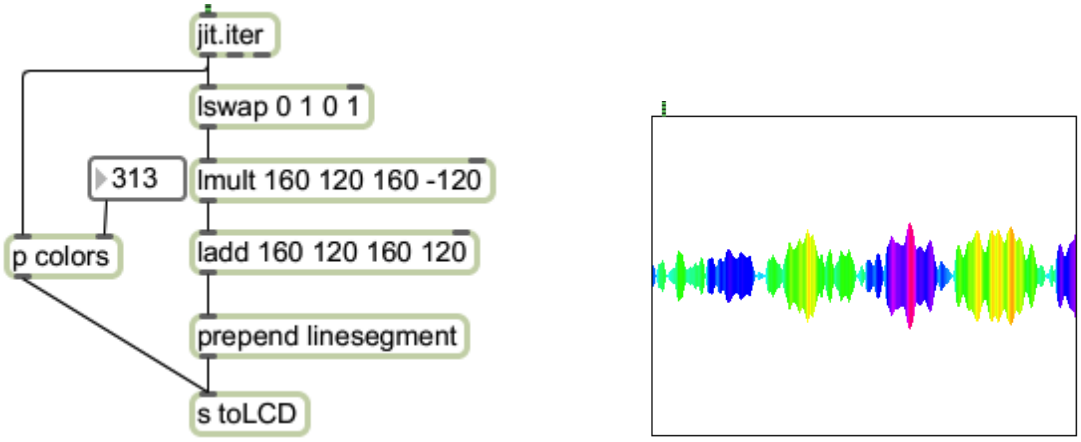


Figure 20.
Of course the secret is in the colors patcher, but you can probably guess what you might find in there.

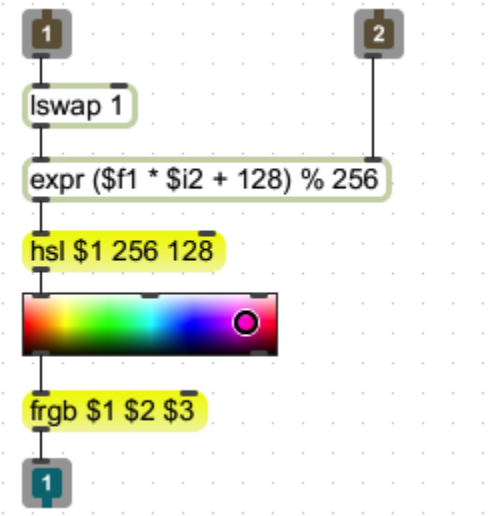


Figure 21.
Figure 21 should just give you ideas. This takes the signal value (Lswap 1 grabs the second member of the input list) and generates a color proportionate to the level. You may wish to set the color by overall amplitude or some other method. (Remember to set swatch to output old style 0-255.)

Polar waves

Polar display of waveforms provides an interesting alternative to the oscilloscope style. To generate polar displays, I've made some changes to the top of the patch. The jit.catch mechanism is modified as shown in figure 22.

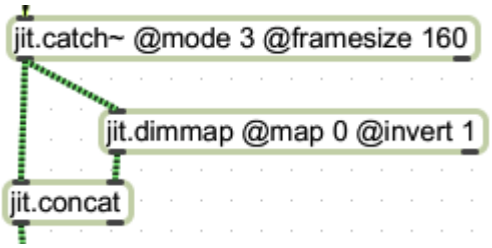


figure 22.

This grabs a half frame, reverses it (via jit.dimmap) and adds that to the original. The result is a palindrome- the beginning and end will be mirror images. (This trick is useful any time you want a symmetrical display.) The coordinates are also modified to run from -3.14 to 3.14, which will be the angles of the display. The meat of the patch is in figure 23.

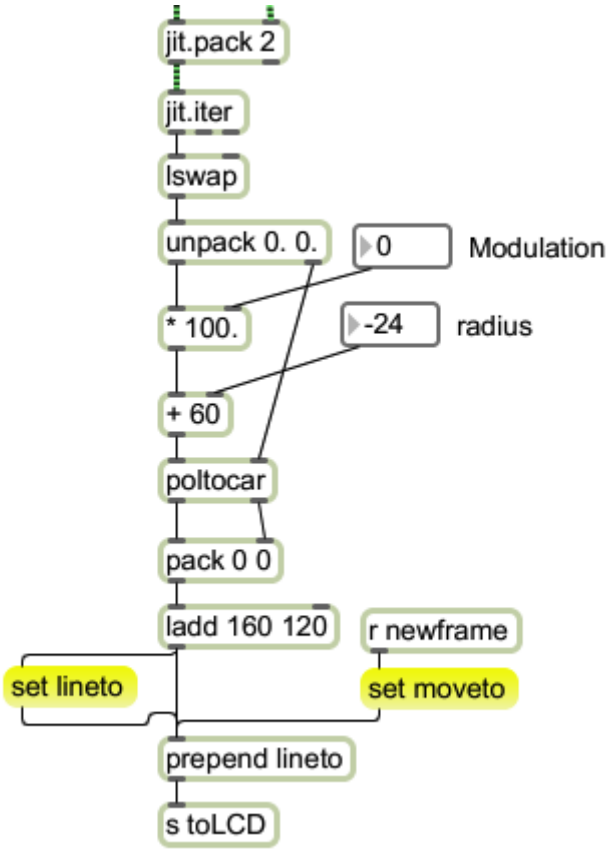


Figure 23.
Here the signal is taken as the radius and the angle from the coordinate generator. Since poltocar requires the angle at the right inlet, lswap is used to exchange the two numbers. These are unpacked, and some math done on the radius. The modified value is applied to poltocar, then treated just like figure 15. The results can be seen in figure 24. These butterfly-like shapes flicker and jump with the audio. The radius control sets the size of the circle generated when there is no signal, and the modulation controls the size of the lobes produced by the signal.

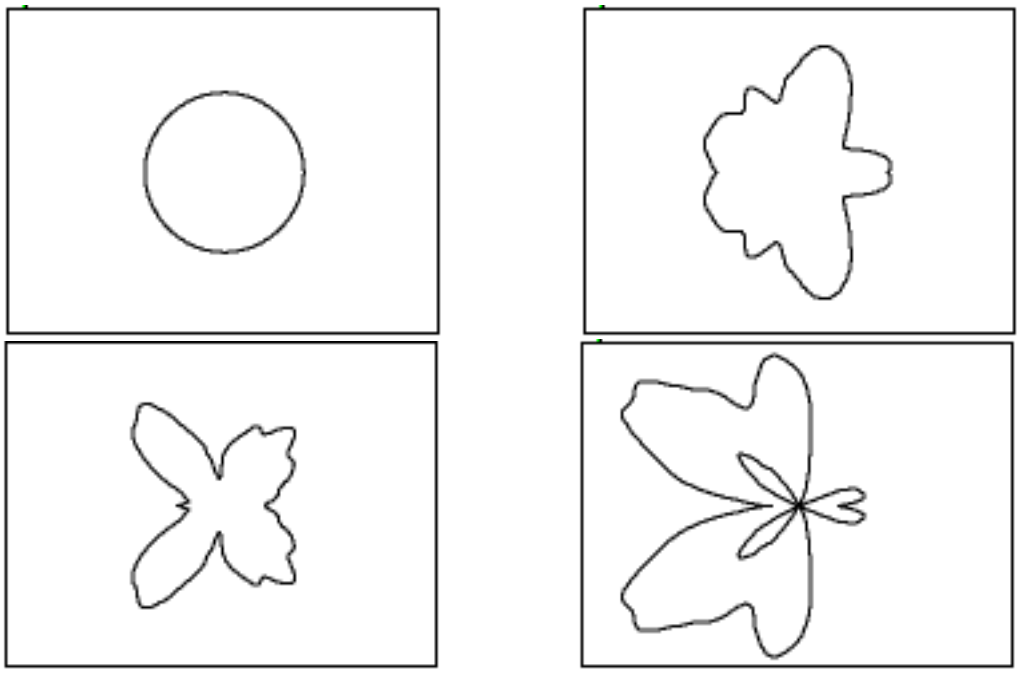


Figure 24.

Some additions will turn this into a solid shape with colors

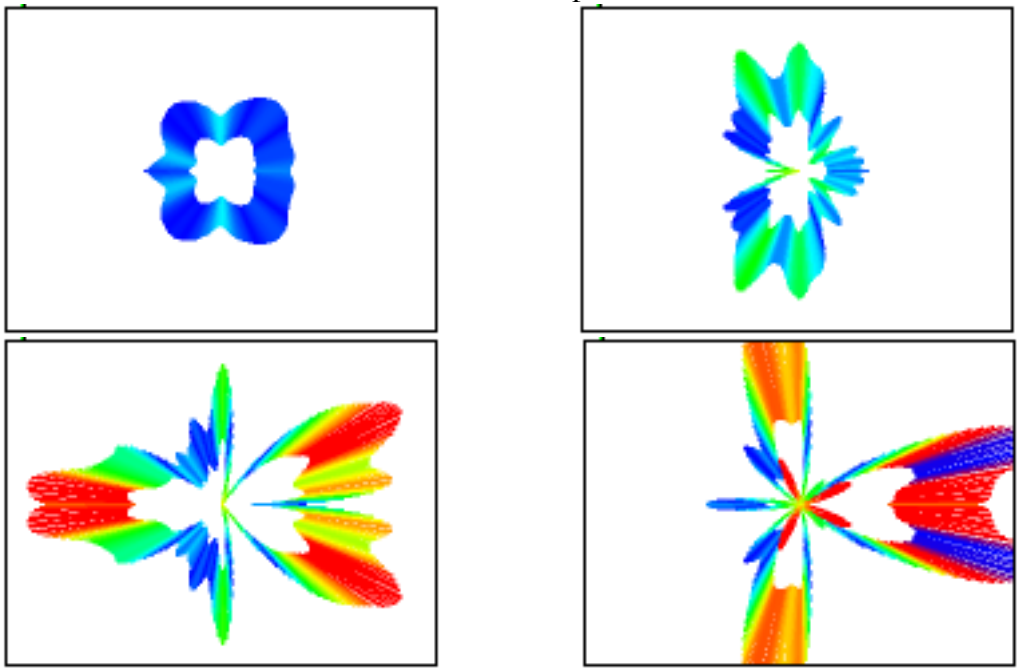


Figure 25
Here's the modified patch:

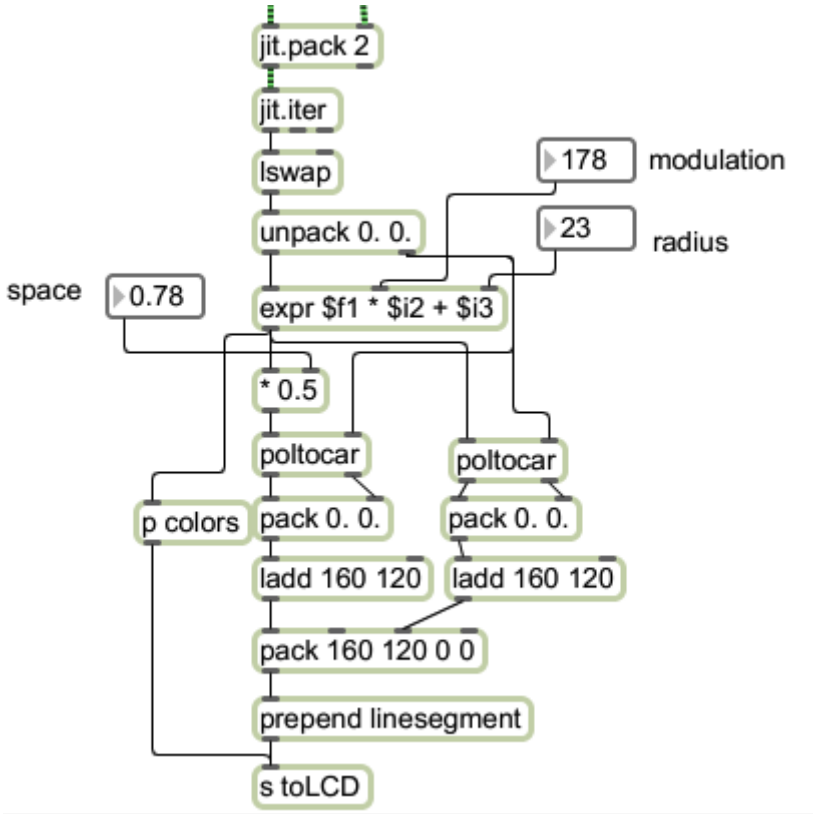


Figure 26.
This calculates two points for each sample value and draws a line between them. The line is part of the radius at each angle-- the end of the line is determined by the signal value, and the start of the line is some percentage of that. Thus the space control adjusts the thickness of the solid section.

X Y Waveforms

Two related but somewhat different waveforms (say a stereo pair) can be displayed in the so-called XY format. One wave provides the X or width value and the other provides the Y for height. The most famous examples of this are the Lissajous figures, which are described in detail in the tutorial "The Art of Lissajous."

To get an XY display, we just replace the coordinate generator with a 2 channel jit.catch object as shown in figure 27.

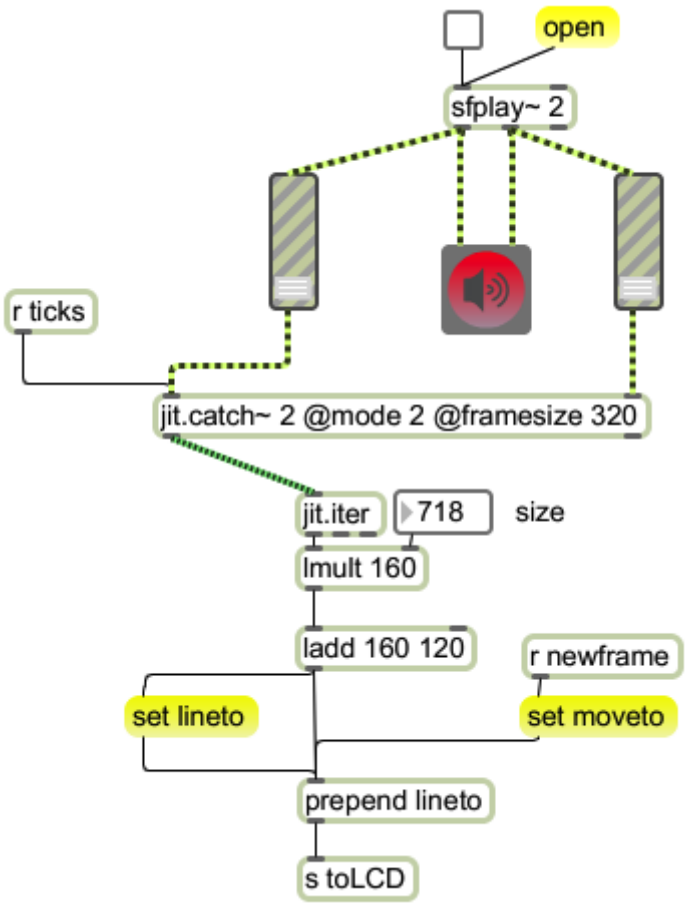


Figure 27.

You will also note the size control has been modified to produce a square image. Initial results are likely to be disappointing, as figure 28 suggests.

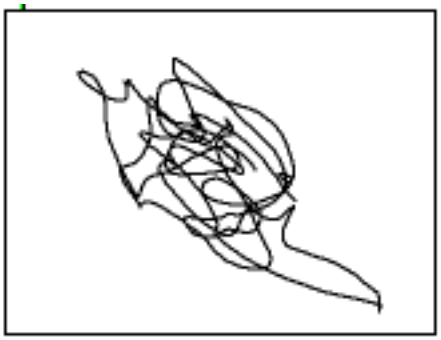


Figure 28.

This is the basic scribble. It's quite dynamic, and will fill the screen sometimes, but with most material it will stick close to the diagonal, and quickly becomes tedious. It can be helped with coloring techniques and various jitter processes.

We can do better by using classic Lissajous figures for the basic shape. This is done by adding a pair of cycles to the recording system.

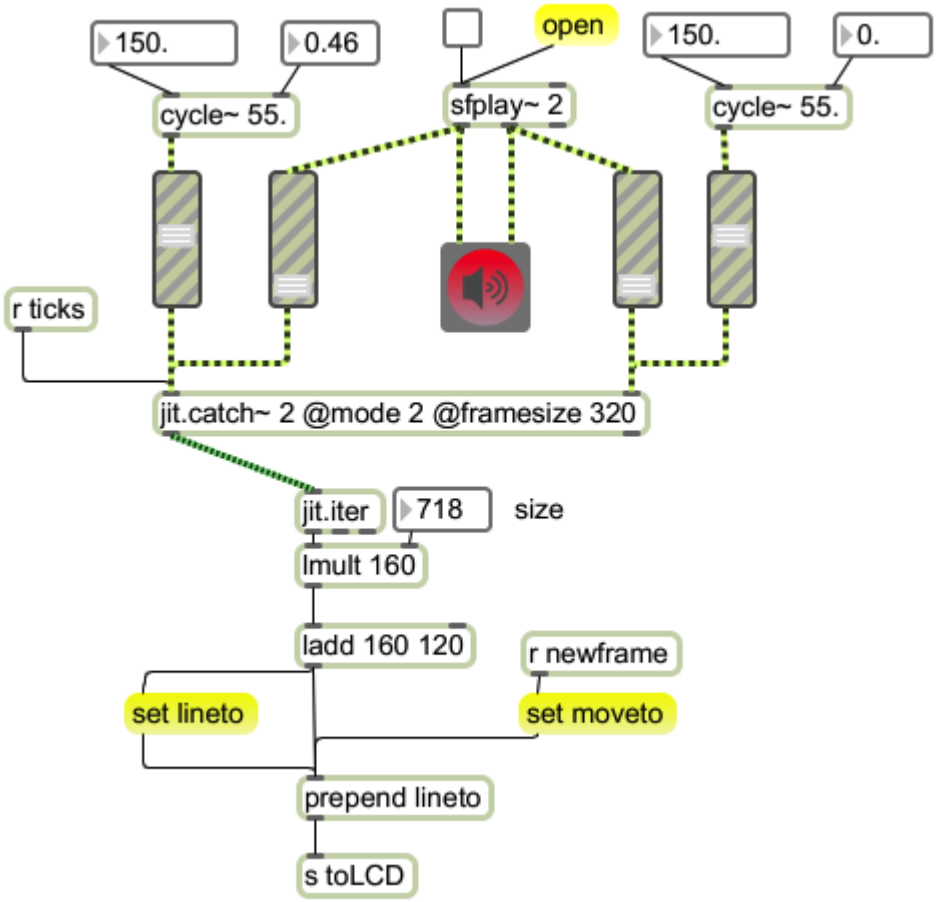


Figure 29.

The frequency, amplitude and phase controls will be manipulated to generate the figures. The left and right input signals are added to these and will turn the smooth curves of Lissajous into something more wiggly. (The ratios are frequency ratios of the base pattern.)

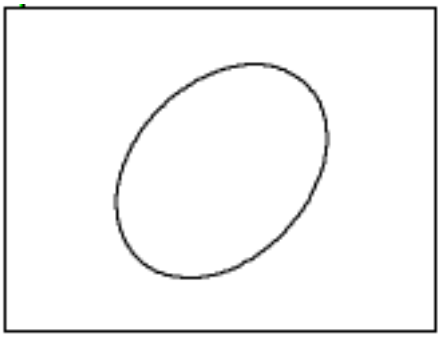
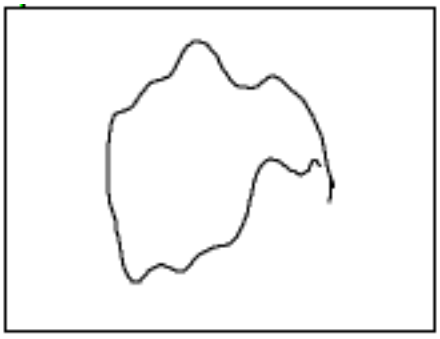
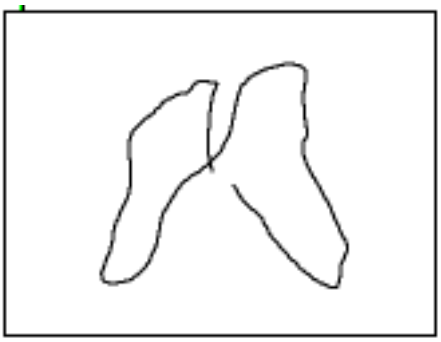


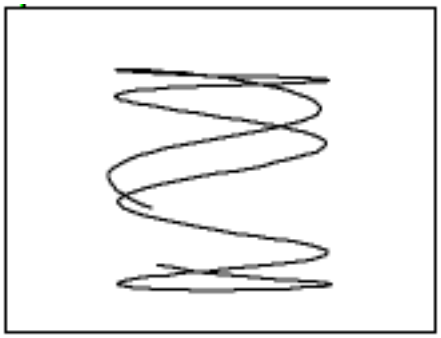
Figure 30A 1:1 with no audio



30B 1:1 , audio added to Y



30C 1:2 with audio added to Y



30D 5:1 with audio on X

The stills don't really do these justice. With moderate amounts of audio added, the Lissajous figures acquire extra depth and dynamic action.

Spectrum Driven Images

The spectrum of an audio signal can also be used to generate images. One approach is to take a fast Fourier transform with the fft~ and put that into the same process that shows waveforms.

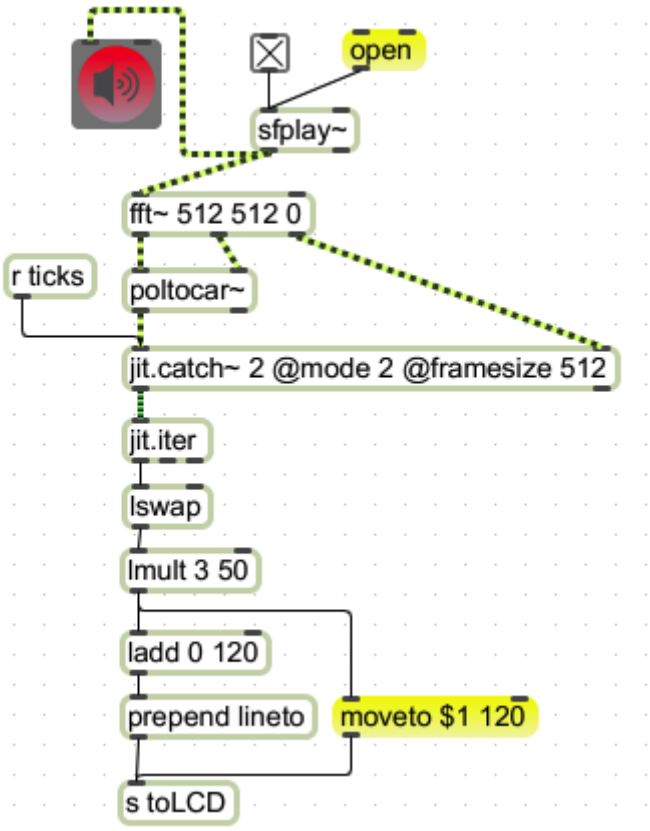


Figure 31.

This is right out of the fft~ help file. The cartopol~ converts the fft data into magnitude and phase form. Note that we record the fft into the left channel of a very short buffer~ and the fft index into the right channel. Displaying this is very much like the XY

waveform technique. The left channel provides amplitude and the right channel assures that the sample will be placed in the proper spot on the screen.

A couple of things should be mentioned here. There are 512 samples in the fft, so the jit.lcd must be expanded to show them all. Also the code is modified to start drawing at the left edge instead of the center. The moveto mechanism displays each bin as a vertical line. If I displayed all of this, the result would be figure 32.

Figure 32.
You can see there's a little bit of action at each end, but the middle is a wasteland. Everything above bin number 60 or so is always so close to 0 that it won't show, and bins above 256 are a reflection of the first half of the fft.[2] The patch in figure 31 actually produces this:

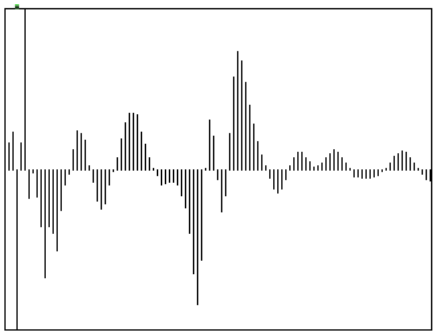


Figure 33
By multiplying the X value (index) by three the image is stretched and the duplication lost.

[2] If this is news to you, you may want to look at the Fourier Notes tutorial.

A radial display of the fft can be interesting:

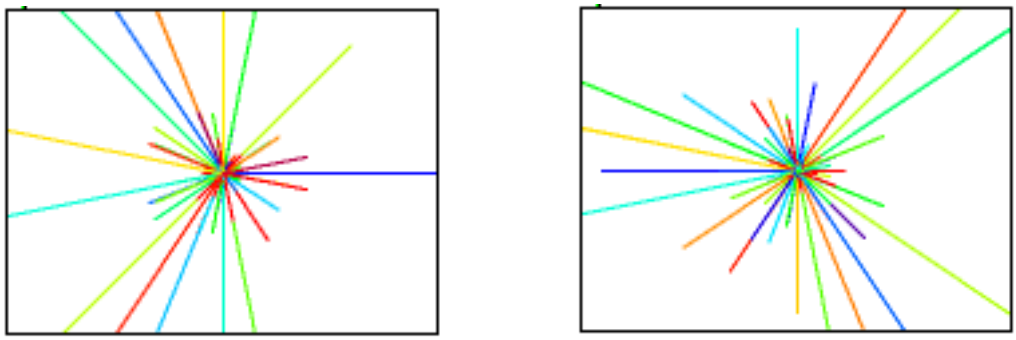


Figure 34

Here is the patch that makes it happen:

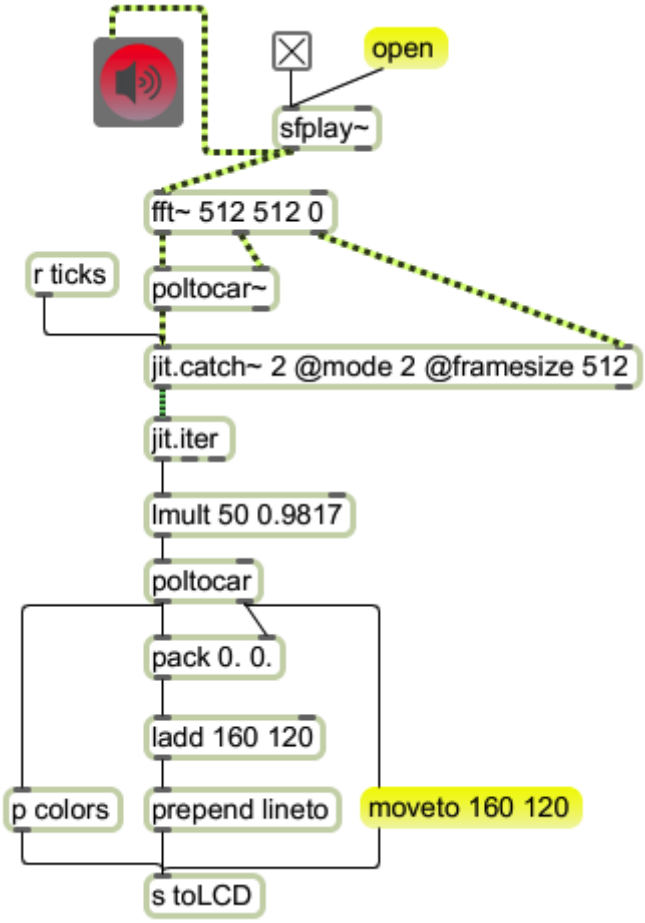


Figure 35

The fft index is multiplied by 0.98171 which divides the circle into 64 spokes. This means 8 different points of the fft will be combined on each spoke. The reflection points will give the image a bit of symmetry.

Deriving Spectra with a Filter Bank

An alternate approach to generating images from spectra of sounds is to use the fast fixed filter bank. This gives an effect like the old analog color organs, where a set of band pass filters would be used to power colored light bulbs. Here's an ambitious example:

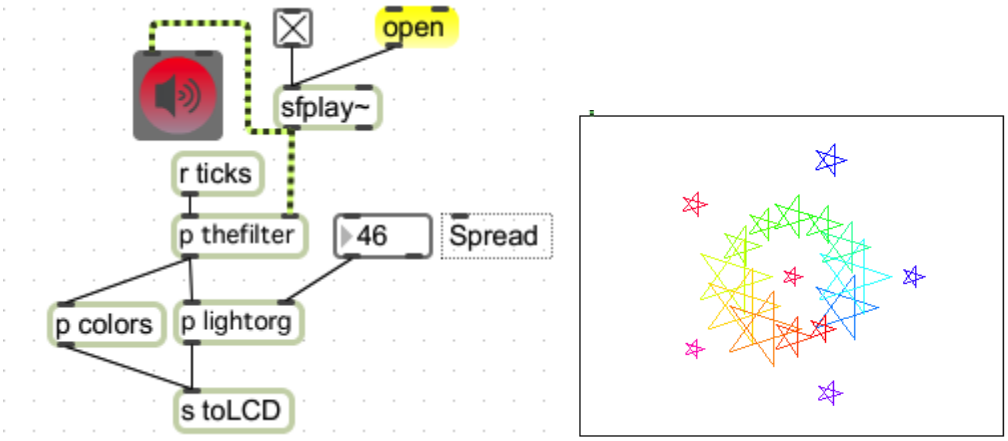


Figure 36.

The broad outlines of the process are show by the sub patchers—the incoming audio is analyzed by a filter, and the filter output generates shapes that are drawn in the jit.lcd and appear in the jit.pwindow.

This filter is daunting, but simple in concept:

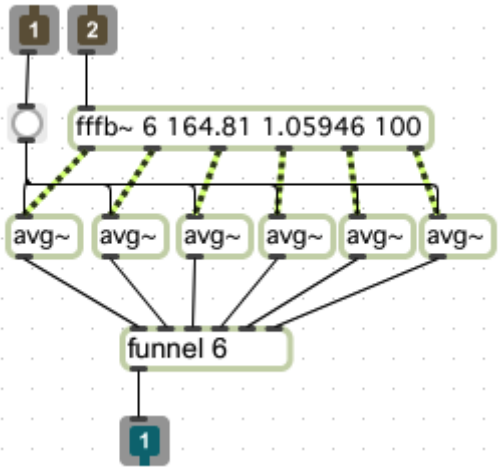


Figure 37.
The fffb~ object is working as a half octave filter in this example. The output of each filter section is averaged over the frame period to give a value that will determine the size of the associated object. The filter can have many more bands. I usually have 18. Funnel will pack each value into a list after an index number that identifies the filter section. These two item lists are used by the light organ subpatcher.

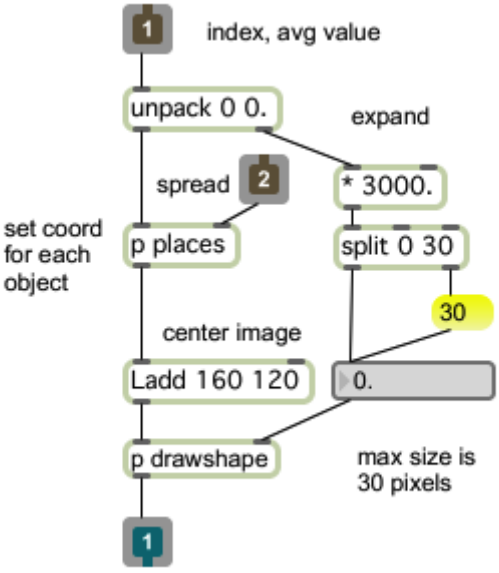


Figure 38 Lightorg

The light organ section contains a subpatch called places :

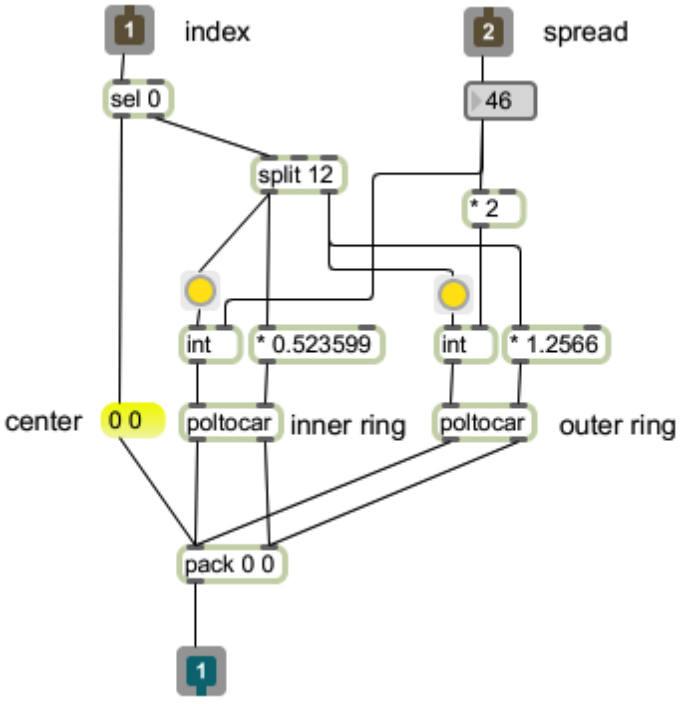


Figure 38 Places

This calculates a center point and size for each object. Note that three radii are used in this version. The 0 object is centered, objects 1- 12 are at the spread value, and 13-18 are twice as far out. The object is constructed in the drawshapes subpatch of lightorg, which could hold the code to draw anything. In this case, star shapes are drawn by the patch in figure 40.

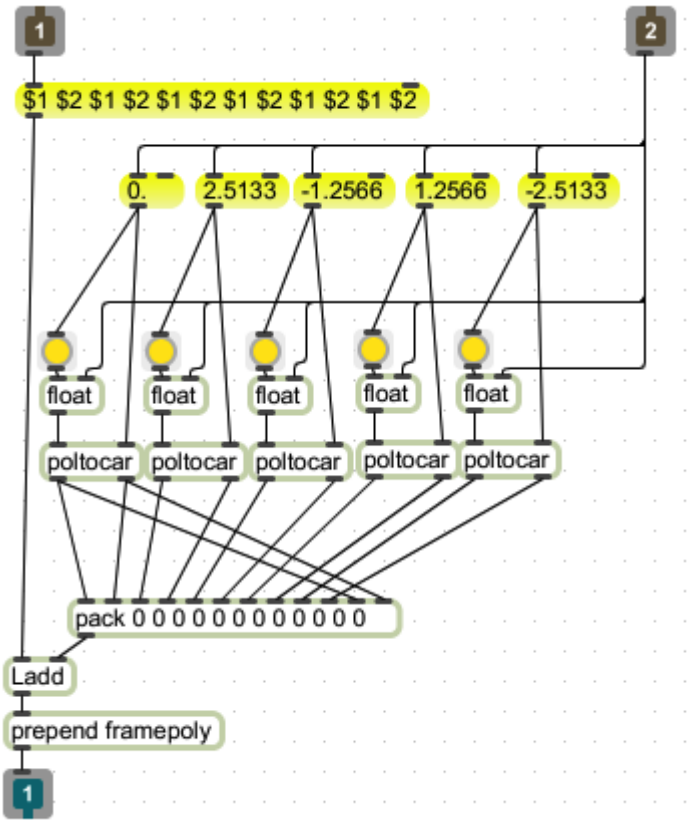


Figure 40.

This calculates the five corners of a pentagram. The size value will arrive first, and is used as the radius of the corner points. (The angles are preset. A fancier version would allow the rotation of the images.) These are packed into a list and stored in the ladd object. Finally the center coordinates turn up and get added to all five points. The framepoly command is a convenient way to draw complicated shapes.

Further….

This is just a sampler of visual effects that can be generated from live sounds. These graphics are deliberately simple, but they can easily be expanded by replacing the drawing modules. They are also excellent sources for jitter effects like rotated feedback. After a bit of experimentation, you will quickly develop a library of your own techniques.