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 version 4.5 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.

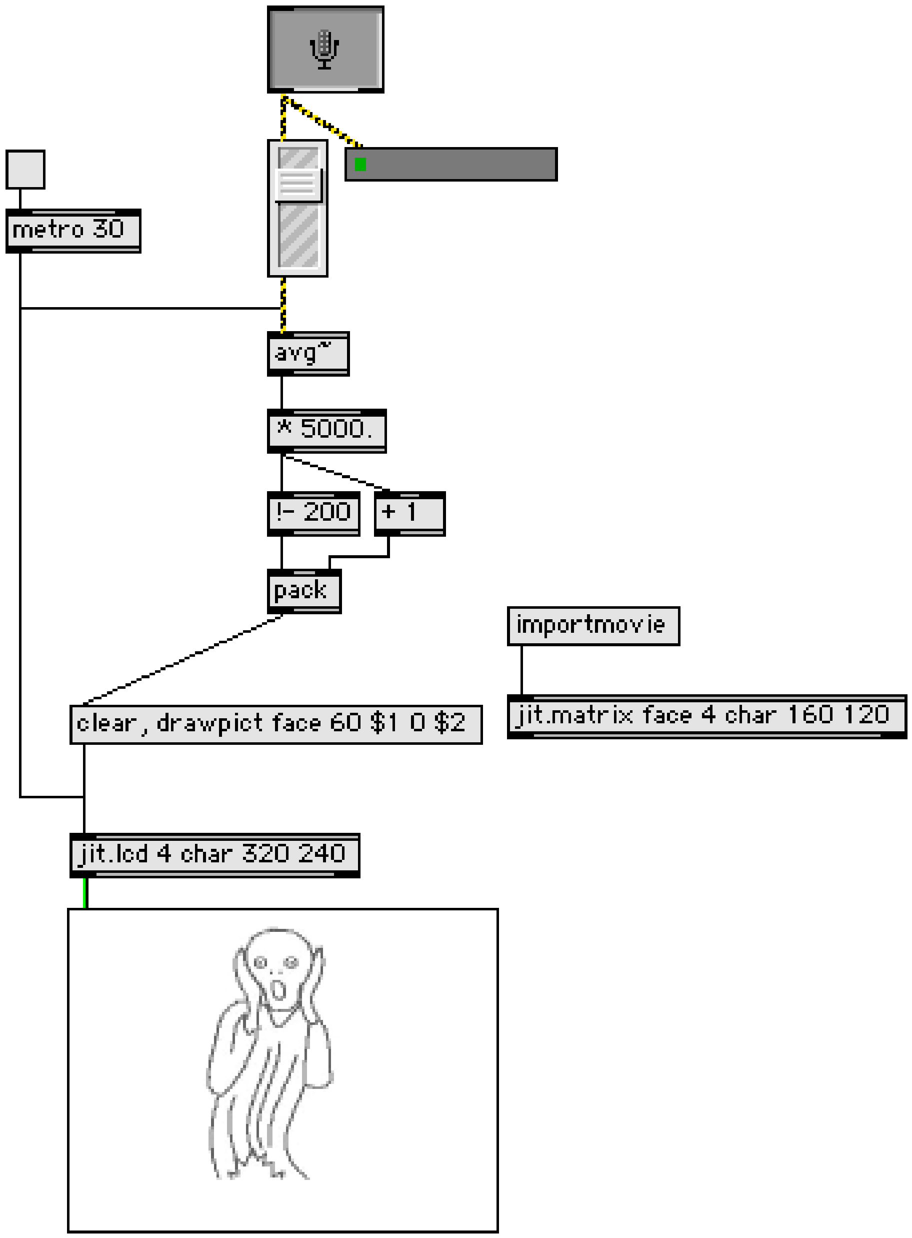


Figure 1. Amplitude controlling image size

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.

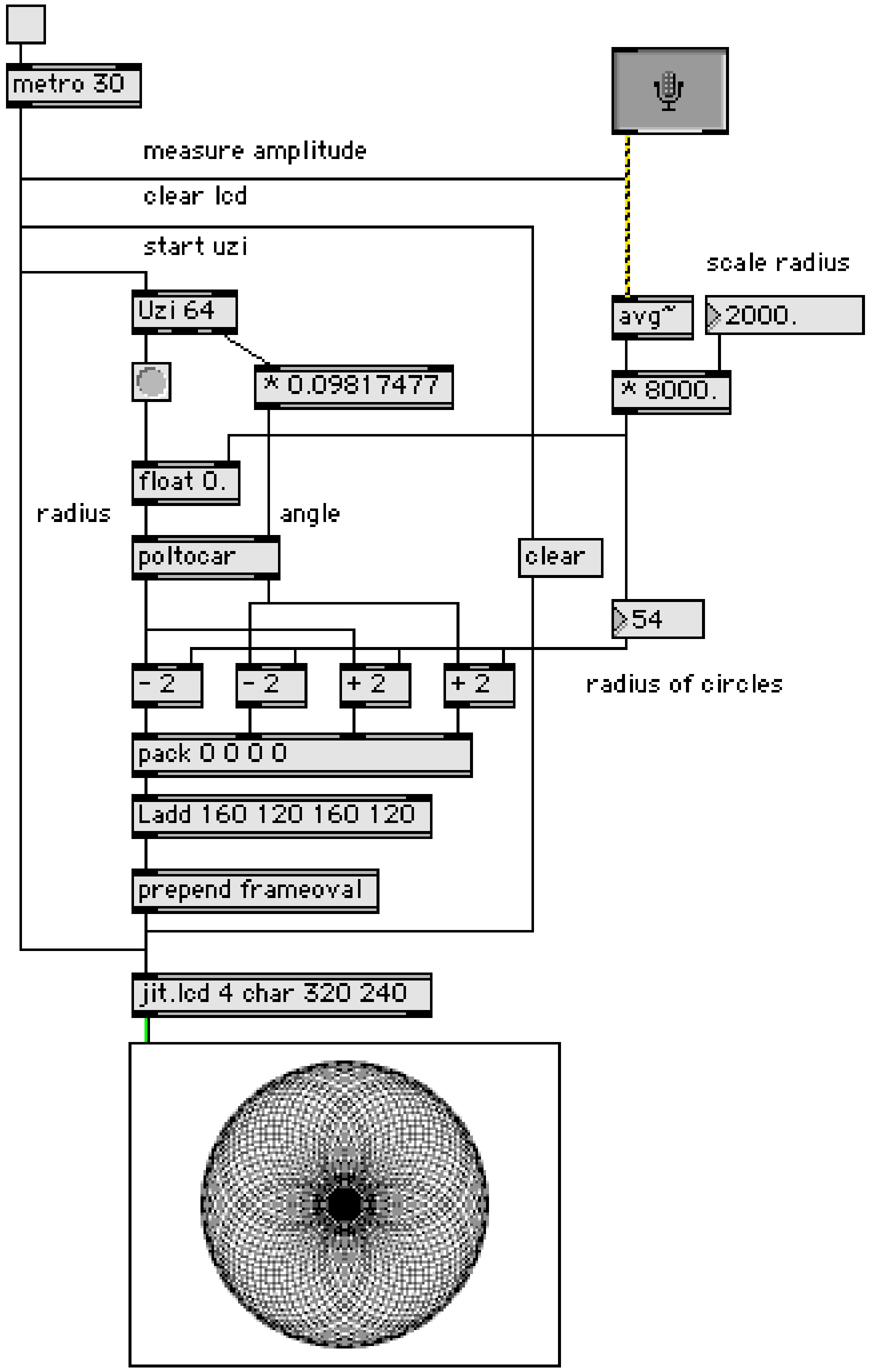


Figure 2.

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:

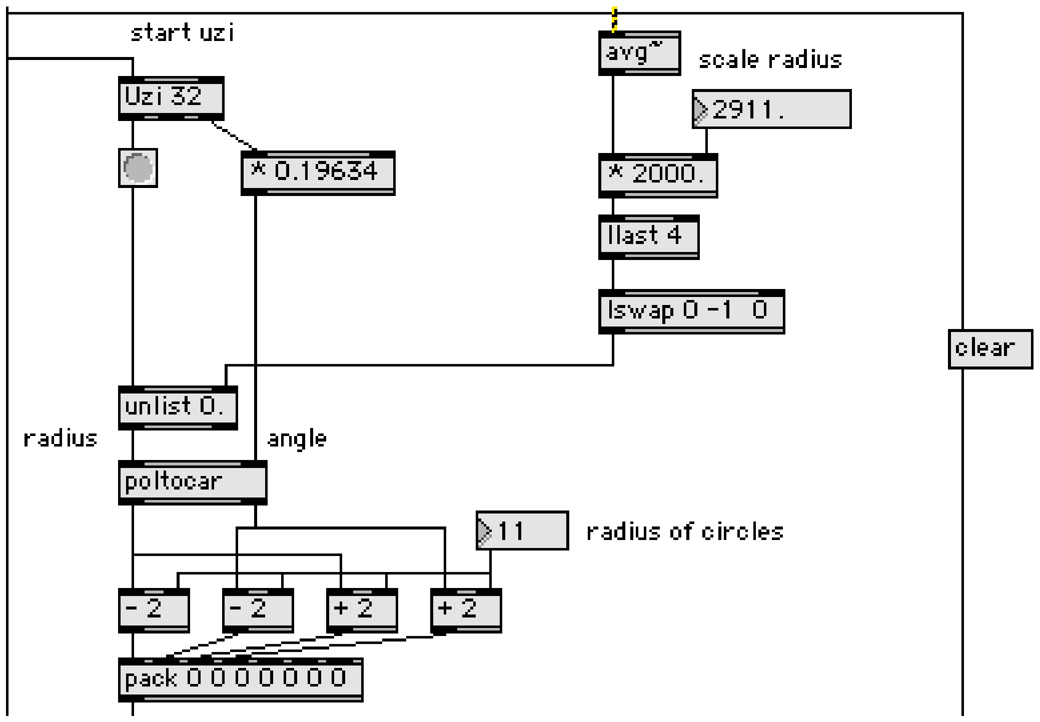
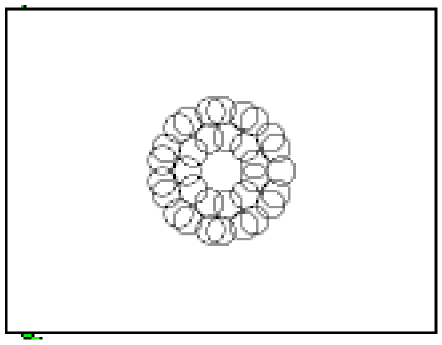
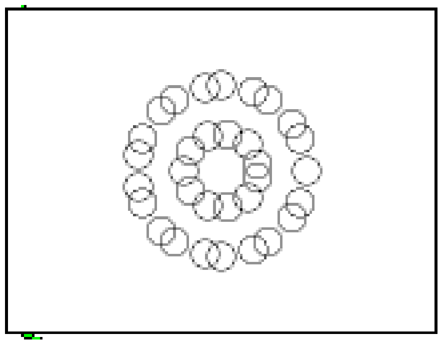


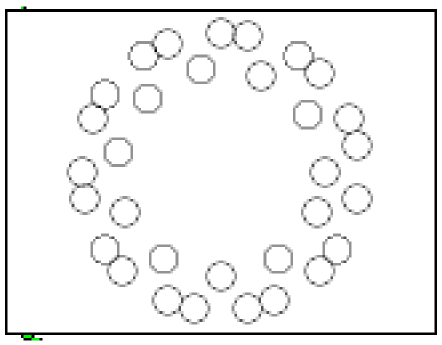
Figure 3.

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.

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


A


B


C

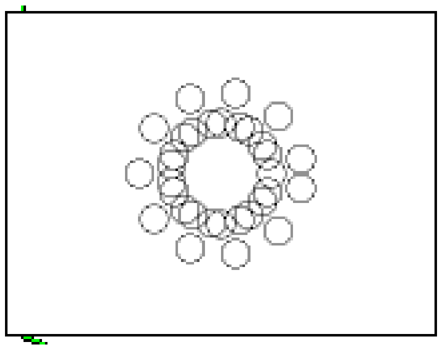
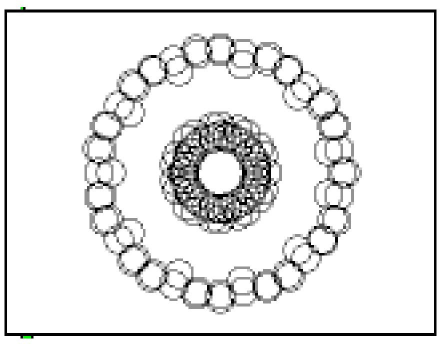

D

Figure 4.
These also appear to spin, because 7 measurements are used to produce the 32 circles. If you look at 5C, 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.



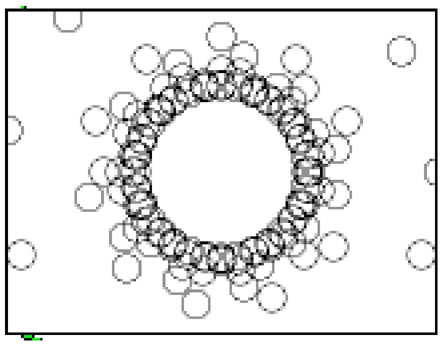


Figure 5.

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.

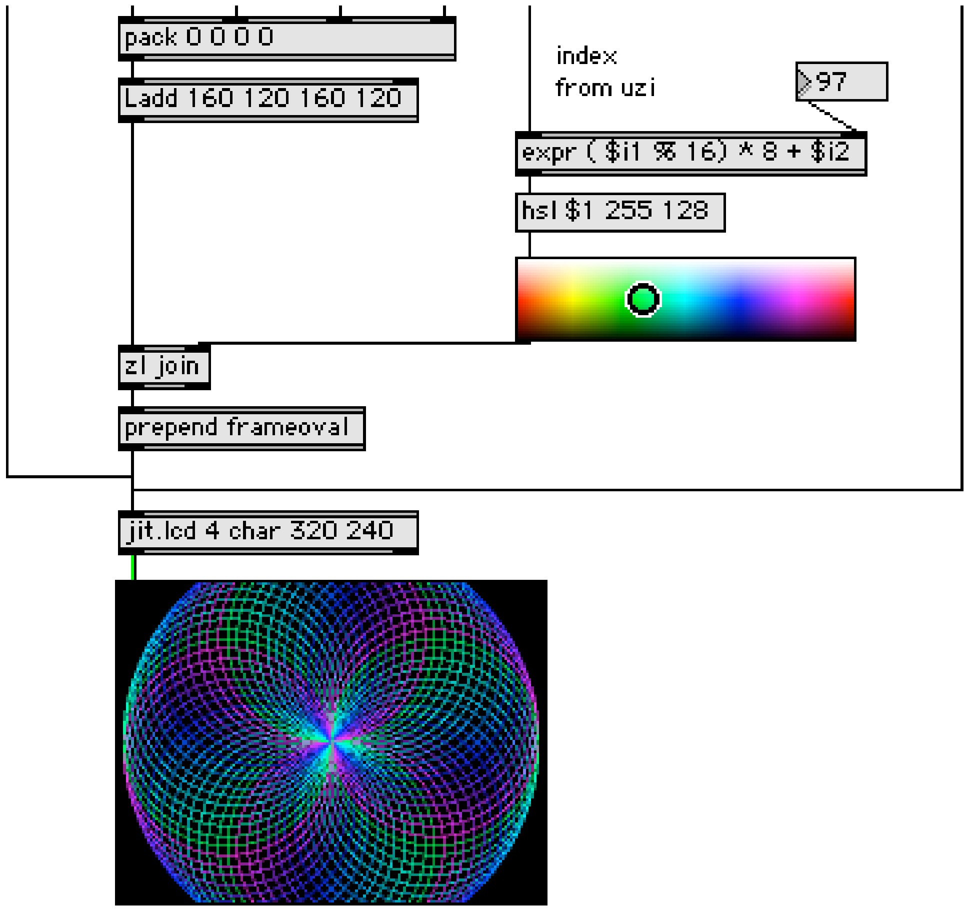
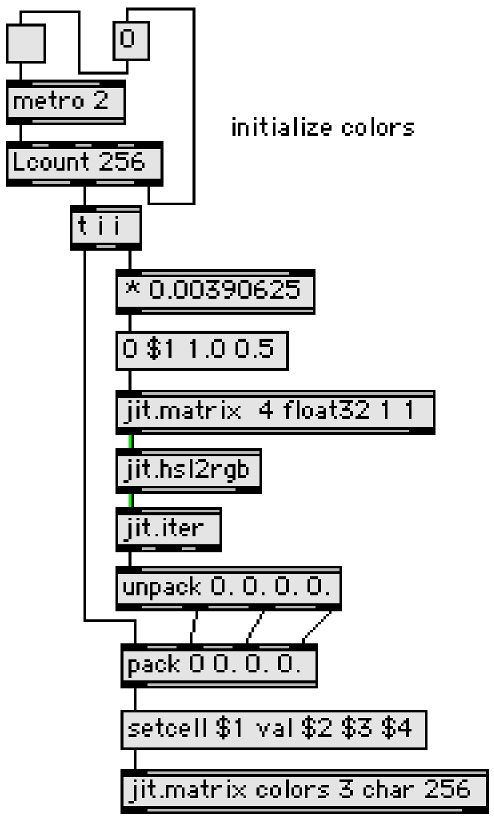


Figure 6. Adding color to jit.lcd drawing

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.

It's not a good idea to include graphic UI objects in jitter intensive patches if it can be helped. Even if the object is in a closed window, it must be polled for possible updates which takes a few clock cycles that may be needed elsewhere. If we accept the same restriction used here (leaving saturation at 255 and luminance at 128) we can reduce the process to a single matrix lookup:



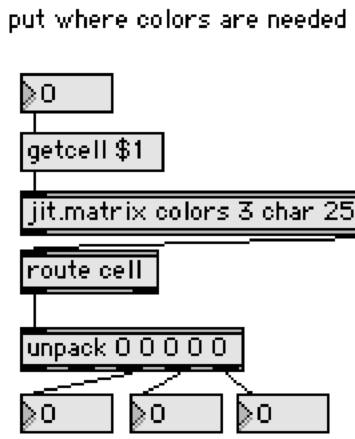


Figure 7.

The left side of figure 7 is in the master patch, and initializes the matrix colors with the same values found in swatch. All that is needed to extract colors is the operation on the right, which will be quite fast. (jit.hsl2rgb uses float values, so 1.0 is equivalent to 255 in swatch.)

Mapping parameters like pitch to color is a philosophical point that has been debated for ages. One common choice is to use blue for low values and red for high ones (scientifically backwards, but emotionally satisfying). We can do that easily enough by manipulating the pointer into the colors matrix, or we can design out own color space. To understand how this is done, consider the color transition tables for swatch:

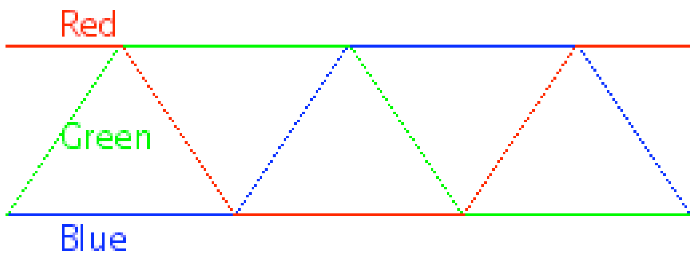


Figure 8.

These three curves represent the values for red, green and blue as hue is increased from 0 to 255. There are seven[2] inflection points, which correspond to the six primary and secondary colors. The top and bottom of the graph are the boundary values of 0 or 255 for full saturation and half level. Other levels of saturation and luminance will change the boundary values—saturation is the distance between boundaries and level is the average of the two.

If we want to redesign the color space, all we need to do is move the corner points around.

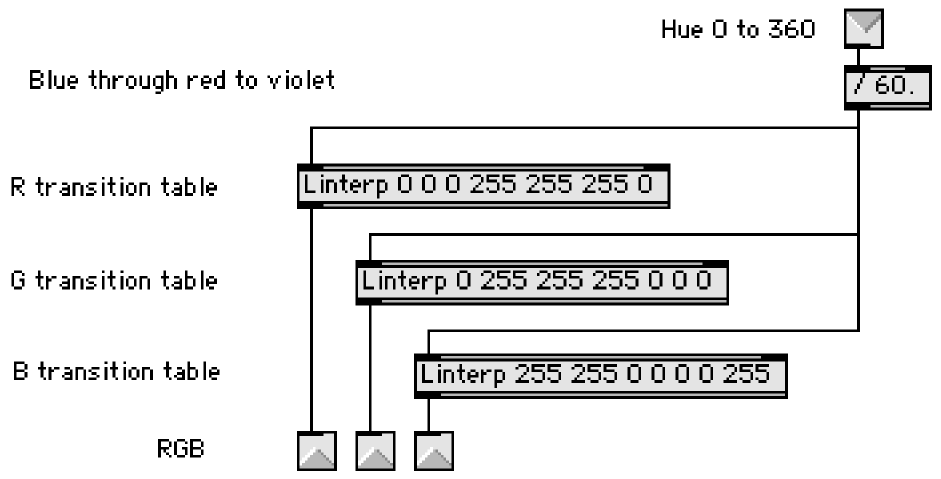

Hue 0 to 360
/ 60.
Blue through red to violet
R transition table
Linterp 0 0 0 255 255 255 0
G transition table
Linterp 0 255 255 255 0 0 0
B transition table
Linterp 255 255 0 0 0 0 255
RGB

Figure 9.

In figure 9, hue (expressed as an angle in degrees) is used to read values from Linterp objects for red, green, and blue. The numbers have been reversed and rotated so the gamut will run from blue to violet. If we use a higher low boundary, the colors will become pastels, and if we lower the top the colors will be darker.

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.

[2] Six really, since it should be circlular.

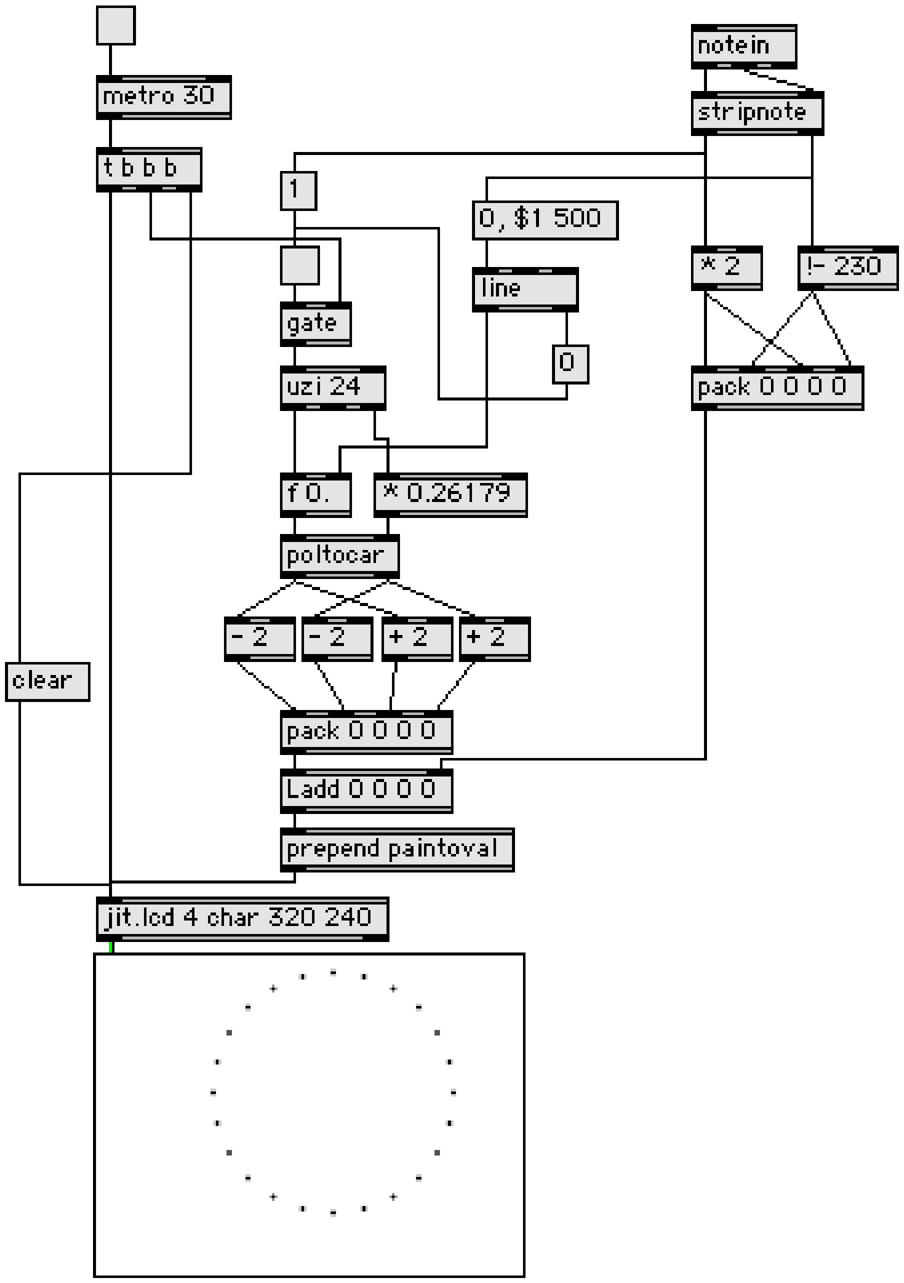


Figure 10.

This has many elements from figure 2. Drawing is triggered by the notein object. 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 turned on by receipt of a note and turned off when the line hits its end. 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 11.

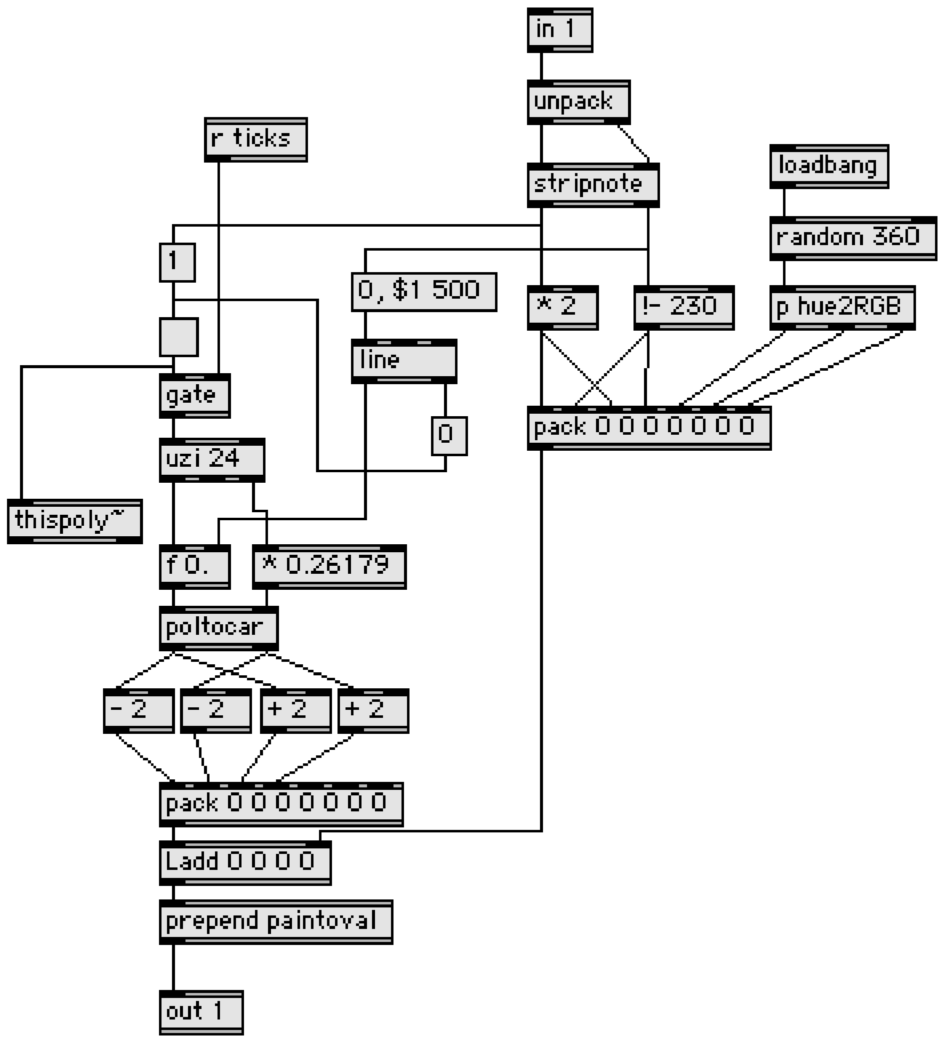


Figure 11.

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 toggle used to gate the ticks to uzi can also serve to provide busy status to thispoly~ .

Figure 12 shows the enclosing patch. There's very little change here. The midinote message to poly~ will trigger drawing from the first available subpatch.

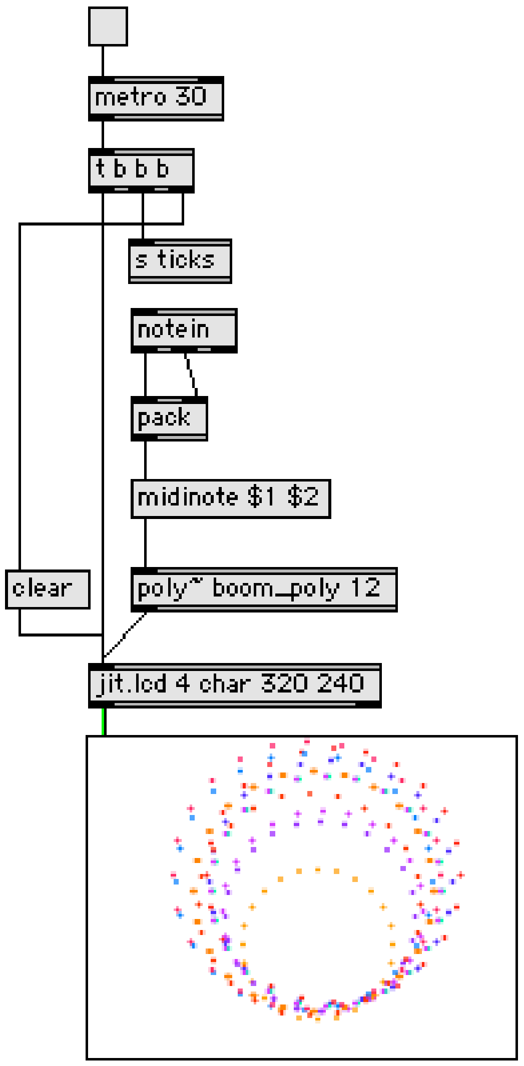

metro 30
t b b b
s ticks
notein
pack
midinote $1 $2
clear
poly~ boom_poly 12
jit.lcd 4 char 320 240

Figure 12.

Drawing Waveforms

To show a waveform in a jitter window, we need a simple recording setup:

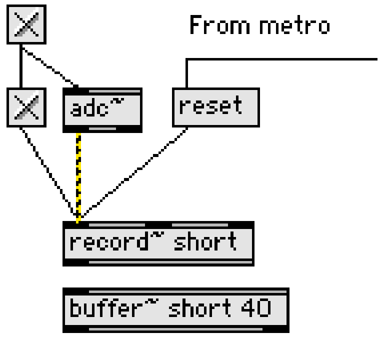


Figure 13.

The buffer named short will get 40 ms worth of signal every time a bang arrives from some distant metro. The metro will be driving the jitter display:

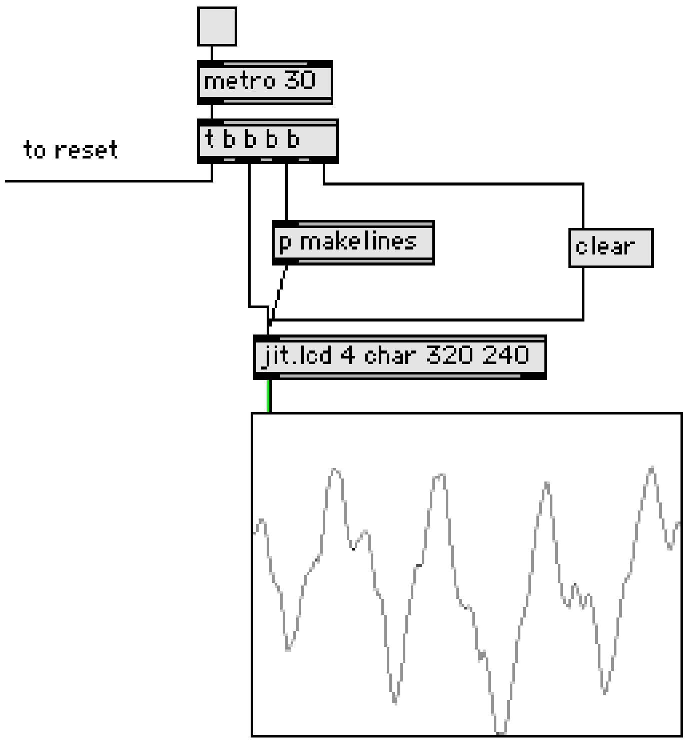


Figure 14.

There are four distinct actions triggered by the metro in figure 14. First jit.lcd is cleared. Then the contents of the makelines subpatcher (figure 15.) come into play. The uzi looks (via peek~) at the first 360 samples in the buffer~ named short. These will range from –1.0 to 1.0, so they are multiplied by 120.0 and added to 120 to fill the display. The index of each sample and the modified value will be the endpoint of a lineto command.

Note that the command is changed to moveto for the first sample. This suppresses a line from the last sample on the right to the first on the left.

Once all of the drawing is done, the jit.lcd is banged to show up in the window, and then the record~ is reset to grab the next chunk of waveform.

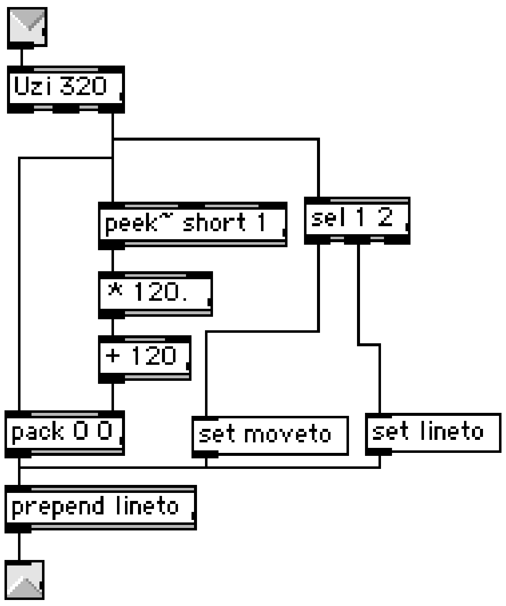


Figure 15.

A plain waveform is not really the most interesting thing to watch for long periods of time. Some simple manipulations of the image are more artistically engaging. Figure 16 shows a version of makelines that draws the wave in a solid form:

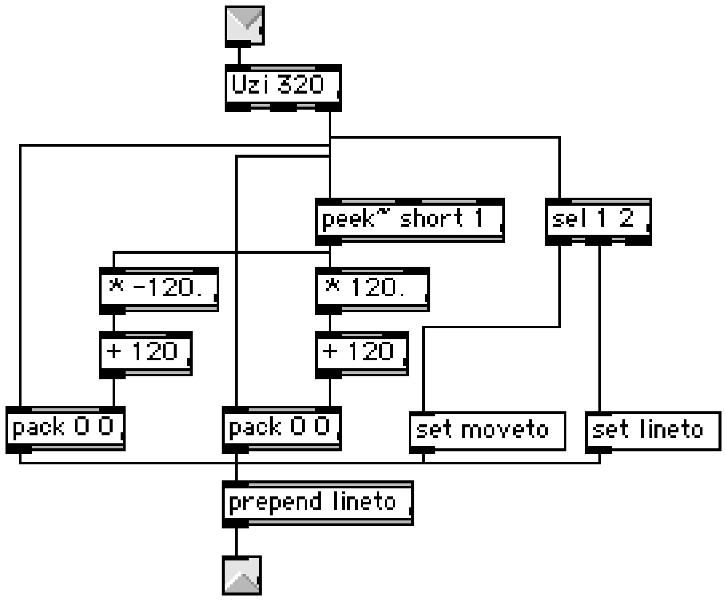


Figure 16A.

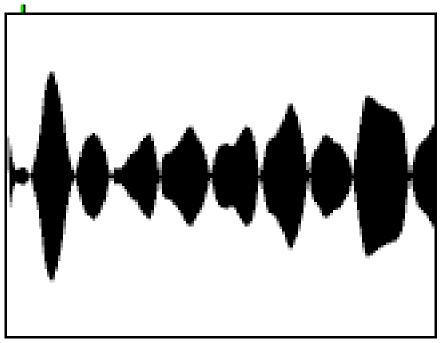


Figure 16B.

This can be made symmetrical by changing the way the uzi scans.

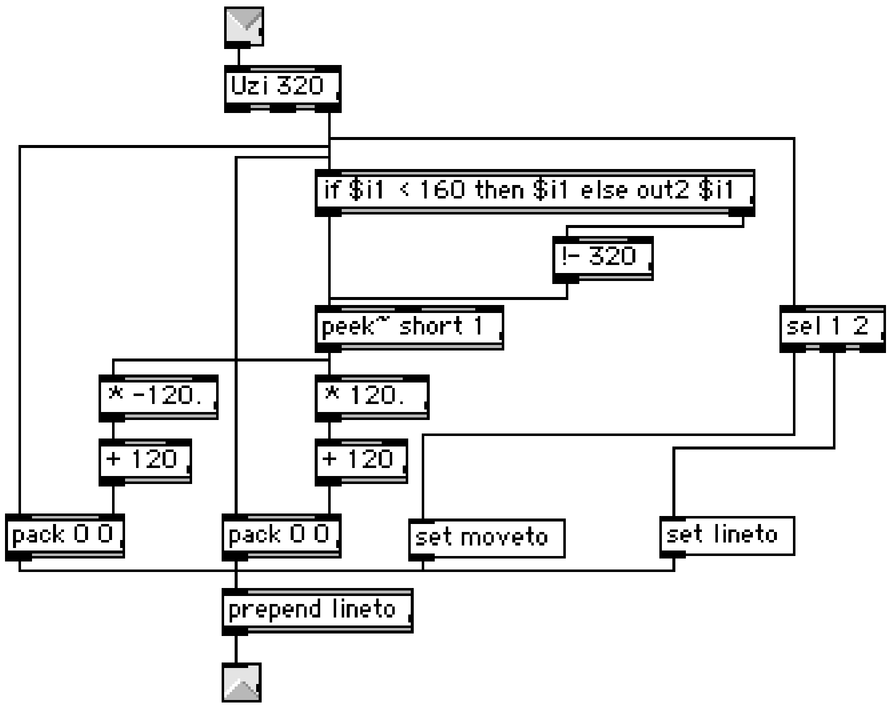


Figure 17A.

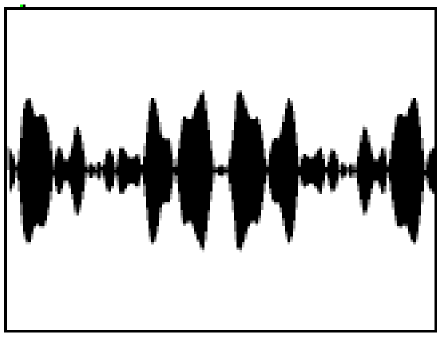


Figure 17B.

Figure 18 shows how to add a bit of color. The hue2RGB subpatcher is the same as figure 9, with a pack on the outputs.

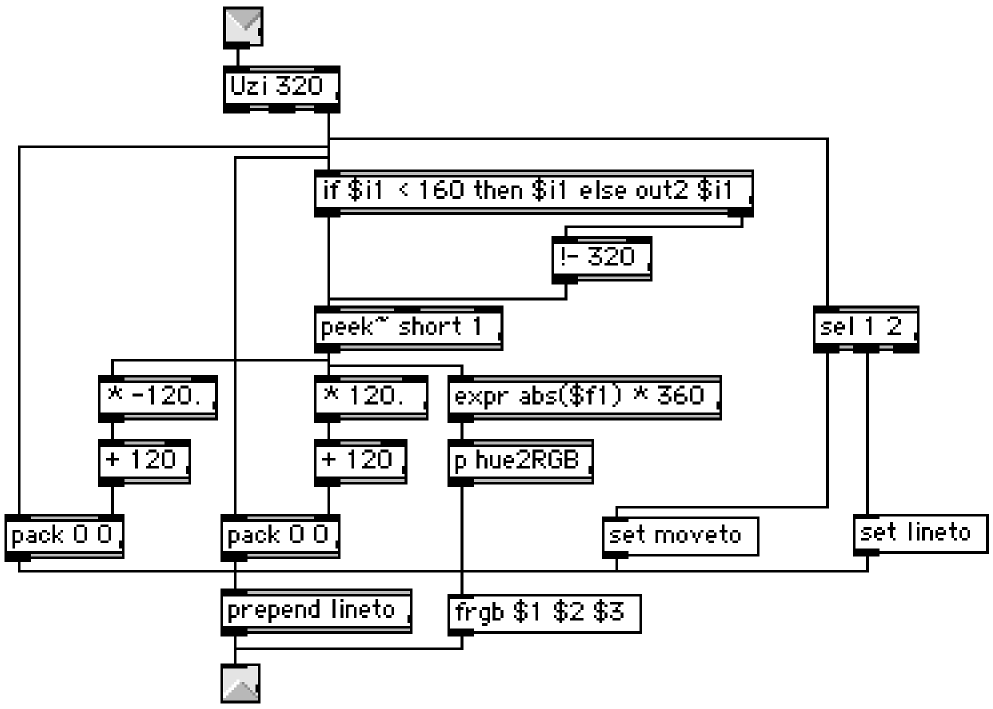


Figure 18A

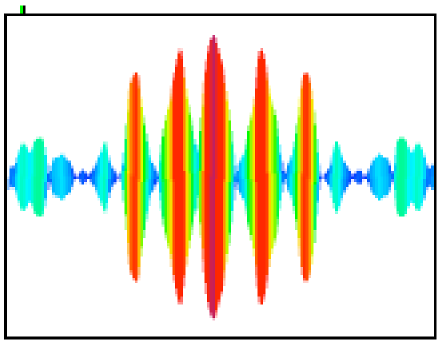


Figure 18B

Polar waves

Polar display of waveforms provides an interesting alternative to the oscilloscope style. The Makelines subpatcher is modified so index from the uzi is converted to radians. This will provide the angle for a cartopol object and the value from peek~ will be modified to give a radius.

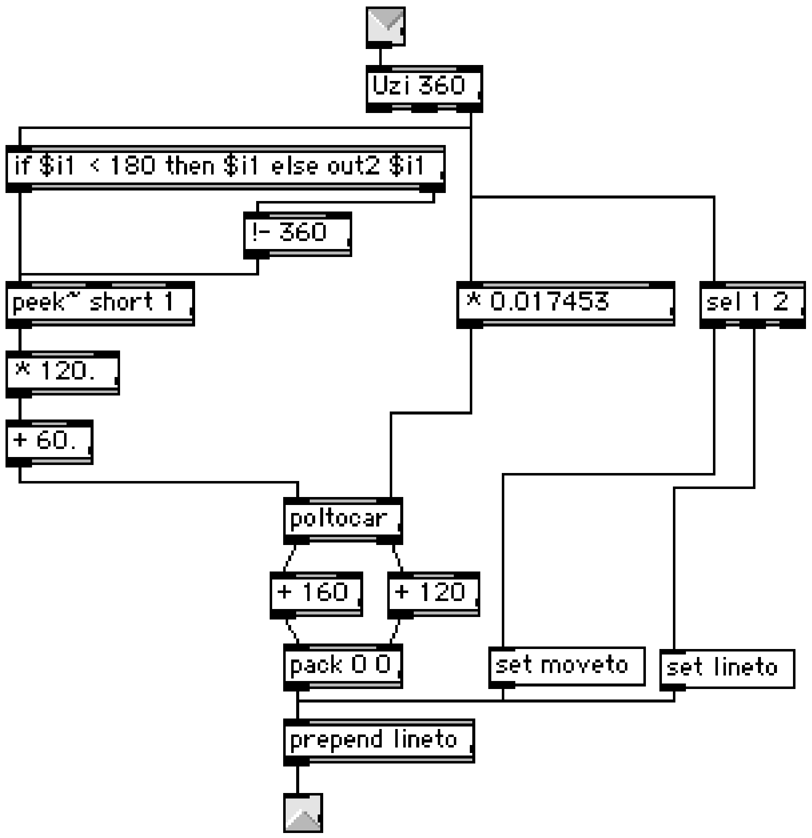


Figure 19
Since the peeked values have 60 added, the basic line with no signal is a circle. When audio comes in we get rapidly changing butterfly shapes.

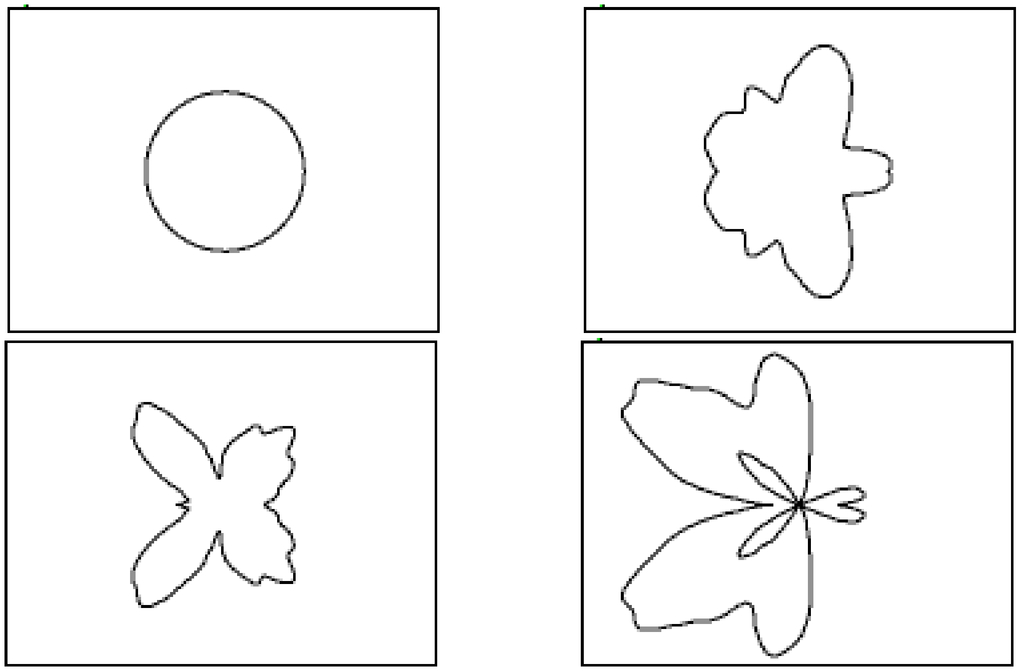


Figure 20

Some additions will turn this into a solid shape with colors

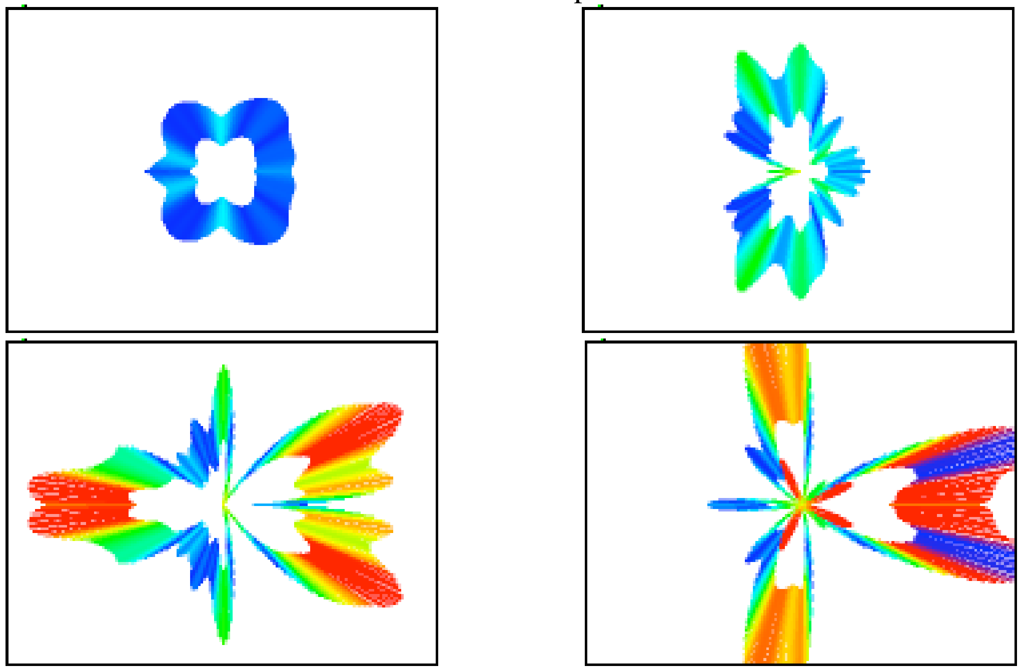


Figure 21

Here’s the modified makelines patch:

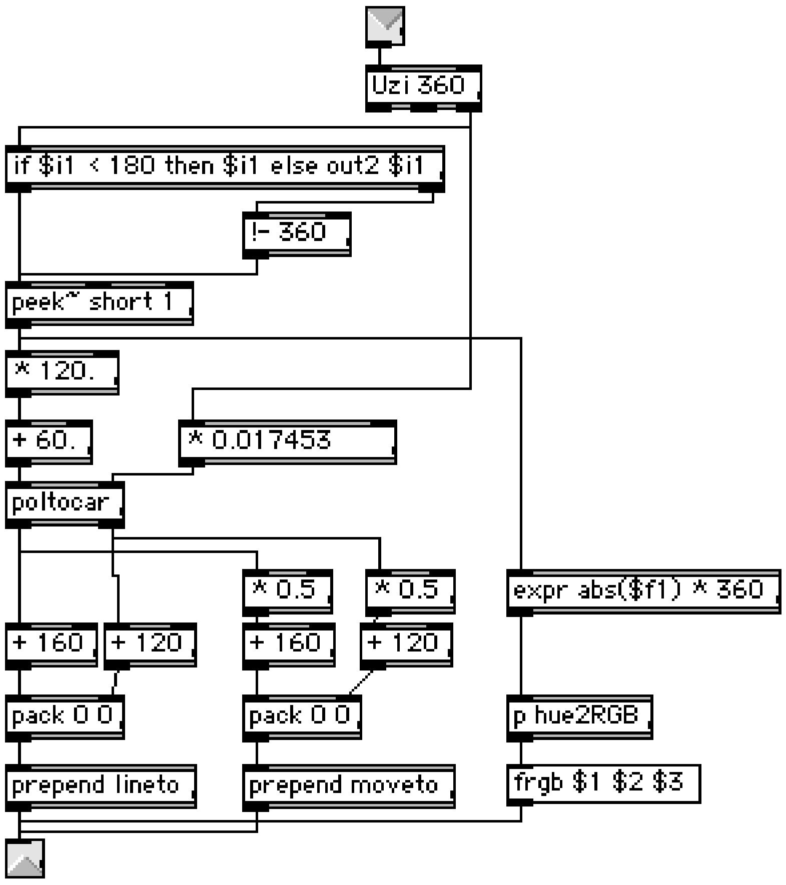


Figure 22.

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 first modify the recording patch to do stereo:

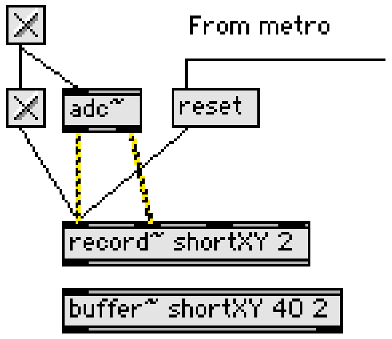


Figure 23.

We also change the display calculator to use the left channel instead of the uzi index for the X value.

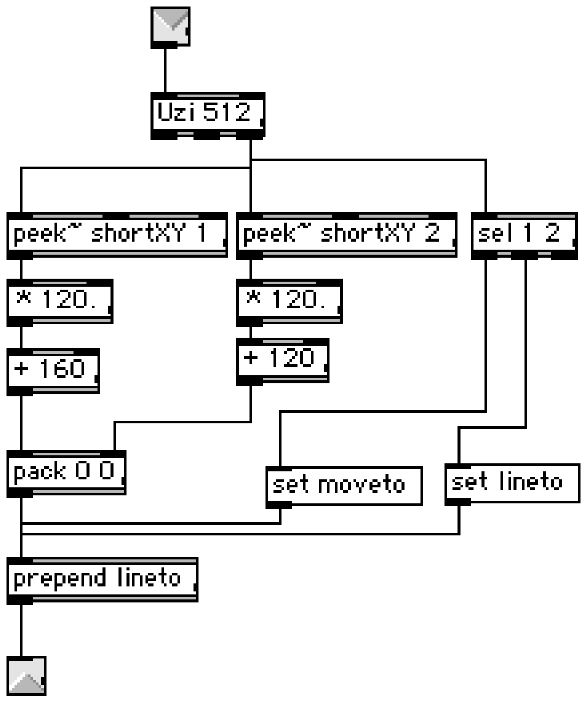


Figure 24.

Initial results are likely to be disappointing, as figure 25 suggests.

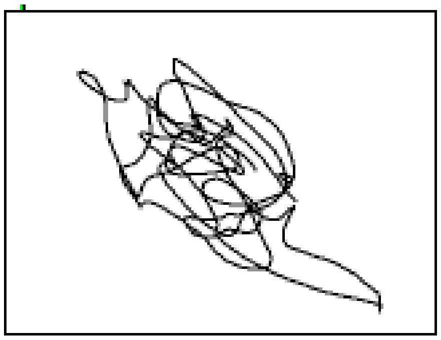


Figure 25.
This is the basic scribble. It's quite dynamic, and will fill the screen sometimes, but with most material will stick close to the diagonal, and ultimately becomes tedious. It can be helped with coloring techniques and various jitter processes.

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.

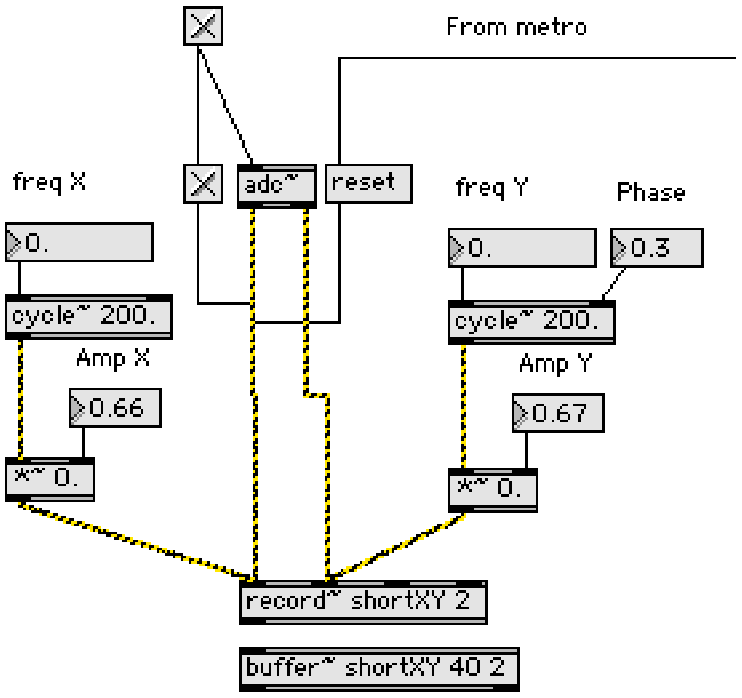


Figure 26.

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.

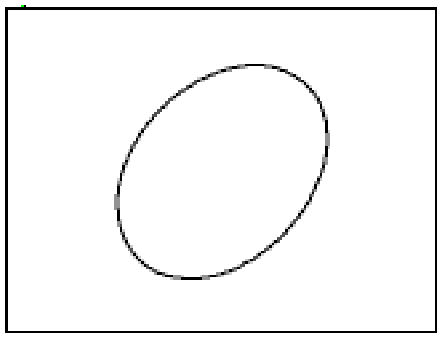
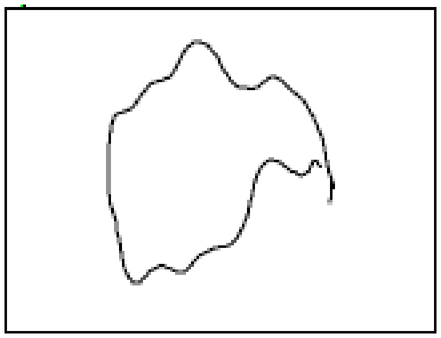
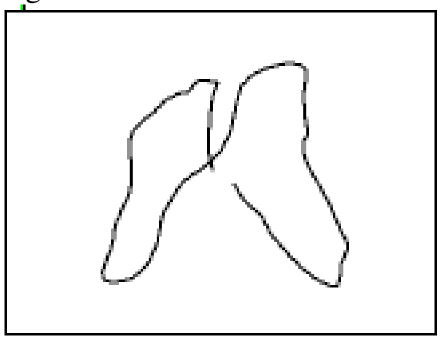
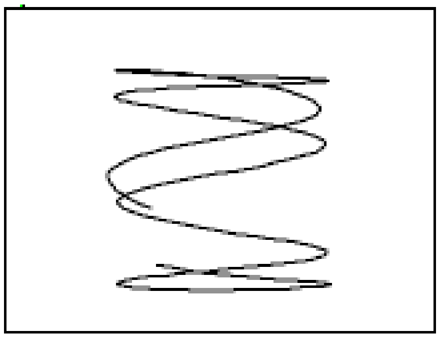


Figure 27A 1:1 with no audio


27B 1:1 , audio added to Y


27C 1:2 with audio added to Y


27D 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.

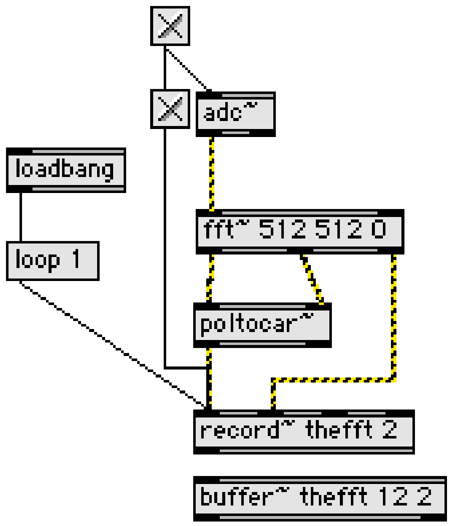


Figure 28.

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.

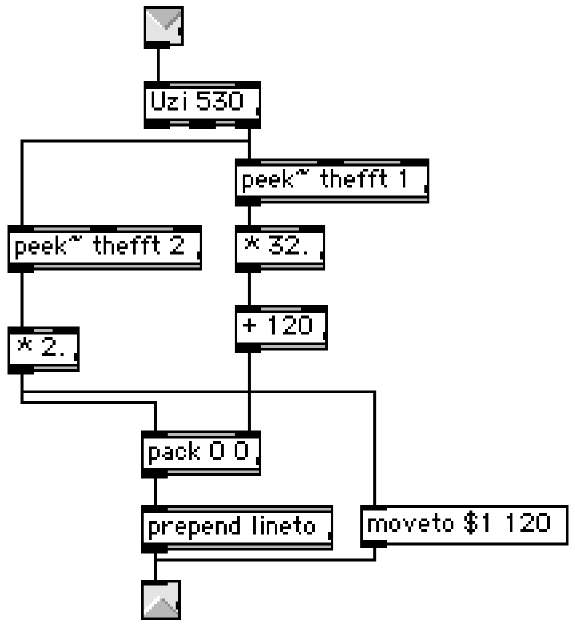


Figure 29.
A couple of things should be mentioned here. There are 530 samples in a 12 ms buffer~, and the Uzi must look at all of them to get a complete display. In figure 29, the index is multiplied by two (with the pensize for jit.lcd is set to 2x2) to spread the shape out a bit. To see why this is desirable, look at figure 30 which shows the entire thing.

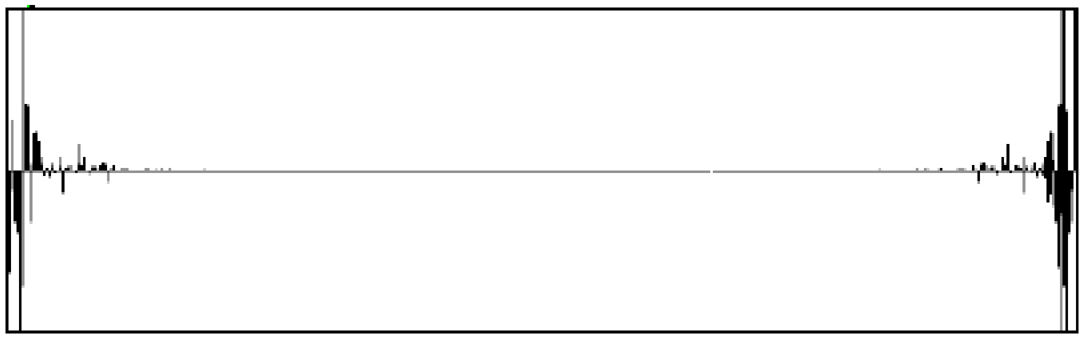


Figure 30
You can see there's a little bit of action at each end, but the middle is a wasteland. Everything above bin number 60 or so is always so close to 0 that it won't show, and bins above 256 are a reflection of the first half of the fft.[3]

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

Figure 31 shows the doctored version:

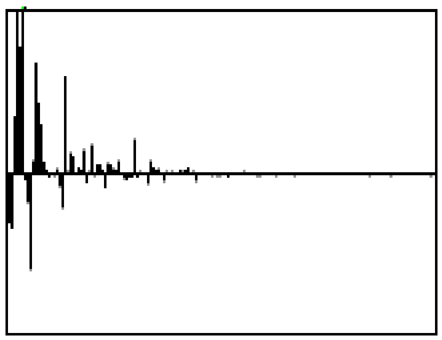
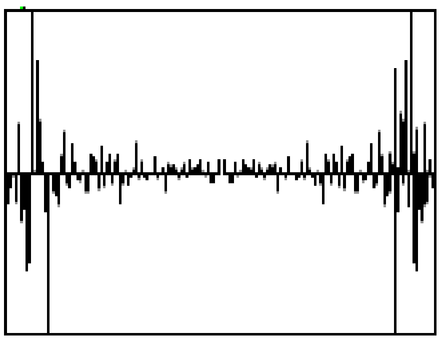



Figure 31A 31B

Version A is produced by the patcher shown in figure 29. Version B displays both ends of the fft chopping out the middle. Figure 32 does this.

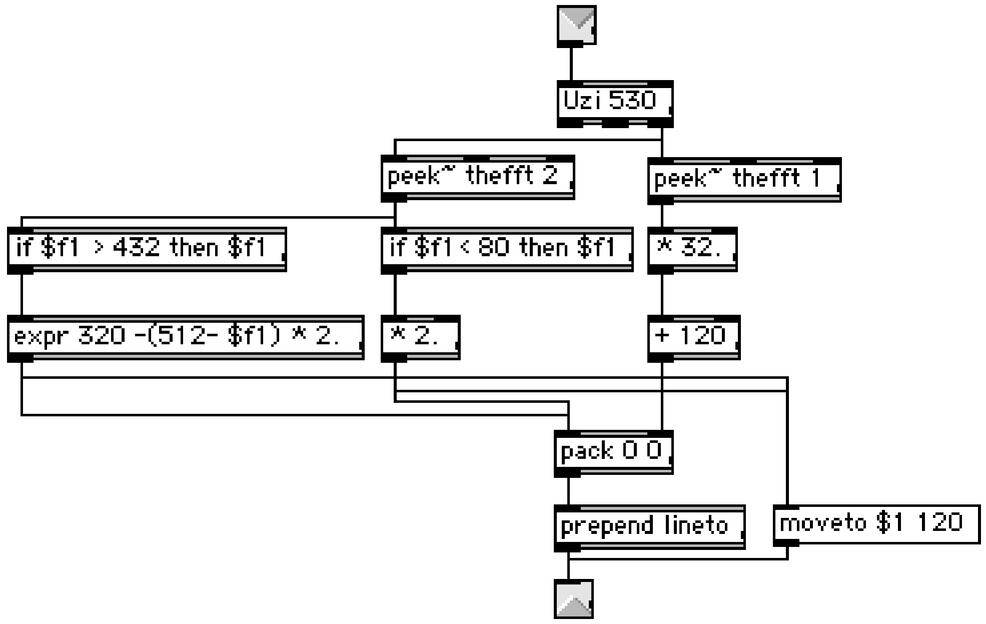


Figure 32.

A radial display of the fft can be interesting:

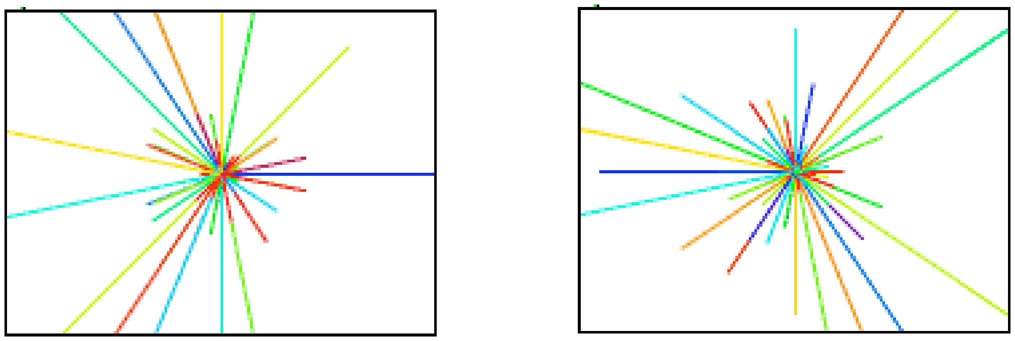


Figure 33

Here is the patch that makes it happen:

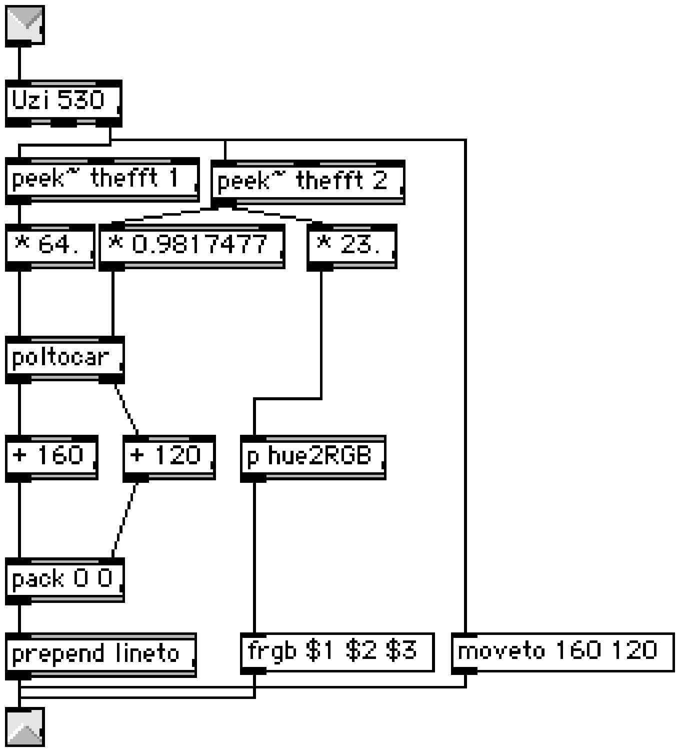


Figure 34
The fft index is multiplied by 0.98171 which divides the circle into 64 spokes. This means 8 different points of the fft will be combined on each spoke. The reflection points will give the image a bit of symmetry.

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:

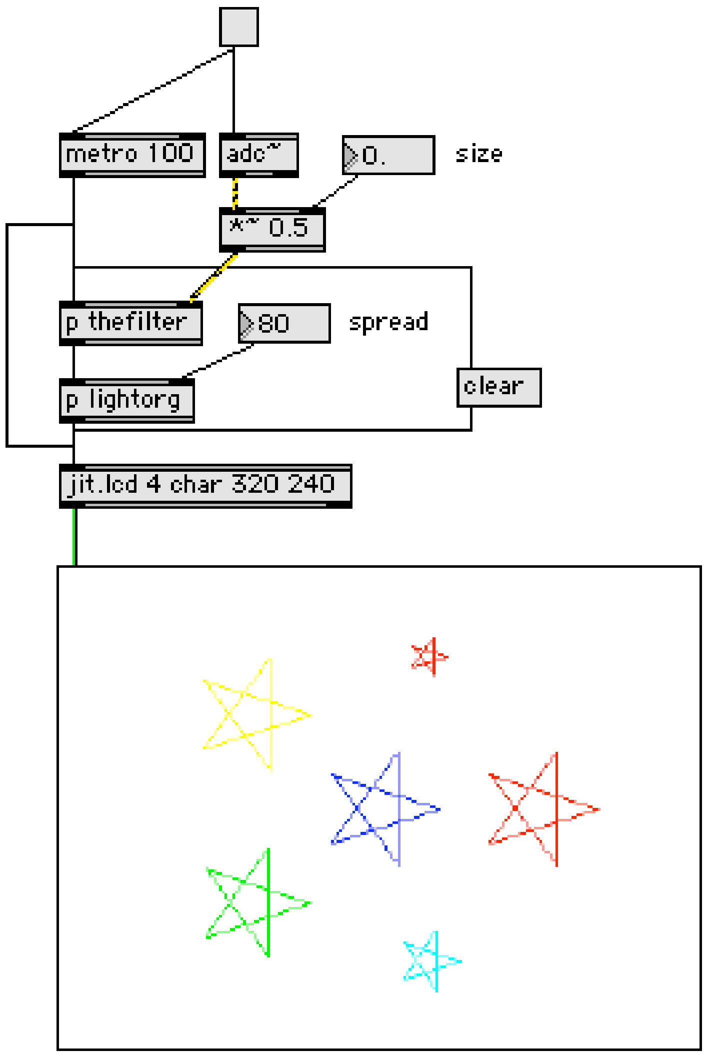


Figure 35.

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:

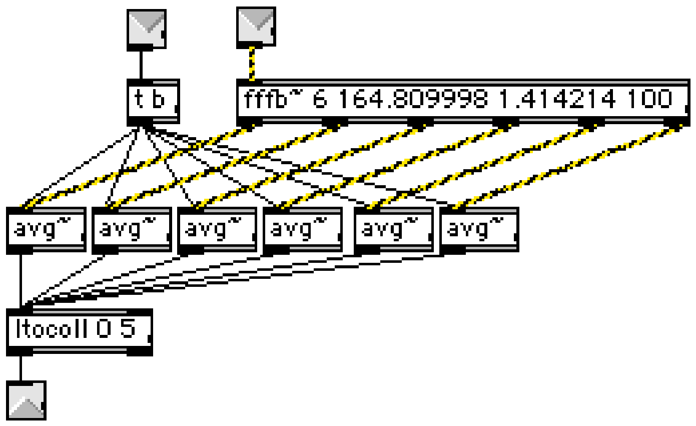
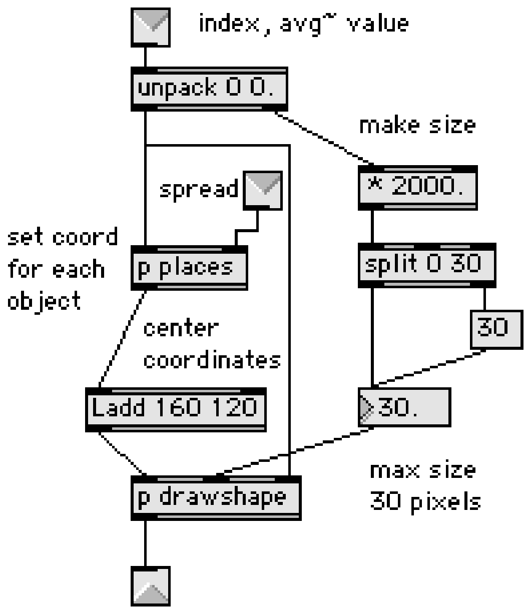


Figure 36.
The fffb~ object is working as a half octave filter in this example. The output of each filter section is averaged over the frame period to give a value that will determine the size of the associated object. The filter can have many more bands. I usually have 18. Ltocoll will pack each value into a list after an index number that identifies the filter section. These two item lists are used by the light organ subpatcher.

The light organ section contains a subpatch called places :



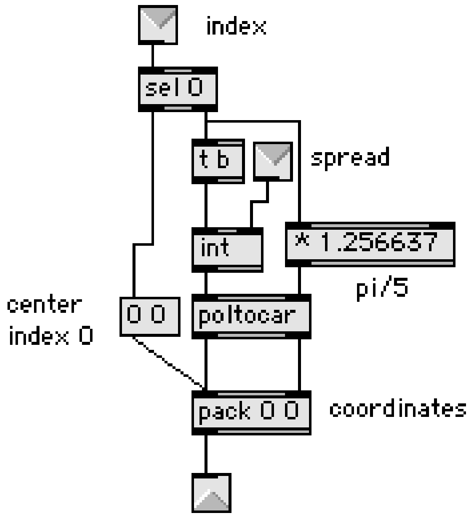


Figure 37   Lightorg   Places subpatcher

This all calculates a center point and size for each object. The object is constructed in the drawshapes subpatch, which could hold the code to draw anything. In this case, star shapes are drawn by the patch in figure 38.

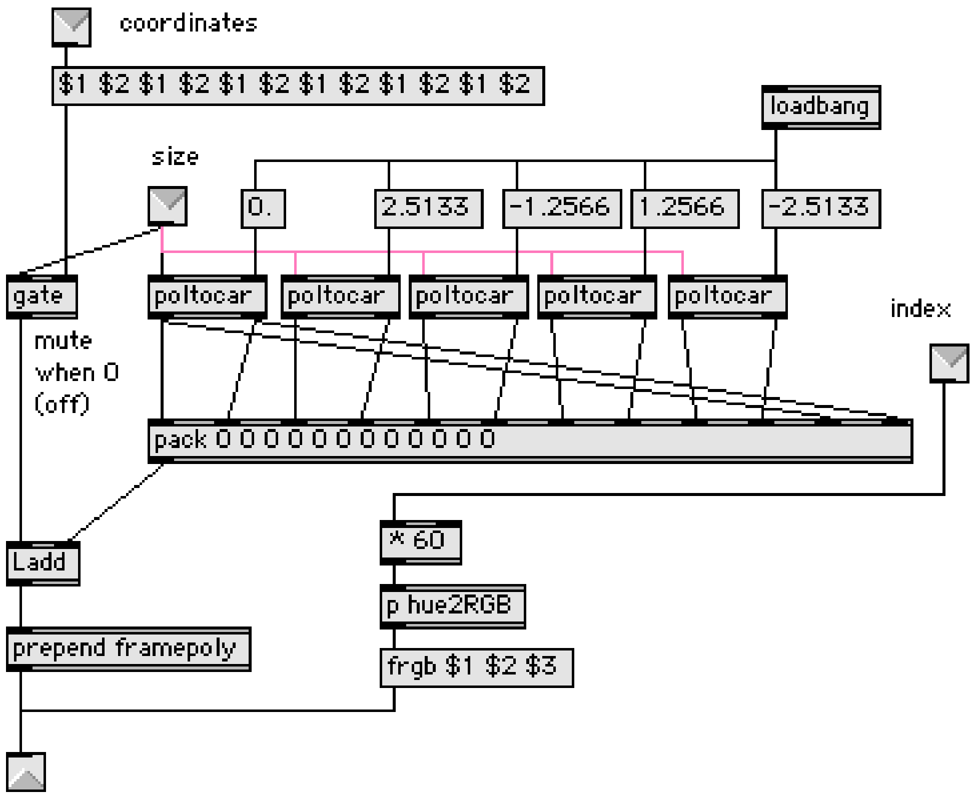


Figure 38.

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.) This index will be4 the next arrive, and is used to pick a color. 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 the 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.