Basic Drawing in Max

Max has had drawing features from the first commercial version, but these were perfunctory until the advent of Jitter in 2003. Once Jitter appeared, artists became interested, and we are starting to see some extraordinary work. Even so, drawing in Max is primitive. The Illustrator pen, Photoshop plugins, and the transformations of Studio Artist are all better at their function than anything available in Max. Art can be output as QuickTime movies or JPEGs, but there is no direct to web connection. So what does Max offer? Graphics in Max are computed at the frame rate according to your instructions, so they can be dynamic, interactive and algorithmic. Further, the output of those other applications can be brought into Max for real time transformations and augmentation.

The computation and patching necessary to produce 2D art are done with basic Max objects. For a quick introduction to Max, see my essay "Max Intro". Then dive into the tutorials found in the max help menu. Max was originally a music program, and the tutorials reflect that fact. Nonetheless, the tutorials are the best way to learn Max, so start with them right away. This essay will just show patches-- the tutorials show how to build them.

Three vital pieces of Jitter

Jitter is the graphical face of Max. It is primarily designed as a video system, so there's a fair amount to ignore for now (if you want to jump in at the video end of the pool, look at my essay titled "Pre-Jitter Studies"). Jitter manipulates images at the pixel level, so some understanding of how screen images are organized is required. If you look closely at a screen or projection, you notice the image is made of dots. Each dot is controlled by three numbers that set the intensity of Red, Green, Blue which combine to produce the color you see. These are 8 bit numbers. Each has a range of 0 to 255, for a total of 16,581,375 possible colors. A typical screen is 1024 by 768, so there are 786,432 pixels, at least. That's a total of 2,359,296 numbers blasted at the screen 60 times a second. Special hardware in the display card (or chip) manages the brute force operations, but the computer has to prepare these numbers so the CPU is working pretty hard too.

Jitter manages images in a data structure called a matrix, which is a collection of cells, one per pixel. Each cell contains the values for R G B with a fourth value to control opacity when images are combined. The forth value is called "alpha", and they are listed in the order ARGB. (Most literature calls alpha "transparency", but an image with an alpha of 0 is invisible.)

The matrix stores images

Image matrices are stored in an object called jit.matrix. (Every jitter specific object is jit dot something.) jit.matrix objects can store other data too, so you have to specify image data by giving the object the arguments *4 char*. These are followed by two more arguments for width and height of image. We'll start out with small images, as illustrated in figure 1.

Figure 1.

Qmetro moves images

The qmetro[1] object produces bangs at regular intervals. These bangs prompt the jit.matrix to produce the image it is holding. The qmetro isn't strictly necessary here (one bang would do) but I always use it to make updates automatic. The importmovie command instructs jit.matrix to load an image file. You can load practically any type of image, not just movies (you only see one frame of a movie.) If the image size does not match the matrix size, jit.matrix will decimate or interpolate to fit the whole image in.

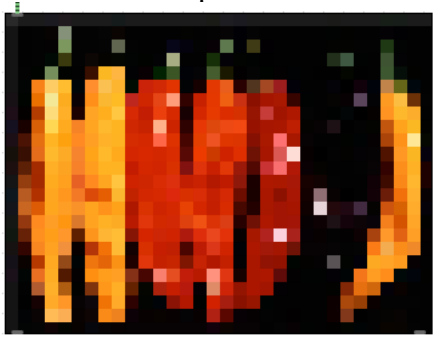

Figure 2. Chilis in a 32 by 24 matrix.

[1] Qmetro does exactly the same thing metro does. Qmetro is more polite about it. Max will freeze up if you ask for too many operations between metro bangs, but qmetro will slow down if too much is going on.

The pwindow Displays Images


The Kitten icon in the object browser will expand to a jit.pwindow. This will show the contents of a 2 dimensional matrix in a patcher. (To show an image in an independent window, use jit.window.) You can resize a pwindow with the mouse, but it is best to open the inspector and set the size exactly. The pwindow will interpolate any matrix to fill the display space, so it should match the image size if possible. The pwindow can do much more than show flat images. For instance, mouse actions can be detected when the cursor is over the pwindow.

Basic Drawing

The drawing canvas is an object called jit.lcd. This is a descendent of a more primitive drawing object called lcd. Lcd is still available[2], but all it can do is show drawings in a patcher. It does not produce matrices. The drawing routines in jit.lcd are based on the original Macintosh graphic routines called QuickDraw.

The drawing space is made up of numbered pixels. Each pixel has two numbers, the first is horizontal location, the second is vertical. The upper left corner is 0 0, and the pixels across the top are 0 0, 1 0, 2 0, 3 0, 4 0, etc. The right edge is 0 0, 0 1, 0 2, and so forth. This is similar to the familiar Cartesian coordinates except that y increases going down the screen. Instead of X and Y, I'm going to call the points H V.

Technically speaking the point H V refers to the upper left corner of the pixel. Thus, when you draw a line from 0 5 to 10 5, and another from 10 5 to 20 5, they will butt against each other instead of overlapping. Some commands draw inside of lines, so you occasionally find a 1 pixel difference on the right and bottom edges.

Drawing is done by commands with arguments that specify H and V. These commands set the color of appropriate pixels. Drawing takes place within the jit.lcd, and is not visible until a bang sends the image out for display or further processing. The commands are cumulative until a clear command is received. If commands overlap, the most recent is "on top". Commands exist to draw lines, arcs, rectangles, ovals (rectangles and ovals may be either framed or filled), polygons, and clips from files.

Colors

Some of the commands can take colors as arguments. There are two ways to specify color: indexed and rgb.

An rgb color is specified by three numbers, which set the intensity of the red, blue or green component. Possible values for each color range from 0 to 255. 0 0 0 is black, 255

[2] Available, but not supported.

255 255 is white. 255 0 0 is a very intense red, and so on. Yellow is 255 255 0, magenta is 255 0 255, and Cyan is 0 255 255. That's 16 million colors.

Indexed colors use a single number that refers to a set of predefined colors. The set defined is rather strange, being permutations of the values 255, 204, 153, 102, 51, and 0. The system creates groups of six that work progressively from pale green to red in sets of 36 that move toward blue. That gets us up to 214. 215 to 224 are darkening shades of red, 225 to 234 are shades of green, and 235 to 244 are blues. 245 to 255 are darkening shades of gray.

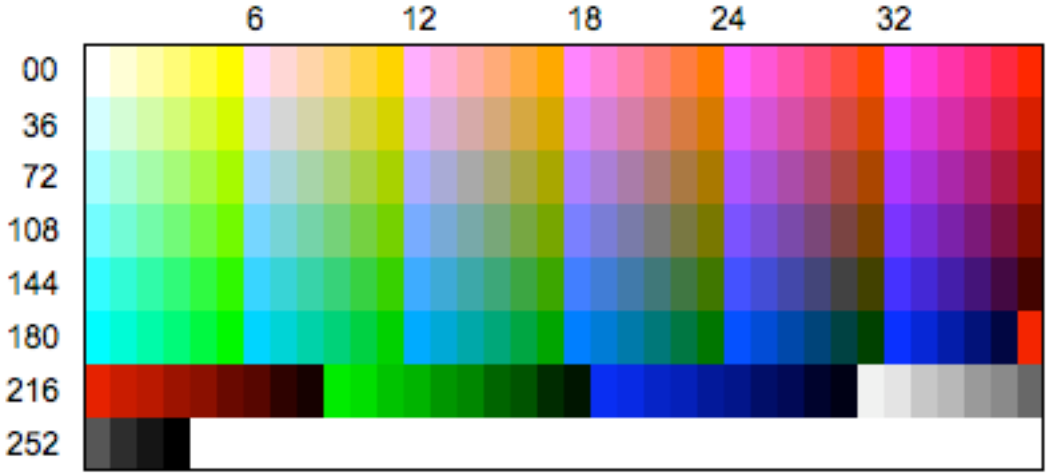

Figure 3.
There is a color designated as the foreground color. This is used by any drawing command that does not specify a color. It defaults to black, and will be changed by any color drawing. Or, it can be explicitly set by the command frgb r g b. There is a color designated as the background color with the command brgb r g b. You won't see it until you issue a clear command. The default is white, but the initial contents of a fresh jit.lcd are 0s, so the image will have a black background until a clear command is received.

The pen

Drawing is done at the location of an invisible pen. To put the pen somewhere, use the command *moveto H V*. The similar command *move H V* is relative to the last location. Thus if the pen is at 15 44, move 5 5 will place it at 20 49.
Getpenloc will tell the position of the pen.

The pen starts out one pixel across. The command pensize H V changes it to a rectangle. This affects line drawing and the various frame commands. If the shape is not square, vertical and horizontal lines will be different widths.

Penmode sets the behavior of the pen in relation to what is already drawn. It determines whether you get the foreground or background color, or something else. Some of these functions use a third color called *opcolor*, which is set by the command *oprgb*.

The modes are called by number, but have names as shown in the help file.
The following table describes what happens with simple shapes, with the results of drawing a pict shown in parentheses. The most common pen mode will be 0 or 4.

Penmodes

0 Copy	You get the foreground
1 Or	You get the foreground of the source and destination
2 Xor	Black if over white space, white if over black (in colors, you get compliments.)
4 Bic	Background color (erases things)
5 NotCopy	Background color (negative of colors of source)
6 NotOr	Doesn't draw at all (negative of pict, destination shows through)
7 NotXor	Doesn't draw (negative of pict, negative of destination)
8 NotBic	Doesn't draw (source & destination where they overlap)
32 Blend	Gives a translucent effect , based on opcolor.
33 AddPin	Adds the source and destination -generally gives stronger colors. Uses OpColor as maximum
34 AddOver	Adds, but with "wraparound" which can be really bizarre.
35 SubPin	Gives the difference, but opcolor is the minimum. Also strange
36 transparent	Will ignore pixels in the source that match the current background. Use this when drawing PICTs if you want the original to show through.
37 Addmax	Will pass the maximum of the source and destination color, which tends toward white.
38 SubOver	Uses the difference between the colors, but if negative wraps around.
39 AdMin	Uses the lesser of the colors.

Lines

The command *lineto H V* draws a line from the pen location to H V.
The command *line H V* draws to a point H pixels right and V pixels down. H and V can be negative numbers to go the other way.
linesegment h1 v1 h2 v2 c draws a line from h1 v1 to h2 v2 with indexed color c.
linesegment is really a moveto and a lineto.

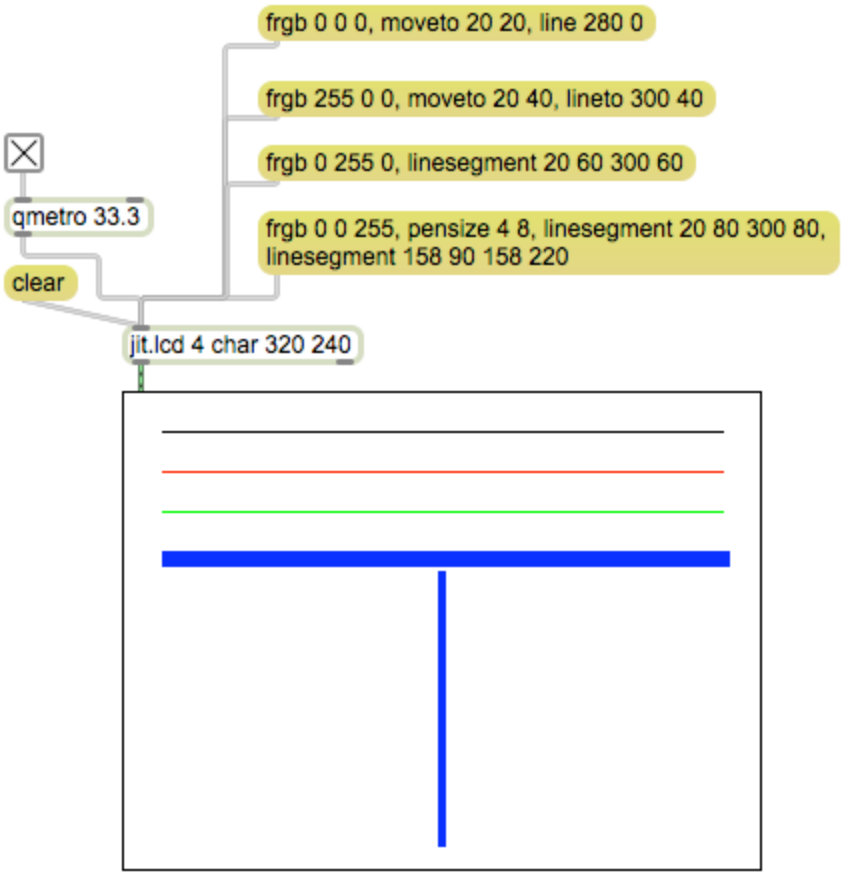

Figure 4.

Shapes

Rectangular and oval shapes can be drawn with a single command. The shapes are fitted into a rectangle you define as H V upper left and H V lower right. The shapes can be painted or framed. Paint gives a solid shape, frame gives the outline.
paintrect
framerect
paintoval
frameoval
paintroundrect (needs 2 more values to specify radius of corners)
frameroundrect

framearc needs two more numbers, which are starting angle and ending angle. These are given in degrees, where 0 is straight up.
setpixel H V R G B sets a single pixel to the specified color

The paint commands paint the area inside the bounding lines, and the frame commands draw the bounding lines, so if you paint on top of a frame with the same numbers, the right and bottom of the frame will still show. Paint commands do not move the drawing pen.

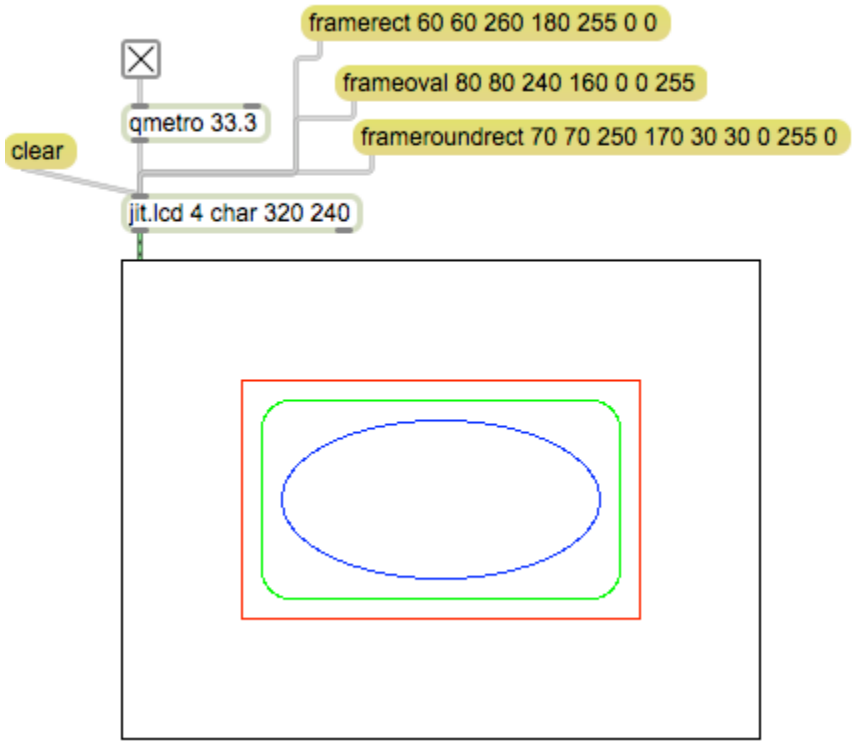

Figure 5.

Poly

Polygons are painted with the paintpoly and framepoly commands. The arguments to the poly are a list of points (H,V) the poly will be drawn to. It's analogous to a moveto followed by series of linetos. If the lines cross, the paintpoly command will only fill areas on one side of any of its lines. It looks for the best enclosure. Note that the poly will only be closed if the last pair of numbers is the same as the first pair.

Regions

You can group a more complicated set of commands than the lines of poly. This is done by the region procedure. To do this, you issue these commands:

- Recordregion
- Any number of drawing operations
- Closeregion somename
- Paintregion somename H V

You won't see any drawing until the paintregion command. The H and V of the paint region will be added to whatever Hs and Vs were in the recorded drawing commands. You may be surprised at what you see when you include linetos in the region recording.

You can have as many named regions as you want to, but when you are done with one, you should call deleteregion to free up the memory. Clearregions deletes all regions, as does reset.

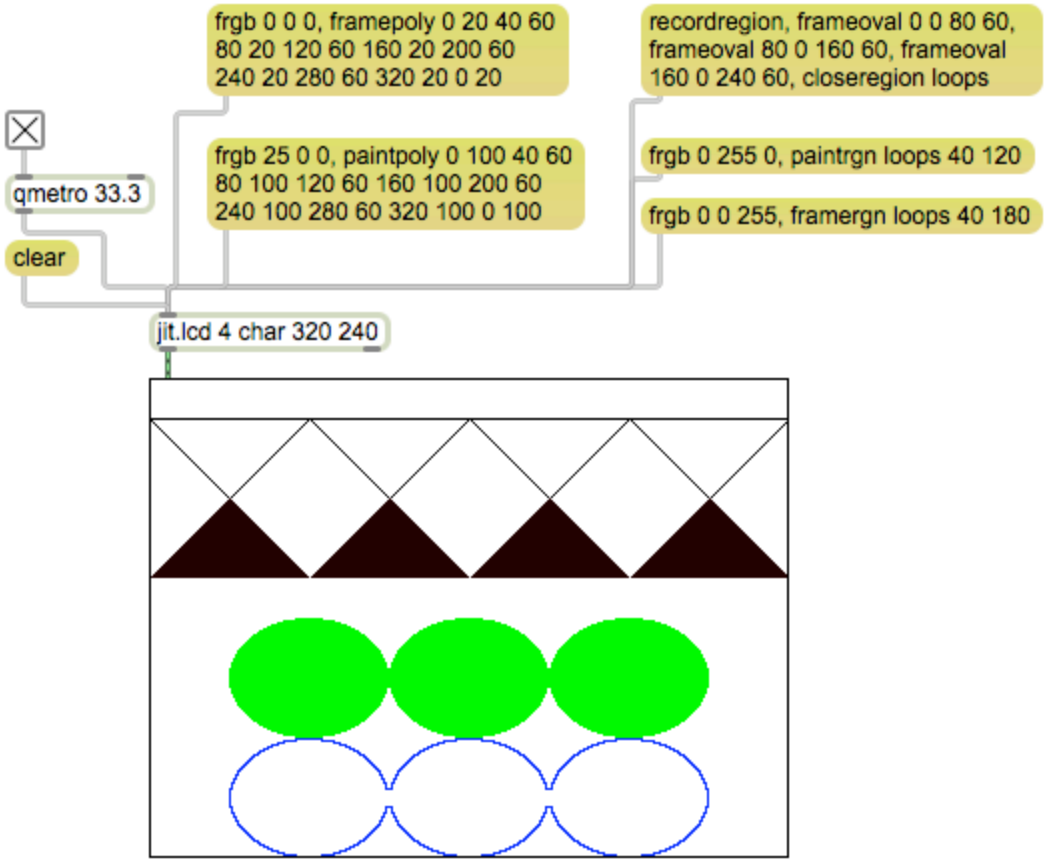

Figure 6.

Clipping Regions

A clipping region is a area that limits drawing, sort of a cookie cutter effect. Once a clipping region is defined, drawing is restricted to that region.
The following define clipping regions.
Cliprect
Clipoval
Cliproundrect
Clippoly

Only one clipping region is in effect at a time. You can have a complex clipping region by making a region, and using the command:
Cliprgn somename H V

If a clipping region is in effect, the clear command only clears the clipping region. The command noclip gets rid of the clipping region.

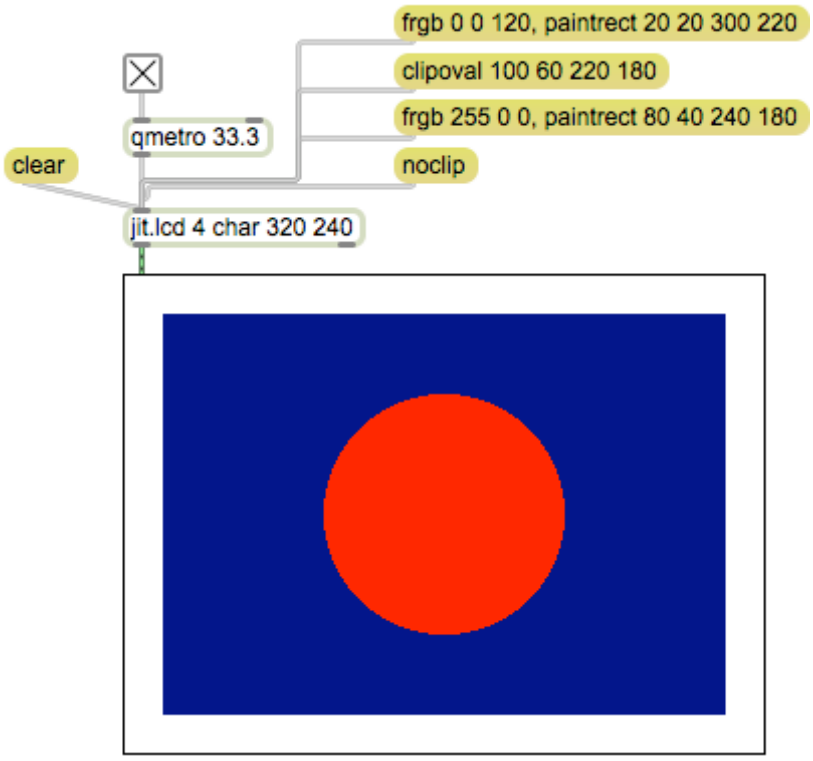

Figure 7.

Scrollrect

Scrollrect is similar to a region command. It copies whatever is in the specified rectangle, erases the rectangle, then draws the copy into a rectangle that is moved by the specified distance. That's all done by
Scrollrect H V H1 V1 mH mV

It's most useful for moving the entire jit.lcd contents. Note that once part of the drawing has been scrolled off the screen, you can't get it back.

Text

Text is drawn starting at the pen location, and moves to the right with each letter. If you want text to restart at the left and next line down when the right is reached, you have to do that yourself.

Font *name* s changes the font to *name* and size s. You can get a list of available fonts from the fontlist object.

The command [write something \, something] will write text. Note that some punctuation, like comma, must be preceded by a backslash to be written properly. The command ascii n n n n will translate the numbers to ascii equivalent letters and write those. There's a chart of ascii at the end of this tutorial.

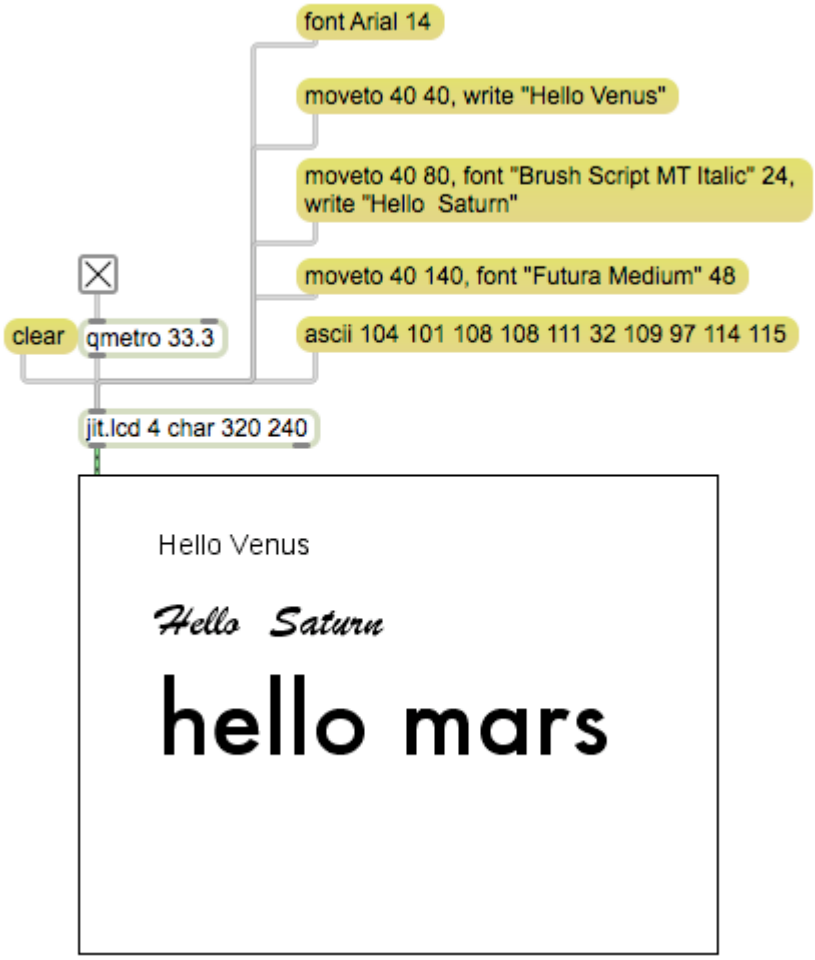

Figure 8.

picts

Pict files are an ancient Macintosh graphics files format. Very few graphics programs can create them. PICTs live on in jit.lcd however. The command

Readpict aname filename will load a pict or jpeg file into memory. The name you give it is just an identifier for it.lcd, it has nothing to do with what's on the drive. The name can also refer to a named matrix. (Currently broken in Max 6)

The *command drawpict aname* will copy it into the drawing. Once this is done, there is no connection between the image and the pict in memory. The arguments to drawpict can make some interesting changes:

drawpict aname H V	will place the upper left corner of the pict at H and V. Many picts include white space around the edges, so you may have to allow for that. (Try transparent mode)
drawpict aname H V w h	will draw the pict at location H V with width w and height h
drawpict aname 0 0 0 0 sH sV sw sh	will draw only part of the picture. SH sV and sw sh determine which part of the picture.
drawpict aname H V w h sH sV sw sh	will draw part of the picture in the rectangle scaled to fit into w h.
deletepict aname	removes the pict file from memory but doesn't erase it from the LCD.
clearpicts	removes all picts from memory.
tilepict H V w h sH sV sw sh	tiles as many picts as will fit in the space defined by H V w h. sH sV sw sh define the part of the pict that is used.

Once you have built up an interesting image in your LCD, you can immortalize it with the *writepict* command.

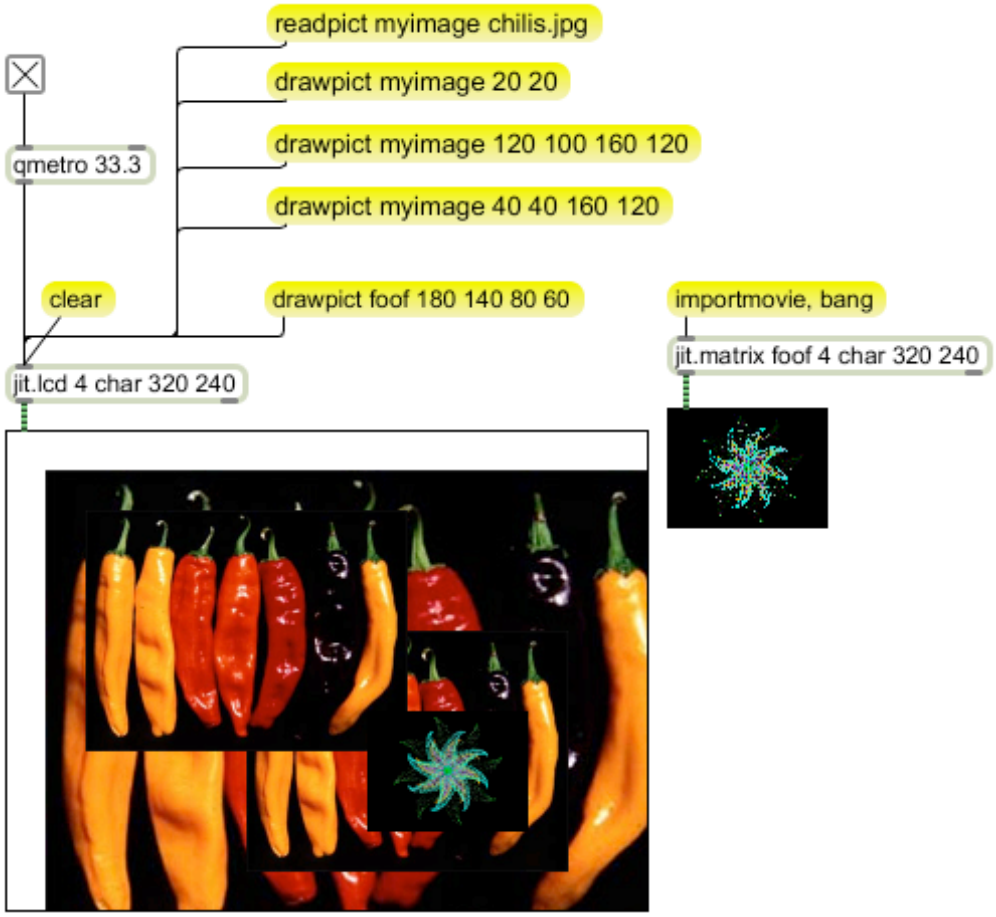
readpict myimage chilis.jpg
drawpict myimage 20 20
drawpict myimage 120 100 160 120
drawpict myimage 40 40 160 120
qmetro 33.3
clear
drawpict foof 180 140 80 60
importmovie, bang
jit.matrix foof 4 char 320 240
jit.lcd 4 char 320 240

Figure 9.

Co-ordinate Systems

Standard

I have described the image as an array of pixels, numbered with a pair of numbers from left to right and top to bottom. This is the way jitter addresses the screen and is usually convenient to use. There are some drawbacks however. We usually invent patches in a low resolution like 320 x 240 so we don't have to worry much about performance. Eventually, keepers are expanded to higher resolution. This requires the recalculation of every point in the drawing, a tedious and error-prone process.

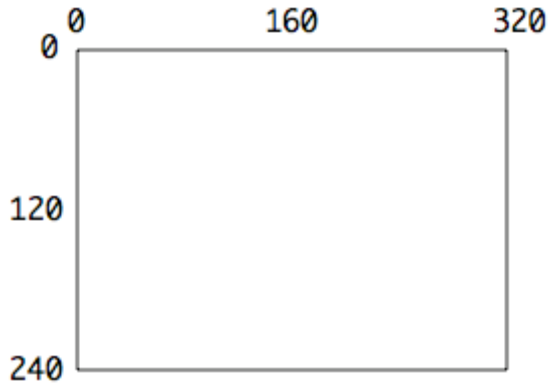

Figure 10. Native co-ordinates

Center Origin

I have found that recalculations are simpler if the origin (point 0 0) is in the center of the image. This scheme is known as Cartesian coordinates[3]. A Cartesian co-ordinate system looks like figure 11.

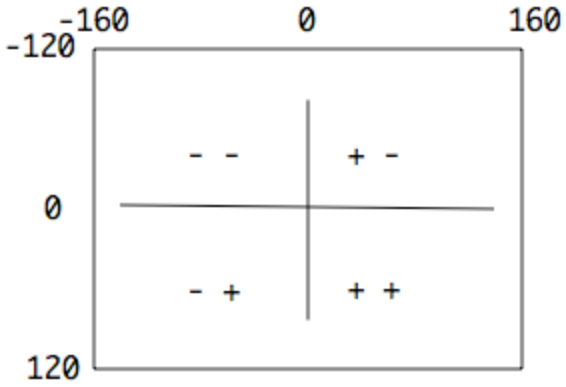

Figure 11. Center Origin or Cartesian coordinates

[3] After mathematician René Descartes

The point 0 0 is in the center of the image. This divides the screen into four quadrants according to the sign of the co-ordinates. The upper left has negative values for H and V. The lower right has positive values for H and V. In the others, H and V have differing signs. One advantage to this system is the image can be flipped right side up by changing the sign of all V values. Another is that the size of many images can be changed simply by multiplying the coordinates by a single value. Of course this scheme has to be converted to standard coordinates for display. This is made easy by the Lobjects Ladd and if necessary Lmult.

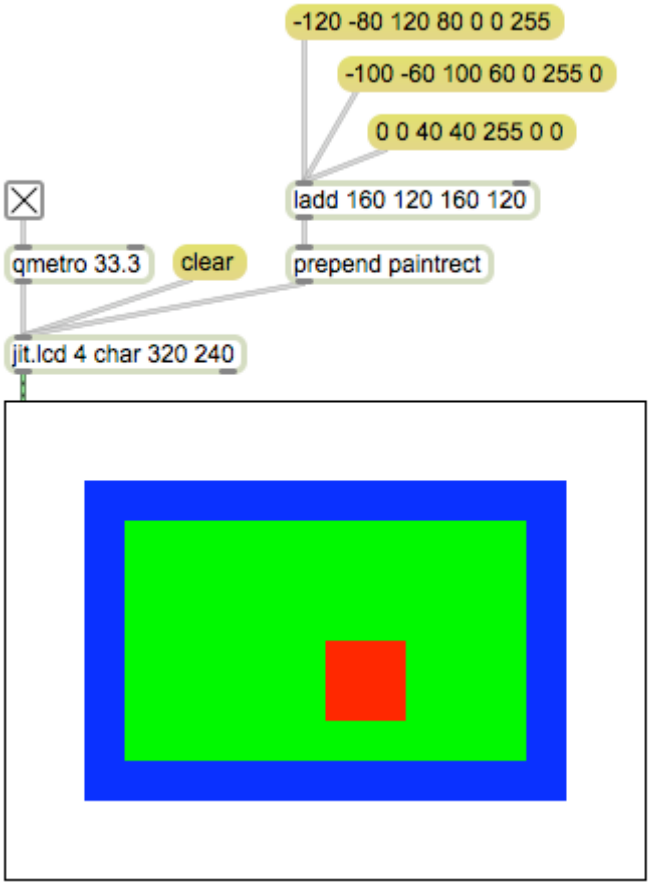

Figure 12. Drawing with center origin coordinates

If we prepend the command to the bottom of the chain of calculations, we save a lot of typing. The Ladd object adds values of two lists member by member. The Lmult object shown in figure 13 multiplies the values of two lists member by member, but if a single value is entered instead of a list, all values of the input list are multiplied by that value[4]. Follow what happens when you click on the message with numbers [-40 -30 40 30]. The list forst passes through lmult, whihc has been set to 3. This changes the list to be [-120 –90 120 90]. These are added to [160 120 160 120] (the center point) and become 40 30

[4] Most Lobjects work that way. One value processes an entire list, two values process the first two members. If an input list is longer than the control list, extra values are passed through unaffected.

300 210]. Prepend puts framerect before this list, so the outer box is drawn. The other bo xes were drawn with different settings for lmult. If lmult were set to 0.5 there would be an even smaller box.

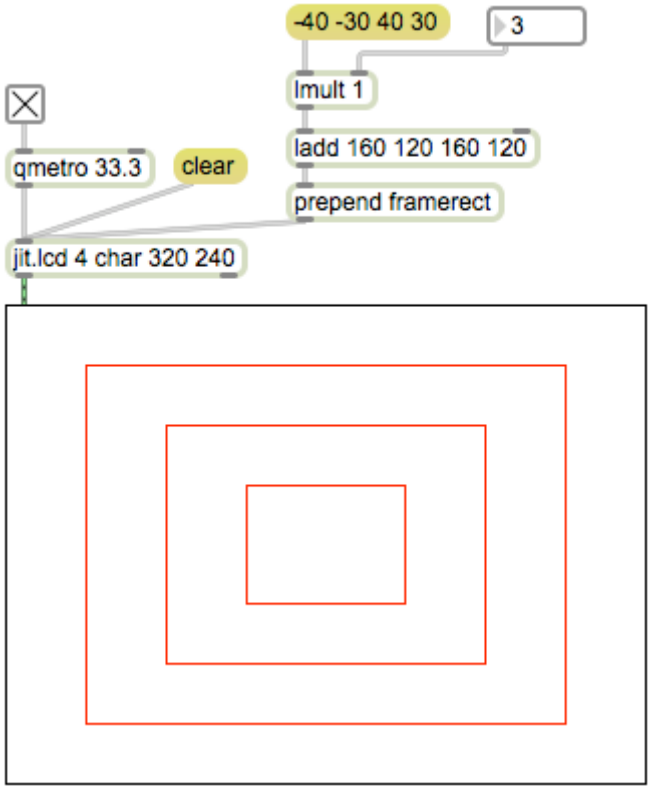

Figure 13. Resizing drawings

Drawing Pixels

I often draw with very small rectangles, even single pixels, but usually the rectangles are 2 x 2 pixels. (Single pixels do not show well in projections.) Rects 2 on a side can overlap and give a consistent line, especially if the overall resolution is 680 x 480 or better. The easiest way to draw these is to build off of the center origin technique as shown in figure 14.

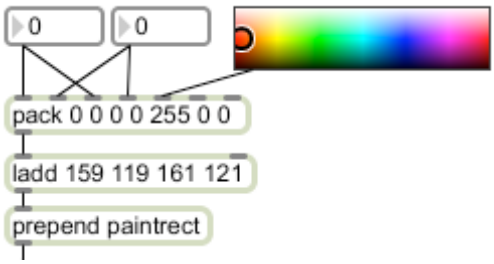

Figure 14. Draw a 2x2 rectangle

The key feature is the pack object that builds up the list of parameters for the paintrect command. The H value is sent to the first and third inlet and the V value to inlets two and four. The last three inlets are for color-- the swatch object shown will provide all three values (open the swatch inspector and set "old style 0-255 value"). This list is passed to

an Ladd object as before, but the numbers in Ladd now produce a larger rectangle. If the location is given as 0 0, the rectangle will be painted as 159 119 161 121, centered in the window.

Figure 15 makes a more elaborate image.

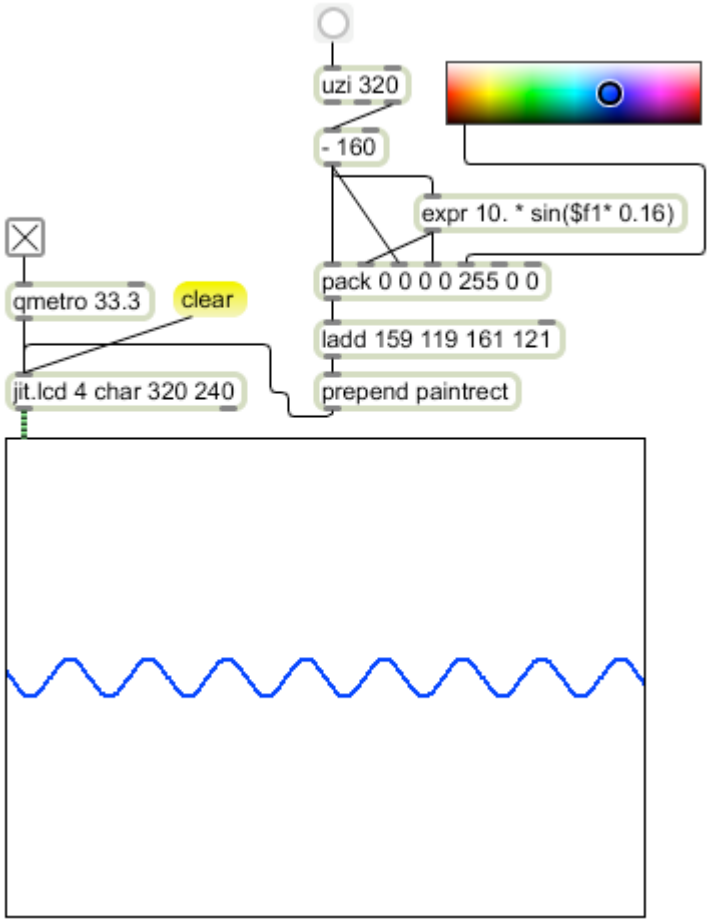

Figure 15.

This is a simple trig function calculated in the expr[5] object. The numbers going into the expr come from the uzi[6] object, modified to range from -159 to 160. These numbers become the H values and the output of the expr is used for V. We are literally drawing a graph of the function. I'll explore this kind of drawing in another tutorial.

[5] Expr lets you write one line of C code.
[6] Uzi bangs a lot of times, as fast as possible. It also produces numbers from the right, counting from 1 to the argument.

Polar Coordinates

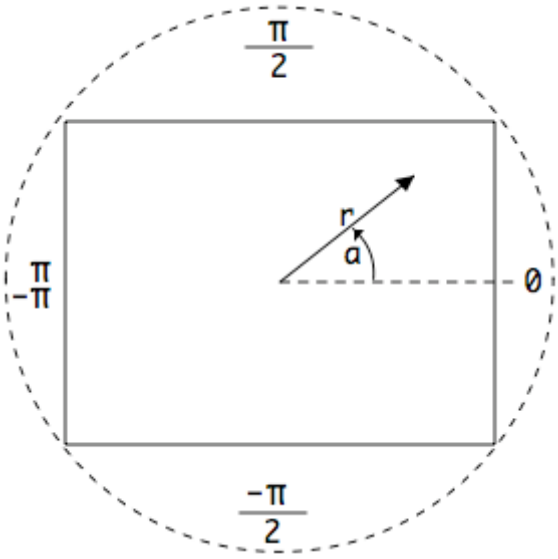

Figure 16. Polar coordinate system

You may remember polar coordinates from a trigonometry course or from looking at response plots for microphones. Polar coordinates specify a location by a distance from the origin (which is usually the center of the plot) and an angle from the horizontal. Since the changing the angle while holding the distance steady produces a circle, the distance is the radius or r. The angle is usually given as a Greek letter but in code we use a. The angle is specified in radians. A radian is an angle of 57.28°, which is not important. What is important is there are 2π or 6.28318... radians in a complete circle. This simplifies a lot of math. (Really!)

Figure 17 shows a mechanism that calculates a circle full of angles in radians.

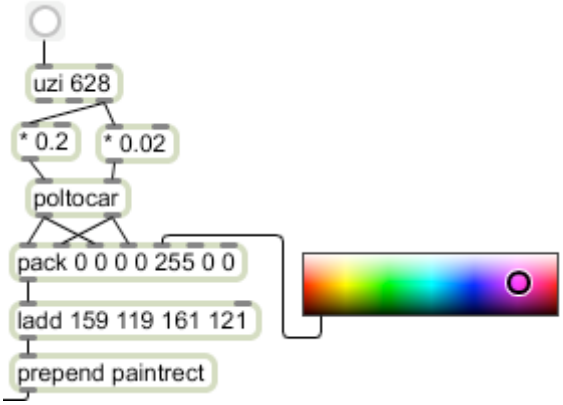

Figure 17.

This uses an uzi to produce numbers from 1 to 628. If we multiplied this by 0.01 it would count in 100ths of a radian from 0 to 2π, once around a circle. Multiplying by 0.02 increases the step so it goes around twice. The radius is calculated from the same numbers that drive the angle, increasing at 10 times the rate. This will produce a simple spiral. The entire patch is shown in figure 18.

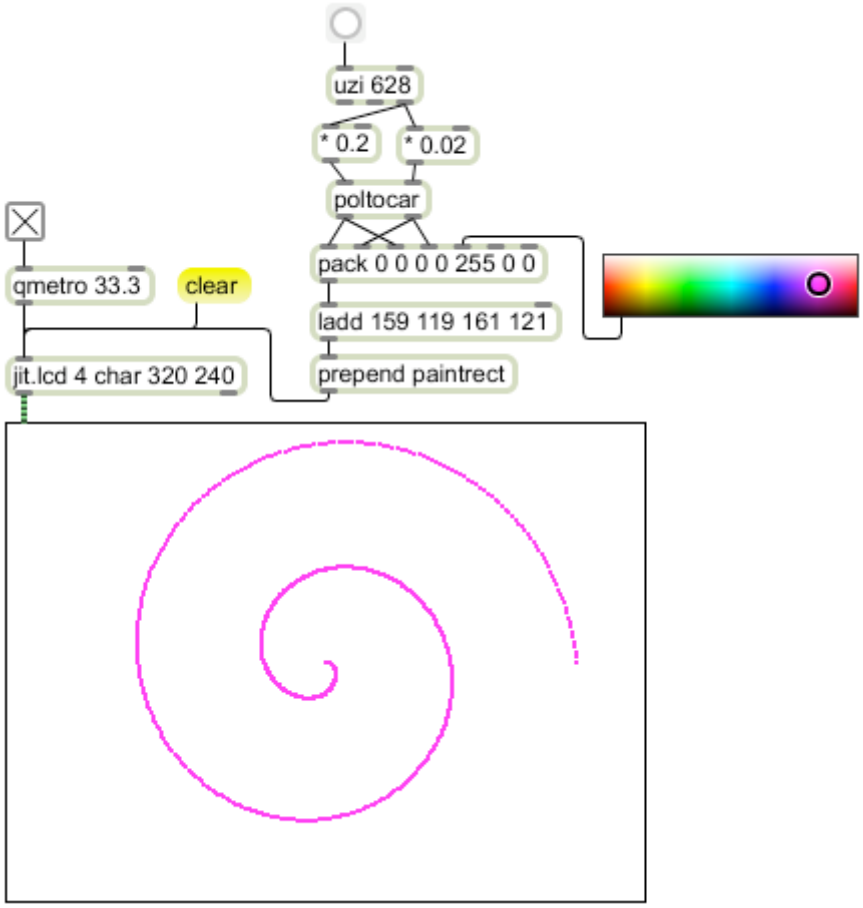

Figure 18.

Polar coordinates can also be used to rotate an image. We do this by converting the Cartesian coordinates to polar coordinates with some trigonometry hidden in the *cartopol* object. We then change the angle and convert back using *poltocar*. Figure 19 uses the drawing of figure 15 to plot a sine curve- however, by converting to polar, adding 1.57 ($\pi/2$) to the angle and converting back, the image is drawn from top to bottom. This may not seem like a big deal, since the original drawing could have been done that way, but using a single value to control the rotation is very handy-- it can even be used in an animated patch to make the image spin. Figure 20 was made by simply changing the angle and drawing again. The angles used were 0, 0.78 and -0.78.

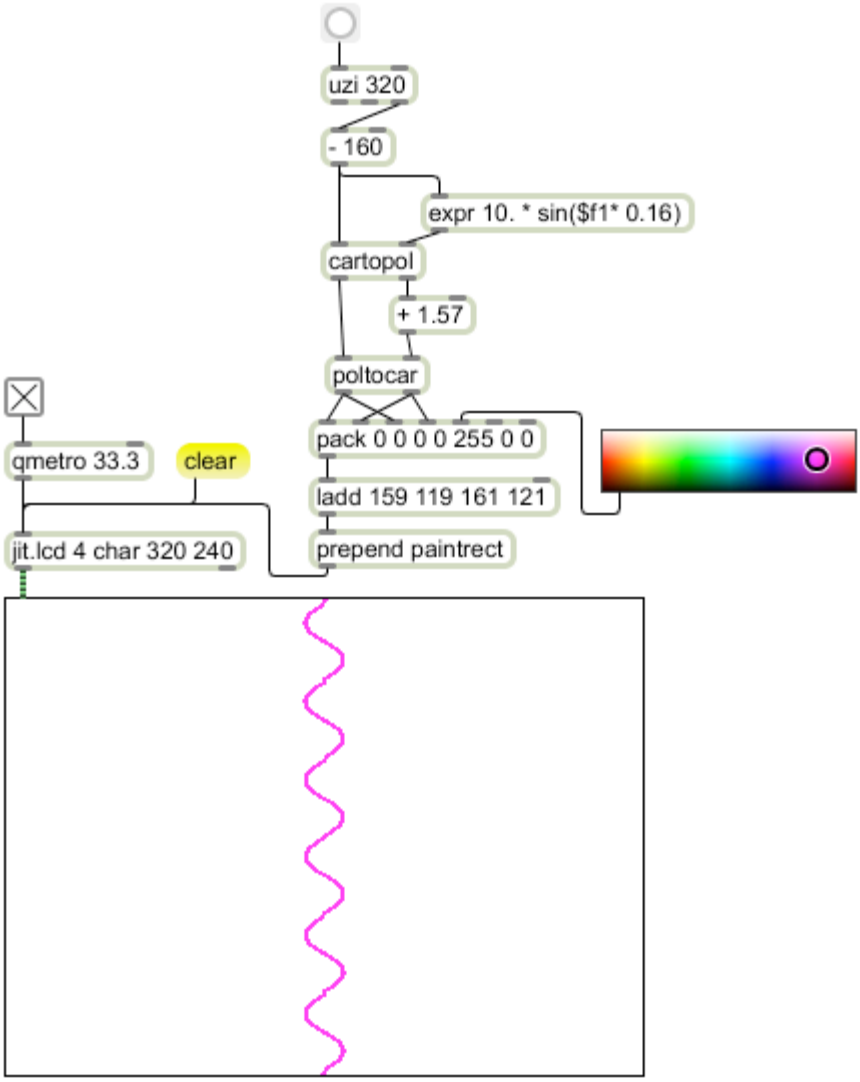

Figure 19.

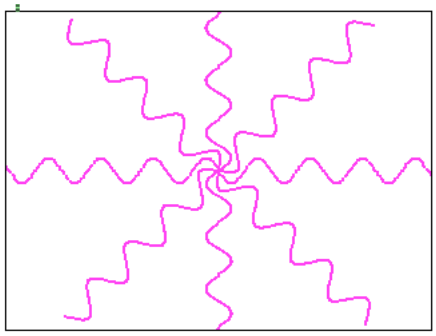

Figure 20.

3D Coordinates

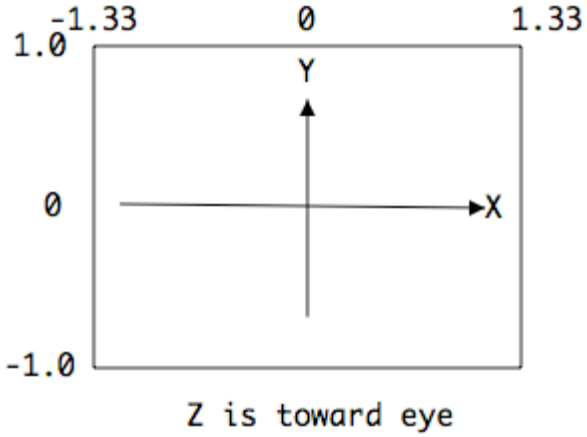

Figure 21.

When we work in openGL with the jit.gl set of objects we will use a 3 dimensional coordinate system. This is the Cartesian system with an added axis that points out of the picture. All points are specified with three numbers relative to an origin of 0 0 0. When we move objects toward negative Z values they will recede into the distance. Distances in openGl are in floating point numbers. When we look into openGL space in jitter, the Y axis at the origin runs from -1.0 at the bottom to 1.0 at the top. (Right side up.) If the window has a 4:3 aspect ratio, the X axis runs from -1.33 to 1.33. The "frame" of the window is at Z = 2.0, so if we move an object in the positive Z direction it will be behind us.

There is a separate tutorial on openGl in Jitter.

ASCII Codes

Here's a table of the ASCII codes. The enigmatic ones like ETB apply to teletypes and the like. You might get them from some serial applications. The parentheses indicate the Mac key that generates them.

0	NUL	34	"
1	SOH (Home)	35	#
2	STX	36	$
3	ETX (enter)	37	%
4	EOT (end)	38	&
5	ENQ	39	'
6	ACK	40	(
7	BEL	41	)
8	BS (delete on Mac)	42	*
9	HT (Tab)	43	+
10	Line feed	44	,
11	VT (Page Up)	45	-
12	Form Feed (Page Down)	46	.
13	CR (return)	47	/
14	S0	48-57	numbers 0 -9
15	SI	58	:
16	DLE (all F keys)	59	;
17	DC1	60	<
18	DC2	61	=
19	DC3	62	>
20	DC4	63	?
21	NAK	64	@
22	SYN	65-90	Capitol A -Z
23	ETB	91	[
24	CAN	92	\
25	EM	93	]
26	SUB	94	^
27	escape (also clear)	95	_
28	cursor right	96	`
29	cursor left	97 - 122	letters a - z
30	cursor up	123	{
31	cursor down	124	|
32	space	125	}
33	!	126	~
		127	DEL