Simple Animation with jit.LCD

Object movement

The jit.LCD object will draw fast enough that we can do simple animation on the fly. The secret to animation (as opposed to gradually filling the screen) is to clear and draw the image once per frame. Usually we start by designing an image, then attaching the animation afterward. If we keep the number of parameters needed to draw to a minimum, it will be relatively easy to add animation. Figure 1 is a patch that will draw a circle anywhere we want it.

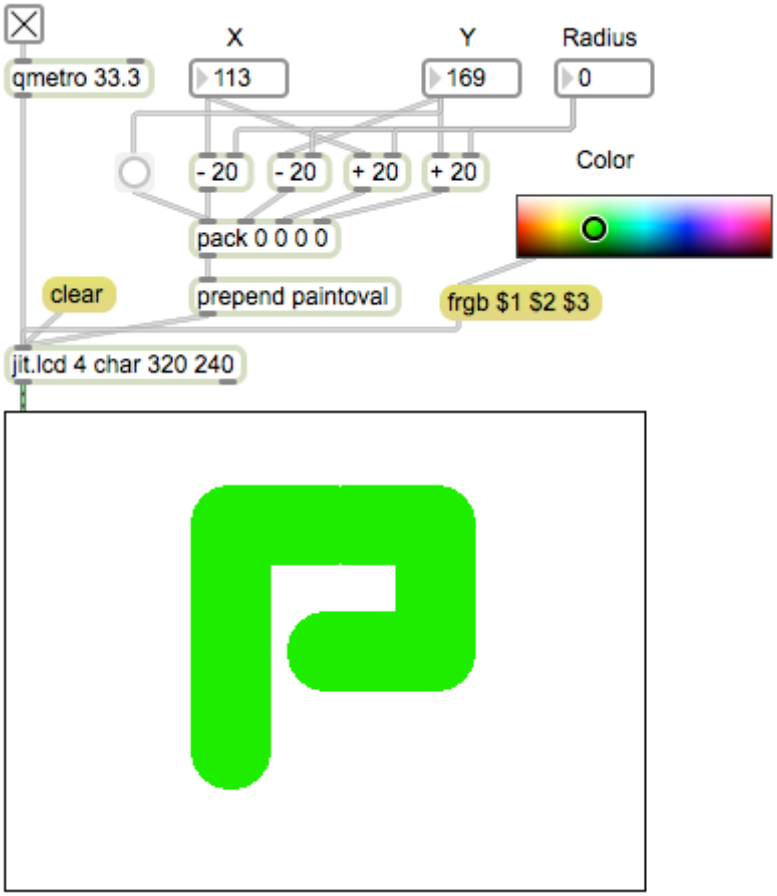

Figure 1.
This patch will work like an Etch-A-Sketch. Adjusting the X and Y number boxes will draw the circle in a new position- this will build up an angular image until the jit.lcd is cleared.

A patch like this is often easiest to understand if we analyze from the bottom up, repeatedly asking the question, where does the data come from? The draw command to jit.lcd is paintoval, which requires arguments for left, top, right, bottom. These arguments

are supplied as a list, to which paintoval is prepended. The list is constructed in a pack object. The data comes from the three number boxes labeled X Y and Radius:

Left = X - Radius
Top = Y - Radius
Right = X + Radius
Botttom = Y + Radius

Pack will produce a list every time the left inlet is changed. That means changing Y would not create a new circle on its own. To make that happen, I added a button to bang the list from pack when Y is changed. It's important that the bang is to the left of the math boxes. Otherwise, the pack would send a list containing the old Y data before the new data was calculated.

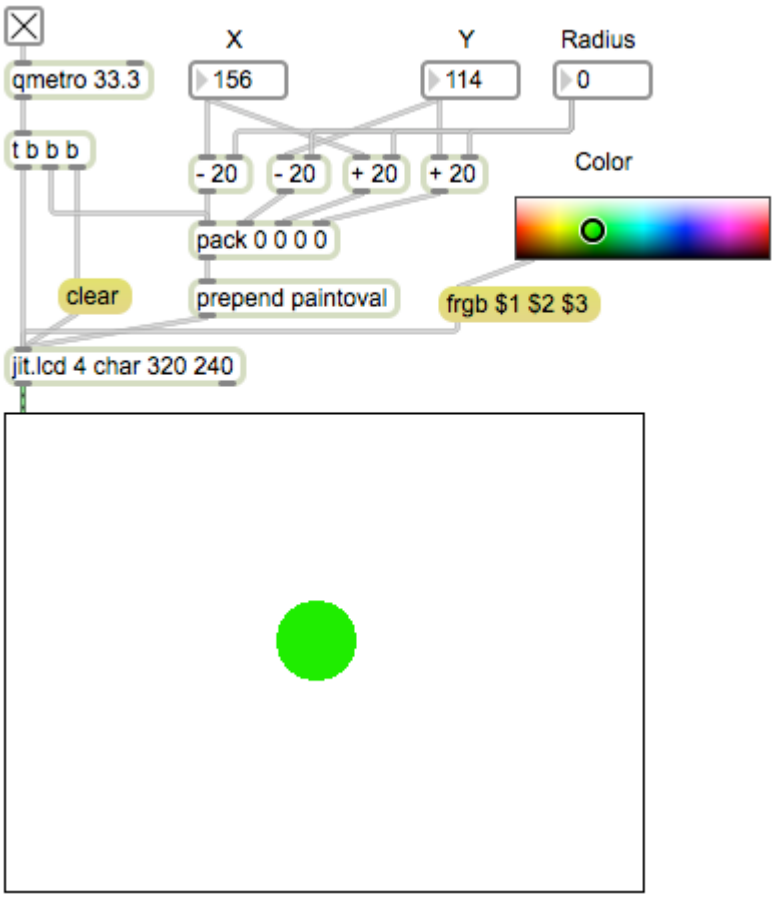

Figure 2.

In figure 2, I have made two changes to the patch. I have replaced the button with a trigger (AKA "t") object attached to the qmetro. This trigger produces three bangs each time it is banged. The right outlet bangs the clear command to the jit.lcd, the middle outlet bangs the pack, which will send its current list along and draw a single circle. Finally the left trigger outlet will send the image out of jit.lcd. What you see is a spot that moves when you adjust X or Y and changes when you drag the circle in the Color

swatch. Changes in radius are not apparent until the X is next changed, but that could be sorted out with another trigger object.

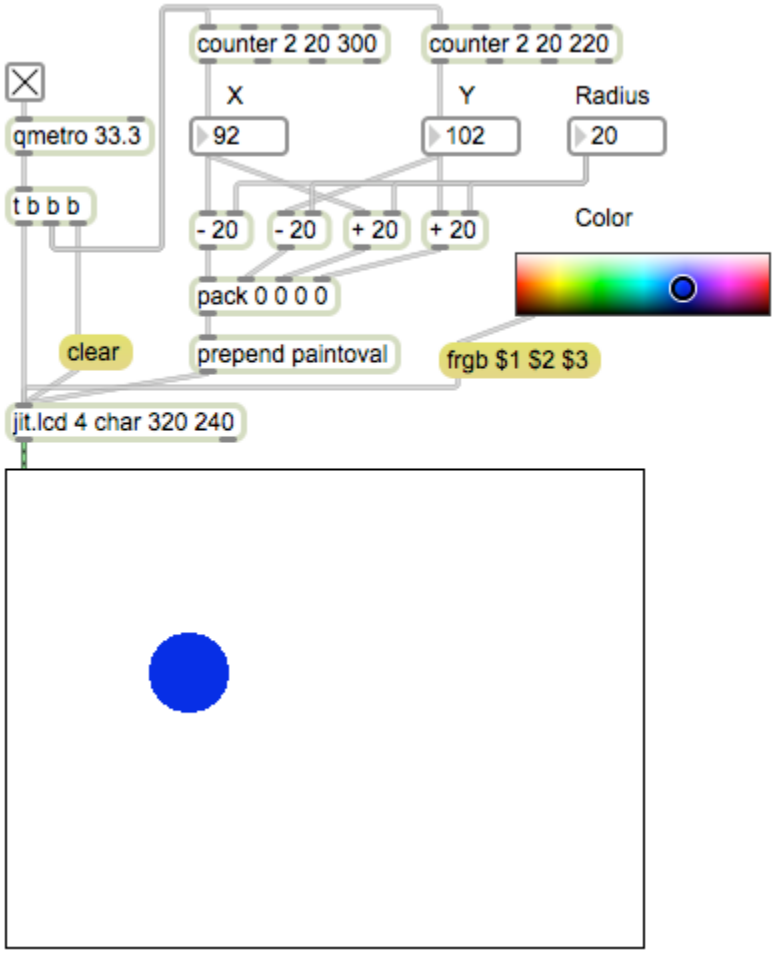

Figure 3.

The simplest way to animate this patch is to add counters to control X and Y. With 2 as the first of three arguments, a counter will count back and forth between the other two arguments. With the patch as shown the blue ball will drift around the screen, bouncing off the edges. This can be refined by adjusting the endpoints of the counters to compensate for changing the radius of the circle. We can change the rate of motion by modifying the output of the counters.

We can get an interesting result by adding a fade effect to the jit.lcd. This is done by copying the output of the jit.lcd to a named matrix, then using a matrix of the same name[1] to replace the clear command. There should also be a jit.op set to multiply this matrix by 0.999. All of this has been added to figure 4.

[1] If matrices share a name, they also share the data. We use two here to avoid a stack overflow error.

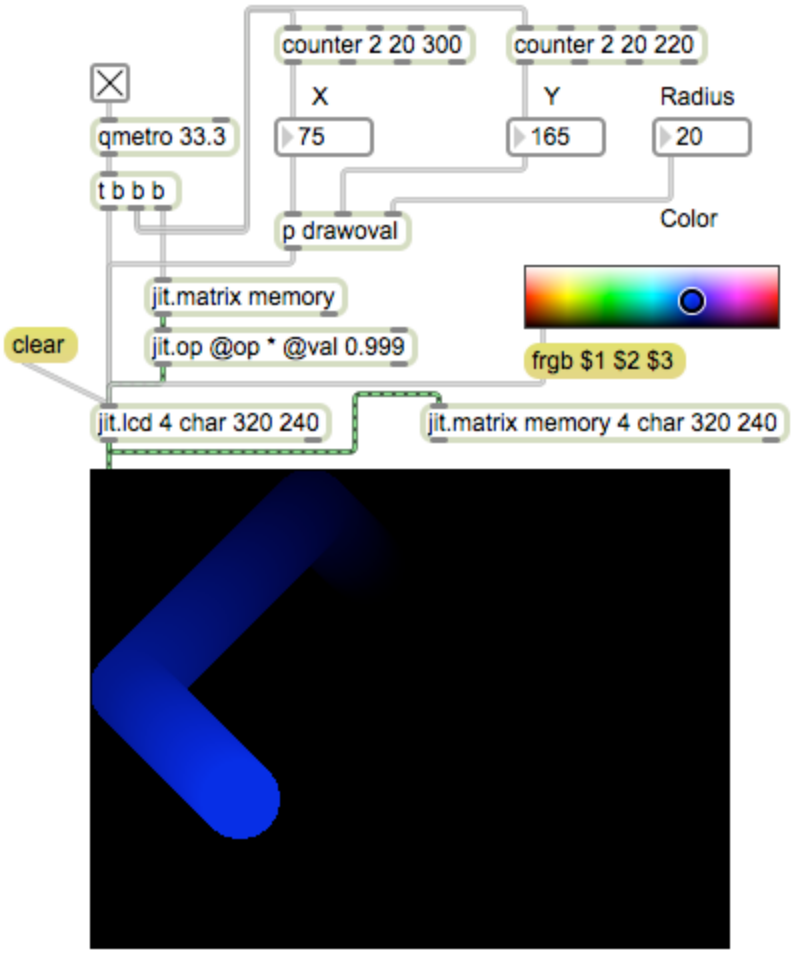

Figure 4.

This it what is going on with each tick:

- The image in the matrix named memory is output to the jit.op
- The jit.op multiplies all values by 0.999, which makes the image slightly darker.
- The darkened image is applied to jit.lcd, replacing the contents.
- A new circle is painted on top of the old image.
- The new, composite image is displayed and copied into the memory matrix, ready for the next frame.

The result is an image that fades out over time. Instead of simply moving around, the circle leaves a tail. This is an example of feedback, which has a tutorial of its own.

In figure 4, I have encapsulated the middle of the patch into the subpatcher drawoval. (Shown in figure 5.) This makes the patch a bit neater, but also raises an important implication: any subpatcher that issues a drawing command when it is banged could be used in this place. That includes images from paintrects. There can also be more than one object- in fact, the complexity of the image is only limited by the amount of drawing that can get done during one tick. So, animation of object movements only requires two

elements-- a mechanism that draws on command with an X and Y parameter to set position, and a mechanism to calculate movement.

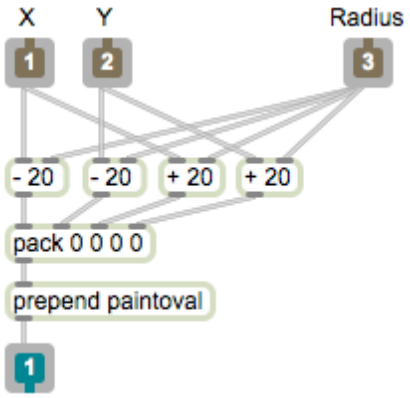

Figure 5. Contents of drawoval

Object Rotation

The tutorial on basic drawing introduced the concept of using polar coordinates to rotate a drawing. Figure 7 began as figure 19 from that tutorial. I have added three elements:

- A mechanism to change the angle of rotation
- A mechanism to change colors
- The feedback scheme described above

I have used my Lcount object to drive the rotation and color change. Lcount is similar to counter, but has a different set of features:

- Lcount counts by an increment that can be other than 1. This gives precise control of rate of motion, and smoother movement.
- Lcount can count by floats-- when doing so, it wraps around instead of returning to the start value when the count exceeds the end value. This produces smooth circular counting.
- The end value of Lcount may be less than the start value.
- You never see the end value from Lcount. If the count range is 0 to 16, you get 16 counts: 0 to 15. This is the type of counting provided in most programming languages.
- The count value comes out of the center outlet. The right outlet bangs when the count hits the high point, the left bangs when the count returns to start. I often use these bangs to stop a counting operation. If the right outlet stops the count, the final value will be the end of the range. If the left outlet stops the count. The final value will be 0 (or the initial value).

Figure 6.

Figure 6 shows how Lcount can generate a series of angles for rotation. Bangs come in from some steady source such as the qmetro. With a target of 6.28 (2π) no extra processing is needed. Each bang increments the count. The increment is set by the first of three arguments or a value in the second inlet. With an incement of 0.1 it will take 62 frams (two seconds?) to get to the top. The higher the increment, the faster the counter will reach the end of the count. If the increment is negative, the count will go backward.

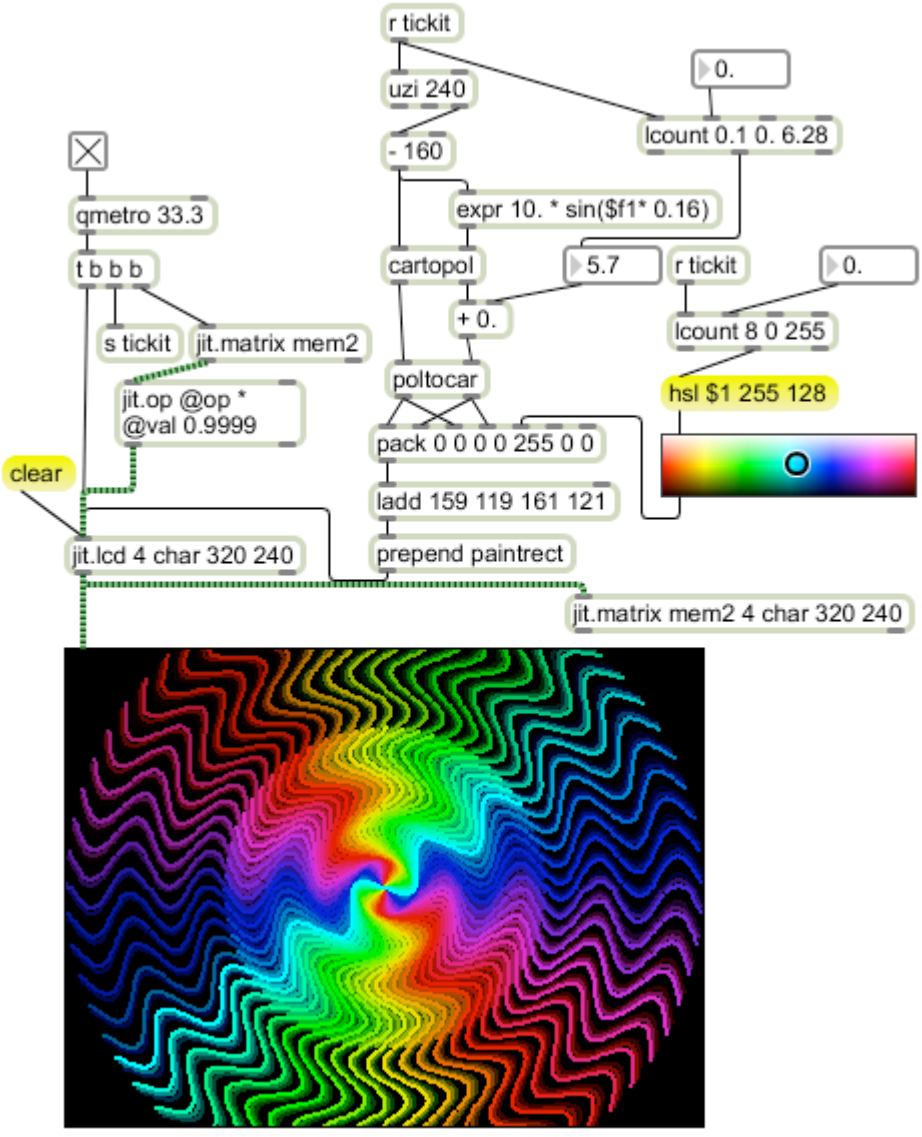

Figure 7.

A bottom up analysis of figure 7 goes like this:

- The draw command is paintrect, which is prepended to a list of data.
- An Ladd offsets the drawing origin (0 0) to the center of the screen, and sets the rect dimensions to 2x2.
- The data is gathered by a pack object- four values for position are derived from two outputs of a poltocar, three values for color are derived from a swatch. (I'll discuss what the swatch is doing shortly.)
- When we see a cartopol connected almost directly to poltocar, we recognize a rotation is being applied. Rotation is accomplished by changing the angle, usually by an addition.
- If we follow the branch that sets the angle, we find the Lcount mechanism recently discussed.
- The Y value going into the cartopol is the sine of the X value, with some modifications to affect the height and frequency of the sine curve. (This is discussed in trigonometry notes.)
- The values for X and the sine expression come from an uzi that generates numbers 1-240 on each tick. 160 is subtracted from these numbers, so they actually run from -159 to 80. When offset to the center origin a line will be drawn from the left edge to halfway between the center and right edge. In other words, the line is not rotated about its center. This produces the effect of a bright disk surrounded by filaments you see in figure 7.
- Uzi is banged by the output of a receive object named tickit. Somewhere there must be a send object with a matching name. (If you double click on the receive, Max will show you the send.) Here it is the center output of the trigger on the qmetro, so drawing happens between the clear and the bang that causes jit.lcd to output.
- The leftmost stack of objects is familiar. The toggle turning on a qmetro is found in every jit.lcd patch. A trigger to provide three bangs usually means an animated patch. (There may be an exam.) in this case, the right outlet of trigger is connected to jit.matrix instead of a clear to produce the fade effect.

Object Color

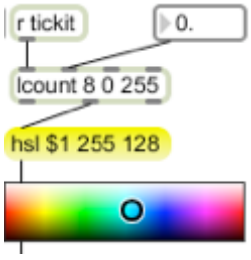

Figure 8.

The colors in figure 7 change slightly on every tick. The color comes as a list from the swatch object, which outputs a list of RGB values when the little circle is moved. The values are from 0-1.0 unless you open the inspector and set "Output old style 0-255 values". The new style is compatible with openGL, but jit.lcd still requires old style. The

swatch cursor can be moved by various commands, the most useful of which is hsl. This stands for Hue-Saturation-Level, which is another way to define a color. Hue is just that (moving the cursor left and right) saturation is the intensity (0 saturation produces shades of grey) and level is the balance between white and black. (It moves the cursor up and down.) These are also the arguments to the hsl command. I like maximum saturation colors of medium level, so a numerical control of hue works most of the time. The $1 token in the message box is replaced by a number from Lcount to constantly shift the cursor from left to right. Lcount gives me adjustable setps. I often use a swatch driven this way by a random object to give unpredictable colors.

Object Size

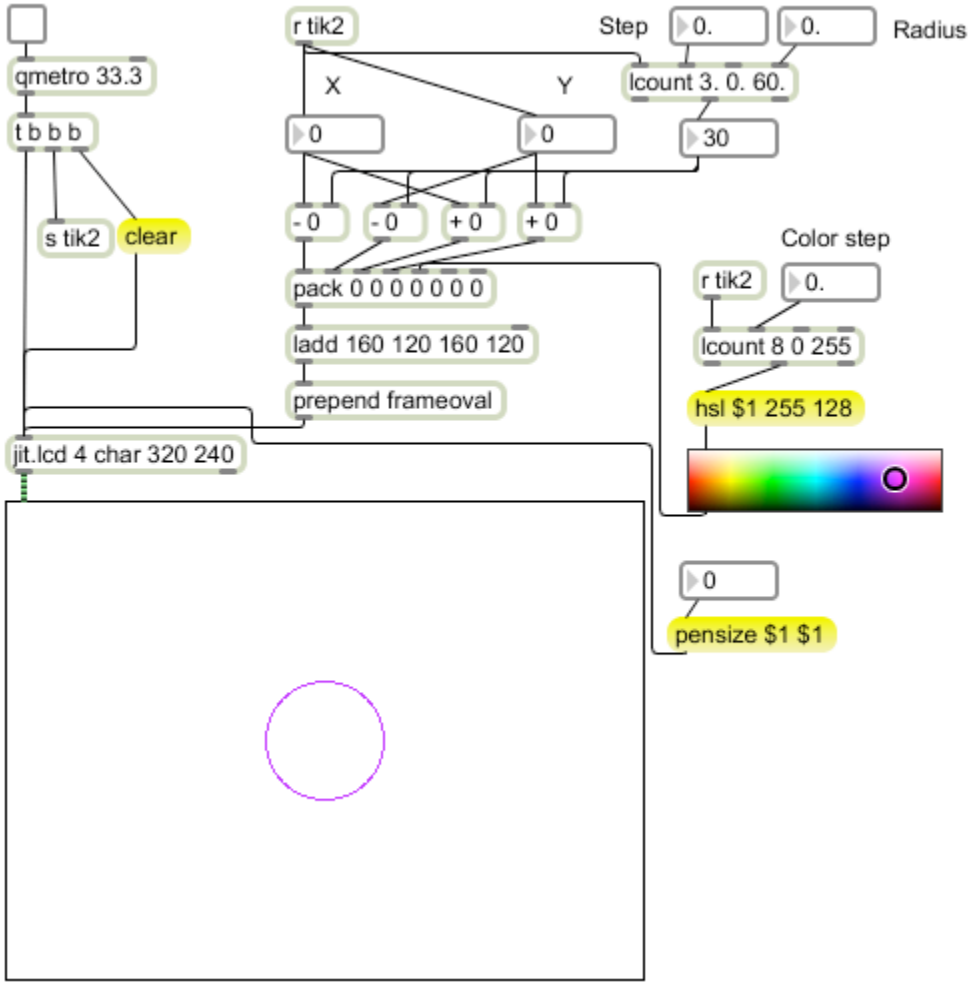

Figure 9.

Figure 9 is another variant of figure 1. This time, I've used Lcount to animate the radius of the circle. I'm also using frameoval to get an empty circle. When this patch is running we see an expanding circle-- this can be located anywhere in the screen, with the origin in the center.

It's interesting to see what you get from this when the clear message is defeated: Figure 10 shows the results with various settings of step and color step. Large steps produce concentric rings of course, and a step of 1 produces smooth transitions. The shading in the third image is a moiré pattern produced by the interference of closely spaced lines and the pixel grid and will shimmer when this is projected. If the image is allowed several cycles of drawing there are interesting blends from adjacent bands of color. This is one application where the odd penmodes can be interesting.

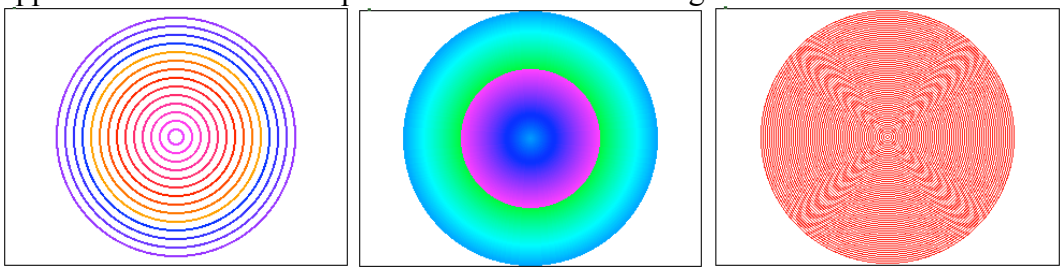

Figure 10. Drawng circles: step of 8, step of 1, step of 2 with pensize 1.

These images are similar to the effect generated by adding feedback, as shown in figure 11.

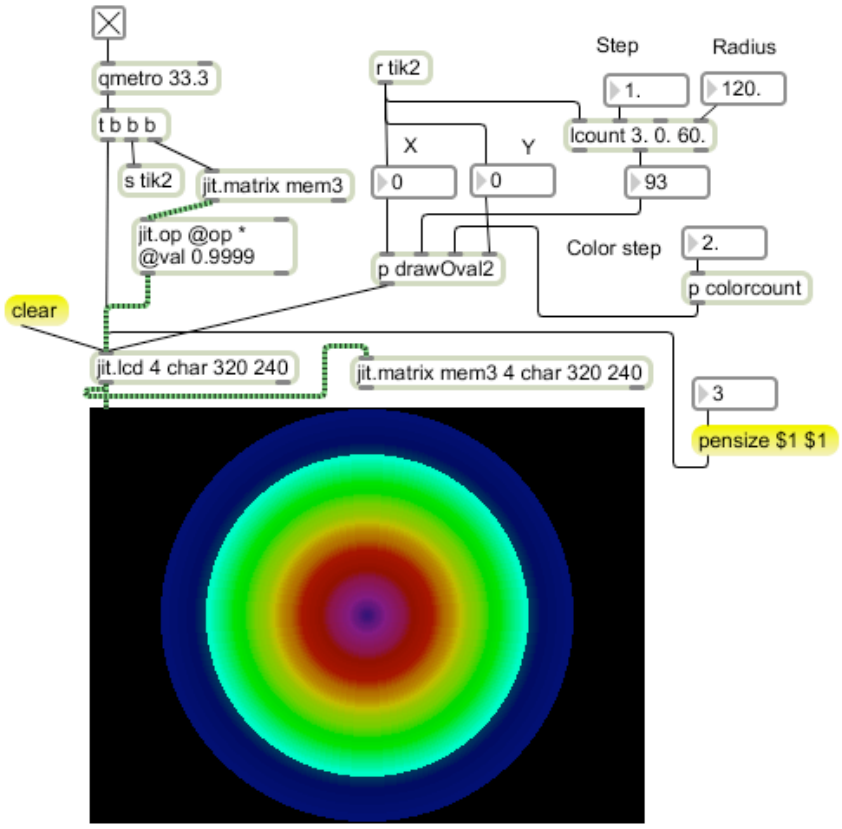

Figure 11.

Multiple Objects

Encapsulation

The oval drawing mechanism has been encapsulated for clarity in figure 11. (DrawOval2) Figure 12 adds two features to the drawing mechanism. There is now a gate object in the path of the bangs from qmetro (via receive tik2). I have shown this path with red patch cords. The gate will pass bangs only when the toggle is set. When the toggle is cleared, drawing will stop. The message 0 will clear the toggle, and that message is banged by Lcount when it restarts at minimum count. The result is a single expanding bubble when the toggle is set. A button makes this easy.

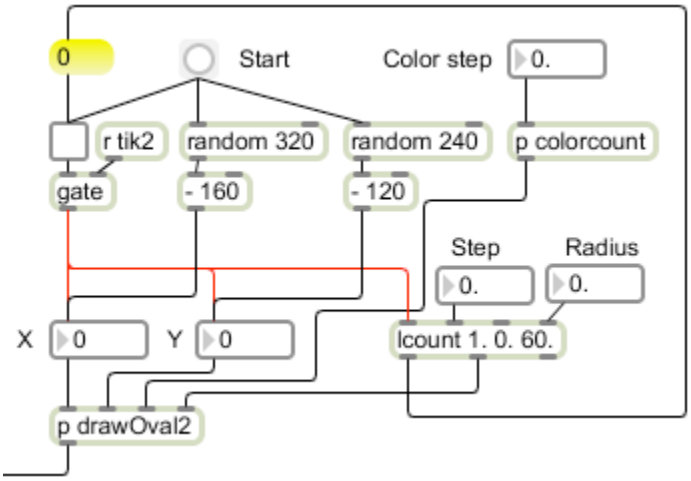

Figure 12.

The second addition is a pair of random objects to make the location of the bubble unpredictable. Random will generate some integer less than its argument each time it is banged. The random 320 will produce something suitable for the X coordinate if 160 is subtracted (this is a center origin patch). That will change the value in the number box. Y is treated the same way. These random objects are triggered by the start button just before the toggle is set-- that results in one bubble popping up somewhere in the window.

Note that the random objects are to the right of the gate mechanism. This ensures that the new coordinates are calculated before the gate is turned on. In fact, the new random X value will trigger a draw, which is why lcount is set to stop on 0 (the shutoff bang originates at the left outlet). That draw operation will have a radius of 0 and won't show.

Figure 13 Shows a further encapsulation

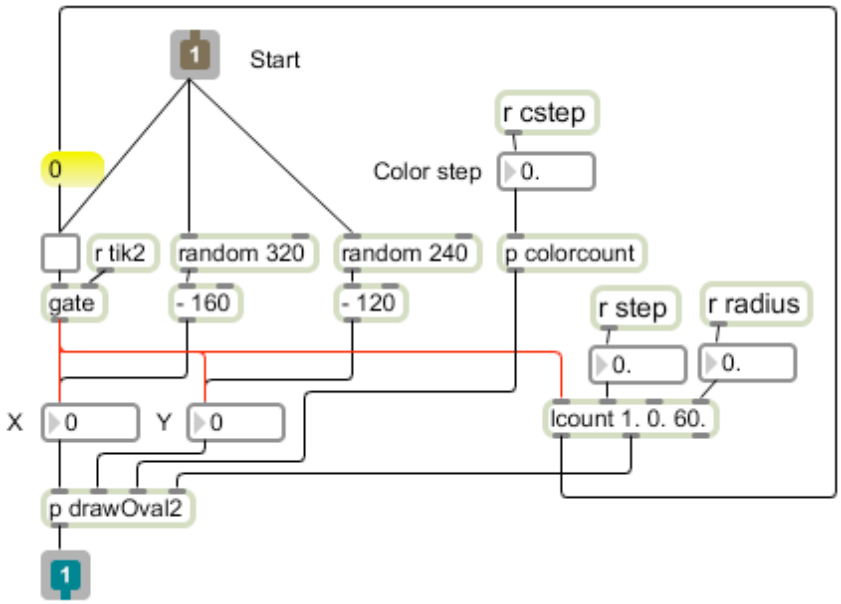

Figure 13.

Figure 13 is entirely self-contained. A bang at the inlet will begin the drawing process, which can be modified remotely with sends for step, radius and color step. The point of this is to build an array of these subpatchers, titled drawBubble in figure 14.

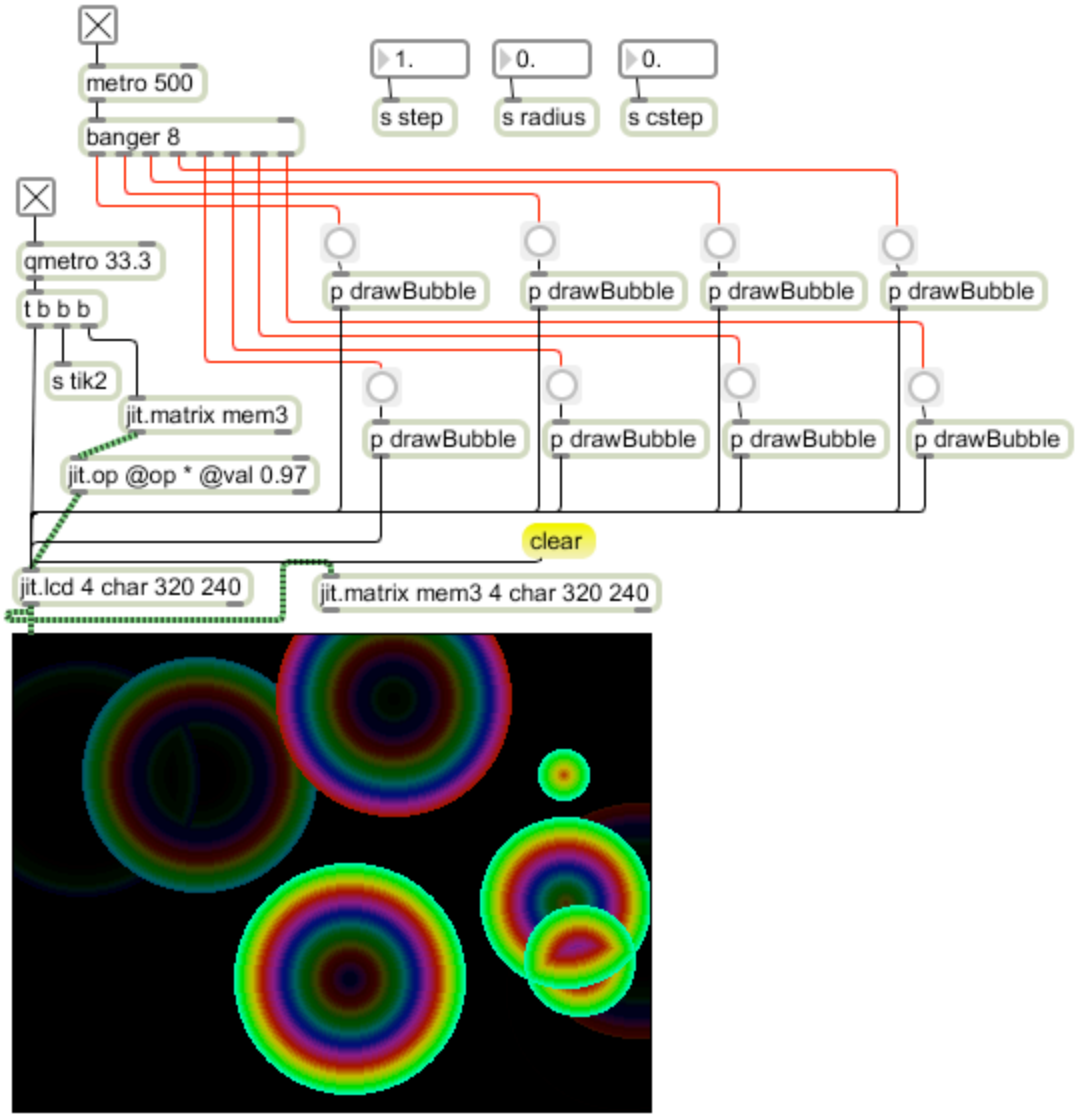

Figure 14.
The individual bubbles are triggered by banger, an Lobject that rotates a bang through its outlets. Since each drawBubble only issues one command per frame, you could use quite a few of them.

Multiple Objects with Poly~

When a patch is built of many copies of a single subpatch, the overpatch can be simplified by the use of poly~.[2] Poly~ is mostly used in MSP, where it encapsulates synthesis voices to build polyphonic instruments, but it works with any kind of code. Poly~ is a wrapper that can contain as many copies of a subpatch as desired. (See Working with poly~ for details.) The first step to using it is to create a patch out of the subpatch and save it like Figure 15.[3]

[2] That's poly~ with a tilde, not to be confused with plain poly.
[3] Some folks call a saved subpatch an "abstraction".

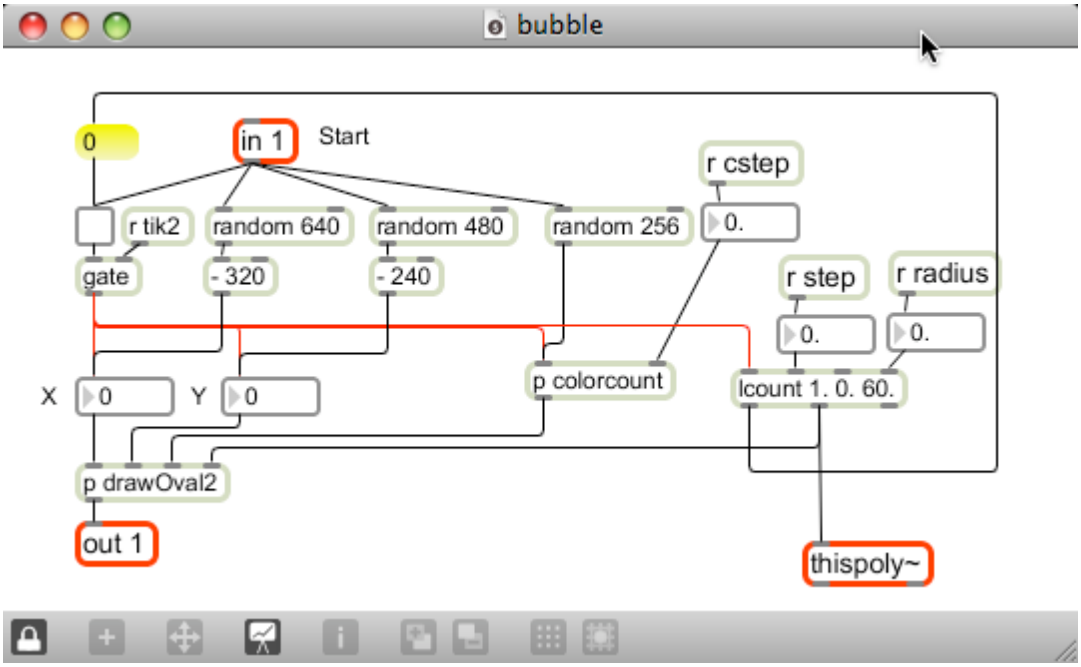

Figure 15.

Figure 15 is the same as the drawBubble subpatch except for the added objects, which I have marked in red:

- The in object will make an inlet appear in the poly~ that owns this patch. The argument is inlet number.
- The out object is similar, creating an outlet.
- Thispoly~ is the connection to the owning poly~ for various messages. Most importantly, a valueof zero indicates the subpatch is idle and available to generate a new note or figure. (A non-zero value indicates the subpatch is busy.)

Another change to this patch is I have doubled the resolution. All 320s have become 640s and all 240s are now 480. That also applies to the contents of drawOval2.

All that is needed to make bubble work is the mechanism in figure 16.

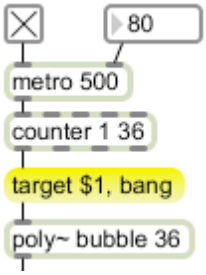

Figure 16.

The poly~ object takes two arguments: the name of the patch to enclose and the number of instances needed. In this case bubble will be loaded 36 times, so 36 expanding circles

can be displayed at once. The output of poly~ will be the combined output of all of the enclosed instances of bubble.

There are several ways to communicate with the enclosed patches. Target is the most flexible. Each instance of the enclosure has a number. The message target n makes instance n the target of any following messages. In the case of bubble, all that is required is a bang. (Remember, a message box with a comma sends two distinct messages.) The counter in figure 16 will step through all 36 of the bubble patches fast enough to keep the screen filled with color.

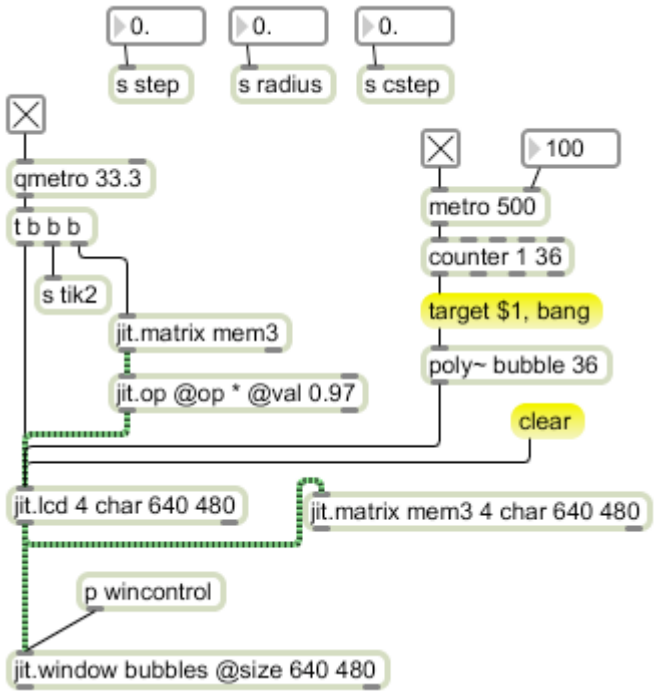

Figure 17.

Figure 17 shows the owing patch. I have replaced the jit.pwindow with a jit.window object for the higher resolution display. I have also added the ability to expand the window to full screen, controlled by the escape key. The mechanism for this is in the subpatcher wincontrol, shown in figure 18.

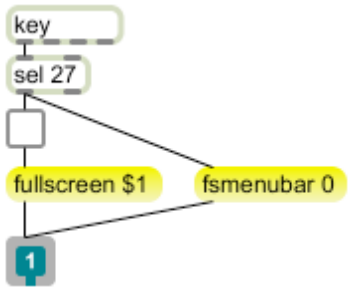

Figure 18.

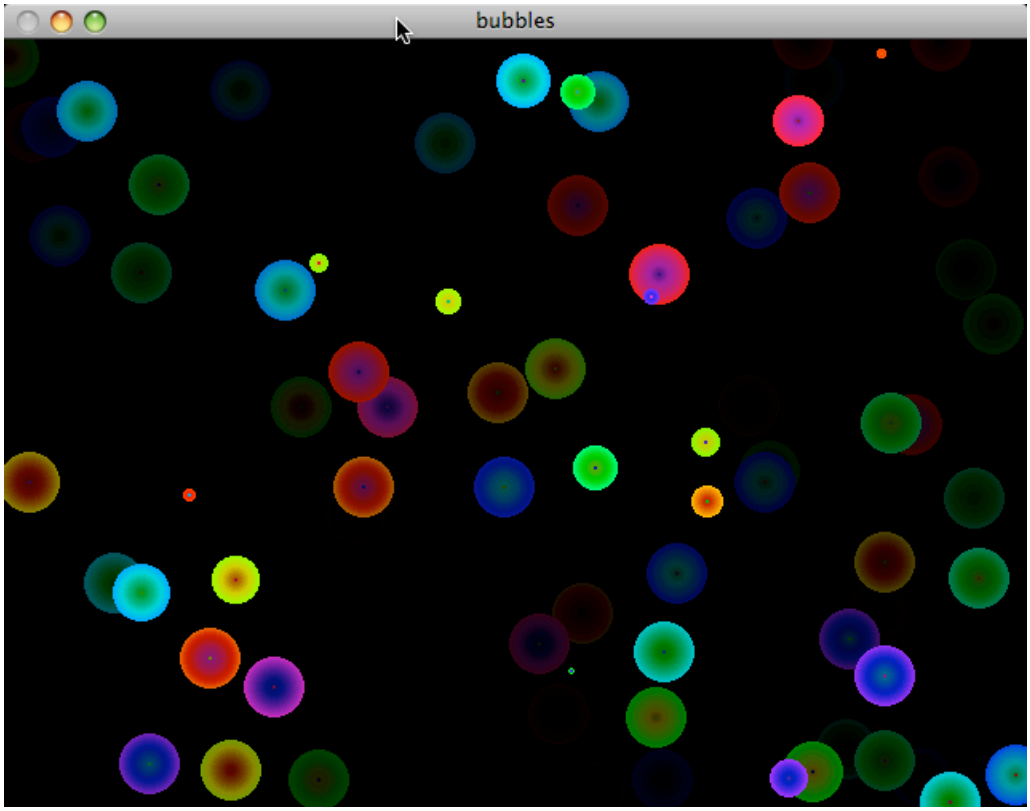

Figure 19.

Figure 19 is the output.