

## Pre-Jitter Studies

Things you need to know before you use Jitter.

### ***How computer programmers manage large chunks of data***

One of the advantages of advanced programming languages is that the programmers do not need to keep track of the actual memory address where anything is stored. If we need to park a number and get it later, we can just give it a name. (Names used for this purpose are called variables.) When we want to group a lot of data together, say all of the numbers that make up an image, we can also use a name. One of the common ways to do this is with an array. When we want to get a particular value from the array, we can access it by the array name and a number that indicates the location of the value. You often see something like `thisArray[5]` to refer to a value. The number five here is the address (in `thisArray`) of the value we want. The actual location of the number in memory is unimportant, but `thisArray[5]` usually means the number found by adding 5 to the address of `thisArray`. If you think about it for a moment, that means the first number in `thisArray` is found at `thisArray[0]`.

Sometimes we want to organize data in a meaningful way, such as a table. We do this with a 2 dimensional array. (Think of a table or spreadsheet.) Now two numbers are used for the address, as in `Array2D[2,5]`. The locations are still found by adding to the array base address, so the first entry is at  $[0,0]$ <sup>1</sup>. Arrays can be expanded to three dimensions or beyond. Egg cartons are a good metaphor for arrays-- they come in one dimension, two dimensions, even three dimensions when you stack them up.



Figure 1.

Arrays can have as many dimensions as you need, although orders beyond 3 are difficult to visualize. I just think of them as arrays of arrays. Jitter does something like that, allowing more than one value per cell. You can think of that as what's in the egg.

### ***About computer graphics***

You probably know the screen is comprised of millions of tiny dots called pixels. Each pixel has three parts (visible with a strong magnifying glass): Red, Green and Blue, which combine to give the appearance of subtly shaded colors. The pixels on computer screens are square, but the pixels in DV cameras are slightly narrowed (or stretched for

---

<sup>1</sup> Of course the second number has to be multiplied by the width of the array first, but that's not important right now.

PAL), which requires some fiddling in the conversion. The intensity of each part of the pixel is controlled by an eight bit number<sup>2</sup>, so twentyfour bits determine the color.

Traditional (NTSC) TV sets have 525 scan lines, but 39 don't show, leaving 486 visible on the screen. The horizontal resolution varies with the tube, but is measured as if there were vertical lines. A good TV will have a horizontal resolution of 648, showing the same amount of detail left to right as top to bottom. HD TV is either 1280 by 720 or 1920 by 1080. Computer monitor resolutions go up with each model, but we pick the resolution in use based on how fine print appears. Choosing higher resolutions shows more picture rather than better picture. For a given video display adapter (video card) you get a list to choose from. Sometimes you will see certain choices recommended. These are the ones that look best given the actual number of pixels on the monitor. If you have a screen with 1024 pixels across, you can easily show 512 by 384 just by lighting up four pixels on the screen for each one in the image. But to show 640 by 480, some kind of fudging is required. This is done by resampling the image, which may give artifacts like unevenly spaced lines or jagged diagonals.

The shape of a screen is called aspect ratio and measures width to height. Movie aspect ratios range from 1.33:1 (or 4:3) for old standard to 2.35:1 for widescreen. Standard TV is 1.33:1 and HDTV is 16:9. Still photographs are also commonly 1.33:1 and 16:9.

The motion in motion pictures comes from projecting a new image or frame 24 times a second. In American TV the images are changed approximately 30 times a second. The eye perceives smooth motion at these rates, but going slower than 20 frames per second will produce flicker. Many computer monitors run 70 frames a second or faster. TVs actually update 60 times per second, but are only showing half the picture at that rate. They paint the even lines on one pass and the odd lines on the second pass, a process called interlacing. A screen that does not interlace is called progressive scan. Interlacing is often indicated by a letter after the vertical resolution-- 720i or 720p.

Images are typically stored (or synthesized) in one of two ways. There may be a list of the color values for each pixel in the image. This is often called a "bitmap" from the black and white days when a pixel was on or off. The more accurate "pixelmap" never caught on. This huge list (921,600 bytes for standard resolution) is usually organized as an array of 640 x 480 x 3 bytes to make the address of the bytes easily related to their location on the screen. In Jitter, a video image is stored in a 2 dimensional matrix that has four "planes". This gives a cell for each pixel location (Horizontal, Vertical) with bytes for red, green, and blue as well as a fourth for opacity. Opacity is called alpha and can be manipulated when images are combined. (Some sources talk about transparency, but use the same numbers for alpha. Thus an alpha of 1 is not transparent.)

The other common way of storing an image is to give instructions for drawing it. This requires much less data storage, especially if the image is pretty much the same from frame to frame. The format for such instructions is rather like a connect-the-dots drawing. Points are specified that represent the corners or vertices of a shape, and the shape is filled in with a specified color. This can easily be done in 3D by using 3 numbers to

---

<sup>2</sup> An eight bit number is called a byte or char.

specify each vertex. Three vertices determine a triangular facet, and the facets can be assembled into an object of any complexity. One of the beauties of this system is that moving or rotating the object requires changing only a few numbers. Of course such an image must be converted (rendered) to pixel values for display on the screen. This is a lot of computing, but most video cards contain a dedicated microprocessor (3D accelerator) to do it. There is a standardized format for drawing instructions called OpenGL, which Jitter understands.

The bitmap approach to graphics is called raster imaging, and the drawing instructions approach is called a vector image. As general rule, we use raster techniques to process video from live sources and do animation with vectors. It's easy to convert a vector image to a raster, so both kinds can be combined in the final result. There's no easy way to convert a video image to a vector image, but a bitmap can be applied as a texture to a vector surface.

### ***Some colorful terms defined***

Luminance is the intensity of a video image or pixel. It's the sum of the R G and B values, weighted for the different sensitivity of the eye to those colors. Luma is similar, but calculated including Gamma Correction which compensates for the nonlinear response of screen pigments and other electrical components in the system. Brightness is the average of the R G and B values. In broadcast TV, luminance is the black and white information of the picture.

Contrast is the ratio of the luminance of the brightest parts of an image to the darkest.

Chrominance or Chroma is the color information signal of broadcast TV. It is actually the difference between the color of a pixel and a reference color (dark green). The actual way this is done varies around the world, but the pertinent fact is that if chroma is 0, there is no color. In jitter, chroma of 1 is the natural color. The European broadcast TV system is often called YUV encoding (as opposed to YIQ for American system). Luminance is Y and chrominance supplies the UV (or IQ) part.

Color space is a method for describing the colors available in a system as well as the method of encoding color. RGB and YUV are examples, but there are many others, including CYMK used for printing and commercial systems like Pantone. Many systems illustrate color space with a color wheel or some three dimensional variant. HSV (Hue, Saturation, Value) for instance is usually illustrated with a circle and a triangle.

Hue is the color regardless of its intensity. If the colors are placed around a circle in spectrum order, hue is the angle measures at the center.

Saturation is the intensity of a color. Low saturations are gray, high saturations often overload the screen display and "bloom" or glare.

Value is the brightness of a color. If you set saturation to 0, Value determine the shade of gray you get. It's also known as level.

## ***Some things to know about Jitter in general***

Jitter is an extension of Max consisting of a batch of specialized objects and a library of code the objects can access. Most jitter objects are named “jit dot something”, a naming convention we started to see when the number of Max objects exceeded a thousand. Most jitter objects are extremely complex; so much so that the old system of a list of numbers to initialize object and special purpose inlets to adjust parameters just won’t work. Instead there is a system of commands and attributes. For instance, the `jit.rota` object rotates images. There is no rotation angle inlet—to change the rotation, you give it the command “theta n” where n is the angle of rotation. To start out with a fixed rotation, include `@theta n` in the arguments. Either will set the theta attribute to n, and `jit.rota` will rotate any input by n radians. Most jitter objects will provide information out of the right outlet in response to the `getattributes` message (a listing of attributes) or the `getstate` message (current settings of all attributes).

Jitter is designed for the efficient manipulation of large batches of data. The general container for this data is the `jit.matrix`. The arguments to `jit.matrix` set the name of the matrix, the number of values per cell, the type of data to store, and the size of each dimension. So `[jit.matrix myData 4 char 320 240]` is a matrix called `myData`. When a matrix receives a bang the only thing output is the name, in the message “`jit_matrix myData`”. Objects that do jitter processing will go to the original `myData` array and change what is found there. This is much more efficient than moving bytes from object to object. Another benefit of matrix names is that you can have another `jit.matrix myData`, in your patch and it will refer to the same data. Most objects will pass along the name once they are through processing, so the data in a matrix connected to a string of objects is processed in the same order as if the data were passing through. You don’t have to give a name to matrices-- Jitter will supply one if you don’t.

The data type can be char (8 bits), long (32 bit integer), float32 (32 bit floating point number) or float64. The ability to specify type not only saves space, it guarantees that operations will be as efficient as possible. It is much faster to process a couple of chars than two longs with 24 0s in each. When each operation is done more than a million times per frame, this kind of detail is important.

An array with multiple values in each cell may seem a bit odd to seasoned programmers, but it is useful. Instead of thinking of a three dimensional array, users can envision a 2D array with cells that each contain 4 values. The only time planes are specifically addressed is if you want to work with just one color value of the pixels. Some jitter objects let you extract arrays of just one of the cell values. These are called planes. If the matrix contains cells with rgba values, plane 0 is the red plane. Other graphics programs may refer to the red plane as the red channel, but audio people have enough channels in their lives.

## ***The Jitter Video Chain***

Jitter uses QuickTime for most video chores. (PC users will have to install QuickTime for Windows before using Jitter.) Video input is done with `jit.qt.grab`, and movie playback with `jit.qt.movie`. Both of these require two arguments to set the resolution of the image. 640 by 480 is a pretty decent screen resolution, but that means at least 1,228,800 bytes of

data must be processed within the 33ms between each frame. This is possible, but you will usually find a 320 x 240 resolution to be more comfortable, giving you four times the performance. (I usually develop patches at 320 x 240, then make a high resolution version of things I like.) You should realize from the beginning that if Jitter can't finish the processing of a frame in time, frame rate will slow down.

If you give jit.qt.movie a name for an argument, it will render the video directly to a jit.window of that name. Otherwise the output is a matrix message.

Jit.qt.movie has enough commands and attributes to actually edit a mov file. The most important are read, start, stop, and loop (0 for off, 1 for on, 2 for back and forth.) Once a mov file has been read, bangs will cause video frames to be output. The frame that you get is the one that is appropriate to the elapsed time since the movie was started or loaded. The start and stop affect the running of the jit.movie clock, not the output of matrices. That's dependent on bangs. You get a frame per bang, so if you are banging faster than the frame rate of the movie, you will get repeats of the same image. It's probably best to use a metro rate close to the actual rate of your monitor, even though many examples use 2 ms. You should also probably use qmetro instead of metro to provide the bangs. If you use metro, heavy processing will make interface objects sluggish. The qmetro object prevents this by giving jitter processing lower priority.

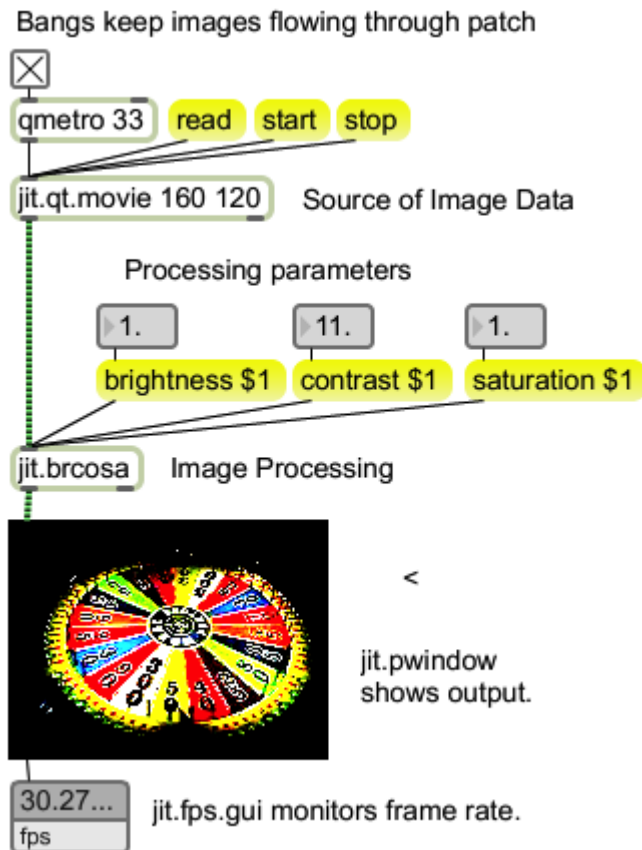


Figure 2.

Matrices output by `jit.movie` or `jit.grab` are of type 4 char, with the dimensions of the resolution. They can be processed by dozens of objects, from simple math to complete effects packages.

After the images are processed, they can be shown in an object called `jit.pwindow` (the palette object with the picture of a cat) or `jit.window`, which opens a new window with just the image. The images can also be immortalized by `jit.qt.record` or `jit.vcr`.

### **Showing your work**

When developing patches, we use `jit.pwindow` to show images. `Pwindow` is found in the object palette, where it has a picture of a cat. There is an inspector for `jit.pwindow` that lets you set its size and a few options. It's generally best to use the inspector to set the window size, because it's difficult to accurately resize it with the mouse. I also usually set the border width to 1, so the `pwindow` is outlined when it's white.

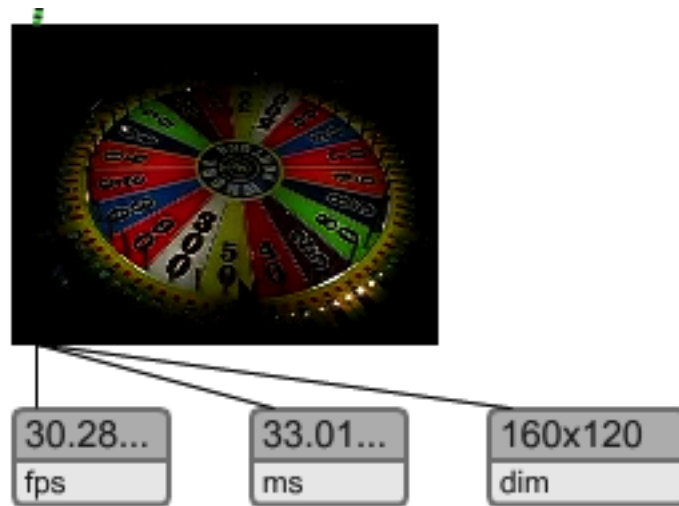


Figure 3.

We often follow the `pwindow` with `jit.fpsgui`, which can (by virtue of a drop menu) show various information about any matrix. `Jit.fpsgui` is also in the palette, labeled `fps`. `Fps` is the most interesting parameter to watch, as it can tell us a lot about the health of our patch.

### **Jit.window**

For serious display, use `jit.window`. This opens a display window outside of the patch. You should give all `jit.window`s names, because the name is displayed in the title bar, and if you don't specify, jitter will call it something like `u87800059`. To position a `jit.window` use the `@rect` attribute to start out and the `rect` command to resize the window. You can change its position with the `position` message, specifying the top left corner.

`Rect` takes four numbers. The first two are the location of the upper left corner of the screen image (not the title). This is the count of pixels right and down from the upper left corner of the primary display. The other two numbers are the position of the lower right corner. (Technically, the corner falls between pixels, so this is the address of the pixel just outside and below the window.) The size of the window is the right minus the left by

the lower minus the upper. To center the window on the display, subtract half the window size from half the display size and use that for the upper left coordinate. So to center a 320 by 240 on a display 1024 by 768, use `@rect 352 264 672 504`. To expand it to 640x480 send the message `rect 192 144 832 624`.

Other important flags for `jit.window`.

- `fullscreen` -- expands the window to fill the display.
- `fsmenubar` – shows the menubar in fullscreen mode. Defaults to 1.
- `floating` – make the window float over all other applications. If it is not floating, the front command will bring a hidden window out.
- `visible` – visible 0 will make it disappear.
- `border` – if 1, the title bar will show. You can't have a floating window with no title bar.

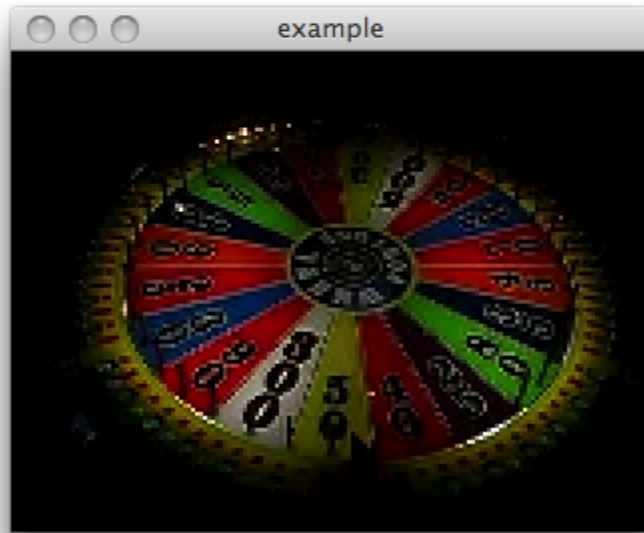
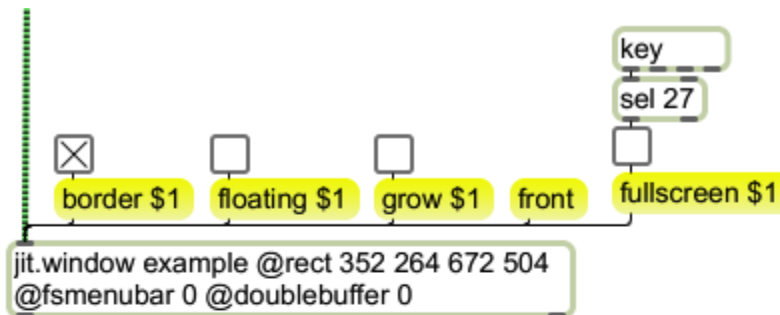


Figure 4.

Figure 4 shows one way to use `jit.window`. Notice that a key (escape) is used to toggle fullscreen mode. Otherwise it would be difficult to get back to the patcher window.

`Jit.window` can be sent to a second monitor (my favorite way to use it). If the monitor is set to the left of the main monitor with a resolution of 800 by 600 `@rect -800 0 0 600` will put the `jit.window` there. If you don't know where the other monitor will be, you can use the `jit.displays` object to find the coordinates. The message cords 1 would get the

message cords 1 -800 0 0 600 in the case described. Jit.displays can do anything the displays preferences panel can, and a bit more.

### **Resizing images**

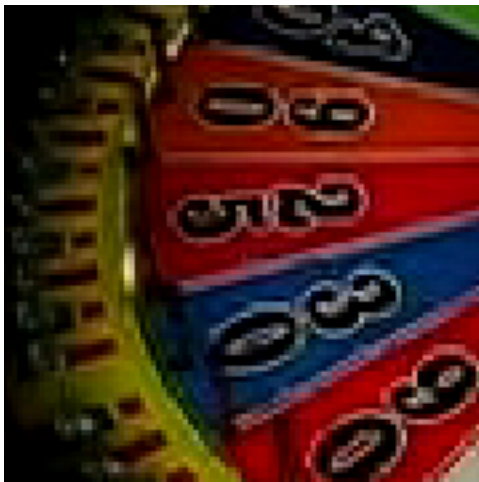
When a small image is sent to a large jit.window, the image is resized without interpolation. If there's much enlargement, the result can be pretty rough looking. The interp flag turns on interpolation, which is quite good, but will look strange if you try to get beyond 2:1.



160 by 120 to 620 by 480 interp 0



160 by 120 to 620 by 480 interp 1



320 by 240 to 620 by 480 interp 0



320 by 240 to 620 by 480 interp 1

Figure 5.

### **Mouse operations**

The jit.window can be dragged and resized with the mouse. To find out where the window is, use the getstate message, with route pos size on the right outlet.

If you click in jit.pwindow or the window opened by jit.window, the right outlet of the object will produce the message mouse, with position of the click in the window, a flag



for mouse down or up and flags for modifier keys: command, shift, caps-lock, option, control. Messages are sent on mouse down and as the mouse is dragged, with a final message on mouse up. The `idlemouse` message enables `mouseidle` output when the cursor is over the window without clicking.

Testing mouse clicks, you will notice you can't get a response from the lower right corner. That's because there's an invisible grow box there for resizing the window. The `grow 0` message will disable resizing, and you will get clicks from the entire window.

## ***Jitter Efficiency***

Using jitter is a constant battle between great ideas and limited processing power. You will watch the fps meter inexorably grind to a low number as you add modules to your patch. The ultimate solution lies with Apple and Intel, but until 8Ghz 8 core machines are a reality, all you can do is follow these guidelines:

- Keep a clean system. Don't install a scanner, disconnect all unneeded peripherals, and close all extraneous applications. Do install lots of memory.
- Fast drives are key.
- Use activity monitor to check what's running and how much CPU each application takes. If it's 0% don't worry about it, but Max should be the biggest user. As an extreme example, Adobe InDesign can easily gobble 40% of the CPU time on a 2 GHz machine.
- Remove all unneeded Max UI objects. Use `[t b]` instead of `button` to convert messages to bangs (It doesn't even matter if they show on the screen). Don't hang number boxes all over, use your midi controller (really-- the number box has to be constantly monitored, Midi only takes attention when it comes in.)
- Keep the use of pwindows to a minimum.
- Keep your MSP operations efficient. MSP can easily use just as much CPU as Jitter. Jitter is more likely to suffer, as graphics have a lower priority.
- Turn overdrive off. Don't use scheduler in audio interrupt if you can help it.
- Use `qmetro` instead of `metro`. `Qmetro` waits until processes have finished before banging. `Metro` interrupts other actions, which uses more processing time in the long run.
- Set the `qmetro` to 33 ms. This will give you a frame rate of 30 fps tops, but there will be less variation than if you try to go faster. The FSPGUI is only counting bangs out of the window, not what's drawn to the screen. The screen is actually updated at 70 fps or so but we don't mind seeing the same frame twice at that rate. There's no point in calculating images you never see, and other parts of the patch will be grateful.
- Never use a complex object if `jit.op` will do. Complex objects are... complex, and they may be doing unneeded operations like multiplying by 1 or copying matrices. Using `jit.brcosa` to control brightness rather than `jit.op` cost 10 frames on an old iMac.
- Never use two objects if the functionality of one is built into another. You wouldn't use `jit.rota` and `jit.resamp` together, since `rota` has a zoom feature.
- Bypass temporarily unneeded objects. If you set `theta` to 0 and `zoom` to 1, nothing is happening, so use a gate to bypass the `jit.rota` object. (Don't use a switch, that

would feed both objects at the same time. Jitter objects do their processing when they get a message, whether there's anything hooked up or not.)