MSP / Jitter Quick Start

being an overview, by no means comprehensive, of the digital media processing capabilities of Max/MSP/Jitter, organized into functional groups of objects with brief annotations.
Based on Max/MSP/Jitter version 5.0.5. Randy Jones, January 2009.

1 INTRODUCTION

The Max visual programming environment is a very effective way to work with digital media, in applications from prototype to finished product. Max was created in the mid 1980's, and its flexible message-passing architecture has enabled it to evolve fairly gracefully since that time, incorporating features unimagined at the time of its initial design. MSP and Jitter are groups of Max objects that deal with signals and matrix data respectively. Each started out as a separate product that extended Max, but both extensions are now becoming more and more part of the standard distribution.

With all of the options available in Max/MSP/Jitter come a fairly steep learning curve. Basic programming in Max itself is relatively quick to learn, but just as there is no way around studying vocabulary when learning a new language, getting to know a large number of the roughly 500 objects in Max/MSP/Jitter is necessary in order to use it well. This document provides a list of some MSP and Jitter objects divided by functional groups. It is by no means comprehensive. I have tried to mention the most useful and commonly-used objects in each of some main categories. After finding an object in a given category, the help file for that object, in particular its "See Also" section, is a good gateway to further possibilities. A basic knowledge of Max patching, such as can be gained from going through the first five Max tutorials or so, is assumed here for brevity.

Many objects can be found by category in Max using the New Object List. This is a text list, different from the UI object palette that appears when the canvas is double-clicked. To use the New Object List, press 'n' to create a new object, then move the mouse to hover over the lower right corner of the object. An equals sign in a blue circle should appear; clicking on this little guy brings up the New Object List.

2 MSP

MSP and its many objects allow digital signal processing (DSP) within the Max framework. DSP is fundamentally different from the message-based processing done by Max. Every signal is an *isochronous* stream of samples: each sample must follow the next at a constant rate, typically 44,100 or 96,000 samples per second. The computer must do all of the DSP calculations fast enough to keep an audio interface suppled with samples at this rate, or we will hear clicks and pops in its output. MSP objects have names ending in a tilde (~) to help avoid confusion with Max objects.

2.1 Audio Fundamentals

We can represent any signal such as an audio channel in two complementary ways: the *time domain* and the *frequency domain.* Signals in the time domain are sequences of samples one after the other; each sample represents an instantaneous value of the signal. Sending the stream of samples to a digital to analog converter using the `dac~` or `ezdac~` objects lets us hear them as sound. Attaching a `scope~` object lets us see a graph of the signal's samples similar to what an oscilloscope would produce. Use this object with some caution though: it skips samples of the signal in order to draw fast enough, and is not as reliable a diagnostic tool as a real oscilloscope.

The frequency domain gives us an alternate representation for sound that more closely parallels our own perception. The *Fourier theorem* states that any continuous, periodic signal can be exactly described as a sum of sinusoidal waves of different frequencies. Where a time-domain graph of a signal shows its changing amplitude over time, a frequency-domain graph shows the amplitude of each sinusoidal frequency that makes up the current signal over a very short time window. The `spectrogram~` object lets us see a signal in the frequency domain.

A sampled signal can store information at a range of frequencies from 0 Hertz (a signal with a constant value) to one half the sampling rate. If a signal contains frequencies outside of this range, and is sampled without proper filtering, *aliasing,* or a non-physical folding of harmonics, will result. Listening to the output of `phasor~`, a geometric sawtooth wave, versus `saw~`, an antialiased sawtooth wave, can give us some feel for what aliasing sounds like. In scientific

measurement, we always want to avoid aliasing. But in music, we don't always—in the pursuit of new and interesting sounds, anything is fair game. MSP even has a special object, degrade~, for introducing aliasing and other distortions.

The adc~ and dac~ objects, and their graphical cousins ezadc~ and ezdac~, transfer digital audio between the computer's audio hardware and MSP patches. The DSP Status window shows the global options that can be set for audio processing, including the sampling rate. The audio hardware being used determines the global sampling rates that are allowed. For efficiency, DSP calculations are done on small batches of samples in each object, rather than on a single sample at a time. The batch size used can be set as a global option, and is called the *signal vector* size. Setting the signal vector smaller results in less overall latency, but increases the CPU time that a patch will take to run.

Using the poly~ object, audio subpatches can be made that run faster or slower than the global sampling rate. The poly~ object also allows multiple copies of a subpatch to be run at once, a very handy feature in an environment like Max, where each copy of a patch would otherwise be a distinct object on the canvas.

2.2 Generators

Audio synthesis is usually defined as the generation of sounds "from scratch," without incorporating any acoustic sources. Synthesis patches begin with objects that generate signals, such as cycle~, saw~, or noise~. The cycle~ and saw~ objects each take a frequency as an input parameter, and produce an output waveform at that frequency. Two more such waveform objects are tri~ and rect~. The rect~ object has an adjustable pulse width.

The noise~ object takes no parameters and produces a steady output of white noise: all frequencies with an equal amplitude. The noise made by pink~ is similar, but the high frequencies are attenuated compared to the lower ones, giving it a less harsh sound.

The train~ object is a generator that produced an evenly spaced train of pulses. The time between pulses as well as the duration of each pulse can be set with input parameters.

The oscbank~ object is a more complicated kind of signal generator that makes many sine waves at once. It can be used for *resynthesis,* making a synthesized reproduction of a previously analyzed recorded sound.

The sig~ object is the simplest generator of all, but a very useful one: it just generates a constant (DC) signal at the amplitude set by the input parameter.

Strangely, none of the above objects are listed in the "MSP Generators" category in Max's New Object List, but they can all be found under "Synthesis."

2.3 Filters

A filter is anything that transforms a signal. So, to say that a synthesis patch is made of signal generators followed by filters is almost a tautology, but it's a good way of categorizing different kinds of objects in MSP as well as Jitter. The words 'filter' and 'effect' are sometimes used interchangeably. While there exists a mathematical definition of what a filter is that includes both intentional and unintentional, perceptible and imperceptible changes, when we talk about effects we are always talking about changes in some aspects of signals we can perceive, and usually about specific kinds of changes with names like delay, flanger, and so on. So it's safe to say that an effect is a particular kind of filter.

Audio signals can be transformed in a remarkable number of ways by simply running them through delays of constant lengths, scaling the amplitudes of these delayed copies, and combining them. In MSP, simple filter blocks can be made with the delay~ and *~ objects. We normally think of a delay in terms of a distinct repetition that we can hear. Very short delays of around ten milliseconds or less, though, do not sound to us like repetitions, rather we hear them as changes in a sound's timbre when they are mixed with the original sound. This effect is called *comb filtering,* and there is a special object, comb~ to apply a small delay with feedback. To put arbitrary kinds of changes in a delay with feedback, it is necessary to use the tapin~ and tapout~ objects. Delays made using these two objects cannot be shorter than the length of the DSP signal vector.

Reverb is another effect that is basically just a combination of delays and scaling, with the addition of feedback. By connecting comb filters and allpass filters (allpass~) of the right lengths in a feedback loop, a patch can be made that quickly diffuses an input click into a wash of sound. Most lush sounding reverbs do a few more tricks than than just delay and scaling though, in order to make smooth and evolving decays. See the MSP example reverb_example.maxpat for an example of how this can be done.

Even the general-purpose filter object biquad~, which can be used in combination with the filtergraph object to create and control many different types of audio filters, is just delays and scaling with feedback. Other MSP objects useful for *subtractive synthesis*, filtering out frequencies from a harmonically rich input, include svf~, lores~, and onepole~.

Some effects that can't be made with just delays and scaling include waveshaping, frequency shifting, and pitch shifting. When an effect or filter can be made entirely with delays and scaling it is said to be *linear.* Those that can't be made in such a way are *nonlinear.* While linear filters can only change the amplitudes of existing frequency components, nonlinear filters may output new frequency components not present in the input. From this description we can see that a pitch

shifter must be a kind of nonlinear filter. The gizmo~ object enables pitch shifting, working in the frequency domain using pfft~. Look at the example pitchshifting-playground.maxpat to see how its approach compares to a pitch shifter made in the time domain. A frequency shifter is like a pitch shifter, but changes input frequencies by a fixed amount of Hertz rather than a fixed ratio. The freqshift~ object does this. Waveshaping can be accomplished in a general way with the lookup~ object, or in a more specific way with kink~.

2.4 Sampling

The simplest way to play back a recorded sound in MSP is by using the sfplay~ object. This object plays sound files from disk into an MSP patch. It plays AIFF and WAV files, and a few other formats, but not compressed audio files like MP3. To play an MP3 file, you first have to import it into a buffer~ object. This object stores all of the data representing a sound in memory at once, and provides the foundation of more flexible ways to play samples. A few objects provide different ways of playing sounds in buffer~ objects. These include groove~, a looping sample player, wave~, a variable size wavetable object, and cycle~, a wavetable oscillator. Listen to the help files and examples for ideas about the different situations these objects can handle.

Sound can be recorded live from the adc~ object into a cycle~. This allows real-time sampling and looping.

2.5 Sequencing

For the most part, sequencing of sounds in MSP is done using Max timing and data-processing objects: metro, coll, counter and so on. There are a few special MSP objects that deal with sequencing though. techno~ is an idiosyncratic, sample-accurate sequencer that's useful for rhythmic music. It comes with a UI helper technoui.js. See the jsui object for more information on Javascript UI objects.

2.6 Analysis

There are numerous ways to get data about different aspects of an audio signal. This data can stay in the signal realm or be sent as messages. Truly sample-accurate timing in MSP can only be achieved if messages are avoided and the entire timing-sensitive portion of the patch is constructed with signals. In practice, the Max scheduler works well enough to an approximately 1ms resolution that this hard-core approach is rarely needed.

The snapshot~ object returns an instantaneous value of a signal when it receives a bang, or based on its internal timer. For rapidly changing signals, this is often not sufficient to characterize the data because transients may be missed entirely. Instead, it's generally better to use the peakamp~ or average~ objects. The change~ object outputs a signal that characterizes the rate of change of an input signal: is it going up, going down, or constant? Note that this object, like a fair number of MSP objects, could be reproduced from lower-level ones. In particular, a one sample delay~ and a -~ can be used to get not just the direction, but the rate of change of a signal, a potentially more useful piece of data to have.

The zerox~ object returnns the number of times the signal crosses zero in a given interval. This can be used as a measure of noisiness, or as a frequency counter for simple signals. The edge~ object, rather than counting zero crossings, treats each crossing as a separate event and outputs a bang. Note that for rapidly changing signals, this could create many bangs in one signal vector, all but one of which will be lost or delayed. The thresh~ object is like textttedge~, but can operate entirely in the signal domain, and allows for separate upper and lower thresholds that can be used to clean up or "debounce" signals.

3 JITTER

Like MSP, Jitter is an extension to Max in the form of a great many objects. As MSP objects end in a tilde, Jitter objects start with "jit.". Jitter objects generally send *matrices,* multidimensional chunks of data, through a patch. Matrices can be used to store video, sound, 3D graphics, or other data, and to convert from one type of data to another. For a full description of matrices, see Jitter Tutorials 1 and 2. The jit.matrix object stores a single matrix of data in memory, and can load and store matrices from hard disk in various formats.

Unlike MSP, data flows through Jitter just like it does in Max. Though you will see special fuzzy lines in your patch when connecting the outputs of Jitter objects that send matrices, each matrix is sent as a regular Max list. This can be verified by sending a matrix to the print object. The symbol jit_matrix is printed, followed by the name of the matrix. The user interface object jit.fpsgui shows more information including the type, dimensions and plane count of matrices sent to it. It also computes the frequency, or frame rate, with which matrices are sent to it.

Jitter added the useful concept of *attributes* to Max. As of Max 5 attributes are now more integrated with all Max objects. An attribute is a named property of an object that can take on a value. Attributes make Jitter programming easier

by giving a common mechanism for setting and querying the many values that affect an object's behavior. See "Jitter Attributes" in the Max 5 Help home page for more information.

3.1 Video Fundamentals

Color video is usually expressed in Jitter as matrices with four planes, two dimensions, and a data type of *char,* eight bits. The four planes of data are in ARGB order: alpha, or opacity, followed by red, green and blue. All of the video objects discussed in this section work on data in this format.

Cropping and scaling data are basic operations on matrices. It's most common to apply these operations to two-dimensional data, but Jitter enables cropping and scaling in two, three and higher numbers of dimensions, all through the `jit.matrix` object itself. Setting the `usesrcdim` attribute of `jit.matrix` tells it to crop its input, by copying data from only a part of an input matrix rather than from the whole matrix. The `srcdimstart` and `srcdimend` attributes set the source coordinates that are used. Likewise, the `usedstdim`, `dstdimstart` and `dstdimend` attributes control the destination coordinates within a matrix to which data is copied. If the source and destination coordinates are of different sizes, the selected part of the input matrix is scaled to match the destination size. In addition to cropping and scaling, many other handy functions including setting and getting data can be accessed just by sending messages to the `jit.matrix` object. See the `jit.matrix` reference for more information.

For taking apart a video or other matrix into a number of equal-sized pieces and putting it back together, the `jit.scissors` and `jit.glue` objects are a good team. Likewise, `jit.unpack` and `jit.pack` disassemble and assemble matrices in a plane-wise fashion. Another pair of complementary objects, `jit.demultiplex` and `jit.multiplex`, take one matrix apart into even and odd rows or columns, and interleave two matrices into one.

3.2 Generators

Like sound synthesis, video synthesis typically starts with one or more generators. The `jit.noise` object is probably the simplest of these. When it receives a bang, it outputs a matrix of random values in any of Jitter's data types, and of the number of planes, rows and columns specified. The *char* data type will get values from 0 to 255, its whole range. The floating-point data types are scaled from 0. to 1.

The `jit.lcd` object lets you draw 2D shapes, and stores the results in an image buffer that can be output as a matrix. The `jit.turtle` object is designed to work with `jit.lcd`, drawing vector-based and procedural designs.

A couple of objects are able to fill matrices efficiently with various functions. The `jit.expr` object is the most general-purpose one. Given strings describing formulas, it iterates over the cells in a matrix determining the value of the formula at each cell. It can send out matrices of any data type, and with an arbitrary number of planes. One can even specify a different equation for each plane of the output matrix. The `jit.expr` object is designed to apply its equations to one or more inputs, so to use it as a generator, one needs to pass it an input matrix. This can be blank, full of noise, or whatever is convenient.

Using `jit.bfg` it's possible to make complicated results, like realistic noise textures, that it would be very hard to reproduce with `jit.expr`. This object has a useful built-in library of complex functions aimed at generating noise and fractals. This approach to generating textures has the prevailing one in high-end computer graphics for a number of years. Many related concepts are explained in detail in the classic computer graphics text *Texturing and Modeling: A Procedural Approach,* by Ebert et al.

3.3 Filters

While an audio signal is a series of single, scalar values over time with no dimensionality, a video is a sequence of two dimensional matrices over time. It's possible in Jitter to make a time-varying sequence of a single row or column of pixels, which would be a one dimensional signal, but uses for this kind of signal are uncommon. With dimensions comes the distinction between filters that operate in the spatial dimensions, and those that operate only in the time dimension, as on an audio signal. We call the former *spatial* filters and the latter *temporal.* An example of a purely temporal filter is `jit.slide`.

Jitter has many video effects built in, both standard ones like `jit.chromakey` and `jit.hue` that you might find on a hardware video mixer, and idiosyncratic ones like `jit.ameba`, `jit.tiffany`, `jit.plur` and more. Choosing "Jitter Special FX" from the New Object List will show a considerable number of these latter effects. Note that the common effect of pixelation is handled, with some optional extra twists, by the `jit.rubix` object. In general, the more flexible and general-purpose effects are grouped in the New Object List by their functions, including Compositing, Colorspace, Blur/Sharpen and Spatial.

When you have a visual filter in mind that doesn't exist as a single Jitter object, and are looking for the right sequence and combination of objects to make it with, `jit.op` is often the best place to start. This object just implements basic math operations in a flexible way. Look at the help file to see the large variety of operations it supports. The `jit.expr`

object is also very handy here. A typical progression of an effect design might be: starting out using individual jit.op objects and rearranging them to get the effect right, then combining some of them into a single jit.expr object by making one expression string that specified the same math.

The objects jit.freeframe and jit.qt.effect both greatly expand the effects available in Jitter by allowing the use of common kinds of plugin effects.

3.4 Sampling

The jit.qt.grab object is for video what adc~ is for audio: real time capture from a live source. With jit.dx.grab the video source is typically the built-in laptop camera, but may also be any QuickTime-compatible video digitizer. With digitizers, as with hardware in general, it's important to note that while the object provides access to many many features, not all digitizers support these features.

The jit.qt.movie object is a very flexible tool for playing QuickTime movies from disk. It supports many arcane features of QuickTime such as effects and matrix transforms. Luckily, it's quite simple to use to play a movie: just send it the message open to open a movie, start to start it, and send a bang every time you want a new frame.

3.5 Analysis

A variety of tools are available to create statistics about Jitter matrices. The simplest to use might be the jit.3m object, and it also does enough analysis for a wide variety of uses. When a matrix is sent to the inlet of jit.3m, it computes the minimum, mean and maximum of all the cells in the matrix, and outputs those values. These numbers can be used to normalize colors in a video, or do simple object detection. Another Jitter object that can detect moving objects in an image is jit.findbounds. Given an input matrix and a range of brightness values, it sends out the minimum and maximum points that contain values in that range. So, given a white object on a black field, it could return the coordinates of the object reasonably accurately.

To detect objects in other kinds of scenes, such as a particular colored object on an arbitrary background, one approach that works is to process the video until it resembles the same problem of a white object against black. Such processing could begin by taking a static image of the background and subtracting it using jit.op @op -. An expression made using jit.expr can determine the distance in RGB color space from the input color to the color of the desired object. Inverting this result, so that close matches appear white rather than black, will turn the arbitrary problem into the simple white on black scenario.

Other useful analysis objects including jit.histogram, jit.fft, and jit.change, are listed under the Jitter Analysis category in the New Object List.

3.6 OpenGL

OpenGL is a device-independent API (Application Programming Interface) for drawing graphics. One set of commands in OpenGL can produce nearly identical images on a wide range of hardware and software platforms from phones to supercomputers. Bindings exist that allow OpenGL to be used in many different languages. Though OpenGL is often thought of as a 3D graphics API, it can be used to draw 2D images as well. On current desktops and laptops, drawing graphics using OpenGL is usually very fast because these computers contain dedicated hardware graphics processing units (GPUs).

Jitter uses OpenGL through a collection of objects with names starting in jit.gl. Currently there are 20 such objects; like the matrix-processing objects described above, they can be categorized into generators, processors and so on. Starting to draw an OpenGL scene is a little more complex than drawing a 2D image, because a 3D scene needs a camera positioned somewhere in it in order to be translated in 2D, and has many other variables that can affect the rendering. A few objects are needed to see any results: a jit.gl.render instance, a jit.gl object to draw, and a rendering destination. The rendering destination is typically a jit.window or jit.pwindow object, but can also be a jit.matrix, in which case the scene is drawn directly into the matrix using a software renderer. The combination of a jit.gl.render object with a named rendering destination is called a *context*.

One common source of confusion in Jitter is that while the color planes in a video matrix are ordered ARGB, the color parameters for OpenGL objects, and the planes of colors in objects such as jit.gl.mesh, are ordered RGBA. This is because the Jitter objects that communicate with each kind of graphics are in the native mode used by that system, and Quicktime video and OpenGL differ in this way. When converting between the two is needed, a jit.pack and jit.unpack pair can do the job.

3.7 OpenGL Generators

Synthesizing OpenGL graphics is really very much like synthesizing sounds or 2D images—we can start with generators and combine them with mathematical operators and filters. Jitter uses some conventions that make generating 3D graphics in real time easier. One of them is that many objects know how to treat a 2D, three plane matrix as a group of points that are connected in a mesh. In OpenGL programming, one typically has to specify each connection between vertices in space as well as the vertices themselves. By letting the gridwise arrangement of vertices in a mesh specify their interconnections, meshes ease the construction of solid 3D shapes. The `jit.gl.mesh` object takes such a vertex matrix as input, as well as other optional matrices specifying texture coordinates, colors, and so on.

While video always flows between objects as matrices in order to be drawn, this is not the default way that Jitter's OpenGL objects work. When an OpenGL object such as `jit.gl.plato`, which draws Platonic solids, is created, it has to be given the name of a rendering context as an argument, and it defaults to appearing in that context without its vertices needing to flow through the Jitter patch. When creating the many objects that for a typical 3D scene, this is obviously a great convenience. Putting it broadly, accessing vertex data directly in matrices is treated in Jitter as an optional and more detailed way of approaching 3D drawing. To enable it, an attribute named `matrixoutput` of many Jitter OpenGL objects can be turned on (set to 1). With `matrixoutput` on, objects send their geometry data as a matrix out of their left outputs, requiring your to send the data to the renderer in your patch. Another simple object that can create various shapes is `jit.gl.gridshape.`

The `jit.gl.model` object can read in 3D model files in .obj format, and output the results, either directly to the renderer or as a vertex matrix. A few sample models come with Jitter, and many free models can be found on the net. The free, open-source software Blender can create .obj files compatible with Jitter. Models can store textures and material parameters as well as geometry, though textures must be in separate files that are referred to by the .obj.

Another way to generate 3D forms is in matrices with some of the same objects that can generate images, such as `jit.noise` or `jit.expr.` Entering `jit.noise 3 float32 16 16` into an object box will cause it to output three plane, 16 by 16 matrices of float32 data. When sent to a `jit.gl.mesh` object, these are interpreted as connected random vertices and something that looks like a spiky ball is drawn. Given the right equation strings, a `jit.expr` object can send out matrices that draw an approximation to any simple, convex 3D shape.

The `jit.gencoord` object is designed for, yes, generating coordinates. On receipt of a bang it returns a 2D grid of values, each of which is scaled linearly across the range of the grid. This is handy as a starting point for both 3D shapes and 2D images.

3.8 OpenGL Filters

Due to Jitter's flexible architecture, effects that are primarily designed for processing one kind of data can by tried out in other situations. All of the video effects built into Jitter can be used to manipulate OpenGL shapes as well. For those building patches in an exploratory mode, these can be a source of interesting and unexpected effects. Simple math operations can be used to create more predictable effects. For example, adding noise to each vertex of a 3D shape does what you might think it would. Transformations can be made with `jit.expr` including mathematical distortions and different coordinate spaces. In the example file `jit.gl.render.sphere,` a network of objects is shown that changes 3D xyz into 3D polar coordinates. This can also be done more simply and efficiently with a single jit.expr object, as shown in the file `viz_poltocar.js.`

Jitter lacks many objects specifically designed to produce 3D distortions, because around the time these might have been added to the program a better solution became available. *Shaders* are a certain type of program designed for running on GPUs. They can manipulate either 2D pixels or 3D geometry. Because they run entirely on the graphics processor they can be much more efficient than the equivalent software on the CPU. Jitter includes an object, `jit.gl.shader,` that allows shaders to be loaded and attached to Jitter GL objects. Writing shaders is not for the faint of heart, but Jitter includes a large library of them, and many more can be found on the net.

There is no section here on OpenGL Analysis because, unlike sound and video, when we have some OpenGL data we generally know what it represents. Someday, when we have cameras that attach to our computers, viewing real-world scenes in stereo and generating 3D graphics as output, we may have more of a need for this category of object.