


JITTER




Tutorial

Version 1.6 / 7 August 2006

Table of Contents

Table of Contents	2
Copyright and Trademark Notices.....	12
Credits	12
About Jitter	13
Video.....	13
2D/3D Graphics.....	13
Ease of Use.....	14
Matrices.....	14
More Details.....	14
How to Use The Jitter Documentation	16
What is a Matrix?	17
A Video Screen is One Type of Matrix	18
What is a Plane?.....	20
The Data in a Matrix.....	21
Attributes: Editing Jitter object parameters	23
What are attributes?.....	23
Setting Attributes	24
Jitter Object Arguments.....	25
Querying Attributes and Object State.....	26
Summary	28
Tutorial 1: Playing a QuickTime Movie.....	29
How Jitter Objects Communicate.....	30
Causing Action by Jitter Objects.....	30
Arguments in the Objects.....	32
Summary	33
Tutorial 2: Create a Matrix	35
What's a Matrix?.....	35
The jit.matrix object.....	35
The jit.print object.....	36
Setting and querying values in a matrix.....	37
The jit.pwindow object — 	39
Filling a matrix algorithmically	40
Other messages to jit.matrix	41
Summary	41
Tutorial 3: Math Operations on a Matrix	43
Operation @-Sign.....	43
Math Operations on Multiple Planes of Data.....	45
Modifying the Colors in an Image.....	46

Sizing it Up	49
Summary	50
Tutorial 4: Controlling Movie Playback	51
Obtaining Some Information About the Movie	52
Starting, Stopping, and Slowing Down	53
Time is on My Side	54
Scrubbing and Looping	55
Summary	56
Tutorial 5: ARGB Color	57
Color in Jitter	57
Color Components: RGB	57
The Alpha Channel	57
Color Data: char, long, or float	58
Isolating Planes of a Matrix	58
Color Rotation	60
Automated Color Changes	62
Summary	63
Tutorial 6: Adjust Color Levels	64
jit.scalebias	64
Math with char data	64
Some more examples of math with <i>char</i> data	66
Adjust Color Levels of Images	67
Adjust planes individually	69
Reassign planes of a matrix	70
Reading and Importing Images	71
Summary	72
Tutorial 7: Image Level Adjustment	73
Brightness, Contrast, and Saturation	73
Hue and Cry	76
Summary	77
Tutorial 8: Simple Mixing	78
Mixing Two Video Sources	78
jit.xfade	78
Automated Crossfade	79
Summary	80
Tutorial 9: More Mixing	81
Mixing and Crossfading Made Explicit	81
Mixing Revisited	81
Combine Matrices Using Other Operators	82
jit.scalebias vs. jit.op @op *	83
Summary	84
Tutorial 10: Chromakeying	85
The jit.chromakey object	87

The suckah object — 	88
The Blue Screen of Death	89
Summary	92
Tutorial 11: Lists and Matrices.....	93
Matrix Names.....	93
jit.fill	93
The offset attribute	94
Using multiSlider	95
Using zl	96
jit.fill with Multiple-plane Matrices	98
jit.fill with 2D Matrices.....	99
jit.spill	100
jit.iter.....	101
Summary	102
Tutorial 12: Color Lookup Tables.....	104
Lookup Tables.....	105
Generating the Lookup Table	106
The jit.gradient object	108
Summary	111
Tutorial 13: Scissors and Glue	112
Cut it Up.....	113
Routing the Matrices	114
The Glue That Keeps It Together.....	116
Summary	116
Tutorial 14: Matrix Positioning	118
Positioning Data in a Matrix.....	118
jit.window	118
From one jit.matrix to another.....	119
Interpolation	120
Isolate a Part of the Matrix.....	121
Flip the Image	123
Resize the Output Matrix.....	124
Moving the Image Data Around in the Matrix.....	125
Changing, Resizing, and Moving the Source Image.....	127
One More Word About Dimensions	127
Hue Rotation.....	127
Full Screen Display.....	129
Summary	130
Tutorial 15: Image Rotation.....	132
Rotating and Zooming with jit.rota.....	132
Basic Rotation.....	132
Automated Rotation.....	133

Zoom In or Out.....	134
Beyond the Edge	135
Some Adjustments—Anchor Point and Offset	136
Rotary Control	136
Summary	139
Tutorial 16: Using Named Jitter Matrices	140
Order of Importance.....	141
What's in a Name?.....	142
The Destination Dimension.....	143
Jumping the Queue	145
Summary	146
Tutorial 17: Feedback Using Named Matrices	147
Jitter Matrix Feedback.....	148
The Game of Life	148
Summary	149
Tutorial 18: Iterative Processes and Matrix Re-Sampling.....	150
Getting Drunk.....	151
The Feedback Network	152
Downsampling and Upsampling	154
Summary	156
Tutorial 19: Recording QuickTime movies	158
Your mileage may vary.....	158
On the clock	159
Off the clock	162
Summary	164
Tutorial 20: Importing and Exporting Single Matrices	165
Import and Export from the jit.matrix object	165
QuickTime export and import.....	165
Jitter binary export and import.....	168
Import and Export from the jit.qt.movie object	169
Exportimage.....	170
General Export from jit.qt.movie	171
The jit.textfile object	173
Summary	177
Tutorial 21: Working With Live Video and Audio Input	178
The Basics of Sequence Grabbing	178
First grab.....	179
Switching between inputs	180
Grabbing for quality	182
Grabbing to disk.....	184
Grabbing video to disk	185
Summary	187
Tutorial 22: Working With Video Output Components.....	189

End of the Line.....	189
Just Passing Through.....	192
Summary	193
Tutorial 23: Controlling Your FireWire Camera	194
Plug and Play.....	194
Basics	195
PLAY and WIND groups (VTR mode)	196
RECORD group	198
Summary	200
Tutorial 24: QuickTime Effects.....	201
The Dialog Box Interface	201
To the Max.....	204
Listing and loading effects	204
Parameter types.....	205
Listing parameters	206
In practice.....	207
Making changes to parameters	207
Tweening	209
Saving and Loading Parameter Files.....	211
Using QuickTime Effects in QuickTime Movies	211
Summary	214
Tutorial 25: Tracking the Position of a Color in a Movie	215
Color Tracking.....	215
jit.findbounds.....	215
Tracking a Color in a Complex Image.....	217
Using the Location of an Object.....	220
Playing Notes.....	220
Playing Tones	222
Deriving More Information.....	223
Summary	224
Tutorial 26: MIDI Control of Video.....	225
The MIDI–Video Relationship.....	225
Mapping MIDI Data for Use as Video Control Parameters.....	225
Using send and receive	228
Using MIDI Notes to Trigger Video Clips.....	229
Routing Control Information	230
Routing Around (Bypassing) Parts of the Patch	232
User Control of Video Effects.....	234
Summary	236
Tutorial 27: Using MSP Audio in a Jitter Matrix.....	238
The Sound Output Component	239
Poke~ing Around.....	241
Sync or Swim.....	243

Putting it all Together.....	245
Summary	246
Tutorial 28: Audio Control of Video	247
Audio as a Control Source.....	247
Tracking Peak Amplitude of an Audio Signal	247
Using Decibels.....	249
Focusing on a Range of Amplitudes	250
Audio Event Detection	251
Using Audio Event Information.....	254
Summary	255
Tutorial 29: Using the Alpha Channel	257
The jit.lcd object	258
Make Your Own Titles.....	261
The Alpha Channel.....	262
Summary	264
Tutorial 30: Drawing 3D text.....	266
Creating a Drawing Context	266
GL Objects in the Context.....	267
Common 3D Attributes.....	269
Summary	273
Tutorial 31: Rendering Destinations	274
Drawing and Swapping Buffers.....	274
Setting Fullscreen Mode.....	275
Setting a jit.pwindow Destination.....	275
Setting a jit.matrix Destination.....	276
Multiple Renderers and Drawing Order	278
Summary	279
Tutorial 32: Camera View	280
Summary	287
Tutorial 33: Polygon Modes, Colors and Blending.....	289
Wireframe Mode and Culling Faces.....	289
RGBA Colors.....	292
Erase Color and Trails.....	293
Blend Modes.....	294
Antialiasing.....	296
Summary	297
Tutorial 34: Using Textures	299
What is a Texture?.....	299
Creating a Texture.....	300
Textures and Color	301
Converting an Image or Video to a Texture.....	302
Interpolation and Texture size	303
Mapping Modes.....	304

Summary	306
Tutorial 35: Lighting and Fog	307
The OpenGL Lighting Model	307
Getting Started	308
Moving the Light	309
Specular Lighting.....	311
Diffuse Lighting.....	313
Ambient Lighting.....	314
That's Ugly!	315
Directional vs. Positional Lighting	315
Fog.....	316
Summary	317
Tutorial 36: 3D Models	318
Review and Setup	318
Reading a Model File	318
Model Attributes	320
Lighting and Shading.....	320
Texture Mapping	321
Drawing Groups	323
Material Modes	323
Summary	326
Tutorial 37: Geometry Under the Hood.....	327
Matrix Output.....	327
Geometry Matrix Details	329
Processing the Geometry Matrix.....	330
Drawing Primitives	331
Summary	332
Tutorial 38: Basic Performance Setup	333
Got It Covered	335
A Little Performance Tip	338
Summary	340
Tutorial 39: Spatial Mapping.....	341
The Wide World of Repos.....	341
Spatial Expressions.....	344
Still Just Matrices.....	350
Summary	351
Tutorial 40: Drawing in OpenGL using jit.gl.sketch	352
The command list.....	353
More about drawing primitives.....	355
Rotation, translation, and scaling	359
Summary	363
Tutorial 41: Shaders.....	365
Flat Shading.....	365

Getting Started	365
Smooth Shading.....	366
Per-Pixel Lighting	367
Programmable Shaders	368
Vertex Programs.....	369
Summary	370
Tutorial 42: Slab: Data Processing on the GPU	372
Recent Trends in Performance.....	372
Getting Started	372
What about the shading models?	373
How Does It Work?	375
Moving from VRAM to RAM	376
Summary	378
Tutorial 43: A Slab of Your Very Own	379
Mixing Multiple Sources.....	379
Fragment Program Input and Output	380
The Vertex Program	382
Wrapping in a Jitter XML Shader (JXS)	382
Ready for Action.....	383
Summary	384
Tutorial 44: Flash Interactivity	385
A Little Background.....	385
Clicks and Pops.....	385
Hips to Scripts.....	387
Summary	390
Tutorial 45: Introduction to using Jitter within JavaScript.....	391
Waking Up	391
The Javascript Route.....	395
Creating Matrices.....	396
Creating Objects.....	398
JavaScript Functions calling Jitter Object Methods	399
The Perform Routine.....	401
Other Functions.....	403
Summary	404
Code Listing	404
Tutorial 46: Manipulating Matrix Data using JavaScript	407
The Wide World of Particles	409
Under the Hood.....	410
The Initialization Phase	413
Door #1: The op() route.....	415
Door #2: The expr() route	418
Door #3: Cell-by-cell.....	420
Other functions.....	421

Summary	425
Code Listing	425
Tutorial 47: Using Jitter Object Callbacks in JavaScript	437
Creating OpenGL objects in JavaScript	438
The callback function	441
Drawing the scene	444
Summary	446
Code Listing	447
Tutorial 48: Frames of MSP signals	451
Basic Viz.....	452
Frames	455
Varispeed	457
Summary	458
Tutorial 49: Colorsaces	459
Color Lookup with a Twist.....	460
Color tinting and saturation	464
Videoplane.....	466
Videoplane Post-Processing.....	467
Summary	468
Tutorial 50: Procedural Texturing & Modeling.....	469
jit.bfg	469
jit.normalize.....	471
Basis Categories	471
Distance Functions	472
Filter Functions.....	475
Transfer Functions.....	478
Noise Functions.....	482
Fractal Functions.....	485
Other Attributes & Messages.....	487
Summary	489
Tutorial 51: Jitter Java	490
Accessing an Input Matrix	491
Operating on a Matrix.....	492
Copying Input Data	496
Object Composition.....	498
Listening.....	501
Summary	504
Tutorial 52: Jitter Networking.....	505
Summary	508
Tutorial 53: Jitter Networking (part 2)	509
Broadcasting.....	509
Mainstream.....	509
File Streaming	511

Using the jit.qt.broadcast object (Mac OS X only)	515
Summary	516
Appendix A: QuickTime Confidential	517
Structure of QuickTime movies	517
Time in QuickTime	519
Optimizing Movies for Playback in Jitter.....	519
Codecs	519
Audio Codecs	520
Video codecs.....	520
Movie Dimensions and Frame Rate.....	522
Our Favorite Setting.....	522
Summary	523
Appendix B: The OpenGL Matrix Format	524
Matrices, Video and OpenGL.....	524
When You Need This Reference	524
GL Matrix Reference	525
Message Format	525
Draw Primitive.....	525
The Connections Matrix	526
The Geometry Matrix	526
Appendix C: The JXS File Format	528
The Jitter Shader Description File	528
JXS Tags	529
Appendix D: Building Standalone Applications with Jitter.....	533
Windows Standalone Format	533
Macintosh Standalone Format	534
Jitter Bibliography: For Your Further Reading Pleasure	536
Jitter Reference List.....	536
Video	536
Image Processing	537
Video Codecs.....	537
OpenGL and 3D Graphics	537
Shaders.....	537
Procedural Texturing and Modeling	538
2D Graphics and Vector Animation.....	538
Video Art.....	538
Generative Art.....	539
Linear Algebra and Mathematical Matrix Operations.....	539
General Programming.....	539
Miscellaneous	539

Copyright and Trademark Notices

This manual is copyright © 2002-2006 Cycling '74.

MSP is copyright © 1997-2006 Cycling '74—All rights reserved. Portions of MSP are based on Pd by Miller Puckette, © 1997 The Regents of the University of California. MSP and Pd are based on ideas in FTS, an advanced DSP platform © IRCAM.

Max is copyright © 1990-2006 Cycling '74/IRCAM, l'Institut de Recherche et Coordination Acoustique/Musique.

Jitter is copyright © 2002-2006 Cycling '74—All rights reserved.

Credits

Jitter Documentation and Reference pages: Jeremy Bernstein, Joshua Kit Clayton, Christopher Dobrian, R. Luke DuBois, Derek Gerstmann, Randy Jones, Ben Nevile, and Gregory Taylor

Cover Design: Lilli Wessling Hart

Cover Photo: Sue Costabile

Graphic Design: Joshua Kit Clayton and Gregory Taylor

Video and image materials: Joshua Kit Clayton, Toni Dove, Susan Gladstone, and Mark McNamara

About Jitter

Jitter is a set of over 150 matrix, video, and 3D graphics objects for the Max graphical programming environment. The Jitter objects extend the functionality of Max4/MSP2 with flexible means to generate and manipulate matrix data—any data that can be expressed in rows and columns, such as video and still images, 3D geometry, as well as text, spreadsheet data, particle systems, voxels, or audio. Jitter is useful to anyone interested in realtime video processing, custom effects, 2D/3D graphics, audio/visual interaction, data visualization, and analysis.

Since Jitter is built upon the Max/MSP programming environment, the limitations inherent in fixed purpose applications is eliminated. You are able to build the programs you want to use, rather than being forced to work around someone else's idea of how things should be done. This power is not to be underestimated, so please use it wisely.

Video

Although the Jitter architecture is general, it is highly optimized for use with video data, and performs with breathtaking speed. A robust set of mathematical operators, keying/compositing, analysis, colorspace conversion and color correction, alpha channel processing, spatial warping, convolution-based filters, and special effects deliver the building blocks for your own custom video treatments.

Jitter includes extensive support for Apple's QuickTime architecture, such as the playback of all QT supported file formats, real or non-realtime file creation, editing operations, import/export capabilities, integrated realtime QT effects, video digitizing, QTVR, file format conversion, and more.

Note: QuickTime must be installed on your system in order to use Jitter.

QuickTime audio may be routed into MSP to exploit MSP's powerful audio processing capabilities. For the production environment, Jitter provides support for digital video (DV) camera control as well as input and output via FireWire, and multiple monitor support for performance situations.

Note: Video Output Components support is only available for the Macintosh at the present time.

2D/3D Graphics

Jitter's integrated 2D/3D graphics support provides the tools to use hardware accelerated OpenGL graphics together with video, including the ability to texture 3D geometry with

video streams in real time, convert audio and video streams directly into geometry data, and render models, NURBS, 2D/3D text, and other common shapes. There is even low level access to geometry data and the majority of the OpenGL API for those who need to be closer to the machine.

Ease of Use

Jitter is tightly integrated with Cycling '74's Max/MSP graphical programming environment which lets you visually connect data processing objects together with patch cords to create custom applications in a similar manner to analog modular synthesizers.

This visual framework provides the power to build your own unique video effects, realtime video mixers, audio visualizers, image to audio synthesizers, algorithmic image generators, batch converter/ processor programs, or whatever your heart desires. You can share the programs you develop with other Max/MSP users and create standalone applications just as is currently possible with Max/MSP. A free Runtime version is available that runs any application created with Max/MSP/Jitter.

Matrices

Jitter's strength and flexibility comes from the use of a single generalized matrix data format when working with video, 3D geometry, audio, text, or any other kind of data. Jitter matrices may be composed of one of four data types: *char* (8 bit unsigned int), *long* (32 bit signed int), *float32* (32 bit floating point), or *float64* (64 bit floating point). Matrices may have up to 32 dimensions, and may have up to 32 planes.

This common representation makes the transcoding of information effortless. You can experiment with interpreting text as an image, converting video images to 3D geometry, turning audio into a particle system, or playing video data as audio. The possibilities are unlimited.

Jitter has all the fundamental mathematical tools required to work with this numerical representation. The **jit.op** object alone provides over 60 arithmetic, bitwise, exponential, logical, and trigonometric operators. The multitude of operators in **jit.op** are particularly useful for experimenting with video compositing. And Jitter's support for linear algebra, particle systems, Fourier analysis and resynthesis, string processing, cellular automata, and Lindenmeyer systems allows for even further experimental possibilities.

More Details

Jitter objects also make available many aspects of their internal state in ways which will be new to even the most seasoned Max/MSP veterans. Jitter introduces the notion of attributes, internal variables which may be set and queried, thus permitting easier

management of object state. As a means of convenience, Jitter objects can be created with attribute arguments of the form “@<attribute-name> <attribute-value>”—greatly reducing the need for excessive use of the **loadbang** object. Jitter objects can work with matrices of arbitrary size and are designed so that they can adapt to the type and size of data that they receive. A single program may have many objects working with different types and sizes of data at once, and there are tools to easily convert from one type or size to another.

All matrices that are passed between objects are named entities similar to the **buffer~** object in MSP. Referenced by name, a single matrix may be accessed by multiple objects, allowing for creative feedback networks, in-place processing, and memory conservation. There is a publicly available Jitter SDK with the source code of over 30 objects taken directly from the Jitter object set, so third party developers can extend the already overwhelming possibilities Jitter provides. This kind of extensibility is one of the strengths Max/MSP is already known for.

We hope that you have as wonderful an experience using Jitter as we had making it. Please do not hesitate to let us know how we can make it better.

Enjoy,

The Jitter Team

How to Use The Jitter Documentation

- The Jitter documentation assumes you understand the basic Max concepts such as object, message, patcher, symbol etc. as described in the Max tutorials.
- This manual contains 37 tutorials that you can use to explore some of the basic capabilities of the software.
- The documentation begins with a basic introduction to the features of Jitter, followed by an introduction to two fundamental concepts in the software: matrices and attributes. If you like getting into things immediately, you can start learning Jitter by skipping to the tutorials, but these two Topics provide a thorough grounding that may be very useful to your understanding.
- Each tutorial is accompanied by a Max patcher found in the *Jitter Tutorial* folder. The idea is to explore the tutorial patcher as you read the chapter in this manual.
- For an in-depth examination of the features of each Jitter object, you can use the *Jitter Reference* folder in HTML format. The Jitter Reference is located in the *JitterReference* folder, located in the *patches* folder in your Max/MSP folder. You can also launch the Jitter Reference by choosing **JitterReferenceLauncher** from the Max/MSP Extras menu.

There is also a help file for each object, and you can launch the object's reference page within the help file. Before delving into the reference materials, you may wish to read the guide to the object reference linked to at the top of the main Object Reference page. This guide is located in the following places:

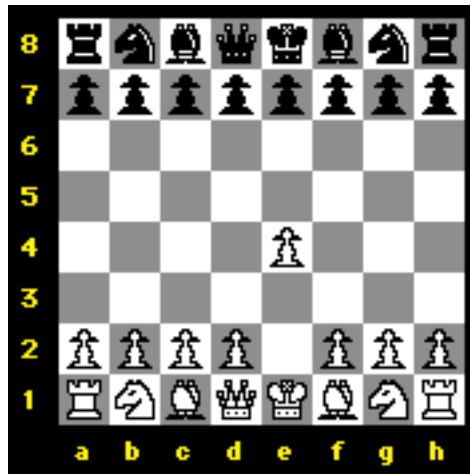
Macintosh: /Applications/MaxMSP 4.5/patches/JitterReference/reference-guide.html

Windows: C:\Program Files\Cycling '74\MaxMSP 4.5\patches\JitterReference\reference-guide.html

- The appendices in this manual provide additional information about QuickTime formats, the OpenGL matrix format, and a list of sources for further reading.

What is a Matrix?

A matrix is a grid, with each location in the grid containing some information. For example, a chess board is a matrix in which every square contains a specific item of information: a particular chess piece, or the lack of a chess piece.



White has just moved a pawn from matrix location e2 to location e4.

For the sake of this discussion, though, let's assume that the "information" at each location in a matrix is numeric data (numbers). Here's a matrix with a number at each grid location.

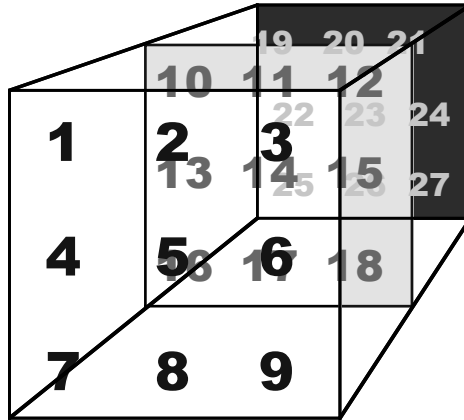
	A	B	C	D
1	0.300	0.258	0.276	0.220
2	0.312	0.280	0.266	0.279
3	0.261	0.295	0.319	0.266

A spreadsheet is an example of a two-dimensional matrix.

We'll call each horizontal line of data a *row*, and each vertical line of data a *column*. On roadmaps, or on chessboards, or in spreadsheet software, one often labels columns with letters and rows with numbers. That enables us to refer to any grid location on the map by referring to its column and its row. In spreadsheets, a grid location is called a *cell*. So, in the example above, the numeric value at cell C3 is 0.319.

The two pictures shown above are examples of matrices that have two dimensions, (horizontal) width and (vertical) height. In Jitter, a matrix can have any number of dimensions from 1 to 32. (A one-dimensional matrix is comparable to what programmers call an *array*. Max already has some objects that are good for storing arrays of numbers, such as **table** and **multislider**. There might be cases, though, when a one-dimensional matrix in Jitter would be more useful.) Although it's a bit harder to depict on paper, one could certainly imagine a matrix with three dimensions, as a cube having width, height,

and depth. (For example, a matrix 3 cells wide by 3 cells high by 3 cells deep would have $3 \times 3 \times 3 = 27$ cells total.)



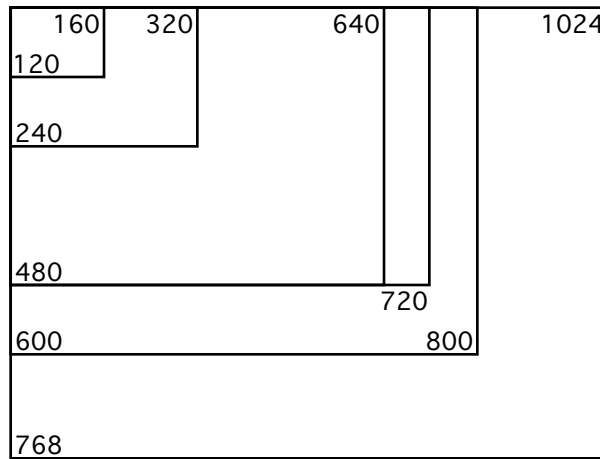
A 3x3x3 matrix has 27 cells.

And, although it challenges our visual imagination and our descriptive vocabulary, theoretically (and in Jitter) one can even have matrices of four or more dimensions. For these tutorials, however, we'll restrict ourselves to two-dimensional matrices.

A Video Screen is One Type of Matrix

A video screen is made up of tiny individual *pixels* (picture elements), each of which displays a specific color. On a computer monitor, the resolution of the screen is usually some size like 1024 pixels wide by 768 pixels high, or perhaps 800x600 or 640x480. On a television monitor (and in most conventional video images), the resolution of the screen is approximately 640x480, and on computers is typically treated as such. Notice that in all of these cases the so-called *aspect ratio* of width to height is 4:3. In the wider DV format, the aspect ratio is 3:2, and the image is generally 720x480 pixels. High-Definition Television (HDTV) specifies yet another aspect ratio—16:9.

In these tutorials we'll usually work with an aspect ratio of 4:3, and most commonly with smaller-than-normal pixel dimensions 320x240 or even 160x120, just to save space in the Max patch.



Relative sizes of different common pixel dimensions

A single frame of standard video (i.e., a single video image at a given moment) is composed of $640 \times 480 = 307,200$ pixels. Each pixel displays a color. In order to represent the color of each pixel numerically, with enough variety to satisfy our eyes, we need a very large range of different possible color values.

There are many different ways to represent colors digitally. A standard way to describe the color of each pixel in computers is to break the color down into its three different color components —red, green, and blue (a.k.a. *RGB*)—and an additional transparency/opacity component (known as the *alpha* channel). Most computer programs therefore store the color of a single pixel as four separate numbers, representing the alpha, red, green, and blue components (or *channels*). This four-channel color representation scheme is commonly called *ARGB* or *RGBA*, depending upon how the pixels are arranged in memory.

Jitter is no exception in this regard. In order for each cell of a matrix to represent one color pixel, each cell actually has to hold *four* numerical values (alpha, red, green, and blue), not just one.

So, a matrix that stores the data for a frame of video will actually contain four values in each cell.

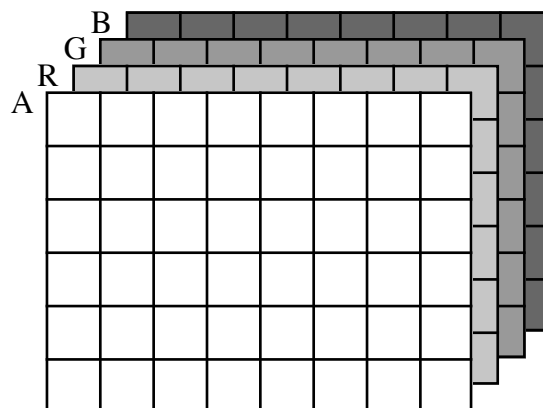
A: 255 R: 218 G: 111 B: 218	A: 255 R: 218 G: 111 B: 218	A: 254 R: 218 G: 112 B: 217	etc.
A: 255 R: 218 G: 111 B: 217	A: 255 R: 218 G: 111 B: 217	A: 254 R: 218 G: 112 B: 217	etc.
etc.	etc.	etc.	etc.

Each cell of a matrix may contain more than one number.

A frame of video is thus represented in Jitter as a two-dimensional matrix, with each cell representing a pixel of the frame, and each cell containing four values representing alpha, red, green, and blue on a scale from 0 to 255. In order to keep this concept of multiple-numbers-per-cell (which is essential for digital video) separate from the concept of *dimensions* in a matrix, Jitter introduces the idea of *planes*.

What is a Plane?

When allocating memory for the numbers in a matrix, Jitter needs to know the extent of each dimension—for example, 320x240—and also the number of values to be held in each cell. In order to keep track of the different values in a cell, Jitter uses the idea of each one existing on a separate *plane*. Each of the values in a cell exists on a particular *plane*, so we can think of a video frame as being a two-dimensional matrix of four interleaved planes of data.



The values in each cell of this matrix can be thought of as existing on four virtual planes.

Using this conceptual framework, we can treat each plane (and thus each channel of the color information) individually when we need to. For example, if we want to increase the

redness of an image, we can simply increase all the values in the red plane of the matrix, and leave the others unchanged.

The normal case for representing video in Jitter is to have a 2D matrix with four planes of data—alpha, red, green, and blue. The planes are numbered from 0 to 3, so the alpha channel is in plane 0, and the RGB channels are in planes 1, 2, and 3.

The Data in a Matrix

Computers have different internal formats for storing numbers. If we know the kind of number we will want to store in a particular place, we can save memory by allocating only as much memory space as we really need for each number. For example, if we are going to store Western alphabetic characters according to the ASCII standard of representation, we only need a range from 0 to 255, so we only need 8 bits of storage space to store each character (because $2^8 = 256$ different possible values). If we want to store a larger range of numbers, we might use 32 bits, which would give us integer numbers in a range from -2,147,483,648 to 2,147,483,647. To represent numbers with a decimal part, such as 3.1416, we use what is called a *floating point* binary system, in which some of the bits of a 32-bit or 64-bit number represent the mantissa of the value and other bits represent the exponent.

Much of the time when you are programming in Max (for example, if you're just working with MIDI data) you might not need to know how Max is storing the numbers. However, when you're programming digital audio in MSP it helps to be aware that MSP uses floating point numbers. (You will encounter math errors if you accidentally use integer storage when you mean to store decimal fractions.) In Jitter, too, it is very helpful to be aware of the different types of number storage the computer uses, to avoid possible math errors.

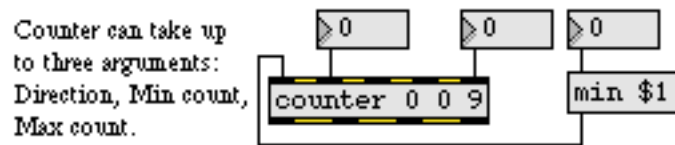
A Jitter matrix can store numbers as 64-bit floating-point (known to programmers as a *double-precision float*, or *double*), 32-bit floating point (known simply as *float*), 32-bit integers (known as *long int*, or just *int*), and 8-bit characters (known as *char*). Some jit objects store their numerical values in only one of these possible formats, so you will not have to specify the storage type. But other Jitter objects can store their values in various ways, so the storage type must be typed in as an argument in the object, using the words *char*, *long*, *float32*, or *float64*.

Important concept: In cases where we're using Jitter to manipulate video, perhaps the most significant thing to know about data storage in Jitter matrices is the following. When a matrix is holding video data—as in the examples in the preceding paragraphs—it assumes that the data is being represented in ARGB format, and that each cell is thus likely to contain values that range from 0 to 255 (often in four planes). For this reason, the most common data storage type is *char*. Even though the values being stored are usually numeric (not alphabetic characters), we only need 256 different possible values for each one, so the 8 bits of a *char* are sufficient. Since a video frame contains so *many* pixels, and each cell may contain four values, it makes sense for Jitter to conserve on storage space when dealing with so many values. Since manipulation of video data is the primary activity of many of the Jitter objects, most matrix objects use the *char* storage type by default. For monochrome (grayscale) images or video, a single plane of *char* data is sufficient.

Attributes: Editing Jitter object parameters

What are attributes?

Attributes are a new way to specify the behavior of Max objects. Most Jitter objects use attributes for the different variables that make up their current internal state.



The good old Max counter object

Many Max objects, such as the **counter** object shown above, take a number of *arguments* to determine how they behave. The order of these arguments after the object's name determines how the object interprets them. In the example above, the first argument to **counter** sets the direction in which the object counts; the second and third arguments determine the minimum and maximum values that the object counts between. Since these values are simply given to the object as numbers, their ordering is important. With some Max objects (**counter** is one of them) the number of arguments you give has some effect on the way in which they are interpreted. If you supply **counter** with only two arguments, they will be understood by the object as the minimum and maximum count, *not* the direction and minimum count. Because the position and number of arguments are crucial, there is no way, for example, to create a **counter** object with a predefined direction and maximum count using only two arguments.

The arguments to an object are often seen as *initial* values, and Max objects typically have ways to modify those values through additional inlets to the object or special messages you send the object. You can change the direction and maximum count of a **counter** object by sending integers into its second and fifth inlets, respectively. These will override the default values supplied by the arguments. Similarly, you can change the minimum count of the object by sending the message **min** followed by an integer into the left inlet.

While this system works well when a Max object only has two or three variables that define its behavior, Jitter objects often have many, many more variables (sometimes dozens). If all of these variables depended on the order of inlets and object arguments, you would spend all day searching the reference manual and never have time to do any work with Jitter!

Setting Attributes

Jitter objects, unlike Max objects, can be told how to behave by using *attributes*. You can type attributes into an object box along with the Jitter object's name, or you can set (and retrieve) attributes through Max messages after the object is created.



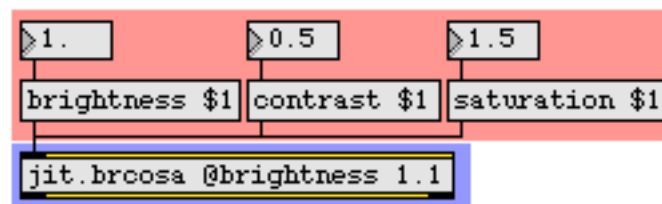
A Jitter object with attributes after the object name

The Jitter object shown above, called **jit.brcosa**, has three typed-in attributes. Typed-in attributes are set in object boxes by using the @ symbol followed by the name of the attribute and one or more arguments (which could be any kind of Max data: ints, floats, symbols, or lists). You can enter as many attributes as the object recognizes, *in any order*, after the object's name. While you may not know what the **jit.brcosa** object does yet, you can infer a little bit about the object based on the names of the attributes and what type of data they have assigned to them.

Important: There is no space between the @ sign and the name of the typed-in attribute you want to set. The @ character tells the Jitter object to interpret the word attached to it as an attribute name instead of an argument value for a previous attribute.

Also Important: Jitter objects can have both typed-in attributes *and* typed-in arguments. See the **Jitter Object Arguments** section below for details.

As with Max objects, the information you give a Jitter object to set its initial behavior is generally something you can change after the object is created. Attributes can be changed at any time by sending messages to the object as shown below.



Attributes can be changed with Max messages

This **jit.brcosa** object has its brightness attribute set to 1.1 initially, but we've changed it to 1.0 by sending the message brightness 1 into the object's left inlet. You can change virtually any attribute by sending a message with the attribute's name, followed by the relevant arguments, into a Jitter object's left inlet.

As with Max objects, Jitter objects have default values for their parameters. The **jit.brcosa** object above only has typed-in attributes initializing its brightness value, but other attributes are set to their default values. We'll show you how to find out what attributes an object uses below. In the example above, we can change the values of the object's contrast and saturation attributes using messages, overriding whatever default values the object has supplied.

Jitter Object Arguments

There are four pieces of information that most Jitter objects use that can be entered either as typed-in attributes or typed-in arguments. In fact, they are always attributes, but Jitter objects automatically handle them correctly when they are used as arguments.



```
jit.rota 4 char 320 240 @theta 0.7
```

Jitter objects can have arguments, too!

The **jit.rota** object, shown above, clearly has one attribute initialized: theta. But what does the other stuff mean?

If you supply arguments for a Jitter object that processes Jitter matrix data (and most Jitter objects do), the arguments are interpreted as:

1. The planecount of the output matrix.
2. The type of the output matrix.
3. The size, or 'dimension list' (dim), of the output matrix.

Now that we know this, we can determine that the **jit.rota** object above will output a matrix that is made up of 4 planes of char (8-bit integer) data with two dimensions of 320 by 240 cells.

Important: Jitter object arguments, if used, must appear *before* any attributes are set. Otherwise the Jitter object will misinterpret the arguments as values for the attribute, not arguments to the object.

These arguments can also be set by attributes that are consistent for all Jitter objects that output matrix data: `planecount`, `type`, and `dim`. They can be set as unordered typed-in attributes or changed with messages. The three objects below, for example, are identical.

```
jit.repos 1 float32 1024 16 @offset_x 8 @mode 1
```

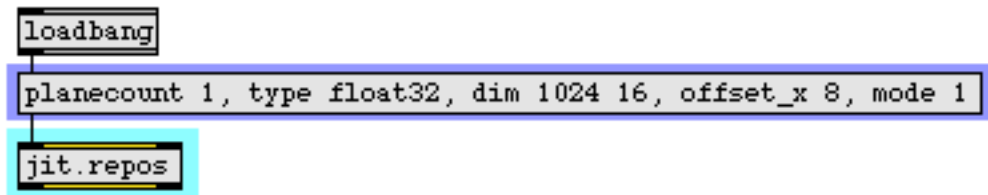
```
jit.repos 1 float32 @offset_x 8 @mode 1 @dim 1024 16
```

```
jit.repos @offset_x 8 @type float32  
@mode 1 @dim 1024 16 @planecount 1
```

Arguments or attributes? You decide.

The first object has its output matrix attributes set using typed-in arguments. The second object has the `planecount` and `type` set using typed-in arguments, but uses a typed-in attribute for the dimension list. The third object uses a typed-in attributes to set everything.

If you prefer, you can initialize an object's attributes using messages triggered from a loadbang object as shown below.

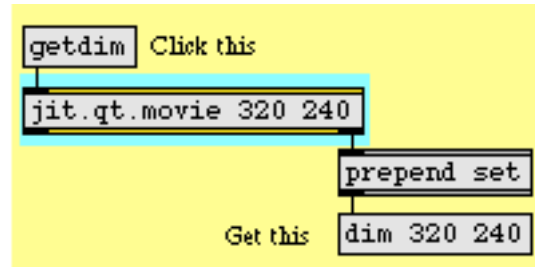


Yet another way to initialize your attributes

Querying Attributes and Object State

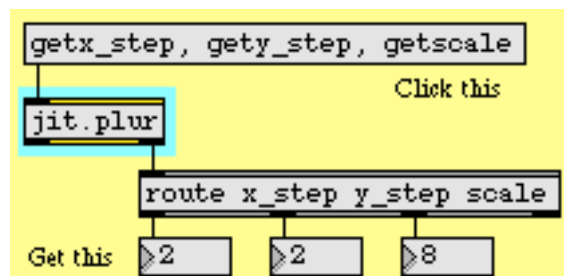
An additional (and very useful) feature of attributes is that you can ask a Jitter object to tell you what value it currently has stored for any given attribute. You do this by *querying* the attribute with the Max message `get` followed (*with no space*) by the name of the attribute you want to know about.

The resulting value is output by the Jitter object as a message (beginning with the attribute's name), sent out the object's right outlet.



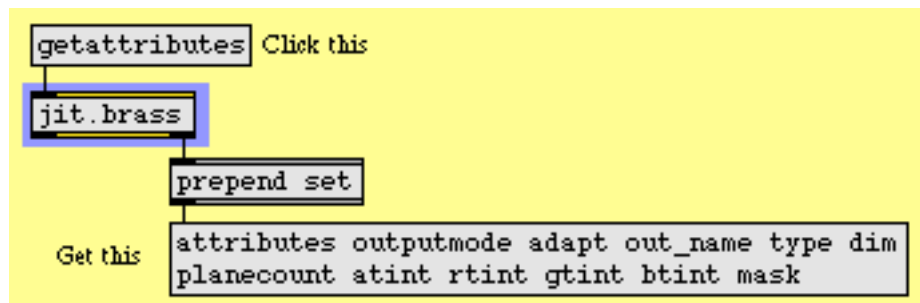
Querying an attribute for a Jitter object

Using `get` to find the values of attributes lets you discover the current value of an attribute, even if you never set the attribute in the first place. For example, the patch below discovers some of the default values of the **jit.plur** object. The Max **route** object lets you easily separate the values for each of the attributes.



Finding out the default values of object attributes

Two messages you can send to any Jitter object, `getattributes` and `getstate`, output all the attributes used by the object.

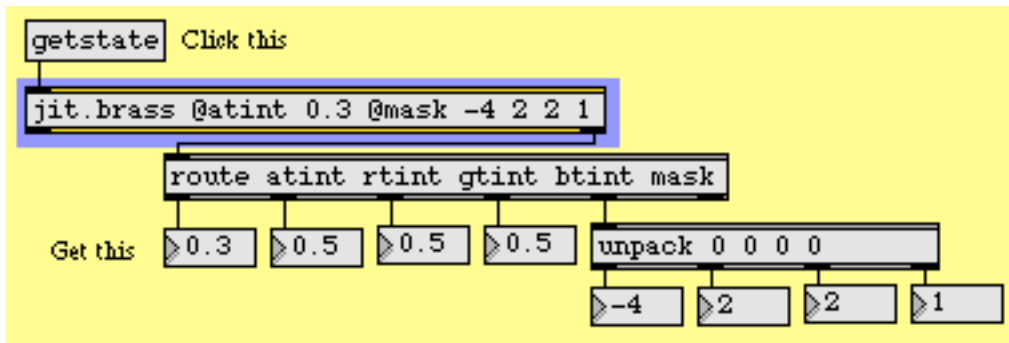


Finding out your options...

The `getattributes` message causes the Jitter object to output the message attributes followed by a list of all the attribute symbols that Jitter object understands. Experimenting with a few Jitter objects will quickly show you that many of these, such as `outputmode`, `type` and `dim`, are

fairly standard. Others (such as `mask` in the `jit.brass` object) will have special meaning for the object that uses them.

The `getstate` message dumps out all the attributes for the Jitter object as if every conceivable attribute query had been performed all at once.



Finding an object's state

You can then use **route**, **unpack**, and other Max objects to extract the attributes as you need them. Later in the tutorials, you will encounter several Jitter objects where the attributes change based on calculations performed on the input matrix (or a file that has just been opened by the object). Querying the relevant attributes is how you can find out the result of the object's calculation.

Summary

Jitter attributes are a powerful tool for managing the parameters of complex objects. You can use attributes to initialize, change, and find out the current values stored in Jitter objects, and the attachment of each value to a fixed attribute name eliminates the need to remember typed-in argument ordering or the role of each inlet in a complex object.

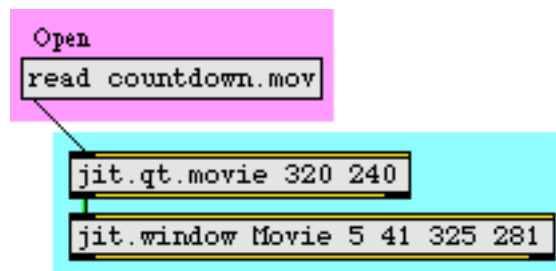
Tutorial 1: Playing a QuickTime Movie

This first tutorial demonstrates one of the simplest yet most useful tasks you can perform with Jitter: playing a QuickTime movie in a window.

- Open the tutorial patch *01jPlayAMovie.pat* in the Jitter Tutorial folder.

There are two Jitter objects in this patch: **jit.qt.movie** and **jit.window**. The **jit.window** object automatically opens a window on your computer screen. The **jit.qt.movie** object allows you to open an existing QuickTime movie, and begins to play it.

- Click on the **message** box containing the message `read countdown.mov`. This causes the **jit.qt.movie** object to open the QuickTime movie file *countdown.mov* and begin reading from it.



The read message opens a QuickTime movie file.

By default a **jit.qt.movie** will begin playing a movie as soon as it opens it. (Alternatively, you can alter that behavior by sending a **jit.qt.movie** object an `autostart 0` message before opening the file, but for now the default behavior is fine.) Notice, however, that even though we've said that the **jit.qt.movie** object is playing the movie, the movie is not being shown in the Movie window. Here's why:

Each object in Jitter does a particular task. The task might be very simple or might be rather complicated. What we casually think of as "playing a QuickTime movie" is actually broken down by Jitter into two tasks:

1. Reading each frame of movie data into RAM from the file on the hard disk
2. Getting data that's in RAM and showing it as colored pixels on the screen.

The first task is performed by the **jit.qt.movie** object, and the second by the **jit.window** object. But in order for the **jit.window** object to know what to display, these two objects need to communicate.

How Jitter Objects Communicate

We've said that **jit.qt.movie** object's primary job is to open a QuickTime movie and read each frame, one after another, into RAM for easy access by other Jitter objects. To do this, the **jit.qt.movie** object claims a particular portion of the Max application's allotted memory, and consistently uses that one location in memory. Jitter keeps track of locations in RAM by assigning each one a name. (You don't always need to know the actual name of the location in memory, but Jitter objects do need to know them in order to find the information that's being held there.)

Important Concept: The most important thing that Jitter objects communicate to each other is a *name*, referring to a *matrix*—a place in memory where data is stored. (We'll explain the meaning of the word "matrix" in more detail in the next tutorial chapter.) Jitter objects output a message that only other Jitter objects understand. That message is the word `jit_matrix` followed by a space and the name of a matrix where data is stored. This message is communicated from one Jitter object to another through a patch cord in the normal Max manner. (But, just as MSP objects' patch cords look different from other Max patch cords, the patch cords from Jitter objects' outlets that send the `jit_matrix` message have their own unique look.) The receiving Jitter object receives the message in its inlet (most commonly the left inlet), gets the data from the specified place in memory, modifies the data in some way, and sends the name of the modified data out its left outlet to all connected Jitter objects. In this way, tasks are performed by each object without necessarily knowing what the other objects are doing, and each object gets the data it needs by looking at the appropriate place in memory. Most Jitter objects don't really *do* anything until they get a `jit_matrix` message from another Jitter object, telling them to look at that matrix and do something with the data there.

In many cases a Jitter object will generate a unique name for its matrix on its own. In other cases, it is possible (and even desirable) to tell an object what name to use for a matrix. By explicitly naming a matrix, we can cause objects to use that same memory space. You will see examples of this in future tutorial chapters.

Causing Action by Jitter Objects

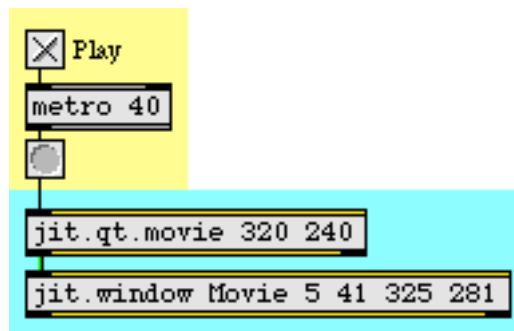
What causes one Jitter object to send a `jit_matrix` message to another object? Most Jitter objects send out a `jit_matrix` message when they receive the message `outputmatrix` or `bang`. (These two messages have the same effect in most Jitter objects.) The other time that an object sends out a `jit_matrix` message is when it has received such a message itself, and has

modified the data in some way; it then automatically sends out a `jit_matrix` message to inform other objects of the name of the matrix containing the new data.

To restate the previous paragraph, when an object receives a `jit_matrix` message, it does something and sends out a `jit_matrix` message of its own. When an object receives `outputmatrix` or `bang`, it sends out a `jit_matrix` message without doing anything else.

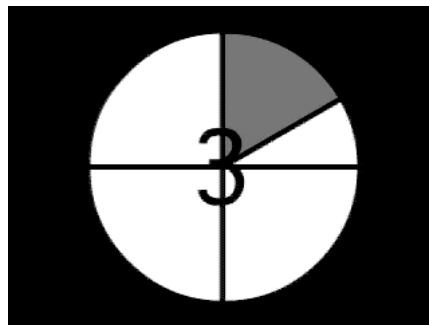
So, in our example patch, the **jit.qt.movie** object is "playing" the QuickTime movie, constantly storing the current frame of video, but the **jit.window** object will only display something when it receives a `jit_matrix` message from the **jit.qt.movie** object. And that will only happen when **jit.qt.movie** receives the message `bang` (or `outputmatrix`). At that time, **jit.window** will display whatever frame of video happens to be currently playing in the movie (that is, the frame that's currently stored by **jit.qt.movie**).

In order to make **jit.window** update its display at the desired rate to show a continuously progressing video, we need to send the message `bang` to **jit.qt.movie** at that rate.



*The movie is playing in `jit.qt.movie`,
but we need to send it a bang each time we want to display a frame.*

- Click on the **toggle** object marked "Play" to start the **metro** object. This will send out bang at the rate of 25 times per second (every 40 milliseconds). That should be fast enough to display every frame of this video. As long as the bang messages continue, you will see the movie displayed in the Movie window.



jit.window displays the contents of a matrix: in this case a frame of a QuickTime movie.

- Click on the **toggle** to stop the **metro**. The **jit.window** object stops updating the Movie window, so you will now just see a still image of whatever frame was last displayed. The movie is still "playing"—and **jit.qt.movie** is still updating its memory frame-by-frame—but **jit.window** is now oblivious because **jit.qt.movie** is no longer sending messages.
- You can verify that the movie is still progressing by clicking on the **button** object just below the **metro**. This will cause **jit.qt.movie** to send a `jit_matrix` message to **jit.window**, which will update the Movie window with the current frame. If you do this a few times, you will see that the movie has progressed in between clicks of the mouse. (The movie is a ten-second countdown, playing in a continuous loop.)

To summarize, **jit.qt.movie** is continually reading in one video frame of the QuickTime movie, frame by frame at the movie's normal rate. When **jit.qt.movie** receives a bang, it communicates the location of that data (that single frame of video) to **jit.window**, so whatever frame **jit.qt.movie** contains when it receives a bang is the data that will be displayed by **jit.window**.

Arguments in the Objects

The **jit.qt.movie** object in this tutorial patch has two typed-in arguments: 320 240. These numbers specify the horizontal and vertical (width and height) dimensions the object will use in order to keep a single frame of video in memory. It will claim enough RAM to store a frame with those dimensions. So, in the simplest case, it makes sense to type in the dimensions of the movie you expect to read in with the `read` message. In this case (since we made the movie in question ourselves) we happen to know that the dimensions of the QuickTime movie *countdown.mov* are 320x240.

If we type in dimension arguments smaller than the dimensions of the movie we read in, **jit.qt.movie** will not have claimed enough memory space and will be obliged to ignore some of the pixels of each frame of the movie. Conversely, if we type in dimension arguments larger than the dimensions of the movie we read in, there will not be enough pixels in each frame of the movie to fill all the memory space that's been allocated, so **jit.qt.movie** will distribute the data it does get evenly and will fill its additional memory with duplicate data.

The **jit.window** object has five typed-in arguments: Movie 5 41 325 281. The first argument is a name that will be given to the matrix of data that **jit.window** displays. That name will also appear in the title bar of the movie window. It can be any single word, or it can be more than one word if the full name is enclosed between "smart single quote" characters. (Smart single quotes are the characters ‘ and ’, obtained by typing *option-]* and *shift-option-]*.) The next two arguments indicate the *x,y* screen coordinates of the upper-left corner of the display region of the movie window, and the last two arguments provide the

x,y coordinates of the lower-right corner of the display region. (Another way to think of these four numbers is to remember them as the coordinates meaning "left", "top", "right", and "bottom".) We have chosen these particular numbers because *a*) they describe a display region that is 320x240 pixels, the same size as the movie we intend to display, and *b*) when we take into account the dimensions of the window borders, title bar, and menu bar that the Mac OS imposes, the entire window will be neatly tucked in the upper-left corner of our useable desktop. (It's possible to make the window borders and title bar disappear with a `border 0` message to **jit.window**, but the default borders are OK for now.)

We have typed the value of 40 in as an argument to **metro** to cause it to send out 25 bang messages per second. The QuickTime movie actually has a frame rate of exactly 24 frames per second, so this **metro** will trigger the **jit.qt.movie** object frequently enough to ensure that every frame is made available to **jit.window** and we'll get to see every frame.

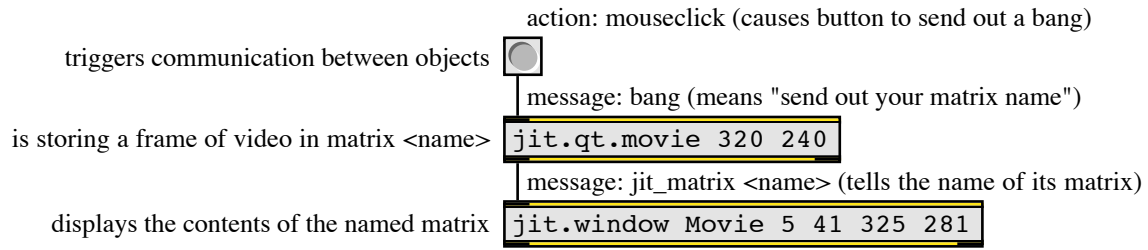
The **jit.qt.movie** object actually understands many more messages besides just bang (way too many to try to explain here). In the upper-right corner of the Patcher window, we've included an example of one more message, simply to demonstrate that the progress of the QuickTime movie can be controlled in **jit.qt.movie** independently of the rate at which the **metro** is sending it bang messages. The message time, followed by a number, causes **jit.qt.movie** to jump immediately to a specific time location in the movie.

- Click on the **button** object labeled "Restart". This sends a message of time 0 to **jit.qt.movie**, causing it to jump to the beginning of the QuickTime movie, and then sends a 1 message to the **toggle** to start the **metro** and begin displaying the movie.

Summary

To play a QuickTime movie, use the **jit.qt.movie** object to open the file and read successive frames of the video into RAM, and use the **jit.window** object to display the movie in a separate window. Use typed-in arguments to specify the dimensions of the movie, and the precise coordinates of the display area on your screen.

Jitter objects communicate the information about a particular frame of video by sending each other the name of a *matrix*—a place in memory where that information is located. When a Jitter object gets a matrix name, it performs its designated task using the data at that location, then sends out the name of the modified data to other Jitter objects. Almost all Jitter objects send out a name (in a `jit_matrix` message) when they receive the message `bang` (or `outputmatrix`). Thus, to show successive frames of a video, send `bang` messages at the desired rate to a **jit.qt.movie** object connected to a **jit.window** object.



Tracing the messages and roles of each object

Tutorial 2: Create a Matrix

What's a Matrix?

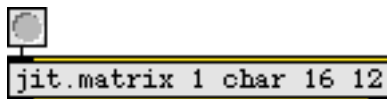
This tutorial chapter demonstrates some ways to manage numerical data in a *matrix*.

If you are in doubt about what a matrix is, please first read the chapter titled "What's a Matrix?" in the Topics section. To summarize here, a matrix is a scheme for storing and modifying large sets of numerical data, by placing the values in a virtual grid. By storing data in a matrix, we can easily identify a particular value by its location in the grid, and we can modify many values at once by referring to all or part of a matrix.

In the previous tutorial chapter, we used the **jit.window** object to open a window and display the contents of a matrix as colored pixels. The matrix being displayed was from the **jit.qt.movie** object, an object that continually fills its matrix with the current frame of a QuickTime video. The fact that **jit.window** was displaying a video, however, was just because that happened to be the contents of the matrix it was being told to display; but in fact *any* numerical values in a matrix can be visualized similarly. The example patch for this tutorial will show an even simpler example that should help strengthen your understanding of a *matrix*, the central idea of Jitter.

The jit.matrix object

- Open the tutorial patch *02jCreateAMatrix.pat* in the Jitter Tutorial folder.



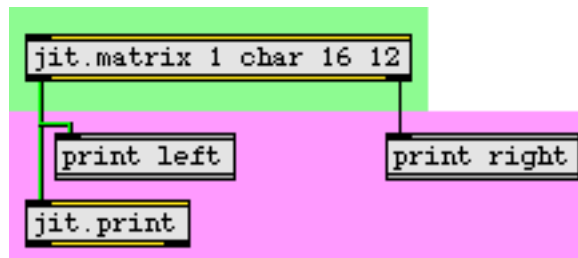
Create a 16x12 storage space for single 8-bit values.

The **jit.matrix** object simply creates a matrix, a storage space in memory. We can then store and retrieve numerical values in the matrix, and we can use other objects to print the values or display them visually. The arguments for **jit.matrix** are an optional [name] argument (not included in this example), the [plane count] (how many values will be stored in each cell of the matrix), the [type] of data (how many bytes to use to represent each number), and then the [dim] (short for "dimensions", describing how large the matrix will be). We use the brackets [] to indicate that this is not the actual word you type in as an argument, but rather a description of the meaning of the argument. The object in this example will create a matrix with 1 *plane* (one number in each matrix location), using the *char* data type (single-byte values), with *dimensions* 16x12 (which equals 192 cells). From this we can deduce that the matrix is capable of holding 192 individual numerical values, with each value ranging from 0 to 255 (that's the range of a single byte).

Note: We always describe the dimensions of a two-dimensional matrix in x,y (width, height) format, meaning that we first state the extent of the horizontal dimension, then the vertical dimension. This is consistent with the way these dimensions are commonly discussed in video and computer screen layout (a 640x480 video image, for example). An alternative way to think of this is that we first state the number of (vertical) *columns* of data, then the number of (horizontal) *rows*. You might want to note, however, if you're applying matrix techniques of linear algebra in Jitter, that this format—columns, rows—is different from the way matrices are commonly described in linear algebra, which states rows first, then columns.

We've connected a **button** object to the inlet of **jit.matrix**. You'll recall that Jitter objects send out their matrix name when they receive a bang message in their left inlet, so this **button** will permit us to trigger **jit.matrix** to send out its matrix name (in the form of a `jit_matrix` message).

The **jit.print** object



Create a 16x12 storage space for single 8-bit values.

Beneath the **jit.matrix** object there is another new Jitter object, **jit.print**. This object accepts a matrix name (i.e., a `jit_matrix` message) in its inlet, and formats the values of the matrix—the sheer number of which can be pretty unwieldy—for printing in the Max window. It prints the formatted values directly to the Max window, much like Max's **print** object, and then passes the matrix name out its left outlet in a `jit_matrix` message (in the normal manner for Jitter objects).

- Click on the **button** object marked "output". This causes the **jit.matrix** object to communicate its matrix name to **jit.print**, which formats the values and prints them in the Max window.

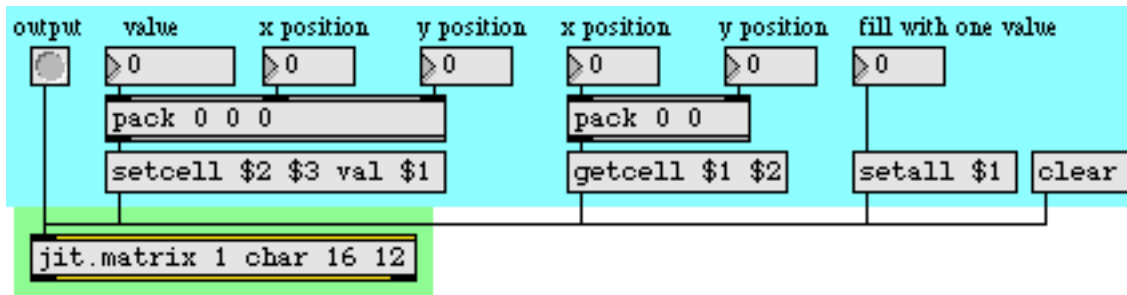
In the Max window you will now see the text `left: jit_matrix[somename]`. The word `left` shows us that this was printed by the **print left** object, so we can see that that is what came out of the outlet of **jit.matrix**. Because we didn't choose a name for that matrix (we didn't provide a name as the first typed-in argument to **jit.matrix**), **jit.matrix** assigned a name of its own choosing. It tries to generate a unique name that's not likely to be used elsewhere, so it

usually chooses something strange like "u330000007". In this case we don't really care what the name is, but it does tell the **jit.print** object what matrix of data to format for printing.

Below that, you will see all of the values in this particular matrix, formatted neatly in 16 columns and 12 rows. Those came from **jit.print**. They're all 0 right now, because we haven't placed anything else in the matrix yet.

Setting and querying values in a matrix

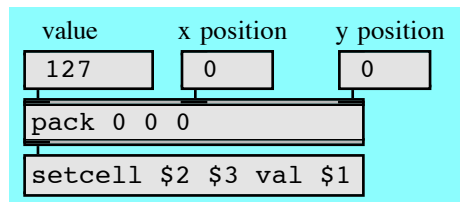
In the previous chapter we saw how an entire matrix could be filled automatically with frames of color data from a QuickTime movie. It is also possible to place numerical values in specific individual cells of the matrix, and retrieve values from specific locations. The objects immediately above **jit.matrix** in this example show a few messages you can use for setting and getting values in a matrix.



The messages `setcell` and `getcell` allow you to access specific values in a matrix.

You can store values in a particular matrix location with the `setcell` message. The syntax for doing so is: `setcell [cell coordinates] val [value(s)]`. For example, the message `setcell 0 0 val 127` to our **jit.matrix** would set the value of the very first cell of the matrix (i.e., the cell in the upper-left corner) to 127. This is because we number the cell coordinates in each dimension starting with 0 and going up to 1 less than the extent of that dimension. So, in this matrix, locations in the *x* dimension are numbered from 0 to 15 and locations in the *y* dimension are numbered 0 to 11. Thus, the cell in the lower right corner would be described with the cell coordinates 15 11.

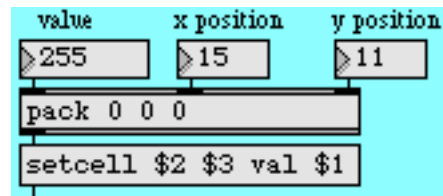
- The combination of the **pack 0 0 0** object and the **message** box helps us to format a proper setcell message for **jit.matrix**. First we set the *x* and *y* positions where we want to store the value, then we specify a value to be stored there. With the *x* and *y* positions at 0 0, use the **number box** labeled "value" to send the number 127 into the left inlet of **pack 0 0 0**. This will cause the message setcell 0 0 val 127 to be sent from the **message** box to the **jit.matrix**.



The message setcell 0 0 val 127 sets the value of cell position 0, 0 to 127.

(If there were more than one plane in this matrix, we could set the values in a particular plane of a cell, or in all planes of the cell. However, this matrix has only one plane, so we'll leave that for another time.)

- Just to demonstrate what we said earlier about the numbering of cell positions, try sending the message setcell 15 11 val 255 to **jit.matrix**. First enter the number 15 into the "x position" **number box** and the number 11 into the "y position", then enter the number 255 in via the "value" **number box**. Now click on the button marked "output" to see how the matrix has been changed. Once again the entire matrix will be printed in the Max window via **jit.print**. Notice that the values in cell positions 0, 0 and 15, 11 have been changed to 127 and 255.



The message setcell 15 11 val 255 sets the value of cell position 15, 11 to 255.

The jit.pwindow object — **jit.**

In your Patcher window you may have noticed a change in the black rectangular region. The upper-left and lower-right corners of it have changed. (If you covered it with your Max window, you may have to click the "output" **button** once more to see this change.)



*The **jit.pwindow** object displays numerical values as colors (or greyscale values).*

This region is a user interface object called **jit.pwindow**. In the object palette it appears like this:



*The **jit.pwindow** cursor in the object palette*

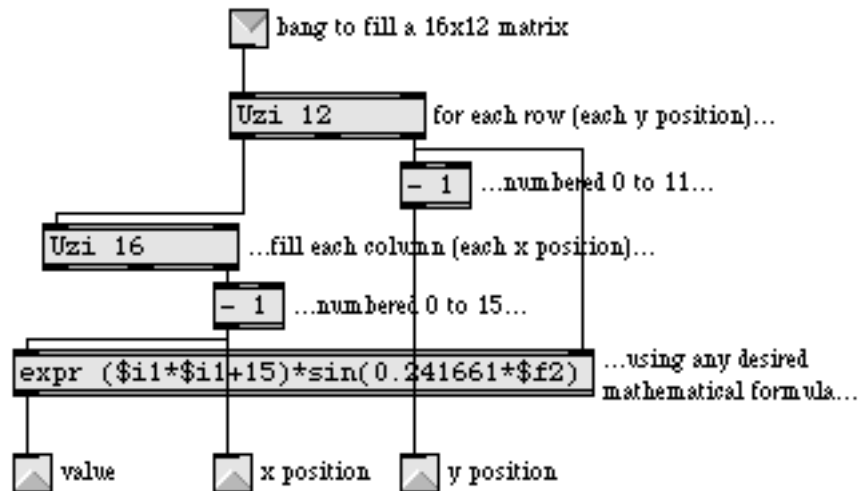
When you choose this object from the palette and click in the Patcher window, it creates a small rectangular object. (You can click in the grow bar in the lower-right corner of the object to adjust its dimensions.) This object is pretty much equivalent in function to the **jit.window** object, but it displays matrix data inside the Patcher window, rather than in a separate window.

So, here we see quite literally the display of numerical values (in this case, *char* data in the range from 0 to 255) as color. Because there is only one plane in this matrix we're displaying, the display is monochrome—that is, greyscale. The 0 values are totally black, and the other values are some level of gray, up to 255 which is totally white. Thus, the 255 value at cell 15, 11 is displayed as white, and the 127 value at 0, 0 is displayed as a 50% gray, halfway between black and white.

You might say, "That's all very well, but it will be pretty tedious to fill a large matrix this way." And you'd be right. But of course Max allows us to write another part of the program that will automate the process.

Filling a matrix algorithmically

- Double-click on the **patcher** fillmatrix object to open the subpatcher window *fillmatrix*. This subpatcher generates 192 different values, one for each position in the matrix, by feeding different numbers into a mathematical expression.



You can generate values algorithmically to fill the cells of a matrix.

When the **Uzi 12** object receives a bang (from the **button** labeled "fill" in the main Patcher window) it quickly counts from 1 to 12 out its right outlet and sends a bang for each count out its left outlet. Those bangs trigger the **Uzi 16** object, so that it sends out numbers from 1 to 16 each time. We subtract 1 from these numbers so that they actually go from 0 to 11 and from 0 to 15, and we use the resulting numbers as *y* and *x* positions in the matrix. For each of the 12 *y* positions, the **Uzi 16** object specifies all the *x* positions, and then uses those numbers in a mathematical expression (in **expr**) to calculate the value to be stored at that position. These numbers are sent out the outlets, and are used to create well-formed setcell messages in the main patch, just as we did by hand earlier.

The mathematical expression here is relatively unimportant. It could be any formula that generates an interesting pattern of data. In this case we have chosen a formula that will produce a sinusoidal gradation of brightness in each column, and the will cause the overall brightness of the columns to increase from left to right (i.e., as *x* increases).

- Close the *fillmatrix* subpatch window and click on the **button** labeled "fill". The matrix is filled with values (generated by the **Uzi** objects in the subpatch) in a single tick of Max's scheduler. Now click on the **button** labeled "output" to view the contents of the matrix. The numerical values will be printed in the Max window, and displayed in the **jit.pwindow**.

Even for a small 16x12 matrix like this, it's tough for us to perceive a trend in numerical data just by looking at a printout of numbers in the Max window. However, the display in

the **jit.pwindow** gives us a very clear and immediate idea of how the values vary within the matrix. This demonstrates one of the benefits of visualization of numerical data.

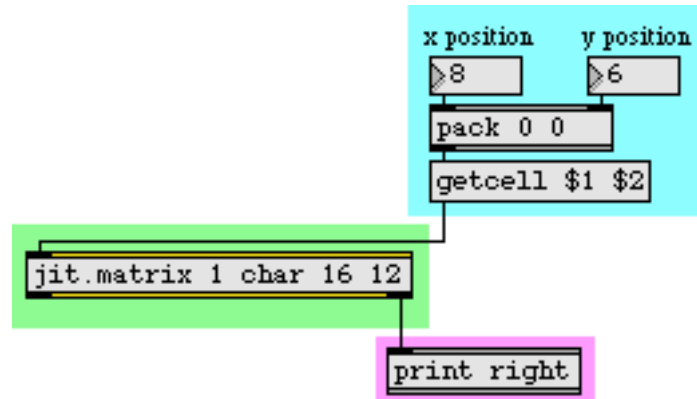
You can no doubt imagine other ways to fill a matrix algorithmically in Max, and in fact we'll demonstrate other ways in later tutorial chapters.

Other messages to **jit.matrix**

There are many other messages understood by **jit.matrix**, more than we can demonstrate fully here. On the right side of the Patcher we show a couple of other handy messages for filling a **jit.matrix** instantly with all the same values. The **clear** message to **jit.matrix** sets all its values to 0, and the **setall** message (the word **setall** followed by a value) sets every position in the matrix to that value.

We also demonstrate the **getcell** message. The word **getcell** followed by a location in the matrix (*x* and *y* positions) will cause **jit.matrix** to send the cell coordinates and value of that position out its right outlet.

- Enter a *y* value and then an *x* value into the **number boxes** above the **getcell \$1 \$2** message box, and observe what is printed in the Max window. Note that the value at that matrix position is reported out the *right* outlet of **jit.matrix**.



Query the value(s) at matrix position 8, 6; reports cell 8 6 val [value(s)]

In future tutorial chapters you will see various ways to use values retrieved from a matrix.

Summary

The **jit.matrix** object creates a storage space for a named matrix of data, with whatever dimensions, planes, and data type you specify. This matrix can be filled with data from another Jitter object (such as **jit.qt.movie**), or by messages such as **setall [value]** to set the value in all cells or **setcell [position] val [value(s)]** to set a specific cell. You can use an algorithm elsewhere in the patch to fill the matrix according to a formula or a set of rules.

To get the data in a specific cell, you can use the `getcell [position]` message. To see all of the numerical data printed out in the Max window, use the **`jit.print`** object to format the matrix data and print it. To see the matrix data displayed as colors, use the **`jit.pwindow`** object. This is similar to the use of the **`jit.window`** object demonstrated in Tutorial 1.

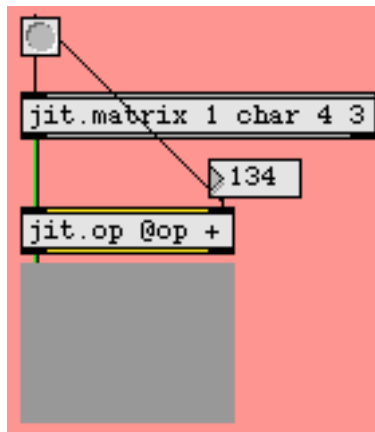
In this tutorial we viewed numerical data that we generated ourselves, rather than digital video as in the preceding chapter. The principle of storage is the same in both cases. Whether a matrix is used to store color information for each pixel from a frame of digital video, or abstract numbers which we wish to view as color, the numbers in both chapters are stored in a two-dimensional matrix and are easily displayed in a **`jit.window`** or **`jit.pwindow`**.

Tutorial 3: Math Operations on a Matrix

This tutorial shows how you can perform simple mathematical operations on the data stored in a Jitter matrix. We'll show you how to use the **jit.op** object to perform arithmetic scaling of matrix cells, or of individual planes within those cells.

- Open the tutorial patch *03jMathOperations.pat* in the Jitter Tutorial folder.

The tutorial patch is split into three simple examples of mathematical operations you can perform with the **jit.op** object. The **jit.op** object performs mathematical operations on entire matrices of data at a time rather than individual numbers.



Adding a constant value to all cells in a matrix.

The first example shows a **jit.matrix** object hooked up to a **jit.op** whose output is viewable by a **jit.pwindow** object. Everytime you change the **number box** hooked up to the right inlet of the **jit.op** object a bang will put out a new matrix from the **jit.matrix** object. As you can see from its arguments, the **jit.matrix** object is generating a 4x3 matrix of single-plane *char* data (i.e. data in the range 0-255). The **jit.pwindow** object will visualize this matrix for you as a greyscale image. Dragging the **number box** will change the level of grey shown in the **jit.pwindow** from black (0) to white (255).

It's important to realize that the **jit.matrix** object is putting out a Jitter matrix that has all its cells set to 0. If you were to connect the **jit.matrix** and **jit.pwindow** objects together and bypass the **jit.op**, you would see a black image, no matter how many times you send a bang message to the **jit.matrix** object. The **jit.op** object is *adding* a value (as defined by the **number box**) to all the cells in the Jitter matrix sent between the **jit.matrix** and the **jit.op** objects.

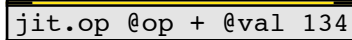
Operation @-Sign

We said above that the **jit.op** object *adds* a value to all the cells in its input matrix. The **jit.op** object adds a value (rather than, say, dividing or multiplying) because of the value of

its **op** attribute. The **op** attribute is a symbol (or a list of symbols, as we'll see in a moment) that defines what math the **jit.op** performs on its input matrix. In this case, we can see that the **op** attribute is set to the value of **+**, which means that it performs simple addition on any matrix that arrives in its left inlet. The integer value in the right inlet is added to all the cells in the matrix. This value is referred to as a *scalar*, because it adds the same value to the entire matrix (in *Tutorial 9* we show how **jit.op** can do math using two Jitter matrices as well).

Important note: Changing the scalar value in the right inlet of the **jit.op** object does not output a new matrix. If you were to disconnect the patch cord between the **number box** and the **button** object, the **jit.pwindow** object would stop showing you anything new. The reason for this is that as with most Max objects, most Jitter objects only output data when something comes into their leftmost inlet. In the case above, each time you change the **number box**, the **jit.op** object stores the new scalar value. As soon as that happens, the **button** object sends a bang to the **jit.matrix** object, causing it to send a new Jitter matrix (with all its values set to 0) into the left inlet of the **jit.op** object, triggering an output matrix which you can see. If you choose the **Enable** command from the Trace menu, and then step through the message order with the **Step** command (-T), you will see how this plays out. (See the "Debugging" chapter of the *Max 4.0 Tutorials and Topics* manual for details about how to trace Max messages with the Trace feature.)

The scalar value can also be supplied as a constant by using the **val** attribute of **jit.op**. For example, if we always wanted to add 134 to all the cells of an incoming Jitter matrix, we could use this object and dispense with the **number box**:

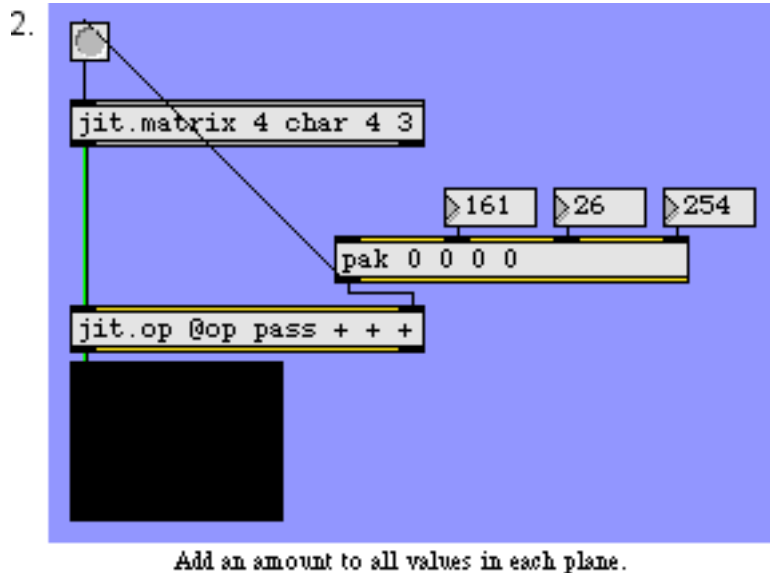
A screenshot of a Max/MSP patch window. It shows a single object named 'jit.op'. The object's message box contains the text '@op + @val 134'. The object is highlighted with a yellow border.

Setting a scalar as an attribute.

Similarly, if we wanted to change the mathematical operation performed by any given **jit.op** object, we could send the message **op** followed by the relevant mathematical symbol into the object's left inlet.

Math Operations on Multiple Planes of Data

The second example shows a more complicated instance of using **jit.op** to add values to an incoming matrix.



Using separate scalars for each plane of a matrix

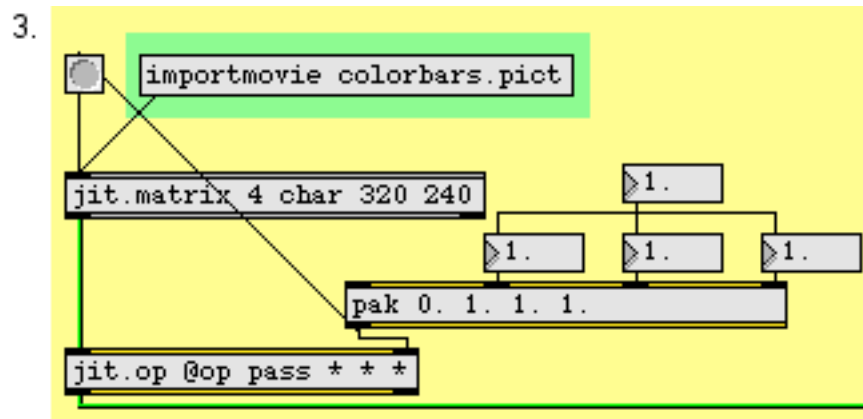
This patch is similar to the first one, with the important difference that we are now working with a 4-plane matrix. This is shown by the first argument to the **jit.matrix** object that generates the matrix we're using. The **jit.pwindow** now shows things in color, interpreting the four planes of Jitter matrix data as separate color channels of alpha, red, green, and blue. Our **jit.op** object in this example has a list of four symbols for its **op** attribute: each symbol sets the mathematical operation for one plane of the incoming matrix. In this patch we're going to pass the first (alpha) plane through unchanged, and add numbers to each of the other planes. (You can mix and match operators like this to your heart's content.)

The **pak** object feeding the right inlet of our **jit.op** object takes four integers and packs them into a list. The only difference between **pak** and the Max **pack** object is that **pak** will output a new list when *any* number is changed (unlike the **pack** object, which needs a new number or a bang in the left inlet to output a new list). The four numbers in the list generated by **pak** determine the scalars for each plane of the matrix coming into the **jit.op** object. In the example above, plane 0 will have nothing added to it (the first argument of the **op** attribute is **pass**). Planes 1, 2, and 3, will have 161, 26, and 254 added to them, respectively. Our **jit.pwindow** object will interpret the cells of the output matrix as lovely shades of magenta (even though we see only one color, there are in fact 12 different cells in the matrix, all set to the same values).

Important Note: If we decided to use only one value for the `op` attribute of the `jit.op` object above (and only used one number as a scalar), `jit.op` would use that mathematical operator and scalar value for *all* planes of the incoming matrix.

Modifying the Colors in an Image

The third example shows a use of `jit.op` on a matrix that already has relevant data stored in it:



Multiplying individual planes with scalars

- Click the **message** box `importmovie colorbars.pict`. The `importmovie` message to `jit.matrix` loads a single frame of an image from a picture or QuickTime movie file into the Jitter matrix stored by the object. It will scale the picture on the disk to the dimensions of its own matrix (in this case, 320 by 240).

Clicking the **button** object shows you image calibration colorbars in the **jit.pwindow** on the right of the patch. In this case, our **jit.op** object has its arithmetic operators set to pass for the alpha plane and ***** (multiply) for the other planes. Since we're working with a 4-plane image, we set each of the scalars using a list of 4 floating-point numbers. Values of 1. in planes 1 through 3 will show you the image as it appears originally:



All scalars at 1.

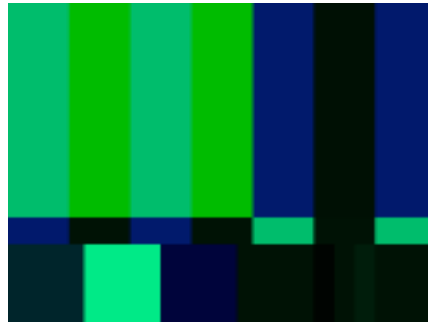
If you set the scalars to 1., 0., and 0., you should see the following image:



The mean reds.

All of the planes (except plane 1) of the matrix containing the colorbars have been multiplied by 0. This will eliminate the alpha, green, and blue planes of the matrix, leaving only the red (plane 1) behind.

Setting intermediate values (such as 0., 0., 1. and 0.5) as the scalars for **jit.op** will give you an image where the colorbars look different:



Pretty, isn't it?

In this case, the alpha channel is ignored and the red channel is zeroed. The blue plane's values are all half of what they were. The green channel (plane 2) is left untouched.

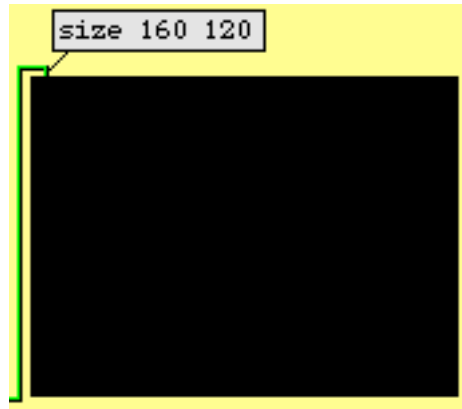
Important Note: Some mathematical scalars in **jit.op** are expressed as floating-point numbers, and some are expressed as integers. This depends on the relevant operator (defined by the **op** attribute), as well as the type of the input matrix. Since all the examples in this tutorial use *char* matrices, it makes sense to use integers when adding to them (any floating-point numbers will be truncated, as the matrix data needs to remain as integers in the range 0-255). If we were using a *float32* matrix as our input, it would make perfect sense to add floating-point numbers to it. Similarly, it's reasonable to multiply a *char* matrix by a floating-point scalar ($240 * 0.5 = 120$, an integer). However, since the matrix output by **jit.op** will still be a *char* matrix (see note below), you will still only get values in the range of 0-255.

If you experiment with the scalar values you will see that you can easily make some of the colorbars disappear or merge with neighboring bars. This is because the colorbars are all set to standard color values with similar ranges. If you show only one channel at a time (by setting all planes but one to 0), four of the seven bars along the top will show color.

We have demonstrated the **+** and ***** operators in this tutorial, but in fact the **jit.op** object can perform a great many other math operations. For a complete list of the possible operators, see the reference page, or double-click on the **p op_list** subpatch in the **jit.op** help file.

Sizing it Up

When you create a **jit.pwindow** object, it will appear in the Max window as 80 pixels wide by 60 pixels tall. You can change its size using its *grow box*, just like many of the user interface objects in Max. If you want to change its size precisely, you can do so using its Inspector or by sending it the `size` message followed by a width and height, in pixels:



Changing the size of a jit.pwindow

If you send a **jit.pwindow** object of a certain size (in pixels) a matrix with a different size (in cells), the **jit.pwindow** object will scale the incoming matrix to show you the entire matrix. If you send a very small matrix to a very large **jit.pwindow**, you will see *pixelation* (rectangular regions in the image where the color stays exactly the same). If you send a small **jit.pwindow** a large matrix, varying degrees of detail may be lost in what you see.

Important Note: in the example above, our **jit.matrix** holding the colorbars had a size (specified by its dim list) of 320 by 240 cells, a plane count of 4, and a type of char. The **jit.op** object (and most Jitter objects you will encounter) recognizes that information and *adapts* to perform its calculation on the entire matrix and output a matrix of the same specifications. If we were to change the **jit.matrix** object to some different size, the **jit.op** object would instantly recognize the change and re-adapt. The **jit.pwindow** object also adapts to the incoming matrix, but in a slightly different way. If the incoming matrix is smaller than its own dimensions, it uses duplicate data to fill all of its pixels. (This results in the pixelation effect described in the previous paragraph.) If the incoming matrix is larger than its own dimensions, it will be obliged to ignore some of the data, and will only display what it can. So, even though the **jit.pwindow** objects in the Tutorial patch never match the size (in cells) of their matrix input, they do their best to adapt to the size of the **jit.op** object's matrix. The **jit.pwindow** in the last example shows you as much as it can of the entire matrix output by the **jit.op** object, but it has to ignore every other row and column in order to fit the 320x240 matrix it receives into its own 160x120 display area.

Summary

The **jit.op** object lets you perform mathematical operations on all the data in a Jitter matrix at once. You can perform calculations on the matrix cells in their entirety or on each plane separately. The mathematical operation that **jit.op** will perform is determined by its **op** attribute, which can be typed in as an **@op** [operator] attribute argument or provided by an **op** [operator] message in the left inlet. For multiple-plane matrices (such as color pictures and video), you can specify the operation for each plane by providing a list of operators (e.g. **op pass****), and you can provide different scalar values for each plane. In *Tutorial 9* you will see how you can use a second Jitter matrix to act in place of a simple scalar.

You can set the size of a **jit.pwindow** object with a **size** [width] [height] message. The **jit.pwindow** will do its best to adapt to the size of any matrix it receives. It will duplicate data if the incoming matrix is smaller than its dimensions, and it will ignore some data if the incoming matrix is larger than its own dimensions. Most Jitter objects do their best to adapt to the dimensions, type, and plane count of the matrix they receive. In the case of **jit.op**, it does not have specified dimensions of its own, so it adapts to characteristics of the incoming matrix.

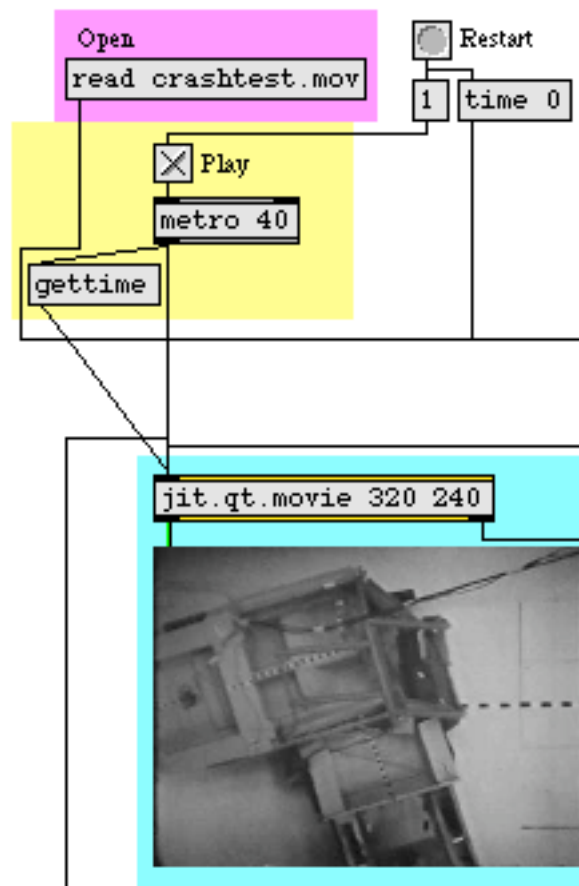
Tutorial 4: Controlling Movie Playback

This tutorial expands on what we've learned thus far about playing back QuickTime movies in Jitter. We'll learn how to get some useful information about the movie that you're playing, as well as how to manipulate the playback of the movie by changing its speed, volume, and loop points.

- Open the tutorial patch *04jControllingMoviePlayback.pat* in the Jitter Tutorial folder.

The two Jitter objects in this patch should already be familiar to you: **jit.qt.movie** and **jit.pwindow**. The rest of the patch will let you experiment with changing the playback behavior of the movie you have loaded into the **jit.qt.movie** object.

The left side of the patch should seem very familiar to you from the very first tutorial:



Open and play the movie.

- Open the file *crashtest.mov* by clicking the **message** box that says read crashtest.mov.

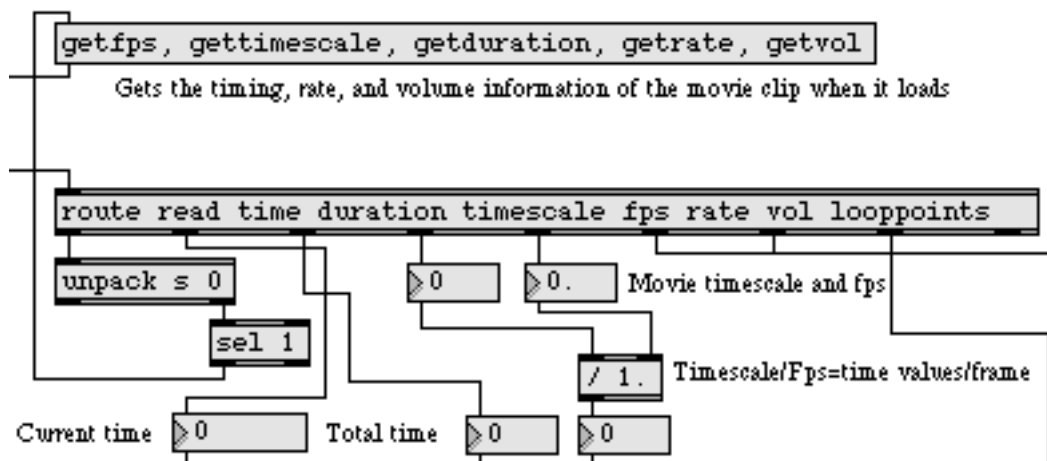
The movie clip should begin playing as soon as it is read into the **jit.qt.movie** object. Since this movie has a soundtrack, you should begin to hear some music as soon as the movie is

loaded. The movie soundtrack will come out of the Sound Manager. If you normally use an ASIO driver with MSP, you will need to connect and set up your Sound Manager outputs so that you can hear them.

You won't see anything in the **jit.pwindow** because, even though the movie is playing, the **jit.qt.movie** object needs a bang message to send a matrix out to the **jit.pwindow**. Start the **metro** object by clicking on the **toggle** box connected to its inlet. You will see the movie's image appear in the **jit.pwindow** object. Don't worry about the **gettime** message yet; we'll get to that below.

Obtaining Some Information About the Movie

The first thing we want to do with this QuickTime movie is get some information about it. The Jitter attribute system lets us query information about Jitter objects at any time, and use that information in our Max patch. Attribute information is always retrieved by sending get messages to a Jitter object's left inlet. We then parse the Max messages the object sends out its rightmost outlet in response (see **What Are Attributes?** for more details).



*Automatically querying the **jit.qt.movie** object.*

The middle of the tutorial patch contains a Max **route** object connected to the right outlet of the **jit.qt.movie** object in our patch. Jitter attributes are always output by objects in the same format that you would set them with in your patch: the *name* of the attribute followed by whatever information the object needs to set that attribute.

When you tell a **jit.qt.movie** to open a movie for playback (by sending it the read message), the object sends a message out its right outlet to tell you that it has found your movie and understood how to play it. If you were to connect a **print** object to the right outlet of the **jit.qt.movie** in the patch and reload the movie in the patch, you would see the following printed in the Max window:

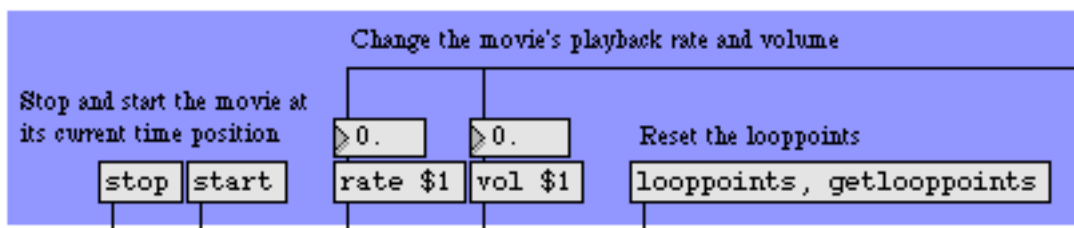
```
read crashtest.mov 1
```

If for some reason the object can't locate the *crashtest.mov* file, you will see a number other than 1 after the name of the file. This message has two purposes: first, to let you know that the movie file has been successfully located and opened; second, so that you can use this message to trigger subsequent actions in the Max patch.

If we look at the first argument to the **route** object, you will see that we've told it to look for a message that begins with read. The rest of that message is sent to an **unpack** object which splits up the remaining list of a symbol (containing the name of the movie) and a number, which indicates success (1) or failure (-1) in opening the movie. The **select** object then sends out a bang message if the movie is opened successfully. The bang then triggers the **message** box above it, which is in turn connected back to the **jit.qt.movie** object.

The **message** box contains the following list of attribute queries, which are parsed by the same **route** object that dealt with the read message described above: getfps, gettimescale, getduration, getrate, getvol. This series of messages will find out the current values stored in the **jit.qt.movie** object for the attributes fps, timescale, duration, rate, and vol. We don't know what those mean yet, but we now have a mechanism by which we can get these attributes every time we successfully load in a new movie into the **jit.qt.movie** object.

Starting, Stopping, and Slowing Down



Some simple movie playback controls

The top of the tutorial patch contains some controls to change the playback behavior of the **jit.qt.movie** object. Sending a stop message to **jit.qt.movie** will freeze the movie's playback at the current point in the movie. Sending a start message will resume playback where you last left off. Any soundtrack that exists in the movie file will stop playing when the movie's playback is halted. Stopping and starting the movie has no effect on the

jit.qt.movie object's matrix output, which is still controlled by the **metro** object. If you stop the movie with the **metro** on, you will still receive a new matrix at the rate of the **metro** object (in this case, 25 times per second), even though all the matrices will be the same.

Changing the rate of the movie will change the speed at which it plays back its video and audio content. Positive rate values will make the movie go forward, with a value of 1 signifying normal playback speed. Negative values will make the movie go backwards. A rate of 0 will stop the movie. The **jit.qt.movie** object takes a floating-point number as the argument to its rate attribute, so a value of 0.5 will make the movie play at half speed, and a value of -2.3 will make the movie play backwards at a bit more than double speed. If you play around with this value, you will note that the soundtrack will speed up, slow down, and play backwards to remain in sync with the video. Once the movie reaches its last frame (or first frame, if you're playing it backwards), it will loop to the opposite end of the file. This behavior can be changed by setting the loop attribute of the **jit.qt.movie** object with a value of 0 (no looping), 1 (regular looping), or 2 (palindrome looping).

The **vol** attribute controls the volume (loudness) of any soundtrack component the movie has. A value of 1 equals full volume, and a value of 0 will turn the sound off.

In this patch, both the rate and the **vol** attributes are initialized by the **message** box in the middle of the patch when the film is loaded. This way they will reflect the values stored in each new QuickTime movie (see below).

Time is on My Side

When a **jit.qt.movie** object opens a new movie, it reads in a lot of information (contained in the movie's *header*) about the movie, including how long it is, how many frames of video are in the movie, and how fast it is meant to be played. We use this *metadata* to control the movie's playback.

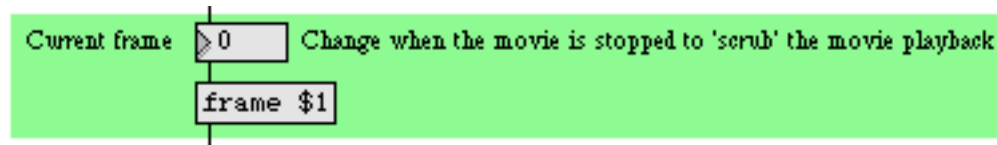
Important note: unlike many Jitter attributes, which are set either by you or the object itself, many attributes used by **jit.qt.movie** are dependent on the current movie file. Different movie files will generate different settings for many of the attributes discussed in this tutorial.

The first three attributes we queried, duration, timescale, and fps, tell us about how the movie file deals with timing. The duration attribute tells us the total length of the movie. This value is not expressed in milliseconds or frames, but in QuickTime time units. The actual length of each time unit depends on the timescale of the movie. The movie's timescale is the timing resolution of the movie per second. Dividing the duration of a movie by its timescale will tell you the approximate length of the movie, in seconds. Our *crashtest.mov* file, for example, has a duration of 2836 time units and a timescale of 600. The movie should run for about 4.73 seconds. If we want to move two seconds into the movie, we could set the

jit.qt.movie object the message time 1200 (1200 time units divided by a timescale of 600 units/second gives us 2 seconds).

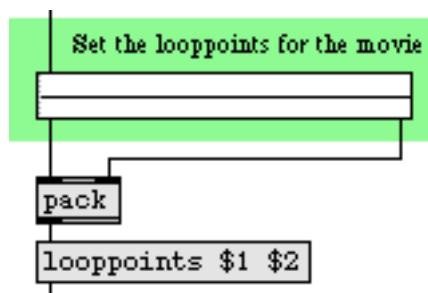
The **fps**, or frames per second, of a movie tells us how many individual video images exist in the movie every second. The higher the **fps** of a movie, the smoother the apparent motion of the movie will be (assuming, that is, that the individual frames are all in some way unique). Some common **fps** rates are 15, 24, 29.97, and 30. Our movie file in this example runs at 15 frames per second, which works out to a new frame of video every 40 time units, or about every 66.7 milliseconds. If we divide the duration of *crashtest.mov* by the number of time units per frame in the movie, we can determine that the movie file has 70 frames. If we wanted to, we could get the total number of frames in the movie by querying the **jit.qt.movie** object with the **getframecount** message, but then we wouldn't get to do the fun math!

Scrubbing and Looping



Displaying and setting the current playback frame

The area at the bottom of the patch contains two controls for further manipulating the movie's playback. The **number box** on the left displays the frame that the movie is currently playing. This value is being updated by the **gettime** message sent into the **jit.qt.movie** object by the **metro** object at the top of the patch; each time a new frame is output the time is updated. If you stop the movie's transport (by sending **jit.qt.movie** a stop message), you can "scrub" through the movie by dragging on the **number box**. The movie will jump to the frame specified as an argument to the **frame** message.



Setting loop points in a movie

Loop points (pairs of time values which specify the beginning and end of a loop) can be sent to a **jit.qt.movie** object by setting the **looppoints** attribute with two integer arguments. The **rslider** in the tutorial patch lets you select regions of the movie that the **jit.qt.movie** object will loop between. The size of the **rslider** has been set to the duration of the movie

through the attribute query we performed when the movie was loaded. You can reset loop points by sending **jit.qt.movie** a `looppoints` message with no arguments (an example of this is at the top of the patch, along with a query message that highlights the entire **rslider**).

Summary

The **jit.qt.movie** object offers a number of simple attributes that allow you to change the way QuickTime content is played. You can stop and start movie playback with those messages. The `rate` attribute lets you change the speed and direction of movie playback. You can control the volume of a movie's soundtrack with the `vol` attribute.

You can get important information about the current movie loaded into the **jit.qt.movie** object by querying attributes such as `duration`, `timescale`, and `fps`. You can go to specific frames in a movie with the `frame` message, and you can set and retrieve `looppoints` for the movie. You can query the current time position of a movie by sending **jit.qt.movie** a `gettime` message.

More powerful functions, such as editing and saving movies, can be accomplished and will be discussed in later tutorials.

Tutorial 5: ARGB Color

Color in Jitter

In this chapter we'll discuss how color is handled in Jitter. We'll focus on the numerical representation of a color, and conversely the visualization of numbers as color. A thorough discussion of color theory—knowledge of how light and matter produce our sensation of color—is well beyond the scope of this tutorial, as is a discussion of the many ways to represent color information digitally. If you wish to learn more about color theory and/or digital representations of color, you can find some other sources of information listed in the *Bibliography*.

Here we present one method of representing color and how that representation is handled in Jitter matrices.

Color Components: RGB

It's possible to produce any desired color by blending three colors of light—red, green, and blue—each with its own intensity. This is known as additive synthesis—generating a color by adding unique amounts of three "primary" colors of light. (The opposite of this is subtractive synthesis: mixing colored pigments, such as paint, which absorb certain colors of light and reflect the rest.) Thus, we can describe any colored light in terms of its intensity at the three frequencies that correspond to the specific colors red, green, and blue.

In Jitter, this is the most common way to describe a color: as a combination of exact intensities of red, green, and blue. For each pixel of an image—be it a video, a picture, or any other 2D matrix—we need at least three values, one for each of the three basic colors. Therefore, for onscreen color images, we most commonly use a 2D matrix with at least three planes.

The Alpha Channel

A fourth plane is often useful for what is known as the *alpha channel*—a channel that stores information about how transparent a pixel should be when overlaid on another image. We won't deal with the alpha channel specifically in this tutorial chapter, but we mention it here because its inclusion is standard in most Jitter objects that deal specifically with representations of color. In most cases, the alpha channel is stored in the first plane (which is plane 0, because planes of a matrix are numbered starting from 0), and the RGB values are in planes 1, 2, and 3.

Color Data: char, long, or float

It's fairly standard in computer applications to use 8 bits of information per basic color value. 8 bits gives us the ability to express 256 (2 to the 8th power) different values. This means that if we use 8 bits for red, 8 for green, and 8 for blue, we can express 16,777,216 (2^{24}) different colors. That's a sufficient variety of colors to cover pretty thoroughly all the gradations we're able to distinguish.

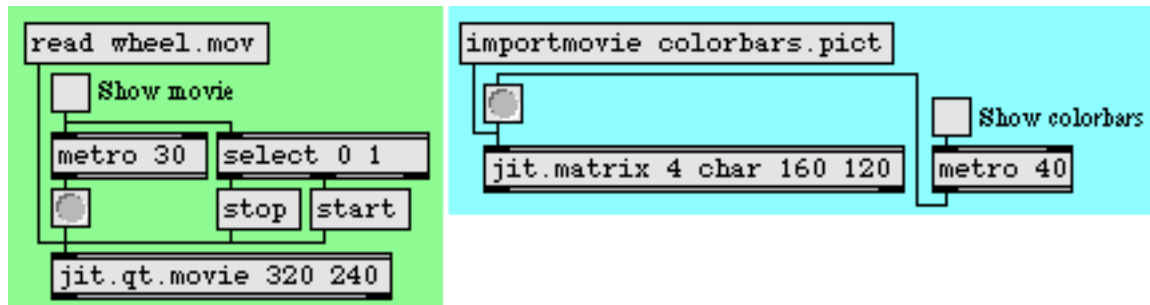
So, if we only need 8 bits of resolution to represent each basic color value, that means that the 8-bit *char* data type is sufficient to represent the value in each plane of a four-plane matrix of color information. We could use the *long*, *float32*, or *float64* data type, and Jitter will certainly let us do that, but we'd be using much larger data types than we really need. Since a full-frame video image can contain a very large number of pixels (a 640x480 image has 307,200 pixels), it often makes most sense—in order to conserve memory and speed up processing—to use the *char* data type.

When a matrix contains 8-bit *char* data, we can think of those 8 bits as representing numbers from 0 to 255, or we can think of them as representing gradations between 0 and 1 (i.e. as a fixed-point fractional number). When a Jitter object containing *char* data receives numerical values from other Max objects, it usually expects to receive them in the form of floats in the range 0 to 1. It will make the internal calculations necessary to convert a float from Max into the proper *char* value. (There are a few exceptions to this rule. For example the **jit.op** object can accept either floats in the range 0-1 *or* ints in the range 0-255 in its right inlet, as demonstrated in Tutorial 3.) For more on the use of the *char* data type in Jitter matrices, see the manual chapter "What's a Matrix?".

Isolating Planes of a Matrix

- Open the tutorial patch *05jARGBcolor.pat* in the Jitter Tutorial folder.

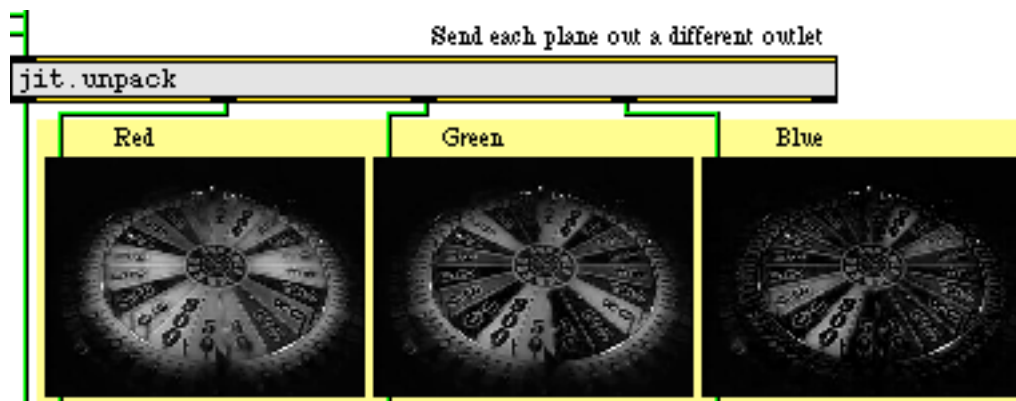
At the top of the patch you are given two different colorful source materials. One is a video of an arcade game, and the other is the standard set of color bars for used for video calibration. You can choose to see one or the other by turning on the one of the **metro** objects (to repeatedly bang the object containing the matrix you want to see).



View a video or a still image

- Click on the toggle marked "Show movie" above the **metro 30** object to view the video.

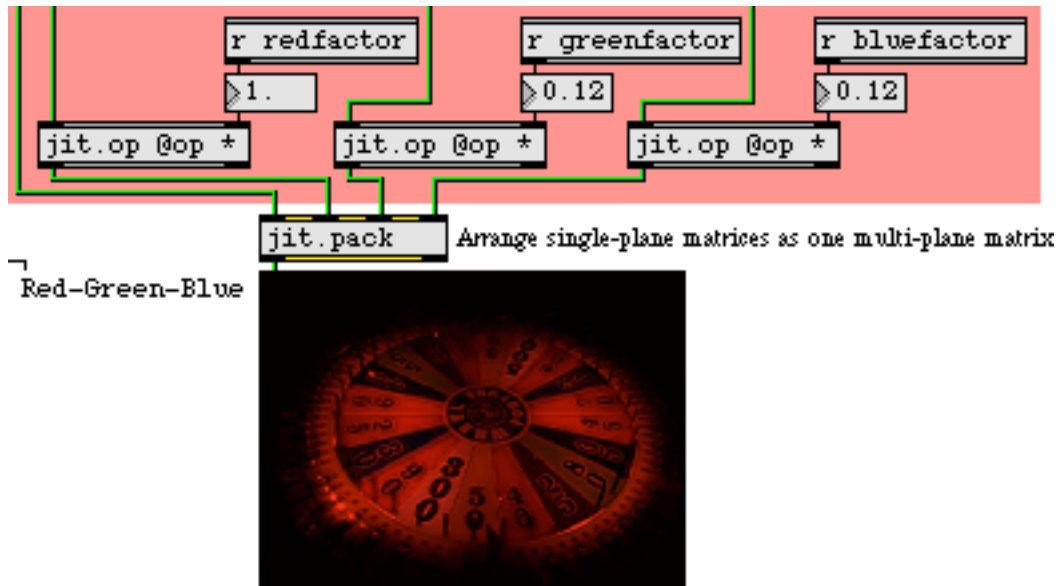
This example patch breaks up a 4-plane matrix of color information into four 1-plane matrices, in order to allow you to see—and modify—each plane individually. We achieve this with an object called **jit.unpack**. Just as the Max object **unpack** breaks a list into individual numbers, **jit.unpack** breaks a multi-plane matrix into individual 1-plane matrices. You can type in an argument to tell **jit.unpack** how many planes to expect in the incoming matrix, but by default it expects four planes since that's the norm for color data. We're interested in seeing the contents of the red, green, and blue planes, so we send planes 1, 2 and 3 to individual **jit.pwindow** objects. In this case we're not interested in the alpha channel, so we don't bother to display plane 0.



Unpacking a multi-plane matrix as single-plane matrices

Here you can see the content of each color plane, in the form of three monochrome images. The lighter pixels indicate higher values for that color. By sending each matrix to a **jit.op** object, we obtain individual control over the strength of each color, and can alter the color balance. We then send the individual (altered) matrices to a **jit.pack** object to recombine them as a 4-plane matrix, for display in the **jit.pwindow**.

- Try, for example, reducing the intensity of green and blue to 0.12, to create a redder image.

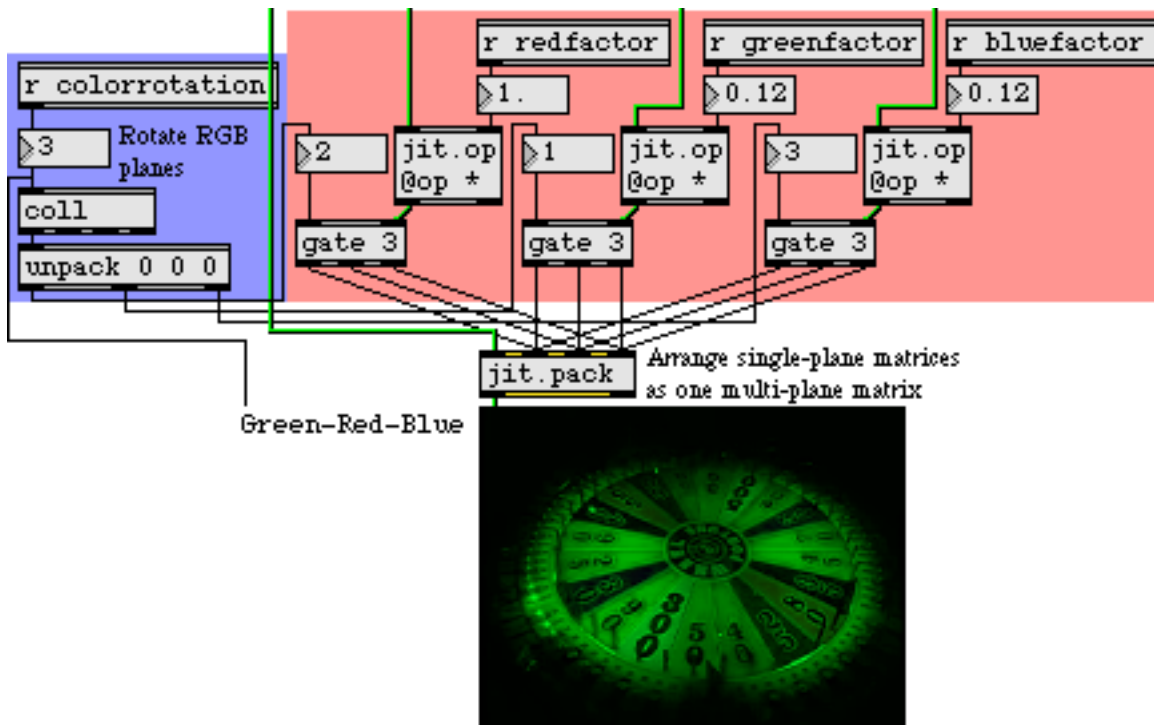


Reducing the intensity of certain colors to alter the color balance

Color Rotation

To demonstrate another color trick, we've sent each color plane through a **gate** object, so that each matrix can be routed to a different inlet (color plane) of **jit.pack**. In this way, the meaning of each plane can be reassigned, and we can try all the permutations of possible color assignments by choosing a setting from the **coll** object on the left side of the patch.

- Drag on the **number box** marked "Rotate RGB planes" to try different reassignments of the three color planes. (Note that plane 0 is sent directly through from **jit.unpack** to **jit.pack**; it's the jit_matrix message coming in the left inlet of **jit.pack** that triggers output of the matrix to the **jit.pwindow**.) If you choose item 3 in the **coll**, you get the result shown below.



The individual color planes are reassigned; the red and green planes are swapped here.

The example above shows the original green and blue planes reduced by a factor of 0.12, and the **gates** are set so that the red and green planes are swapped when they're sent to **jit.pack**, resulting in an image with more green in it. The **coll** object contains all the possible permutations of assignments of the RGB planes.

- Double-click on the **coll** object to see its contents.

0, 0 0 0;

1, 1 2 3;

2, 1 3 2;

3, 2 1 3;

4, 2 3 1;

5, 3 1 2;

6, 3 2 1;

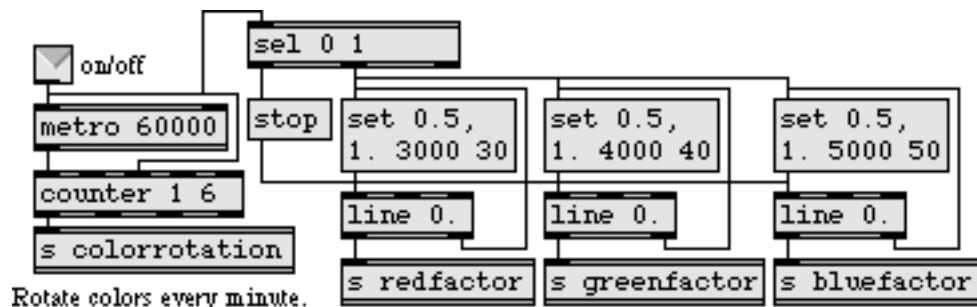
Permutations of RGB plane assignments

The elements in the list coming out of the **coll** are unpacked and sent to assign the outlets of the three **gate** objects. The number sent into **coll** is also sent to a **umenu** (in Label mode) to show the color rotation in words—in this case "Green-Red-Blue".

Automated Color Changes

For one more exercise in color modification, we've made an automated process for continually changing the scaling and rotation of colors.

- Click on the **toggle** marked "Automate color changes". The scaling factors for the three color planes all change continually. Double-click on the **patcher** colorgames object to see the contents of the subpatch.



Red factor goes from 0.5 to 1 every 3 seconds.
 Green factor goes from 0.5 to 1 every 4 seconds.
 Blue factor goes from 0.5 to 1 every 5 seconds.

[colorgames] subpatch

The subpatch uses **line** objects to send values that progress from 0.5 to 1 for each of the color scaling factors. The red factor changes over the course of 3 seconds, green in 4 seconds, and blue every 5 seconds. (The three **line** objects thus come into sync once every 60 seconds.) Every 60 seconds, a new color rotation is selected by the **metro-counter** combination.

- Close the *[colorgames]* subpatch window. You can try all of these same color modifications on different source material. Click on the **toggle** marked "Show movie" to stop the **metro**. (Note that we also use this **toggle** to start and stop the movie playing in the **jit.qt.movie** object. There's no reason to keep the movie playing if we're not even watching it.) Now click on the **toggle** marked "Show colorbars" to display the colorbars. Experiment with changing the scaling factors and rotation on this image.

Summary

When a **jit.window** or **jit.pwindow** receives a single-plane 2D matrix, it displays the values as a monochrome (greyscale) image. When it receives a 4-plane 2D matrix, it interprets the planes as alpha, red, green, and blue values, and displays the resulting colors accordingly. This ARGB representation in a 4-plane matrix is the most common way of representing colors in Jitter.

Because each of the basic colors only requires 8 bits of precision to represent its full range, it's common in Jitter to use the *char* data type for color data. Thus, most of the QuickTime-related objects (such as **jit.qt.movie**) and many of the objects specifically designed for manipulating colors (such as **jit.brcosa** and **jit.colorspace**, which are demonstrated in later tutorial chapters) expect to receive 4-plane 2D matrices of *char* data. (Many other objects adapt readily to other types of data, though. Check the reference documentation for the object in question when you're in doubt.) You can think of the *char* data as representing values 0 to 255, or you can think of them as representing fractional values from 0 to 1. Most of the time, objects that contain *char* data expect to receive numerical values from other Max objects specified as floats in the range 0 to 1.

The **jit.unpack** object splits a multi-plane matrix into individual single-plane matrices. The **jit.pack** object combines single-plane matrices into a single multi-plane matrix. By treating each plane separately, you can control the color balance of an image, or even reassign the meaning of the individual planes.

Tutorial 6: Adjust Color Levels

jit.scalebias

This tutorial elaborates on the color discussion of the previous chapter, and introduces an object that is specially designed for modifying the ARGB color planes of a matrix:

jit.scalebias.

The term *scale* in this instance refers to the process of scaling values by a certain factor; that is, *multiplying* them by a given amount. The term *bias* refers to offsetting values by *adding* an amount to them. By combining the processes of multiplication and addition, you can achieve a linear mapping of input values to output values.

Because **jit.scalebias** is concerned with modifying ARGB color information in an image, it handles only 4-plane matrices of the *char* data type. (See Tutorial 5 for a discussion of ARGB color and *char* data.)

Math with char data

As mentioned in the previous chapter, 8-bit *char* data can be represented as whole number values from 0 to 255 or as fractional values from 0 to 1. In *Tutorial 2*, for example, we saw that the **jit.print** object reports *char* data as integer values from 0 to 255. However, in many cases where we modify *char* values within a matrix (by changing one of its attributes), the Jitter object will expect to receive the attribute value as a float. Because this can be a little confusing, we've provided a demonstration in this tutorial patch.

in each plane of each cell (there's only one cell in this matrix), and we can see that the value (when stored in the matrix as a *char*) is 251 (or 0.984 on a scale from 0 to 1).

(Just as a mental exercise, can you calculate the values **jit.scalebias** will produce in the red and blue planes in the above example? Since the values in those planes in the original matrix are 0, the values in the resultant matrix will be 0 times 2.0 plus 0.2, which equals 0.2, which is equal to 51 on a scale from 0 to 255. So the RGB values being displayed by the **jit.pwindow** object at the bottom are 51 251 51.)

Some more examples of math with *char* data

If the preceding explanation was crystal clear to you, you might want to skip over these additional examples. But in case you're still not totally clear on how math operations with *char* data (and **jit.scalebias** in particular) work, here are a few more examples.

- One by one, click on each of the presets in the **preset** object, proceeding from left to right. In the paragraphs below we explain each of the preset examples.
1. The value in the green plane of the original matrix is 255. (This is equal to 1.0 on a scale from 0 to 1.) The **jit.scalebias** object multiplies this by 0.5, which results in an internal value of 127.5; however, when storing the value as a *char*, **jit.scalebias** truncates (chops off) the fractional part, and stores the value as 127.

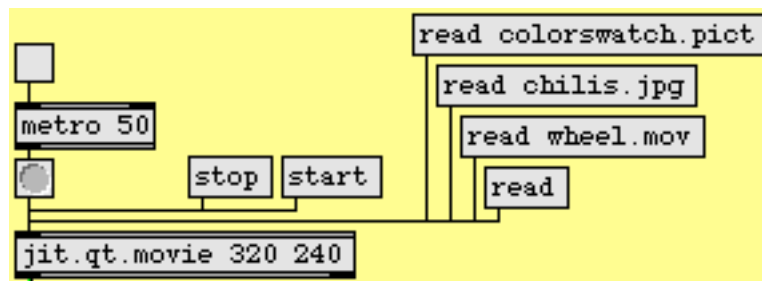
This yields a rather imprecise result. (127 on a scale from 0 to 255 is equal to 0.498 on a scale from 0 to 1, as opposed to the 0.5 we would expect.) But that's the best we can do with 8-bit *char* data. In cases where you need greater precision than that, *char* data is not for you. You would need to use a matrix of *long*, *float32*, or *float64* data, and use **jit.op @op *** and **jit.op @op +** objects instead.

2. The original value is 100, and we double it (scaling factor 2.0) and get the expected result of 200. There is no loss of precision in this case.
3. The original value is 100 (0.392). We scale it by a factor of 1.0, which leaves it unchanged, then we add -0.2 to it—that is, we subtract 0.2 from it—to get a result of 49 (that is, 0.192).
4. $0.392 \text{ times } 2.0 \text{ plus } 0.2 = 0.984$. On a scale from 0 to 255, that's 251.
5. This example and the next one show what happens when the result of the multiplication and addition operations exceeds the capacity of an 8-bit *char*. **jit.scalebias** will simply clip (limit) the result to the maximum or minimum limit of a *char*. Here, $0.392 \text{ times } 4.0 \text{ equals } 1.568$ (that is, $100 \text{ times } 4 \text{ equals } 400$), so the result is set to the maximum allowable, 255.

6. In the other direction, 0.392 minus 0.5 equals -0.108, so the result is set to 0.
 7. It's noteworthy that these imprecisions and limits only occur at the point when the result is re-stored as a *char*. Up until then, the values are calculated internally as *floats*, so the precision is retained. Even though the multiplication takes the internal value beyond the 0-1 range, no limiting occurs internally, and the addition operation can bring it back into range. Here, 0.392 times 3.0 (= 1.176) minus 0.5 equals 0.676. When this is stored as a *char*, however, a small imprecision occurs. 0.676 on a scale from 0 to 255 equals 172.38, but the fractional part is truncated and the stored value is 172 (i.e., 0.675).
 8. For no change, the scaling factor should be 1 and the bias offset should be 0.
- Try some more values yourself, until you're satisfied that you understand **jit.scalebias** and the results that occur with 8-bit *char* data. When you have finished, close the *[explain_scalebias]* window.

Adjust Color Levels of Images

Now let's apply **jit.scalebias** to color images. In the top left corner of the tutorial patch, you can see a familiar configuration: a **jit.qt.movie** object with a read **message** box to load in a movie and a **metro** object to trigger **jit_matrix** messages from **jit.qt.movie**. In this patch we'll modify the matrices with multiplications and additions in **jit.scalebias**.

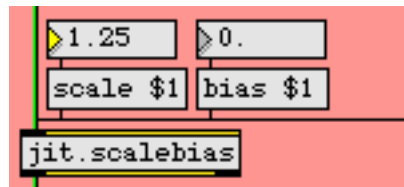


Load in a picture or a movie.

- Click on the **message** box *read chilis.jpg* to read in a JPEG picture. Note that we're reading a still image—not a video—into the **jit.qt.movie** object. QuickTime can handle a wide variety of media formats, including still images in PICT or JPEG format. **jit.qt.movie** treats still images just as if they were 1-frame-long videos.

The output of **jit.qt.movie** will go to **jit.scalebias** for processing, and will then be displayed in the **jit.pwindow**. (You can ignore the **jit.matrix** object for now. We'll discuss its use later in this chapter.) The *scale* and *bias* values can be changed by modifying the *scale* and *bias* attributes of **jit.scalebias**.

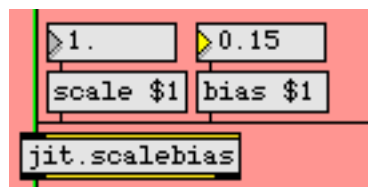
- Click on the **toggle** to start the **metro**. Try dragging on the **number box** above the **scale \$1** message box, to increase the value of the *scale* attribute to 1.25.



Boost the brightness of the image; with scale, higher values get boosted more.

This will increase all non-zero values in all four planes of the image by a factor of 1.25 (a 25% increase). Note that multiplication has the effect of increasing larger values by a greater amount than smaller values. For example, if the red value of a particular cell in the original matrix is 200, it will be increased to 250 (a net increase of 50), while the blue value of the same cell might be 30 and would be increased to 37 (a net increase of 7).

- Try increasing the *scale* attribute to a very large value, such as 20. Values that were 13 or greater in the original matrix will be pushed to the maximum of 255 (and even the very small values will be increased to a visible level), creating an artificially "overexposed" look.
- Try decreasing the *scale* attribute to a value between 0 and 1. As you would expect, this darkens the image. A *scale* value of 0 or less will set all values to 0.
- Return the *scale* attribute to 1. Now try adjusting the *bias* attribute. This adds a constant amount to all values in the matrix. Positive values lighten the image, and negative values darken it.



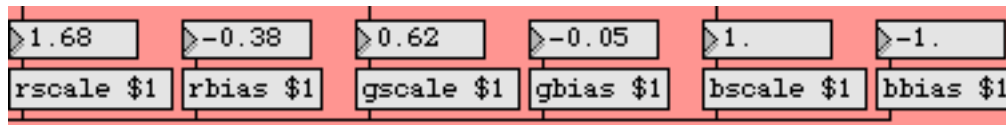
Boost (or reduce) the level of all values by a constant amount.

- Here are a couple of more extreme *scale* and *bias* settings you might want to try. Set the *scale* value to 40 and the *bias* value to -20. This has the effect of pushing almost all values to either 255 or 0, leaving only a few colors other than white or black. Now try setting the *scale* value to -1 and the *bias* value to 1. This has the effect of inverting the color scheme by making all high values low and all low values high. Reducing the *scale* value even further (to, say, -4 or -8) creates a similar inversion, but only values that were low in the original will be boosted back into the 0-1 range by the positive *bias* value.

Adjust planes individually

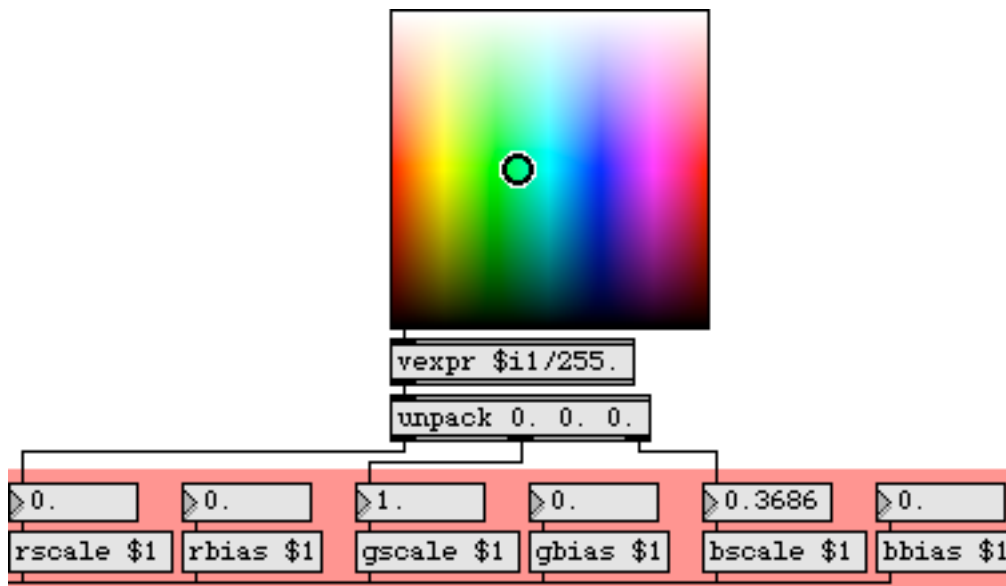
You can adjust the levels in each plane individually in **jit.scalebias**, using the attributes *ascale*, *abias*, *rscale*, *rbias*, etc.

- Set the *scale* value back to 1 and the *bias* value back to 0. Then experiment with adjusting each color plane independently by providing new values for the appropriate attributes.



Adjust levels for each color plane

We've made the process a bit more "interactive" by giving you a controller that lets you adjust the scaling of all three color planes at once. When you click or drag in the **swatch** object, it sends out a three-item list representing the RGB color values at the point where your mouse is located. Those values are expressed on a scale from 0 to 255, so we use the **vexpr** object to change all values in the list to the 0 to 1 range. We then use **unpack** to break that list into three separate floats, and we use those values to alter the *rscale*, *gscale*, and *bscale* attributes of **jit.scalebias**.



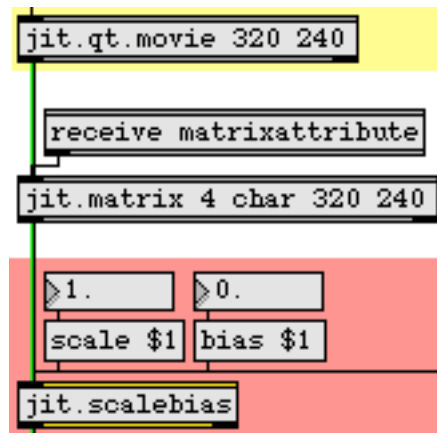
*Values from **swatch** used as attribute values for **jit.scalebias***

- Drag on the **swatch** object to scale the RGB planes all at the same time. (Since this produces scaling values in the range 0 to 1, you're effectively reducing all the levels, so this will generally darken the resulting image somewhat.)

- You can try all of these operations on different images. Read in some other colorful images such as *colorswatch.pict* or *wheel.mov* (or any other image) and experiment with adjusting the color levels.

Reassign planes of a matrix

In the previous tutorial we used **jit.unpack** and **jit.pack** to reassign the planes of a matrix. There is another way to do that, using the **planemap** attribute of the **jit.matrix** object. In this example, we pass the matrix output of **jit.qt.movie** through a **jit.matrix** object just so that we can demonstrate the **planemap** attribute.



*We can reassign the planes of a matrix as it goes through **jit.matrix***

The **planemap** attribute of **jit.matrix** allows us to "map" (assign) any plane of the incoming matrix to any plane of the outgoing matrix. The word **planemap** is followed by as many numbers as there are planes in the matrix (four in this case). Each place in the list stands for an output plane (the first place stands for output plane 0, the second place for output plane 1, etc.), and the value of that number states the input plane that will be assigned to it. By default, the **planemap** values are 0 1 2 3 (etc.), so that each plane in the input matrix is assigned to the same plane in the output matrix. But we can change those assignments as we like. For example, if we send **jit.matrix** the message **planemap 0 3 2 1**, we are assigning input plane 3 to output plane 1 (since the value 3 is in the list location for output plane 1), and input plane 1 to output plane 3. Effectively, we're swapping the red and blue color planes of the image.

- Click on the **message** box read *wheel.mov* and start the **metro** to display the movie. (Set the scale attribute of **jit.scalebias** to 1 and the bias attribute to 0, so that you're seeing an unaltered image in the **jit.pwindow**.) Now, in the bottom right portion of the patch, click on the **message** box **planemap 0 3 2 1** to swap the red and blue planes in the matrix. Click on the **message** box **planemap 0 1 2 3** to go back to the normal plane mapping.

If we set all three of the RGB output planes to the same input plane, we get equal values in all three RGB planes, resulting in a greyscale image.

- Click on the **message** box `planemap 0 1 1 1` to see this effect. The value 1 is in the list location for each of the three RGB planes, so the red plane of the original is used for all three RGB planes of the output matrix.

To rotate through all of the different color plane rotations, we've filled a **coll** object with various plane mappings (similarly to the way we did in the previous chapter), and we will send those assignments to the **jit.matrix** to change the settings of its `planemap` attribute.

- Double-click on the **patcher** `rotatecolorplanes` object to see the contents of the subpatch. It simply counts from 1 to 6, to step through the different mappings stored in the **coll** object in the main patch. (And when it's turned off it sends out the number 1 to reset to the default plane mapping.) Close the `[rotatecolorplanes]` window.
- Click on the **toggle** above the **patcher** `rotatecolorplanes` object to step through the different plane mappings at the rate of one setting per second. Change the rate **number box** above the right inlet to a smaller value (say, 80) to see a flickering effect from rapid plane reassignment.

In the next tutorial chapter, you'll see how to rotate image hue in a subtler way, using **jit.hue**, and you'll see other ways to adjust color levels with the **jit.brcosa** object.

Reading and Importing Images

In this tutorial patch, we loaded three different kinds of images into the **jit.qt.movie** object: PICT and JPEG still images, and a QuickTime movie. It may seem a little strange to read still images into an object made for playing movies, but in fact QuickTime can read many types of media files, and **jit.qt.movie** knows how to read them all. (You could even read an AIFF audio file into **jit.qt.movie**, listen to it with `start` and `stop` messages, jump to different locations with the `time` attribute, etc.! Of course, you won't see any visual matrix info in that case.)

For still images, it's just as easy to load them directly into a **jit.matrix** object with the `importmovie` message, as demonstrated in *Tutorial 3*. If you import a QuickTime movie that way, only one frame of the movie will be stored in the **jit.matrix**.

In this patch, we used **jit.qt.movie** to load in all the images. The first reason is because one of the things we wanted to load was a movie (not just one frame of the movie). The second reason is because we wanted to demonstrate the `planemap` attribute of **jit.matrix**. The `planemap` attribute is only appropriate if there is an actual input matrix (a `jit_matrix` message

coming in the left inlet). If we imported the images directly into **jit.matrix** with `importmovie`, the `planemap` attribute would have no effect.

Summary

The **jit.scalebias** object uses multiplication and addition to modify all the values in a particular plane of a 4-plane *char* matrix—or all planes at the same time. The `scale` attribute is a factor by which each value in the matrix will be multiplied; the `bias` attribute is an amount to be added to each value after the multiplication takes place. The `scale` and `bias` attributes affect all four planes of the matrix. To affect only one plane at a time, use the attributes for that particular plane, such as `ascale`, `abias`, `rscale`, `rbias`, etc.

You must supply the values for these attributes as floats (numbers containing a decimal point). To perform the multiplication and addition operations, **jit.scalebias** treats the *char* values as fractional values in the range 0 to 1, performs the math with *floats*, then converts the results back to *chars* (whole numbers from 0 to 255) when re-storing them. Results that exceed the range 0 to 1 will be limited to 0 or 1 before being converted back to *char*.

You can reassign planes of a matrix using the `planemap` attribute of **jit.matrix**. The arguments to `planemap` are the output planes listed in order, and the values in the list are the input planes to be assigned to each output plane. So, for example to assign the plane 1 of an input matrix to all four planes of the output matrix, the attribute setting should be `planemap 1 1 1 1`.

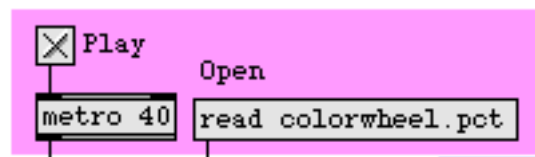
jit.scalebias provides a powerful tool for adjusting color levels in a 4-plane *char* (ARGB color) matrix. More such tools are presented in the next tutorial chapter.

Tutorial 7: Image Level Adjustment

This tutorial shows how to adjust levels of brightness, contrast, and saturation in Jitter matrices that contain image data. We will also look at the concept of hue and hue rotation.

- Open the tutorial patch *07jImageAdjustment.pat* in the Jitter Tutorial folder.

The tutorial patch has two new objects in it: **jit.brcosa**, which allows you to control brightness, contrast, and saturation of an image stored in a Jitter matrix, and **jit.hue**, which lets you rotate the hue of an image:



Open and view the image.

- Open the file *colorwheel.pct* by clicking the **message** box that says read colorwheel.pct. View the movie by clicking the **toggle** box to start the **metro**.

You should see a color wheel appear in the **jit.pwindow** at the bottom of the patch:



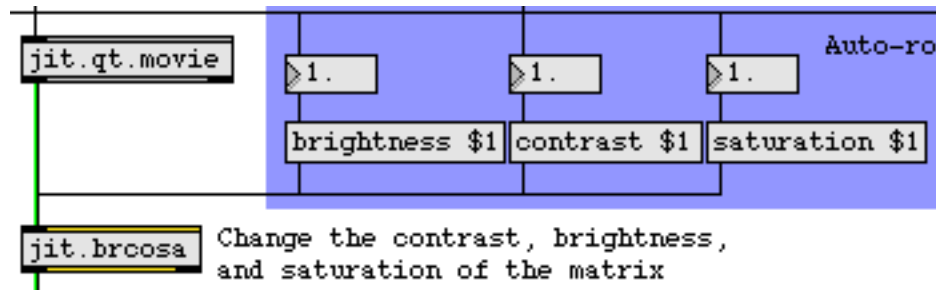
The Color Wheel of Fortune

The **jit.pwindow** object shows us the image after it has passed through our two new objects. We'll talk about **jit.brcosa** first, and then get to **jit.hue**.

Brightness, Contrast, and Saturation

The **jit.brcosa** object takes a 4-plane *char* Jitter matrix and, treating it as ARGB image data, allows you to alter the brightness, contrast, and saturation of the processed matrix. The three attributes are called, not surprisingly, brightness, contrast, and saturation. The default

values of 1.0 for the three attributes result in the matrix passing out of the object unchanged:



*Using the **jit.brcosa** object*

- Change the attributes of the **jit.brcosa** object and observe how the output matrix changes.

The **brightness** of an image refers to its overall lightness or darkness when compared to a reference color (usually black). Changing the brightness attribute is equivalent to multiplying the values in the matrix by that value. A brightness value of 0 will turn the image to black; values above 1.0 will gradually increase all non-0 cell values until they clip at white (255). A value of 0.5 will significantly darken the image, reducing its natural range of 0-255 to 0-127. Some brightness values are shown below on the color wheel:



The color wheel with brightness values of 0.5, 1.5, and 10, respectively

Note that cell values clip at 0 and 255 when altered in this way. This is why the rightmost image, with a brightness of 10, is mostly white, with color showing only in areas where one or more of the visible color planes (RGB, or planes 1, 2, and 3) are 0 in the original matrix.

Image **contrast** can be expressed as the amount the colors in an image deviate from the average luminosity of the entire original image (see below). As the contrast attribute of **jit.brcosa** is increased above 1.0, cell values above the average luminosity of the entire matrix are lightened (increased), and values below the average are darkened (decreased). The result is a dynamic expansion of the matrix image so that light values get lighter and dark values get darker. Settings for contrast below 1.0 reverse the process, with darker tones getting lighter and lighter tones getting darker until, at a contrast value of 0.0, you only

retain the average grey luminosity of the entire image. Negative values invert the color of the image with the same amount of overall contrast.

Technical Detail: The average luminosity of a matrix can be computed by averaging the values of all the cells in the matrix, keeping the planes separate (so you get individual averages for Alpha, Red, Green, and Blue). The three visible planes are then multiplied by the formula:

$$L = .299 * \text{Red} + .587 * \text{Green} + .114 * \text{Blue}$$

The value L will give you the average luminance of the entire matrix, and is used by **jit.brcosa** to determine the threshold value to expand from when adjusting contrast.

Here are some example contrast settings:



The color wheel with contrast settings of 0.3, 2, -1, and 100

The first example shows the color wheel with its contrast drastically reduced (the cell values are all close the average luminosity of the matrix). The second example shows an increased contrast. Note how the lighter shades in the middle of the color wheel begin to approach white. The third example shows a negative contrast setting. The colors are inverted from the original, but the average luminosity of the matrix is the same as in the original. The last example shows the result of a massive contrast boost. Cell values in this example are polarized to 0 or 255.

Image **saturation** reflects the ratio of the dominant color in a cell to the less dominant colors in that cell. As saturation values decrease below 1.0, all color values in a cell will become more similar and thus *de-saturate* towards grayscale. Values above 1.0 will push the colors farther away from one another, exaggerating the dominant color of the cell. As with contrast, a negative value for the saturation attribute will invert the colors but retain the same luminosity relationship to the original.

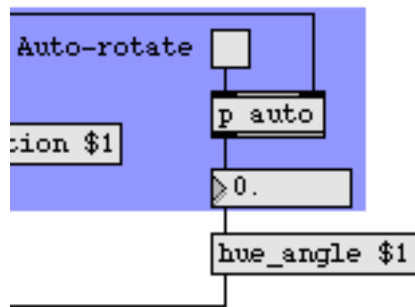


The color wheel with saturation values of 0.2, 2 and -1

The first image is de-saturated, so each cell value in the matrix is being drawn toward the value of the luminosity of that cell. The second image is over-saturated, so the individual colors are much brighter (or darker) than their originals. The third image maintains the original luminosity of the color wheel but inverts the colors.

Hue and Cry

The **jit.hue** object allows you to *rotate* the hue of the input matrix. Setting the `hue_angle` attribute rotates the hue of the input matrix a specified amount in degrees:



The `hue_angle` attribute rotates the hue of the input matrix.

The **hue** of a matrix cell can be thought of as its basic color (e.g. magenta). Image hue is visualized as a rainbow (or a color wheel like the one we've been using) which goes from red to green to blue and back to red again. This value (which is specified in degrees from 0-360), along with the saturation and lightness of the image, can be used to describe a

unique color, much in the same way that a specific series of RGB values also describe a unique color. By rotating the hue of an image forward, we shift the red part of the color spectrum so that it appears green, the green part of the spectrum to blue, and the blue part to red. A negative hue rotation shifts red into blue, etc. Hue rotations in increments of 120 degrees will transpose an image exactly one (or more) color planes off from the original hue of the image.

Technical Detail: Our eyes perceive color through specialized receptors in our retina called *cones* (another type of receptor exists that responds to low levels of light but doesn't distinguish color—these receptors are called *rods*). Cones in our eyes are differentiated by the wavelength of light they respond to, and fall into three categories: L-sensitive receptors which respond to long wavelength light (red), M-sensitive receptors which respond to medium wavelength light (green), and S-sensitive receptors which respond to short wavelength light (blue). Just as our auditory system is weighted to be more perceptive to frequencies that are within the range of human speech, the distribution of cones in our eyes is weighted towards the middle wavelengths which are most crucial to perceiving our environment. As a result, we have roughly twice as many green receptors in our eyes as the other two colors. This explains why the luminance formula (above) allocates almost 60% of perceived luminosity to the amount of green in an image. Camera technology has been developed to emulate the physiology of our eyes, so cameras (and film) are also more sensitive to green light than other colors.

- Click the **toggle** box to automate the hue_angle of the color wheel. Note that when the hue_angle reaches 360 degrees, the original matrix image is restored.



Our color wheel at various hue rotations (0-360 degrees)

Summary

The **jit.brcosa** and **jit.hue** objects allow you to modify the brightness, contrast, saturation, and hue of an input matrix. You use the two objects to perform tasks such as dynamic level adjustment (e.g. autoexposure), color correction, or idiosyncratic hue displacement.

Tutorial 8: Simple Mixing

Mixing Two Video Sources

One of the most common and useful things to do with video is to mix two images together.

The simplest form of video mixing is achieved by just adding the two images together, with the intensity of each of the two images adjusted in whatever proportion you desire. By fading one image up in intensity while fading the other down, you can create a smooth *crossfade* from one image to the other.

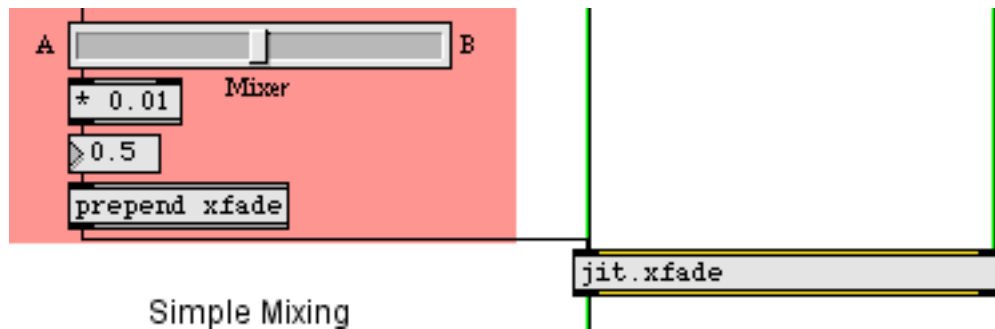
Jitter provides an object that accomplishes mixing and crossfading for you, called **jit.xfade**.

Technical Detail: This type of mixing involves adding each value in one matrix to the corresponding value in the other matrix—cell-by-cell and plane-by-plane—and outputting a matrix that contains all the sums. If we just did that, however, we would get an output image in which all the values are greater than in either of the input images, so the result would be lighter than the originals (and some of the values might even clip at 255 if the matrices contain *char* data). Therefore, it's normal to scale down the intensity of one or both of the images before adding them together. For example, to get an equal mix of the two images, we would scale them both down by an equal amount (say, by a factor of 0.5) before adding them.

jit.xfade

The **jit.xfade** object takes a matrix in each of its two inlets, scales the values of each matrix by a certain amount, adds the two matrices together, and sends out a matrix that contains the resulting mix. The scaling factor for each of the two input matrices is determined by the object's *xfade* attribute. The *xfade* value is a single (float) number between 1 and 0. That value determines the scaling factor for the matrix coming in the right inlet. The matrix coming in the left inlet is scaled by a factor of $1 - \text{xfade}$. So, if you gradually increase the *xfade* value from 0 to 1, the output matrix will crossfade from the left input to the right input.

- Open the tutorial patch *08jSimpleMixing.pat* in the Jitter Tutorial folder. Two source videos are read in automatically by the **loadbang** object. Start the **metro**. You'll see only the left video at first. Drag on the **hslider** to change the xfade value, which will give you a mix of the left and right matrices.

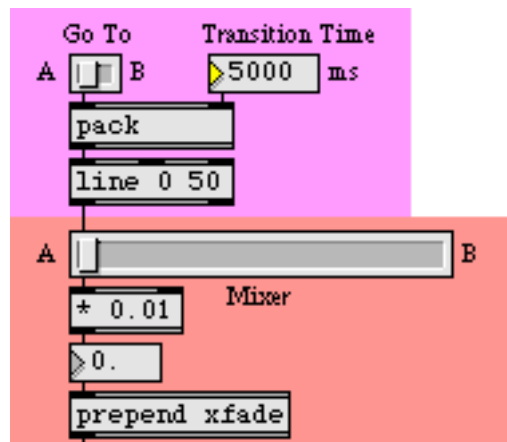


A value of 0.5 gives an equal mix of left and right matrices

Automated Crossfade

A crossfade is one of the most common ways of making a transition from one image to another. It might be very gradual—over the course of several seconds—or it might be quite rapid, lasting a fraction of a second, to make a sudden transition that is slightly smoother than a jump cut.

In the upper left portion of the patch, we've made an automated fader from video A to video B (or vice versa). The crossfade can take any amount of time; you can specify the transition time with the **number box**.



Automated process to send a continuously changing xfade value

- Using the **number box**, set a slow transition time (say, 5000 ms) so that you can see the effect of the crossfader. Click on right side of the **Go To** switch to fade to video B.

The *Go To* switch is actually a small **hslider** with a range of 2 and a multiplier of 100, so the only two values it can send out are 0 and 100. Clicking on the right side of the switch sends out the value 100, the **pack** object sends out the message 100 5000, and the **line** object sends out continuous values from 0 to 100 (one new value every 50 ms) over the course of five seconds. Those values are then multiplied by 0.01 to give a smoothly changing *xfade* value from 0 to 1.

Summary

Adding two matrices together is the simplest way to do A-B mixing of video images. To achieve the desired balance of the two images, you first scale each matrix by a certain factor. You can crossfade from one image to another by decreasing the scaling factor of one image from 1 to 0 as you increase the scaling factor of the other image from 0 to 1.

The **jit.xfade** object makes it easy to mix and/or crossfade two matrices. Its *xfade* attribute determines the balance between the two matrices. Changing the *xfade* value continuously from 0 to 1 performs a smooth A-B crossfade. You can use **line** or any other Max timing object to automate crossfades.

Tutorial 9: More Mixing

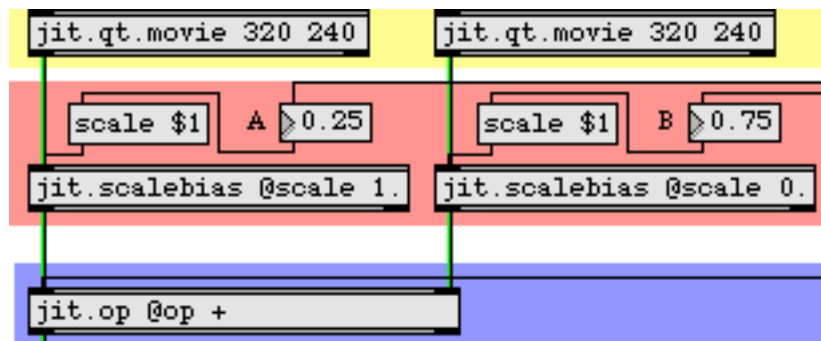
Mixing and Crossfading Made Explicit

In the previous chapter we explained how the **jit.xfade** object uses scaling (multiplication) and addition to mix two matrices together in varying proportions. In this tutorial we'll use the **jit.scalebias** and **jit.op** objects to perform those mathematical operations ourselves.

This will provide us with a few advantages. First, it will allow us to demonstrate the mathematics of mixing and crossfading explicitly. Second, it will allow us to demonstrate how **jit.op** can perform math operations using two matrices as its inputs. (In *Tutorial 3* we showed **jit.op** in action with scalar values operating on a single matrix.) Third, it will allow us to specify balance settings (scaling factors) for the two matrices independently, providing more flexibility than the **jit.xfade** object. Finally, because **jit.op** can implement so many different kinds of math operations, we can try other means of combining matrices to see different visual effects.

Mixing Revisited

- Open the tutorial patch *09jMoreMixing.pat* in the Jitter Tutorial folder.



*Multiplication and addition to mix /crossfade matrices (replicating **jit.xfade**)*

Here you see each of two different videos being scaled down (darkened) by some factor between 0 and 1 with **jit.scalebias**. Below that, you see a slightly new wrinkle in the use of **jit.op**: the input to *both* inlets is a matrix. When we do this, **jit.op** performs the specified math operation on every single value individually, pairing each value in the left matrix with the corresponding value in the right matrix. This lets us add all the values in the two matrices, effectively mixing the images.

Note: In cases where the dimensions, planecount, and data type of the two matrices are not the same, **jit.op** uses the attributes of the matrix that's coming in the left inlet. If the matrix in the right inlet is smaller than the matrix in the left inlet (has fewer dimensions or planes, or fewer cells in a dimension), a value of 0 is assumed for the missing locations. If the data type of the matrix in the right inlet does not match the data type of the matrix in the left inlet, values in the right inlet's matrix will be converted before calculations are performed. (For example, if the right matrix is of type float32 and the left matrix is of type long, values in the right matrix will have their fractional part truncated before calculations are performed.)

Another note: In MSP you can add to audio signals together simply by sending them in the same inlet, because all audio signal inlets perform an automatic addition of signal vectors internally when more than one signal patch cord is attached to an inlet. *This is not the case with Jitter matrix inlets.* Therefore, if you want to add two matrices cell-by-cell, as we're doing in this example, you should use a **jit.op** object with the **op** attribute set to +.

The result of these multiplications and this addition is comparable to what the **jit.xfade** object performs internally. You can verify this by using the controls in the top right part of the patch—which are nearly identical to those of the previous chapter—to crossfade the videos.

- Start the **metro** and use the *Mixer* slider to perform a crossfade from video A to video B.

Note that we send the crossfade value directly as the scale attribute for the B video, and at the same time we use a **!- 1** object to scale the A video by 1 minus that value. That way, the sum of the two scaling factors always equals 1, as it does in **jit.xfade**.

Combine Matrices Using Other Operators

Addition is perhaps the most obvious operation to perform with two matrices, but it's not the only one possible. By changing the **op** attribute of the **jit.op** object, we can try out many other operations to see what kind of visual effect they create.

- Set a very gradual crossfade time in the *Transition Time* **number box** (say, 10000 ms). Choose an operator other than + in the *Operator* pop-up menu. Now click on the *Go To* switch to begin the crossfade. You can see how that particular operator looks when implemented with two video matrices.

The pop-up menu contains a few of the many operators provided by **jit.op**. Here's a brief description of each operator in the menu.

- + Add the values of B to A.
- m Subtract the values of B from A, then perform a modulo operation to wrap the result back into the desired range.
- max Use whichever value is greater, A or B.
- absdiff Subtract the values of B from A, then use the absolute value of that difference.
- | "Bitwise Or"; using binary number values, whenever a bit is 1 in either A or B, set it to 1 in the result.
- ^ "Bitwise Exclusive Or"; using binary number values, whenever the bits of A and B are not the same, set that bit to 1 in the result, otherwise set the bit to 0.
- > If the value in A is greater than the value in B, set the result to 1 (or *char* 255), otherwise set it to 0.
- < If the value in A is less than the value in B, set the result to 1 (or *char* 255), otherwise set it to 0.
- >p If the value in A is greater than the value in B, use the A value in the result, otherwise set it to 0.
- <p If the value in A is less than the value in B, use the A value in the result, otherwise set it to 0.

If you want to see what other operators are available, check the *Object Reference* documentation for **jit.op**.

- If you'd like, you can drag directly on the **number boxes** above the **jit.scalebias** objects, to set the balance levels independently (i.e. differently from the way our crossfade scheme sets them). You can also try values that exceed the 0 to 1 range.

jit.scalebias vs. jit.op @op *

We chose to use the **jit.scalebias** object in this patch to perform the scaling multiplications instead of using **jit.op** with the ***** operator. Why? When **jit.op** is performing operations on *char* data (as we are doing in this patch), it limits its *val* attribute to the range 0.-1. (when specified as a float) or 0-255 (when specified as an int). In cases where we want to multiply *char* data by some amount from 0. to 1., **jit.op** is just fine. But if we want to multiply *char* data by some other amount, then **jit.scalebias** is the correct object to use because it permits scale factors that exceed the 0 to 1 range. **jit.scalebias** is only for handling 4-plane *char* matrices, but that's OK because that's what we're scaling in this example. So, in this patch, since we're operating on 4-plane *char* matrices, and since we want you to have the ability to try scaling factors that exceed the 0 to 1 range, we have used **jit.scalebias**.

Summary

You can use the **jit.op** object to perform various math operations using the the values from two different matrices. **jit.op** performs the specified math operation on every value individually, pairing each value in the left matrix with the corresponding value in the right matrix. When the dim, plane count, and type attributes of the two matrices differ, **jit.op** uses the attributes of the matrix in the left inlet. Different math operators can create a variety of visual effects when used to combine two video images.

Tutorial 10: Chromakeying

This tutorial explains how to perform chromakeying with two source movies using the **jit.chromakey** object. We will also learn how to find out the color of any pixel on the screen.

- Open the tutorial patch *10jChromakey.pat* in the Jitter Tutorial folder.

When you open the tutorial patch, Max will automatically read two movies (*oh.mov* and *traffic.mov*) into two **jit.qt.movie** objects by sending appropriate read messages to those objects with a **loadbang**:

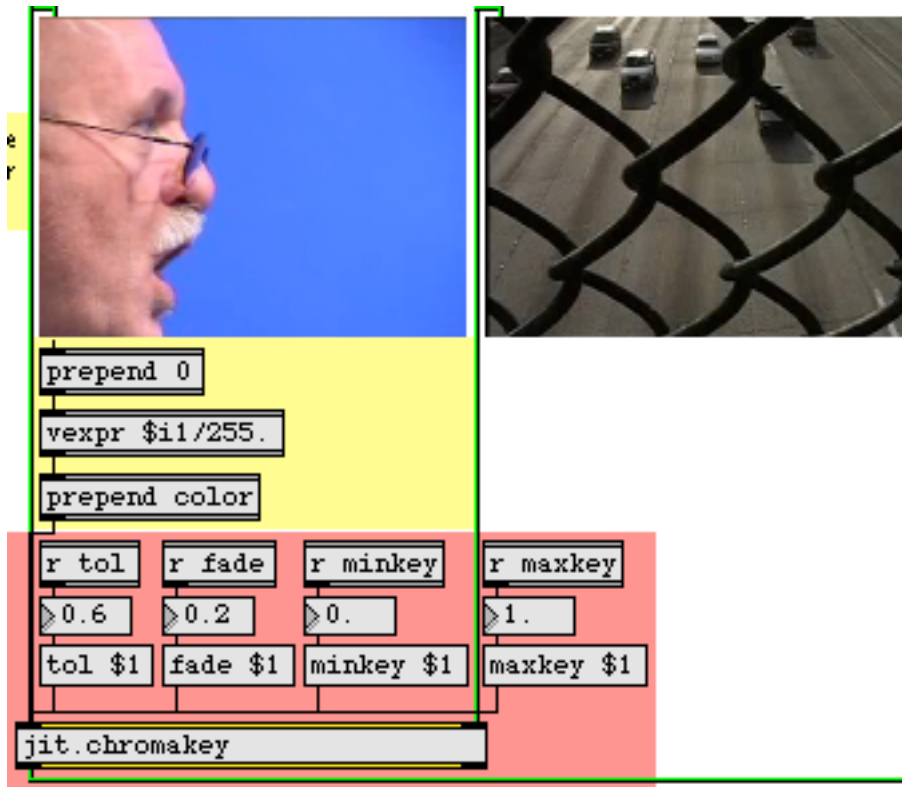


*Initializing the patch via **loadbang**.*

Additional parameters we need for this patch are also initialized by the **loadbang**, which is connected to the **message** box on the right of the patch. The **message** box initializes the rest of the patch by sending messages to named **receive** objects elsewhere in the patch (see **Tutorial 25: Managing Messages**, in the *Max4TutorialsAndTopics.pdf*).

- Click the **toggle** box to start the **metro**. You should see images appear in the three **jit.pwindow** objects in the patch. Note that the **toggle** box not only starts and stops the **metro**, but also starts and stops the movie transport of the two **jit.qt.movie** objects.

The lower half of the tutorial patch (with two of the three **jit.pwindow** objects) looks something like this:



*The **jit.chromakey** object*

- Click with the mouse on the blue region of the left-hand **jit.pwindow** object (i.e. the area behind the man's head in the movie).

The third **jit.pwindow** object (in the lower-right hand of the patch) will look like this:



How the heck did he get in front of that fence?

The `jit.chroma` object

*Chroma*keying—the process of superimposing one image on top of another by selective replacement of color—is accomplished in Jitter by the `jit.chroma` object. By specifying a color and a few other parameters, `jit.chroma` detects cells containing that color in the first (left-hand) matrix and replaces them with the equivalent cells in the second (right-hand) matrix when it constructs the output matrix. The result is that cells from the first matrix are superimposed onto the second.

- Since any color is fair game for the chromakey, try clicking elsewhere in the lefthand `jit.pwindow`. Different colors will be knocked out of the man's face to reveal the traffic.



The disappearing face trick (part one)

Historical note: Bluescreen compositing, or the process of shooting live footage against a blue matte background only to replace the blue with a separate image later, has been around since the late 1930s. Originally a very expensive film process involving expensive lithographic color separation, bluescreen (and its slightly less common sibling, greenscreen) has evolved into the most commonplace (and effective) special effect in film, television, and video. The ability to perform chromakeying (the technical term for the process) using analog (and later digital) video superimposition has only made it more ubiquitous. Video chromakeying is often referred to in the television industry as CSO (Colour Separation Overlay), the name given to the process by the BBC team that developed it in the 1960s. Petro Vlahos, a bluescreen innovator in the 1960s, was awarded a Lifetime Achievement Award by the Academy of Motion Picture Arts and Sciences in 1994, an acknowledgment of how indispensable the technology had become.

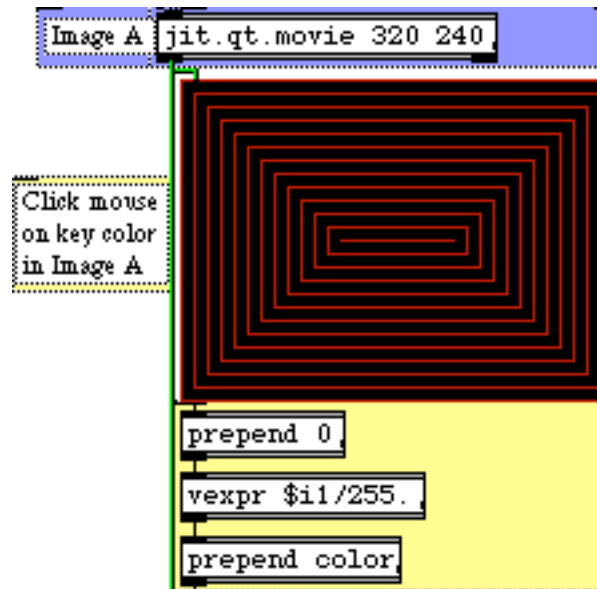
The `jit.chroma` object uses the `color` attribute to define the center color to in the chromakey (called the reference color). This attribute is set as a list of values for as many planes as exist in the matrices that are being keyed. The `tol` attribute specifies a range of

values around the key color. Colors within this range will be keyed as well. When using **jit.chroma**key with *char* matrices (e.g. video), the attributes are specified in a floating point range 0 to 1, which is then mapped to the 0-255 range necessary for *char* data. To set the color attribute for a solid green chromakey, therefore, you would set the attribute as `color 0 0 1.0 0`, **not** `0 0 255 0`. A `tol` range of 0.5 will key all values within half of the chromatic distance from the reference color (computed as the sum of the magnitudes of difference in each plane between the reference color and the actual cell value). A `tol` range of 0 will treat only the exact reference color as part of the chromakey.

- Try clicking on the blue region in the lefthand movie again, and play with the `tol` attribute to see how the chromakey output changes. At low tolerance, some of the bluescreen in the left image will remain in the keyed output. At a very high tolerance, parts of the man's face may disappear.

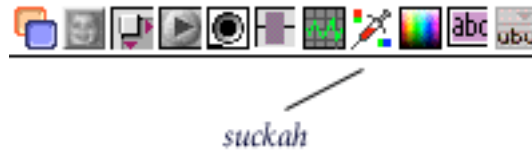
The suckah object —

In the tutorial patch, the color attribute to **jit.chroma**key is set by clicking on an invisible object. If you unlock the patch, you will see a region of concentric red squares that sit on top of the left-hand **jit.pwindow** object:



The suckah object

The region is a Max user interface object called **suckah**, which appears on the object palette like this:



*The **suckah** object in the object palette*

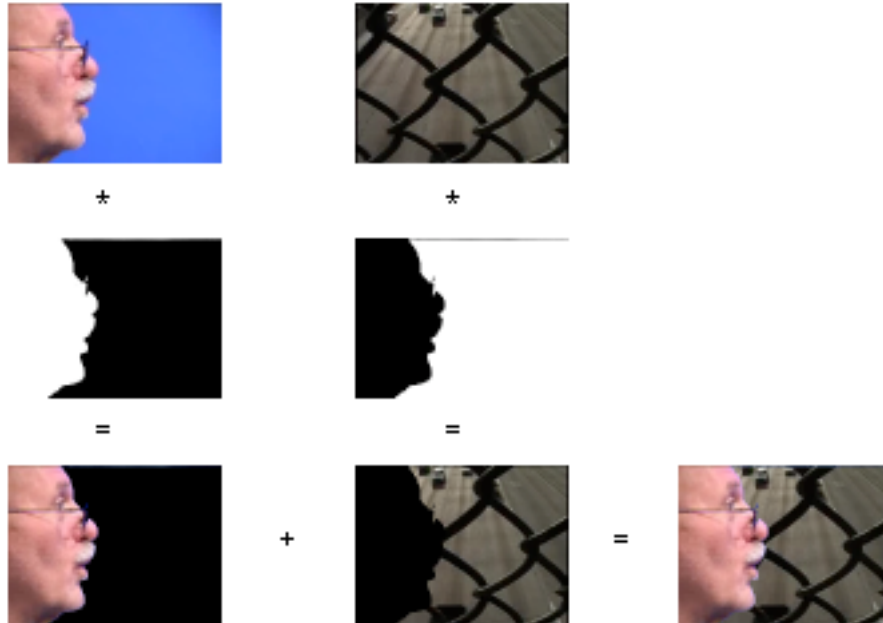
The **suckah** object reports the RGB values of any pixel on the screen that the **suckah** object overlays. It reports these values as a list of integers out its outlet when you click in the object in a locked patch. For example, clicking on a region of solid blue that has a **suckah** on top of it will cause the **suckah** to send out the list 0 0 255.

To set the color attribute for our **jit.chromakey** object, we take the RGB list that comes out of the **suckah** object and send it through a **prepend 0**, which adds an alpha value of 0 to the front of the list. The resulting ARGB list is then divided by 255 using the **vexpr** object to scale it to the range of 0-1. The message is then completed by the **prepend color**. The final message is then sent to **jit.chromakey**.

The Blue Screen of Death

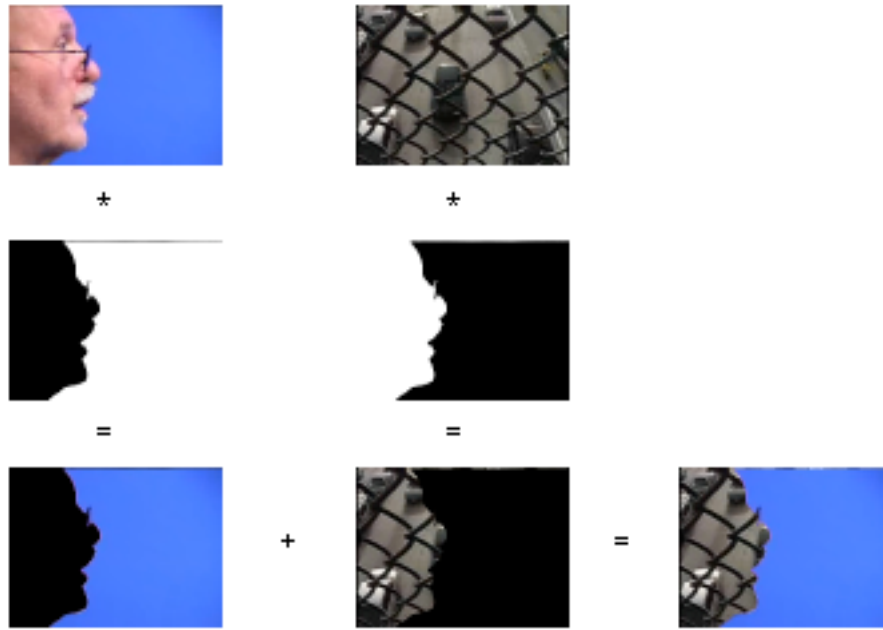
The **jit.chromakey** object has additional attributes: **minkey**, **maxkey**, and **fade**. When a matrix arrives in the left inlet, **jit.chromakey** creates a greyscale (1-plane) mask internally, based on that matrix. Cells in the incoming matrix that have color values within the tolerance (**tol**) range are set to the **maxkey** attribute's value (the default is 1) in the mask. Regions outside the tolerance range are multiplied by the **minkey** attribute (default is 0). If the **minkey** and **maxkey** are set to 0 and 1, the resulting image should look white where the keying should take place, and black where the original image is to be retained.

The resulting mask and its inverse are then multiplied by the right and left matrices, respectively. The results of the multiplication are then added to form the composite image. The following diagram shows you a pictorial overview of the process:



The two sources, their masks (with minkey at 0 and maxkey at 1) and the composite chromakey.

As you can see, the maxkey attribute sets the strength of the righthand matrix in the output, while the minkey attribute sets the strength of the lefthand matrix. If we were to reverse the minkey and maxkey attributes, the chromakey would be reversed, and the following would happen:



The composite effect with the minkey at 1 and the maxkey at 0 (reverse chromakey).

The fade attribute allows for an amount of interpolation between the area being keyed and the area not being keyed. This lets you create a soft edge to the chromakey effect. Colors in the left matrix that are slightly out of bounds of the key tolerance range, yet that are within the range of $\text{tol} + \text{fade}$ from the reference color, are interpolated between their original (unkeyed) color and the color in the same cell of the right matrix. The amount of interpolation is based on how great the fade value is, and how far the color in question lies outside the tolerance range.

- Try experimenting with different tol, fade, minkey, and maxkey values for different colors. Watch how the five attributes interact for different keying effects, and how the minkey and maxkey values complement one another.

Accurate chromakeying can be a challenging process. Correct values for the tol and fade attributes are essential to make sure that the correct regions in the first image are keyed to the second image. In general, very detailed key images will show slight aliasing in spots where the colors rapidly move between keyed and non-keyed regions. In addition, a single key color (e.g. blue) almost never suffices for a complete key, so a range of values must always be used. You will often find, however, that the color you want keyed out of part of the image is somewhat present in the region you want to retain! Balancing all of

these factors to get the most convincing effect is the hardest part of using the **jit.chroma**key object.

Summary

The **jit.chroma**key object lets you do two-source chromakeying in Jitter. You can set a color range for the key using the `color` and `tol` attributes, and use the `fade`, `minkey`, and `maxkey` values to define how the two matrices work in a composite. The **suckah** user interface object allows you to easily select colors as they appear on the screen by setting the object over a **jit.pwindow**. Clicking the **suckah** object will give you the color of the pixel just clicked on the screen.

Tutorial 11: Lists and Matrices

This tutorial shows how to use Max *lists* to fill all or part of a matrix, and how to retrieve all or part of a matrix's contents as a list. We will also demonstrate the use of matrix *names* to access the contents of a matrix remotely, a concept that will be demonstrated further in *Tutorials 12, 16, and 17*.

Matrix Names

- Open the tutorial patch *11jListsAndMatrices.pat* in the Jitter Tutorial folder.

In the yellow region in the upper left corner of the patch, you'll see a blue **jit.matrix** object. The first argument gives the matrix a specific name, `smallbox`. The remaining arguments say that the matrix will have 1 plane of char data, and that the matrix will have only one dimension with 12 cells.

```
jit.matrix smallbox 1 char 12
```

This matrix has a unique name: smallbox.

In *Tutorial 2* we explained that every matrix has a name. If we don't give a matrix a name explicitly, Jitter will choose a name arbitrarily (usually something strange like "u040000114", so that the name will be unique). The name is used to refer to the place in the computer's memory where the matrix's contents are stored. So, why give a name of our own to a matrix? That way we'll know the name, and we can easily tell other objects how to find the matrix's contents. By referring to the name of a matrix, objects can share the same data, and can access the matrix's contents remotely, without actually receiving a `jit_matrix` message.

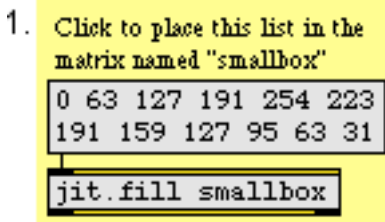
Jitter's use of the matrix name to refer to its memory location is analogous to the way Max's **value** object works. You can have many **value** objects with the same name, and you can store a numeric value in any one of them and retrieve the same value from any other one of them. But there is really only one memory location for that name, so they are all sharing the same data. In a like manner, you can have more than one **jit.matrix** object with the same name, and they will all share the same data. Some other objects, such as **jit.fill**, can access the contents of that matrix just by knowing its name.

jit.fill

In *Tutorial 2* we showed how to place a numeric value in a particular matrix location using the `setcell` message, and how to retrieve the contents of a location with the `getcell` message. Now we will show how to use the **jit.fill** object to place a whole list of values in a

matrix. (Later in this chapter we'll also show how to retrieve many values at once from a matrix.)

In the upper left corner of the patch there is a **message** box containing a list of twelve numeric values. It's attached to a **jit.fill** smallbox object. The smallbox argument refers to a matrix name.



jit.fill puts a list of values in the named matrix

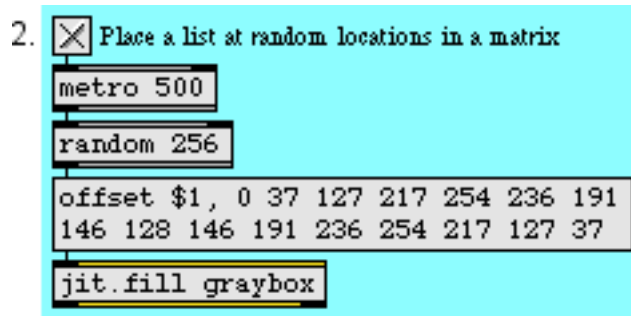
- Click on the **message** box to send the list of values to the **jit.fill** smallbox object. The **jit.fill** smallbox object places those values in the matrix named "smallbox". To verify that this is true, click on the **button** above the **jit.matrix** smallbox object to display the contents of the "smallbox" matrix. The values are printed in the Max window by **jit.print**, and displayed as levels of gray by **jit.pwindow**.

In this example, the list was exactly the right length to fill the entire matrix. That need not be the case, however. We can place a list of any length in any contiguous portion of a 1D or 2D matrix.

The offset attribute

By default, **jit.fill** places the list of values at the very beginning of the matrix. You can direct the list to any location in the matrix, though, by setting **jit.fill**'s offset attribute.

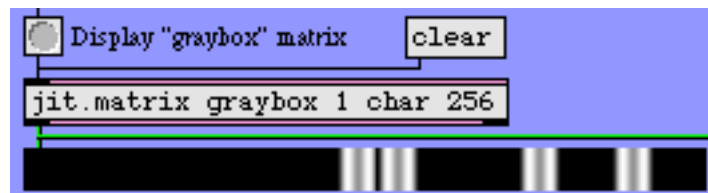
The portion of the patch numbered 2 demonstrates the use of the offset feature.



First specify the offset, then provide the list

This example chooses a cell index at random, uses that random number as the argument to an offset message to the **jit.fill** graybox object, then sends a 16-element list to be stored starting at that index in the graybox matrix.

- Click on the **toggle** to start the **metro**. Every half-second, the 16-element list will be written to a new location in the "graybox" matrix. Find the **jit.matrix** graybox 1 char 256 object in the center of the window. Click on the **button** above it to display its contents in the **jit.pwindow**.



The list has been written to four locations in the "graybox" matrix

- You can use the clear message to zero the contents of the "graybox" matrix, then display it again as the **metro** writes the list into new random locations. When you're done, turn off the **metro**.

Using multiSlider

So far we've shown how to put a predetermined list of values into a matrix. When you want to generate such a list of numbers interactively in Max and place them in a matrix in real time, you'll need to use a Max object designed for building lists. We'll look at two such objects: **multiSlider**, and **zl**.

The **multiSlider** object displays a set of individual sliders, and it sends out the position of all of its sliders at once as a list of values. It sends out the whole list when you click in the window to move any of the sliders, and it sends the list again when you release the mouse button. In part 3 of this patch, we've set up a **multiSlider** to contain 256 sliders that send

values from 0 to 255, so it's just right for sending a list of 256 *char* values to the **jit.fill** graybox object.

- Use the mouse to draw in the **multiSlider**, setting its 256 sliders. When you release the mouse button, the list of 256 values is sent out to the **jit.fill** graybox object. Notice how the brightness of the matrix cells corresponds to the height of the sliders.

As soon as **jit.fill** receives a list in its inlet, it writes the values into the named matrix (at the position specified by the offset attribute). As soon as this is done, **jit.fill** sends a bang out its left outlet. You can use that bang to trigger another action, such as displaying the matrix.

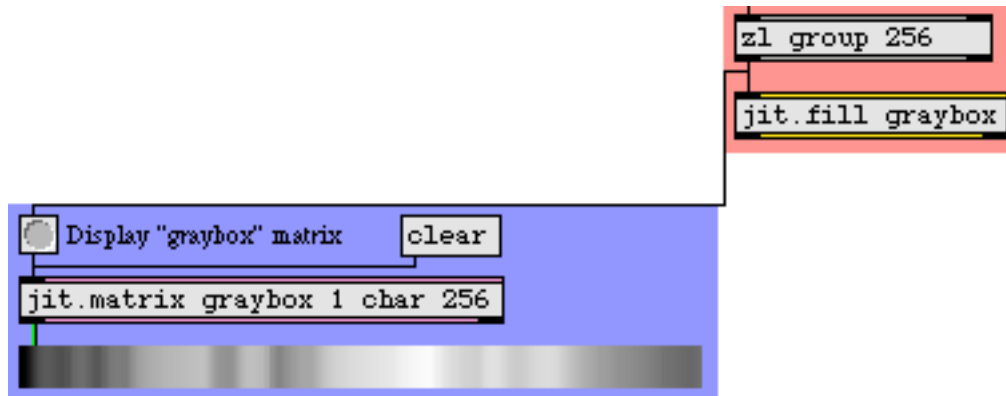
In the first two examples we deliberately avoided using the bang from the left outlet of **jit.fill**, in order to make it quite clear that **jit.fill** writes into the named matrix remotely without being physically connected to the **jit.matrix** object. The bang out of **jit.fill**'s left outlet is convenient, though, for triggering the output of the matrix as soon as it has been filled.

Using zl

In some situations you might want to use a matrix to store numeric messages that have occurred somewhere in the patch: MIDI messages, numbers from a user interface object, etc. The **setcell** and **getcell** messages to **jit.matrix** are useful for that, but another way to do it is to collect the messages into a list and then place them in the matrix all at once with **jit.fill**.

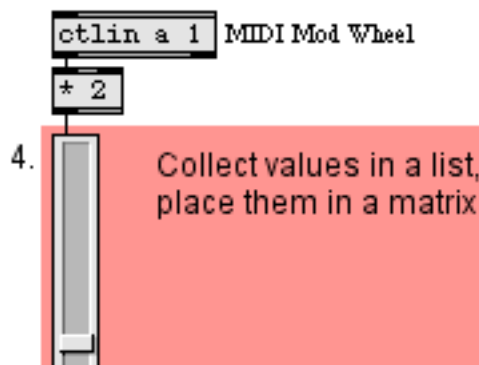
The **zl** object is a versatile list-processing object that can help you collect individual numbers into a list. It has many possible modes of behavior, depending on its first argument. When its first argument is **group**, it collects the messages received in its left inlet until it has amassed a certain number of them, then sends the numbers out as a single list. (The values are grouped in the order in which they were received.) So, in part 4 of the patch, we have placed a **zl group 256** object that will collect 256 values in its left inlet, and when it has received 256 of them it will send them out its left outlet as a list (and clear its own memory).

- Move the **uslider** up and down to generate 256 input values for the **zl** object. When **zl** has received 256 numbers, it sends them as a list to **jit.fill graybox**—which writes them into the "graybox" matrix—then bangs the **jit.matrix graybox 1 char 256** object to display the matrix.



zl sends a 256-element list into the "graybox" matrix, then bangs **jit.matrix** to display the result

- If you have OMS installed and have a MIDI keyboard controller attached to your computer, you can use the modulation wheel of the MIDI keyboard to move the **uslider**. (The interaction between MIDI and Jitter is explored in detail in later tutorial chapters.)



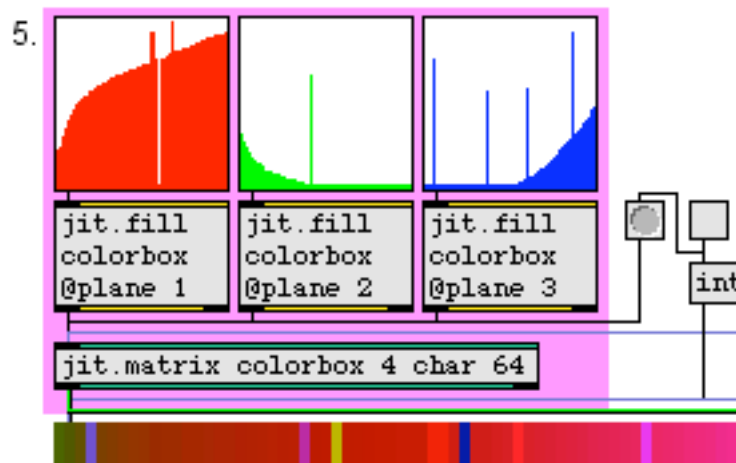
Values are doubled to occupy the range 0-254, making them useful as char data for the matrix

You can change the length of the list that **zl** collects, by sending a new list length in the right inlet from the *List Length* **number box**. And you can say *where* in the matrix you want to put it, by sending an offset message to **jit.fill** from the *Location* **number box**. By varying the list length and location, you can put any number of values into any contiguous region of the matrix.

- Try changing the *List Length* of **zl** (to, say, 100) and setting the *Location* of **jit.fill**'s offset attribute (to, say 50), then move the **uslider** some more to put a list of values into that particular location in the matrix.
- You can combine parts 2, 3, and 4 of the patch to fill the "graybox" matrix in different ways.

jit.fill with Multiple-plane Matrices

jit.fill works fine with multiple-plane matrices, but it can only fill one plane at a time. The plane that **jit.fill** will access is specified in its plane attribute. In part 5 of the patch, we've created another matrix, with four planes of *char* data this time, named **colorbox**. We've set up three **multiSliders** and three **jit.fill** objects, each one addressing a different color plane of the "colorbox" matrix.

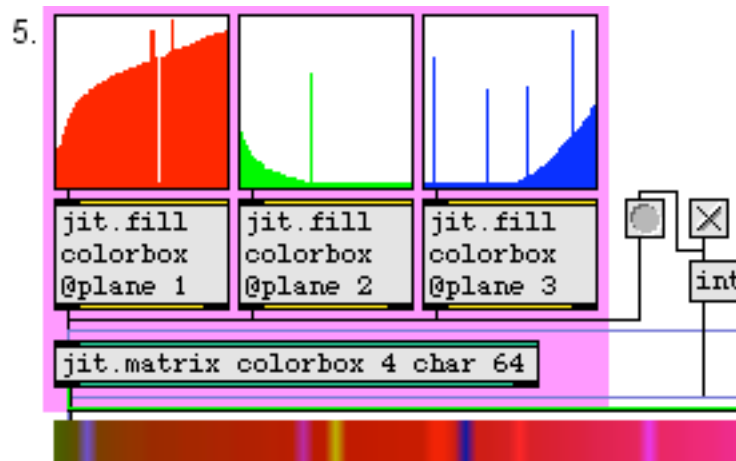


Filling each plane independently

- Drag on the three colored **multiSlider** objects to fill each of the three color planes.

This is a convenient way to generate different curves of intensity in the RGB planes of a matrix. The **jit.pwindow** that's showing the matrix is actually 256 pixels wide, so each of the 64 cells of the matrix is displayed as a 4-pixel-wide band. If you turn on the *interp* attribute of the **jit.pwindow**, the differences between adjacent bands will be smoothed by interpolation.

- Click on the **toggle** above the **interp \$1 message** box to send the message **interp 1** to **jit.pwindow**. (Note that this also sends a bang to **jit.matrix** to re-display its contents).



*The same as the previous example, but with interpolation turned on in **jit.pwindow***

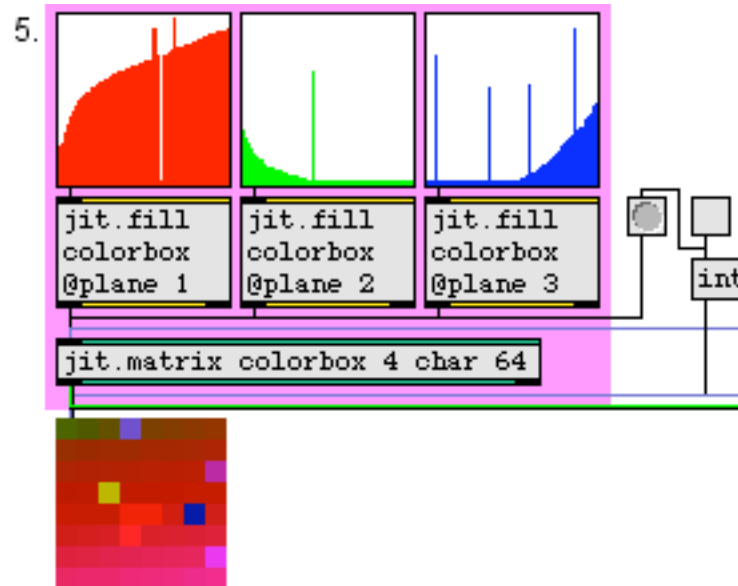
jit.fill with 2D Matrices

So far, all of our examples have involved one-dimensional matrices. What happens when you use a list (which is a one-dimensional array) to fill a two-dimensional matrix via **jit.fill**? The **jit.fill** object will use the list to fill as far as it can in the first dimension (i.e. it will go as far as it can the specified row), then it will wrap around to the next row and continue at the beginning of that row. We've made it possible for you to see this wrapping effect in action.

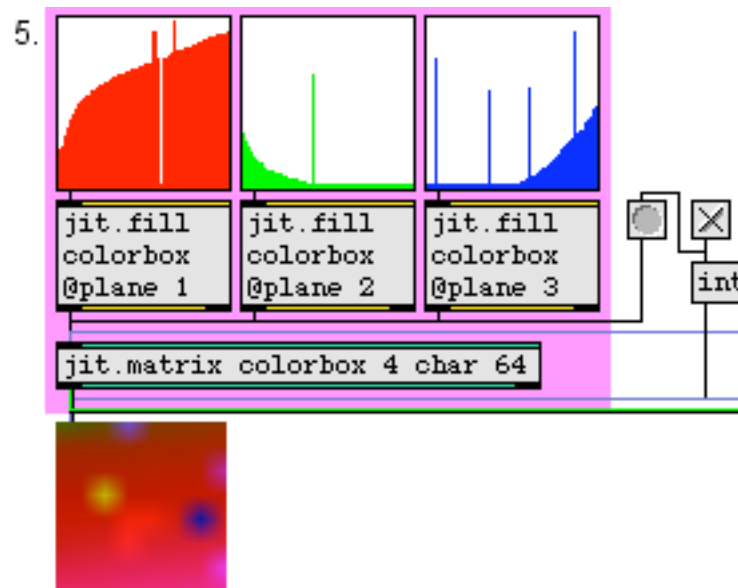
- Click on the button labeled **2D**. This will change the **jit.matrix** colorbox object to contain a two-dimensional 8x8 matrix, and will also resize the **jit.pwindow** to a more appropriate shape. Whenever you change the dimensions of a matrix, it loses its contents, so you will need to click in the three **multiSliders** again to fill the matrix anew. You are still sending a 64-element list to each of the **jit.fill** objects, and they fill each of the eight rows of the matrix with eight elements.

Important: Although we don't demonstrate the use of the **offset** attribute with a 2D matrix in this patch, it's worth mentioning that when the **name** attribute of **jit.fill** names a 2D matrix, the **offset** attribute requires *two* arguments: one for the *x* offset and one for the *y* offset.

***jit.fill** only works for 1D and 2D matrices.*



The same example, with each list wrapped around in an 8x8 matrix (shown uninterpolated)

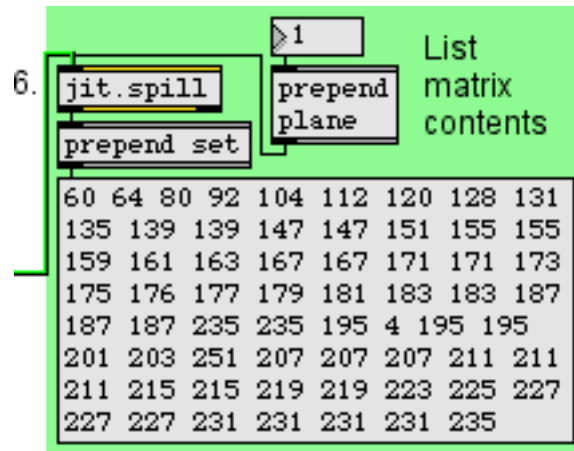


The same example, displayed with interpolation

jit.spill

The complementary object to **jit.fill** is **jit.spill**. It takes a `jit_matrix` message in its inlet, and sends the matrix values out its left outlet as a Max list.

You may have noticed that while you were using part 5 of the patch, the **jit.spill** object in part 6 was sending the values of plane 1 (red) out its left outlet and setting the contents of a **message** box.



The contents of plane 1 of the "colorbox" matrix, displayed as a Max list

Although not demonstrated in the patch, **jit.spill** also has **listlength** and **offset** attributes that let you specify exactly *how many* values you want to list, and from exactly *what location* in the matrix.

If you need to have the values as an immediate series of individual number messages rather than as a single list message, you can send the list to the Max **iter** object.



Get a few values from the matrix and make them into separate messages

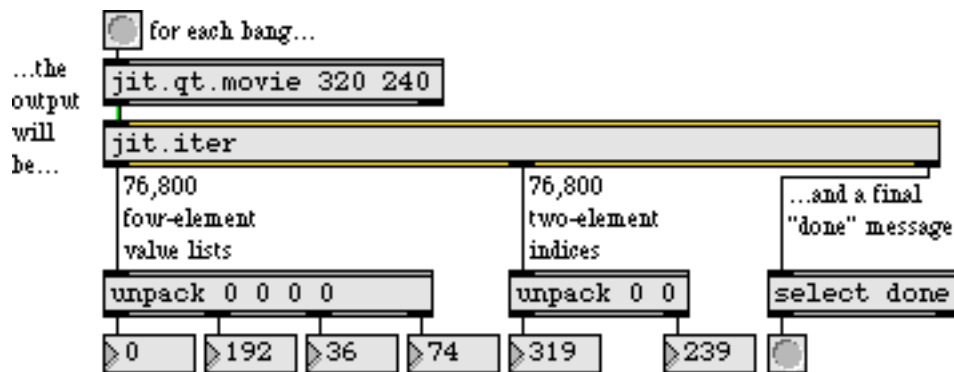
jit.iter

For times when you need to retrieve *every* value in a matrix, there is an object called **jit.iter**. When it receives a **jit_matrix** message in its inlet, it sends out an as-fast-as-possible sequence of messages: the cell index (out its middle outlet) followed by the value(s) in that cell (out its left outlet) for every cell of the matrix in order. For a large matrix, this can be an awful lot of Max messages to try to send out in a single tick of Max's scheduler, so when it's done reporting all of the values in a matrix **jit.iter** sends a **done** message out its right outlet.

In part 7 of the patch there is a **jit.iter** object which receives the matrix information from the **jit.matrix** graybox 1 char 256 object. We use a **swap** object to switch the order of the cell index (coming out the middle outlet of **jit.iter**) and the cell value (coming out the left outlet of **jit.iter**). We then use the value of that cell as the *y*-value we want to store in the **table** object, and we use the cell index as the *x*-axis index for the **table**.

- Click on the **multiSlider** object to send its contents to **jit.fill** (which will in turn bang the **jit.matrix** object and communicate its contents to **jit.iter**). Then double-click the **table** object to open its graphic window and see that it contains the same values as the "graybox" matrix.

Note that this technique of using **jit.iter** to fill a **table** works well with a modest-sized one-dimensional one-plane matrix because a **table** is a one-dimensional array. However, the matrix of a **jit.qt.movie** object, for example, has two dimensions and four planes, so in that case the output of **jit.iter**'s middle (cell index) outlet would be a two-element list, and the output of the left (value) outlet would be a four-element list.



What are ya gonna do with all those numbers?

Still, for one-dimensional matrices, or small 2D matrices, or even for searching for a particular value or pattern in a larger matrix, **jit.iter** is useful for scanning an entire matrix.

Summary

For placing individual values in a matrix, or retrieving individual values from a matrix, you can use the **setcell** and **getcell** messages to **jit.matrix** (as was demonstrated in *Tutorial 2*). For placing a whole list of values in a matrix, or retrieving a list of values from a matrix, use the objects **jit.fill** and **jit.spill**. These objects work well for addressing any plane of a 1D or 2D matrix, and they allow you to address any list length at any starting cell location in the matrix.

The **multiSlider** and **zl** objects are useful for building Max list messages in real time. With **multiSlider** you can "draw" a list by dragging on the sliders with the mouse. With **zl** group

you can collect many individual numeric values into a single list, then send them all to **jit.fill** at one time.

You specify the starting cell location in the matrix by setting the **offset** attribute of **jit.fill** (or **jit.spill**). The **jit.fill** object requires that you set its **name** attribute (either by sending it a **[name]** message or by typing in a **[name]** argument), specifying the name of the matrix it will fill. It accesses the matrix using this name, and sends a **bang** out its outlet whenever it has written a list into the matrix. You can use that **bang** to trigger other actions. In *Tutorials 12, 16, and 17* we show some practical uses of accessing a matrix by its name.

To output every value in an entire matrix, you can send the matrix to **jit.iter**.

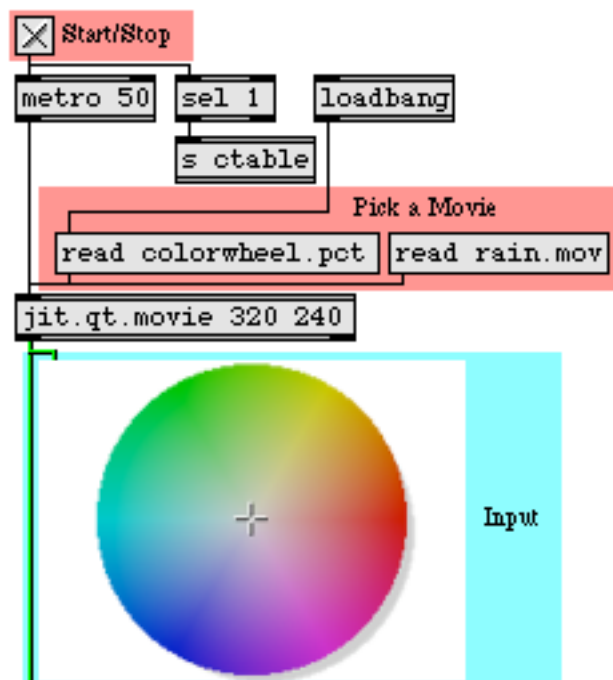
Tutorial 12: Color Lookup Tables

In this tutorial we will explore how to use color lookup tables to remap data inside a Jitter matrix. We'll also look at different strategies for generating lookup tables as matrices.

Lookup tables are simply arrays of numbers where an input number is treated as an address (or position) in the array. The table then outputs the number stored at that address. Any function—a graph where each x value (address) has a corresponding y value (output)—can be used as a lookup table. Max objects such as **funbuff**, **table**, and the MSP **buffer~** object are common candidates for use as lookup tables. In this tutorial, we'll be using Jitter matrices in much the same way.

- Open the tutorial patch *12jColorLookup.pat* in the Jitter Tutorial folder.

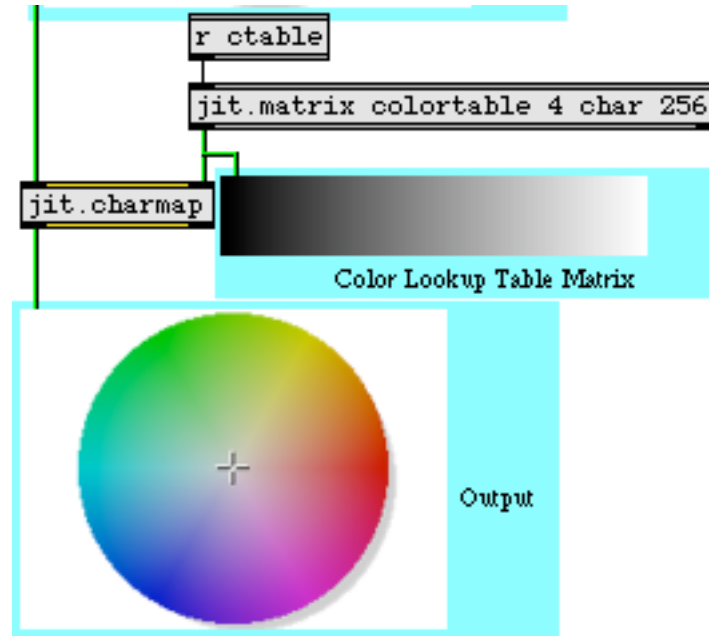
The tutorial patch shows two new objects: **jit.charmap**, which maps input cell values to new output values according to a lookup table matrix, and **jit.gradient**, which generates color gradients.



Read the images

The top left of the patch shows a **jit.qt.movie** object in which you can read two different files. The object is initialized (via **loadbang**) with the file *colorwheel.pct* loaded into it. You can also load in the movie *rain.mov* by clicking on the **message** box that says read rain.mov. You should feel free to alternate between the two image sources throughout the tutorial.

- Start the **metro** by clicking the **toggle** box at the top of the patch. You will see the color wheel appear in both the **jit.pwindow** at the top and the **jit.pwindow** at the bottom of the patch. In addition, you will see a gradient appear in a third (rectangular) **jit.pwindow** at the bottom.



*The output of **jit.charmap** and the lookup table matrix*

The bottom of the patch contains the **jit.charmap** object, which we will use in this tutorial to remap cell values in the image. The object has two inlets, the left one of which is connected from our **jit.qt.movie** object at the top of the patch. The right inlet has a one-dimensional, four-plane *char* **jit.matrix** connected to it with a name (*colortable*) and a width of 256 cells. This is the lookup table that **jit.charmap** uses to remap the color values of the cells in the lefthand matrix. The **receive** object (abbreviated **r**) with the name *ctable* receives data from elsewhere in the patch and sends it to the **jit.matrix**. For example, at the top of the patch, turning on the **toggle** box will send a bang to the **jit.matrix**, causing it to send out its matrix message (*jit_matrix colortable*) to both the **jit.pwindow** and the right inlet of **jit.charmap**.

Lookup Tables

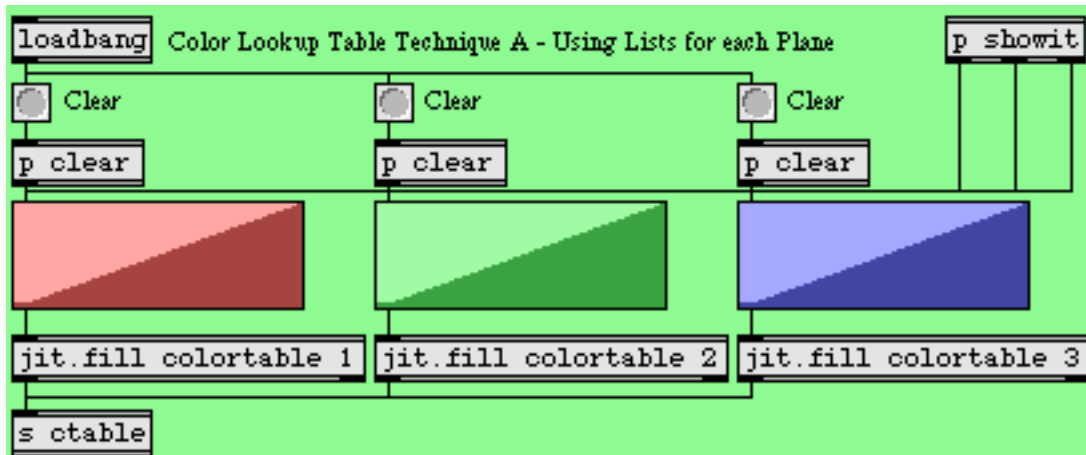
Lookup tables (which are often called transfer functions) are arrays of numbers where an input number is 'looked up' as an index in the table. The number stored at that index (or address, or position) is then retrieved to replace the original number. The **jit.charmap** object replaces every value in every plane in every cell of its (leftmost) input matrix with values stored at the relevant indices in the lookup table that arrives as a matrix in its right inlet.

For example, assume that the matrix we send to **jit.charmap** contains a cell with the values 100 50 35 20 in its four planes. The object looks up each plane individually at the relevant position in its lookup table matrix and replaces it. If our lookup table has the value 73 at cell 100 in the first plane, 25 at cell 50 in the second plane, 0 at cell 35 in the third plane, and 203 at cell 20 in the fourth plane, our output cell will contain the values 73 25 0 203.

Lookup tables for **jit.charmap** should be one-dimensional matrices of 256 cells with the same number of planes as the matrix you want to remap. This is because the possible range of values for char matrices is 0-255, so 256 numbers are needed to cover the full range of the lookup table.

Generating the Lookup Table

The upper-right side of the tutorial patch contains three **multiSlider** objects that let you design the transfer functions for planes 1-3 of the lookup table matrix **colortable**:



*Filling the lookup table matrix with values from a **multiSlider***

The **multiSlider** objects (which have 256 integer sliders in the range 0-255) send their lists to the **jit.fill** objects below them. These objects replace the values currently stored in planes 1-3 (i.e. Red, Green, and Blue) of the matrix **colortable** with the values from the **multiSlider** objects. (See *Tutorial 11*.) When the matrix has been edited with the new values, the **jit.fill** objects send out a bang, which we **send** to the **jit.matrix** on the right of the patch that connects to the right inlet of **jit.charmap**. We're ignoring plane 0 in this tutorial because it only contains Alpha values when we treat 4-plane Jitter matrices as video.

The **jit.matrix** and **jit.fill** objects in our patch share the same name (**colortable**). As a result, the two objects read from and write to the same matrix, allowing one object (**jit.fill**) to generate data that the other object (**jit.matrix**) can read from without having to copy data between two separate matrices. This is similar to how many MSP objects (e.g. **peek~**,

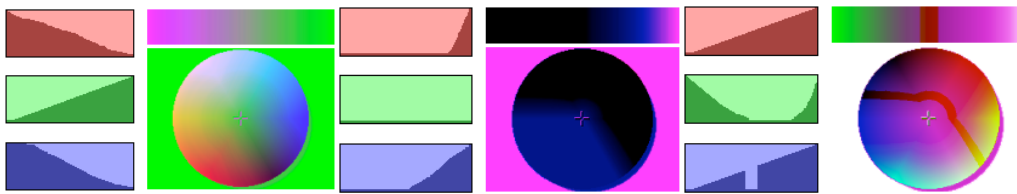
play~, **groove~**) can share sample data stored in a single **buffer~**. See Tutorials 11, 16, and 17 for more information on using named matrices.

- Do some freehand drawing in the **multiSlider** objects, to see how this affects both the lookup table (the smaller of the **jit.pwindow** objects) and the output image from the **jit.qt.movie** object. Remember to switch back and forth between the two image sources.

If you want to reset any of the planes to a $y=x$ transfer function (i.e. a straight ascending line that leaves all the values unchanged), you can click the **button** object above the relevant **multiSlider** object. The subpatchers called **p clear** initialize the **multiSlider** objects with an **Uzi**.

Important note: Like many Max objects, Jitter objects retain matrices stored in one inlet even if a new matrix arrives in another inlet. The **metro** object in this patch, therefore, only needs to trigger the **jit.qt.movie** object. The **jit.matrix** that contains the lookup table to **jit.charmap** only needs to output its value to the object when the data stored in it actually changes.

Here are some lookup tables and their results:



*Three sets of **multiSlider** objects and their resulting color lookup tables and output color wheels*

In the first example, the red and blue transfer functions are approximately inverted while the green is normal. The result is that high values of red and blue in the input image yield low values on the output, and vice versa. This is why the white background of the color wheel now looks green (the cell values of 0 255 255 255 have been mapped to 0 0 255 0).

The second example has the green plane completely zeroed (the transfer function set to 0 across the entire span of input values). The red and blue planes are also set to 0 up to a threshold, at which point they ramp up suddenly (the red more suddenly than the blue). As a result the majority of the colorwheel is black (especially in the 'green' area). The red plane only becomes visible in very high values (i.e. the magenta in the background of the color wheel).

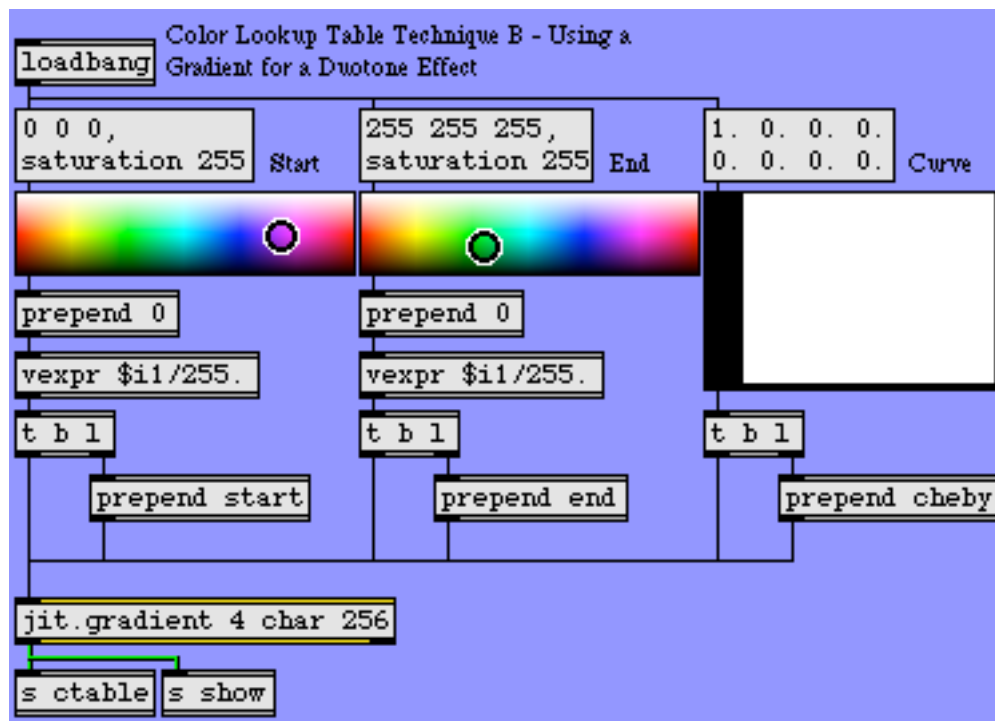
The third example has the red plane mapped normally. The green plane has a parabolic shape to it, where the extreme values are mapped high and the medium shades are mapped low. The blue plane is normal except for a range in the midtones, where it is

zeroed. This nonlinearity is visible as a red 'fault line' across the top and down the right side of the colorwheel.

As you can see, there are infinite ways to remap the cell values of these matrices. We'll now investigate another object, which lets us remap the color values in a more precise manner.

The jit.gradient object

The lower right area of the tutorial patch shows a method for generating lookup tables using the **jit.gradient** object:



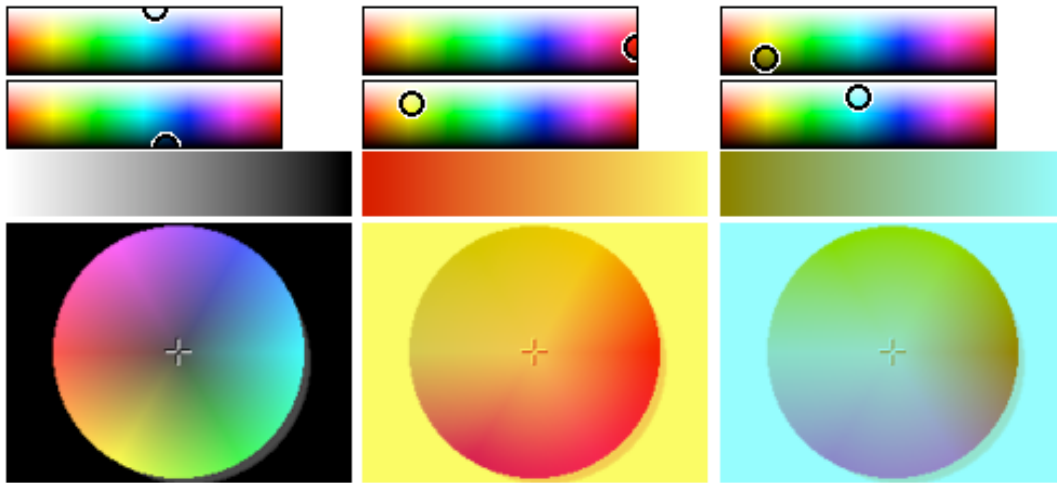
*Using the **jit.gradient** object*

The **jit.gradient** object generates single dimension char matrices that transition smoothly between two specified cell values. The start and end attributes are lists that specify these cell values. For example, a start attribute of 0000 and an end attribute of 00.5 1.0 0.5 will generate a gradient that goes from black (at cell 0 in the matrix) to pale green (at the last cell in the matrix). We've given our **jit.gradient** object the relevant arguments to make it 256 cells wide, so that it can be stored in our colortable **jit.matrix** when it changes. Note that **jit.gradient** takes floating point numbers in its attribute lists to specify char values (i.e. a value of 1.0 in the attribute specifies a char value of 255).

The attributes are formatted by taking the RGB list output of the Max **swatch** objects and converting them to ARGB floats. After the attribute has been sent to the **jit.gradient** object,

the object is given a bang from the **trigger** object to cause it to output its matrix into the **jit.matrix** object on the left of the patch.

- Try selecting some colors in the **swatch** objects. The start and end attributes will specify the boundaries of the lookup table, so values in the input image will have a duotone appearance, morphing between those two colors. The **multislider** objects at the top of the patch will reflect the correct lookup tables generated by the **jit.gradient** object.



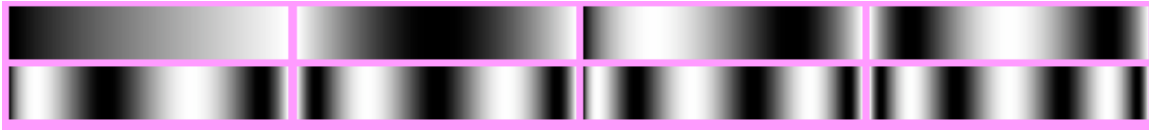
Using color gradients as lookup tables

The first example shows an inverted image. The start of the lookup table is white (start 0 1.0 1.0 1.0) and the end of the lookup table is black (end 0.0. 0.0.). As a result, input values of 0 are mapped to 255, and vice versa ($y=255-x$).

The second and third examples show duotone gradients that remap the color wheel's spectrum to between red and orange (example 2) and olive and cyan (example 3). Notice how, depending on the original colors at different points in the color wheel, the gradient curve becomes steeper or more gradual.

The third attribute of the **jit.gradient** object is the **cheby** attribute, which specifies a curve to follow when morphing between the start and end values in the matrix. The **cheby** attribute takes a list of floating point numbers as arguments. These arguments are the amplitudes of different orders of *Chebyshev polynomials* (see below). These special function curves create different effects when used in lookup tables.

The **multiSlider** in the tutorial patch that sets the cheby attribute lets you specify the relative amplitude of the first eight Chebyshev polynomial curves, which have the following shapes (if you view them as progressing from black to white):

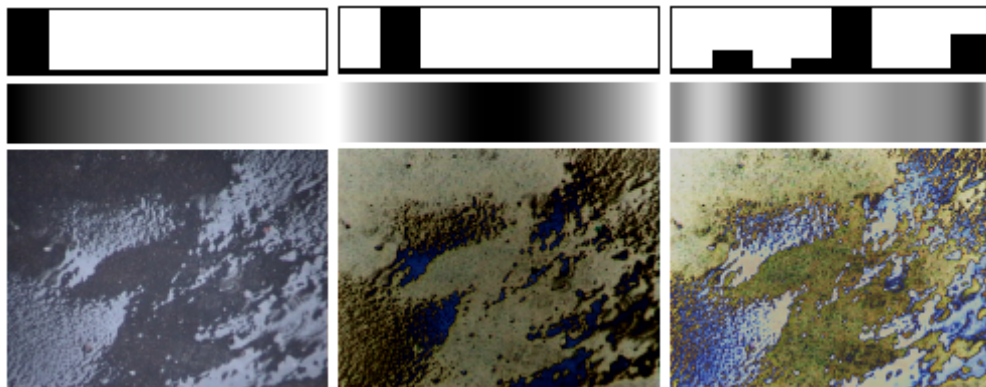


Gradients generated using Chebyshev orders 1-4 (top row) and 5-8 (bottom row)

Technical note: Chebyshev polynomials are commonly used as transfer functions in waveshaping audio signals in digital synthesis algorithms (they have special properties that allow them to distort sinusoidal waveforms into harmonic spectra equivalent to the amplitudes of different orders). The MSP **lookup~** object can be used with a function loaded into a **buffer~** to do the equivalent process in audio signal processing that we're doing in this tutorial with image. See *Tutorial 12: Synthesis: Waveshaping* in the MSP manual for more details.

- Reset the start and end points of the gradient (by clicking the **message** boxes above them) and slowly change the **multiSlider** that controls the cheby attribute. Watch how the color wheel changes as colors disappear and reappear in different regions.

When you use the cheby attribute in the **jit.gradient** object, you can get some very interesting color warping effects even if you leave the start and end points of the gradient at black and white. Here are some examples with our movie clip *rain.mov*:



The effect of different gradient curves on the color spectrum of the rain

The lefthand image shows an unprocessed still image from the rain movie. The middle image shows what happens to the color spectrum when the gradient is generated using a second order Chebyshev polynomial (the darkest area in the image is now in the middle of the color spectrum). The righthand image shows a more complex gradient, where the color spectrum shows numerous peaks and troughs.

- The **multislider** objects at the top of the patch reflect the current state of our lookup table (the matrix generated by our **jit.gradient** object is sent to a **jit.iter** object inside of the **p showit** subpatch, where the numbers are grouped to set the state of the **multislider** objects). Try generating a gradient and then modifying the lookup table by hand by changing the **multislider** objects. This lets you use the **jit.gradient** object as a starting point for a more complicated lookup table.

Summary

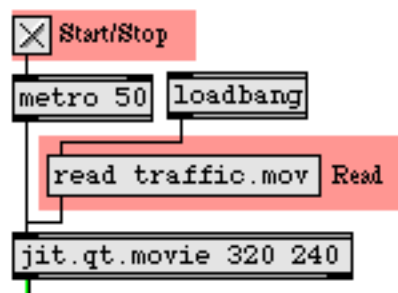
You can map cell values in *char* Jitter matrices using the **jit.charmap** object. The right inlet of **jit.charmap** takes a 256-cell matrix that defines the lookup table (or transfer function) to be applied to the incoming matrix data. You can define the lookup table using several strategies, including using **jit.fill** to generate the matrix from Max lists, or using the **jit.gradient** object to generate color gradients between a start and end cell value according to a curve shape specified by the *cheby* attribute.

Tutorial 13: Scissors and Glue

In this tutorial we'll learn how to use two simple objects to slice and combine rectangular regions of two-dimensional Jitter matrices.

- Open the tutorial patch *13jScissorsAndGlue.pat* in the Jitter Tutorial folder.

The tutorial patch shows two Jitter objects that neatly complement each other: **jit.scissors**, which cuts a matrix into equally sized smaller matrices, and **jit.glue**, which pastes multiple matrices into one matrix. We'll also take a brief look at a Max object called **router**, which lets you easily route Max messages from multiple sources to multiple destinations.



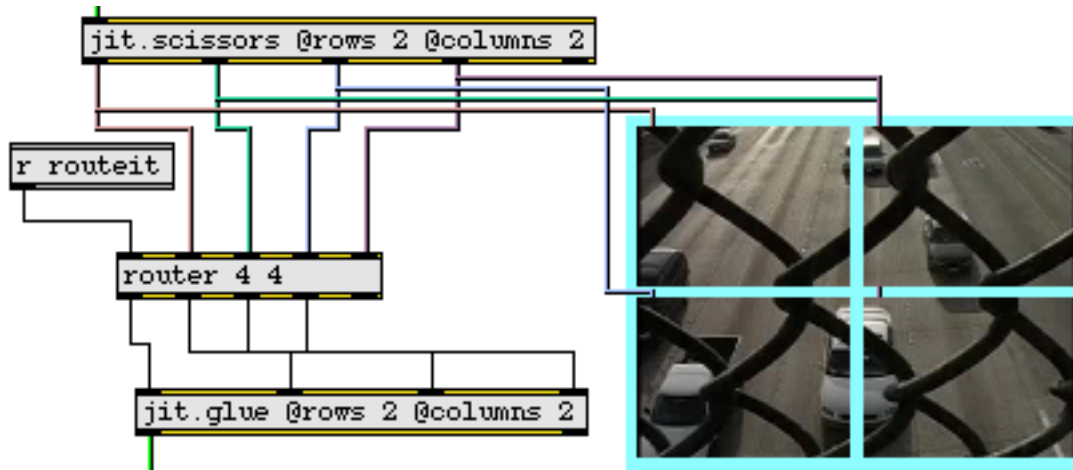
Read the movie

The top left of the patch is straightforward enough. The **loadbang** object automatically sends the `read traffic.mov` message to the **jit.qt.movie** object, which then loads our movie of traffic footage.

- Start the **metro** by clicking the **toggle** box at the top of the patch. You will see the traffic appear in the large **jit.pwindow** at the bottom of the patch. More interestingly, you will see the traffic image cut into quadrants, each of which appears in a separate **jit.pwindow** object off to the right side.

Cut it Up

The **jit.scissors** object is responsible for splitting the Jitter matrix containing the traffic footage into four smaller matrices:



*The **jit.scissors** object*

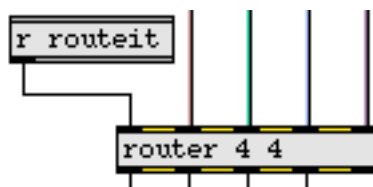
The **jit.scissors** object cuts a Jitter matrix of any size, type, or plane count into smaller Jitter matrices that are then sent out independent outlets of the object. The rows and columns attributes specify how many smaller matrices are created each time the object receives a new matrix in its inlet. In our tutorial patch, the **jit.scissors** object is splitting the image into four smaller matrices (2 columns and 2 rows). These separate matrices come out individual outlets of the object in *column-major* order (i.e. the object assigns outlets to the smaller matrices from left-to-right and then from top-to-bottom).

Two very important things you should know about `jit.scissors`:

- 1) The number of outlets that **`jit.scissors`** has is determined at object creation. Therefore the `rows` and `columns` attributes will only create outlets when they are specified in the object box. For example, typing **`jit.scissors @rows 10 @columns 2`** will create an instance of **`jit.scissors`** with 20 matrix outlets (plus the usual right outlet for attribute queries), but simply making a **`jit.scissors`** object with no arguments will only give you one matrix outlet. You can change the `rows` and `columns` attributes with Max messages to the object, but you won't be able to add outlets beyond what those initially created by the object.
- 2) The size (dim) of the matrices put out by **`jit.scissors`** is equal to the size of the slices of the matrix, not the entire original matrix. For example, the four smaller matrices in our tutorial patch are each 160x120 cells, not 320x240.

Routing the Matrices

The four smaller matrices output by **`jit.scissors`** in our patch are each sent to two different places: to **`jit.pwindow`** objects so we can see what's going on, and to a Max object in the middle of the patch called **`router`**. The colored patchcords illustrate where each smaller matrix is sent.

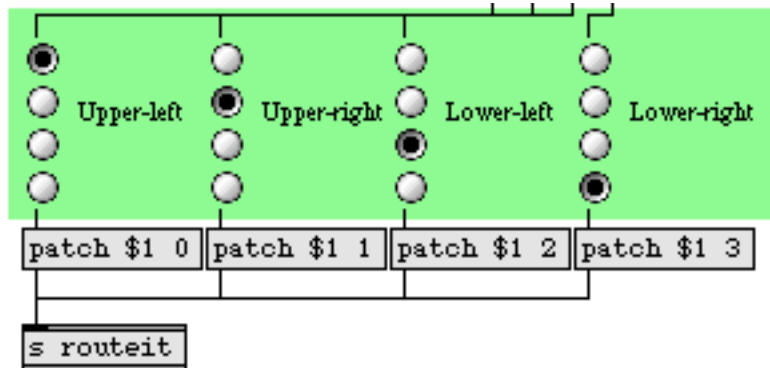


*The Max **`router`** object*

The **`router`** object is a combination of the Max **`gate`** and **`switch`** objects. It takes two arguments (the number of routeable inlets and the number of routeable outlets) and is controlled by messages sent to the leftmost inlet. Most of the messages that **`router`** understands are identical to the MSP object **`matrix~`**. As a result you can use **`router`** with the **`matrixctrl`** object with ease.

The four inlets to the right of the **`router`** object take their input from the four matrix outlets of our **`jit.scissors`** object. A **`receive`** object assigned to the symbol `routeit` gets messages from the lower-right of the tutorial patch, which controls our **`router`** object. The four

leftmost outlets of the router object are connected to a **jit.glue** object, which we'll talk about in a moment.



*Controlling the **router***

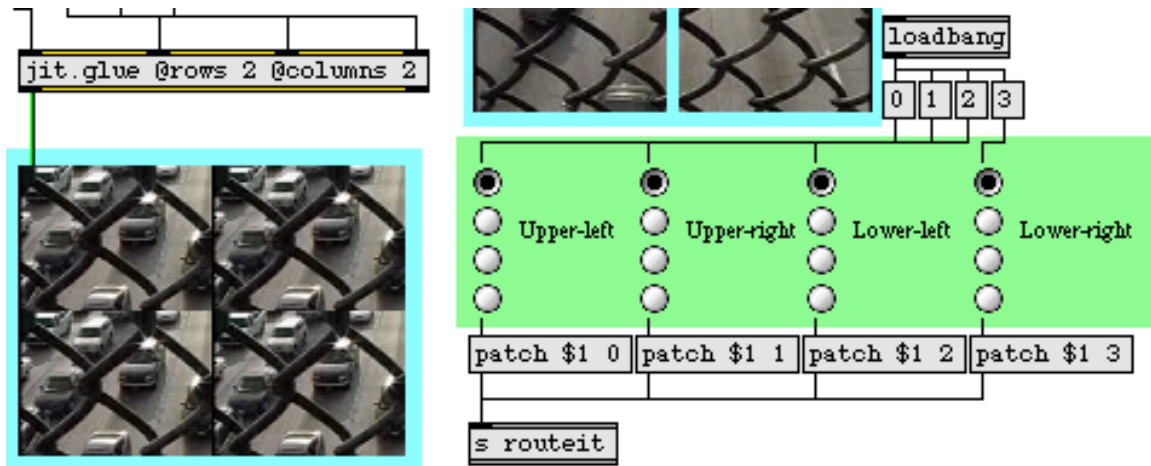
Sending the message patch followed by an inlet number and an outlet number to a **router** object will make a virtual connection between that inlet and that outlet in the object. Any message arriving at that inlet will be instantly forwarded to the relevant outlet. If an inlet was previously connected to that outlet, a patch message will sever that connection in favor of the new one.

The **radiogroup** objects in this patch control which outlets of the **router** our four small Jitter matrices (arriving at the inlets) are sent to. The inlets and outlets number up from 0, so the message patch 2 1 makes a connection between the third routeable inlet and the second outlet of the **router** object.

- Click on some of the **radiogroup** controls, and watch how the output image in the lower jit.pwindow changes. Notice how with the **router** object you can make the matrices cut from the traffic image appear in any of the four quadrants of the composite image at the bottom.

The Glue That Keeps It Together

The **jit.glue** object at the bottom of the patch does the effective opposite of **jit.scissors**. The rows and columns attributes specify inlets, not outlets, and a composite matrix is output which is made up of the incoming matrices laid out in a grid.



*Sending the same matrix to all four inlets of **jit.glue***

Important Note: As with **jit.scissors**, **jit.glue** can only create new inlets and outlets when the object is created, so the rows and columns attributes present in the object box will determine how many inlets the object has. Also, the size (dim) of the output matrix generated by **jit.glue** will be equal to the size of all the smaller matrices put together (e.g. our four 160x120 matrices in this patch will yield one 320x240 matrix).

One final point worth making about **jit.glue** is that its default behavior is to only output a composite matrix when a new matrix arrives at its *leftmost* inlet. If we were to disconnect the leftmost inlet of our **jit.glue** object, we would no longer get any new output matrices from the object. The syncinlet attribute lets you make **jit.glue** send its output in response to a different inlet. A syncinlet value of -1 will cause **jit.glue** to output new composite matrices when it gets new matrices at *any* inlet. While this sounds like a good idea in theory, it can quickly bog down the frame rate of your Jitter processes with lots of redundant work.

Summary

The **jit.scissors** object cuts a matrix into smaller, equal-sized rectangular matrices. The **jit.glue** object takes equal-sized rectangular matrices and pastes them back together into a composite matrix. The rows and columns attributes of both objects determine their number of outlets or inlets, respectively, when given at object creation, as well as the way in which the matrix is sliced up or composited. The **router** object lets you arbitrarily connect Max

messages from multiple inlets to multiple outlets in a similar fashion to the MSP **matrix~** object.

Tutorial 14: Matrix Positioning

Positioning Data in a Matrix

In this tutorial we discuss some ways to take a portion of one matrix and place it in some different location in another matrix. There are various reasons you might want to reposition the location of data. We'll be focusing especially on visual effects, but the techniques we show here are useful for any sort of task that involves moving matrix data around.

We'll show how to isolate a region of a matrix, place it at a particular position in another matrix, resize it (which can be useful for visual effects such as stretching, *pixelation*, and blurring), and move it around dynamically.

- Open the tutorial patch *14jMatrixPositioning.pat* in the Jitter Tutorial folder.

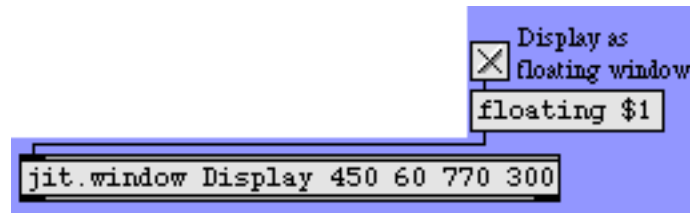
jit.window

In the bottom-left corner of the patch there is a **jit.window** object. We introduced this object in *Tutorial 1*; it creates a separate window for displaying the contents of a matrix. In most of the other tutorial chapters we have used the **jit.pwindow** object instead.

jit.window and **jit.pwindow** are pretty similar—aside from the obvious difference that one opens a separate window while the other uses a rectangular region within the Patcher window—and they share many of the same attributes and messages. There are a few differences, though, so we'll use **jit.window** this time in order to demonstrate a couple of its unique characteristics.

You probably can't see the *Display* window that has been opened by the **jit.window** object, because it's hidden behind the Patcher window. However, if we want to, we can make the *Display* window be a floating window—one that always "floats" on top of every other window in Max while still letting us interact with the foreground Patcher window. To do this, we must turn on the floating attribute of **jit.window** with a floating 1 message. (The floating attribute is 0 by default.)

- Click on the **toggle** box labeled *Display as floating window* to send a floating 1 message to **jit.window**.



Make the window "float" in front of all other windows

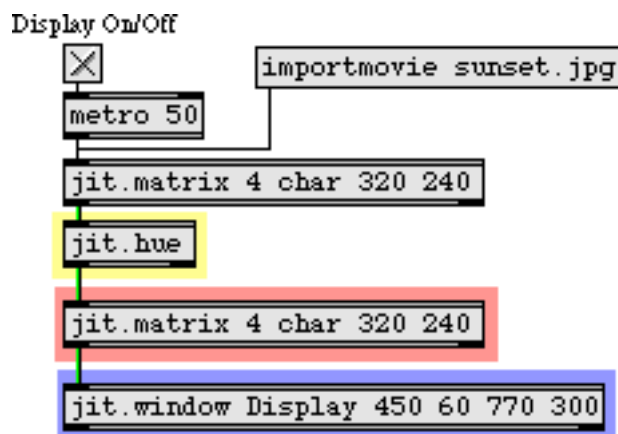
Note that the screen coordinates we've typed into the **jit.window** object for the display area—450 60 770 300—specify a display area 320 pixels wide by 240 pixels high. (For an explanation of how to specify screen coordinates for **jit.window**, see *Tutorial 1* and/or the Note later in this chapter.)

From one jit.matrix to another

Now we will load in a picture and try some modifications.

- Click on the **message** box `importmovie sunset.jpg` to load a picture into the **jit.matrix** object at the top of the patch. Turn on the **metro** labeled *Display On/Off* to begin sending bang messages to **jit.matrix**.

The bang sends the matrix (through **jit.hue**) to a second **jit.matrix** object before displaying the image with **jit.window**. In that second **jit.matrix** object we'll be able to modify attributes to change what part of the matrix we display.

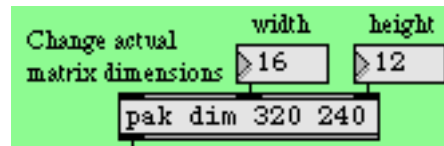


The matrix goes (through jit.hue) to another jit.matrix, then to jit.window

We've saved several preset configurations for the window's user interface objects in a **preset** object in the middle of the patch.

- In the **preset** object, click on preset 1.

This changes the dimensions of the lower **jit.matrix** object to 16x12, by sending a dim 16 12 message to it.



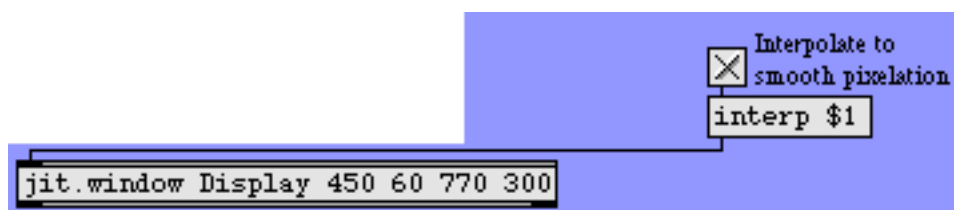
*The dim message changes the dimensions of the matrix in **jit.matrix***

The matrix coming in has the dimensions 320x240, but the receiving **jit.matrix** has dimensions of only 16x12, so it tries its best to display the entire matrix it receives, but it necessarily has to discard much of the information. This results in a very pixelated image. (The term *pixelation* refers to the mosaic effect that results from using an insufficient viewing resolution—an insufficient number of pixels—to represent an image faithfully.) Even though the **jit.window** object is capable of displaying the full-resolution 320x240 image (because of the window dimensions typed in as arguments), the matrix it is receiving is only 16x12 now. It "expands" the 16x12 matrix to 320x240 for display purposes, duplicating pixels as it needs to.

- Try dragging on the two **number boxes** labeled *Change actual matrix dimensions* to see different pixelation effects.

Interpolation

- Now set the **number boxes** back to 16 and 12, and click on the **toggle** labeled *Interpolate to smooth pixelation* in the blue region at the bottom of the patch to send an **interp 1** message to **jit.window**.

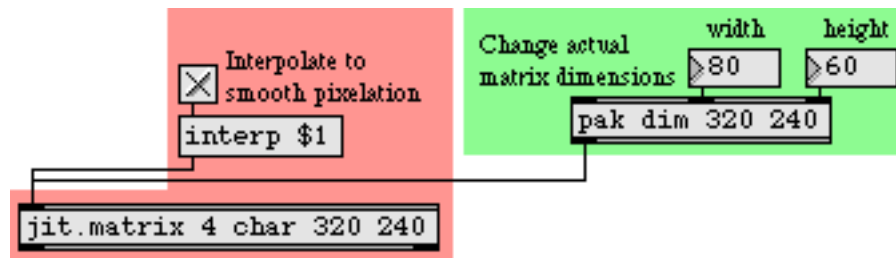


*Turn on interpolation in **jit.window***

Now the **jit.window** object—instead of simply duplicating pixels of the 16x12 matrix to make a bunch of 20x20-pixel blocks—interpolates between values in the incoming matrix as it expands it to 320x240. That is, as it expands the image, it creates a smooth gradation of colors between each cell value and its neighboring values in the incoming matrix, so all of the transitions from cell to cell in the displayed 320x240 matrix are as gradual as

possible. The interpolation causes extreme blurring because the size difference between the incoming matrix and the display is so great.

- Click on the **toggle** again to turn off the interpolation. This sends an `interp 0` message to **jit.window**, setting its `interp` attribute to 0 (off). Enter some new matrix dimensions into the **number boxes** labeled *Change actual matrix dimensions* so that the image is not quite so pixelated: say, 80 and 60. (You should see that now the pixelated blocks are each only 4x4.) Click on the **toggle** to turn interpolation back on. Notice that in this case the blurring is not so extreme because interpolation occurs over only 4 pixels. Click on the **toggle** again to turn off the interpolation.
- Now find the other **toggle** with the same label, just above the **jit.matrix** object, and click on it to turn on interpolation within **jit.matrix** (not **jit.window**).



Interpolation doesn't do much when you're reducing the size of the matrix

Notice that this doesn't have very much effect. That's because **jit.matrix** still only has a 80x60 matrix to send out. Interpolation in this case (when we're reducing the size of the matrix rather than enlarging it) is pretty ineffectual.

- Click on that **toggle** again to turn off interpolation in **jit.matrix**.

Isolate a Part of the Matrix

Now we'll look at ways to focus on a particular portion of a matrix.

- In the **preset** object, click on preset 2. Now you see only a small portion of the picture.

This preset restores the dimensions of our **jit.matrix** object back to 320x240. But we can still isolate a particular part of the matrix, without altering the actual dimensions of the full matrix, using some different attributes: `srcdimstart`, `srcdimend`, and `usesrcdim`. Notice that we have sent three new messages to **jit.matrix** to set those three attributes: `dimstart 40 150`, `dimend 119 209`, and `usesrcdim 1`. These messages let us specify a subset of the full matrix coming in the inlet, and send those values out as a full-sized (in this case 320x240) matrix. This smaller subset of the incoming matrix gets "expanded" (cells are duplicated as needed) within **jit.matrix** itself, to fill the size of the outgoing matrix. The `usesrcdim 1` message says, "Use the input matrix subset that I've specified, instead of the full input

matrix." (By default the `usesrcdim` attribute is set to 0, so the `srcdimstart` and `srcdimend` attributes are ignored.) In the messages for setting the `srcdimstart` and `srcdimend` attributes, the words `srcdimstart` and `srcdimend` are followed by cell indices describing the starting and ending points within each dimension. With our `dimstart 40 150` and `dimend 119 209` messages, we have told **jit.matrix** to use a specific 80x60 region from cell 40 to 119 (inclusive) in the horizontal dimension and from cell 150 to 209 in the vertical dimension.

Note: In this chapter we've discussed three different ways of specifying rectangular regions! It's important to be clear what we're specifying in each case.

In **jit.window** we typed in *screen coordinates* for the display area of the window. In the computer's operating system, screen coordinates are specified in terms of the point at the *upper-left corner of a pixel*. The upper-left corner of the entire screen is 0,0; the point two pixels to the right of that (the upper-left corner of the third pixel from the left) is 2,0; and the point 5 pixels down from that (the upper left corner of the sixth pixel down) is 2,5. To describe a rectangular area of the screen, we type in arguments for the left, top, right, and bottom limits of the rectangle's coordinates.

In the `dim` attribute to **jit.matrix**, we provided *dimension sizes* for the object's matrix: the *number of cells* in each dimension.

In the `srcdimstart` and `srcdimend` attributes, we're stating (inclusive) *cell indices* within the matrix. Remember that cells are given index numbers that go from 0 to one less than the number of cells in that dimension. (Planes are indexed similarly, by the way.) So for a 320x240 matrix, the indices for the cells in the first dimension go from 0 to 319, and the indices for the cells in the second dimension go from 0 to 239. To set the source dimensions for **jit.matrix**, we need to specify the range of cells we want to start at, using `srcdimstart` followed by a starting cell index for each of the matrix's dimensions, and using `srcdimend` followed by the cell indices for the end of the range in each dimension.

These different ways of describing regions can be confusing, but if you think carefully about exactly *what* it is that you're specifying, you'll be able to deduce the proper way to describe what you want.

We're using only an 80x60 pixel range of the incoming matrix as the source, but the destination matrix is 320x240. Once again, this expansion of a smaller matrix into a larger one causes a pixelation effect. This time, though, the expansion occurs inside **jit.matrix** (i.e. between its "source" region and its "destination" size), rather than between **jit.matrix** and

jit.window (as we did earlier when we reduced the actual dimensions of the **jit.matrix**). Therefore, if we want to smooth out the pixelation by interpolating, we must do it in **jit.matrix**. There's no point in turning on interpolation in **jit.window**, since it's already receiving a 320x240 matrix from **jit.matrix**.

- If you want to verify this, turn on the **toggle** to send an interp 1 message into **jit.window**. It has absolutely no effect because we're trying to interpolate a 320x240 matrix into a 320x240 display area, so no change occurs. Turn off that same **toggle** to set the interp attribute of **jit.window** back to 0. Now use the other **toggle** to send an interp 1 message into **jit.matrix**. This time we get the smoothing effect we desire.
- Try entering new values into the **number boxes** to change the arguments of the **srcdimstart** and **srcdimend** attributes. This lets you isolate any particular region of the picture as your "source" area. Of course, the dimensions you choose for your source area will determine the distortion that the picture undergoes when it's expanded to fill a 320x240 output matrix.

Flip the Image

You might assume that the arguments of the **srcdimend** attribute (the ending cell indices of the source region) should be greater than the index numbers for the **srcdimstart** attribute. But that need not necessarily be so.

- In the **preset** object, click on preset 3. Now the picture is has been flipped vertically.



The top and bottom have been flipped in the second dimension.

This example shows that if you specify an ending cell index in the vertical dimension that is less than the starting index, **jit.matrix** will still relate those indices to the starting and ending points in the vertical dimension of the destination matrix, effectively reversing the up-down orientation of the values. (This statement assumes that you have not done the same sort of flip to the orientation of the destination matrix!)

You could do the same sort of flip in the horizontal (first) dimension to flip the image horizontally. If you flip the source region in *both* dimensions you get the same visual effect as if you had rotated the image 180°.

- In the **preset** object, click on preset 4.

In this example we've flipped the source region in both dimensions, reduced the size of the source area to 160x120, and smoothed out the pixelation by turning on the `interp` attribute.

Resize the Output Matrix

Just as we specified the "source" region of the matrix, we can also specify a destination for that source. This still does not change the size of the output matrix; that will still be 320x240, as determined by the `dim` attribute. However, this does change the region into which the specified "source" region will be placed. The source region of the input matrix will be placed in the destination region of the output matrix (with expansion/contraction as necessary). Cells of the output matrix that lie outside the destination region will remain unchanged.

- In the **preset** object, click on preset 5. The entire input matrix has been squeezed into an 80x60 rectangle in the center of the output matrix.

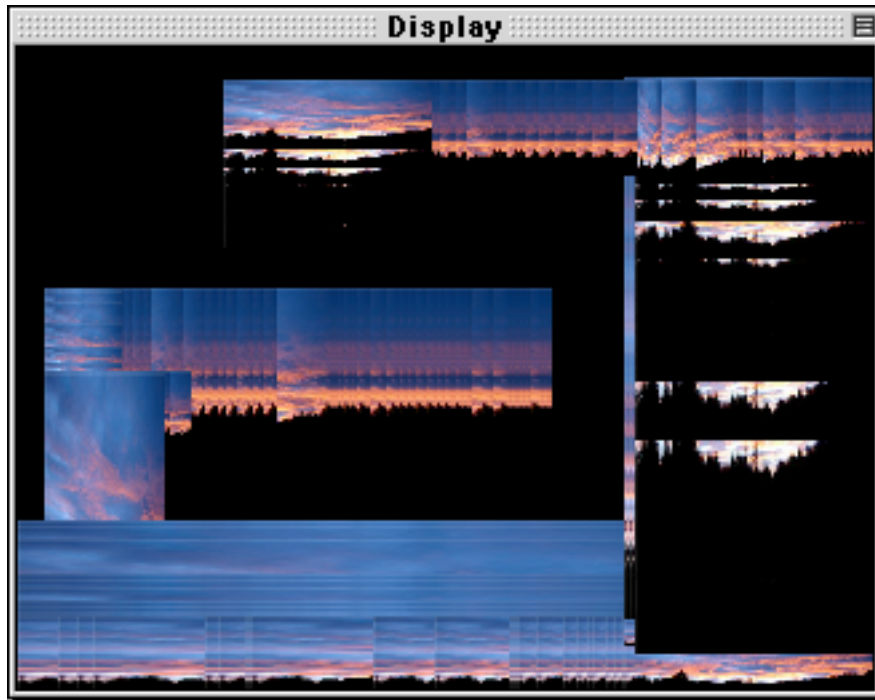
The first thing to notice is that the `usesrcdim` attribute has been turned off, so that we're back to using the entire input matrix as the source. (The `srcdimstart` and `srcdimend` attributes are now being ignored.) The `usedstdim` attribute has been turned on, so the input will be placed in whatever part of the output matrix we specify. The `dstdimstart` and `dstdimend` attributes have been set to specify the cells in the center of the matrix as the destination: `dstdimstart 120 90` and `dstdimend 199 149`. We've turned the `interp` attribute off because we're contracting the image rather than expanding it.

Notice also that we've turned on the **toggle** labeled *Erase previous image*. This sends the number 1 into the `if $2 then clear` object. The *if* part of the statement is now true, so every time the object receives a message in its left inlet it will send out the message `clear`. This clears the contents of the **jit.matrix** object immediately after displaying the image, to prepare **jit.matrix** for the next matrix it will receive. That ensures that the values in all the cells outside the destination region will be 0, so the unused region of the output matrix will be displayed as black.

Change some of the values in the **number boxes** that provide the destination dimensions, to move (and resize) the picture within the Display window.

Now turn off the **toggle** labeled *Erase previous image*, to suppress the `clear` messages. Change the arguments of `dstdimstart` and `dstdimend` some more, and notice what's different this time. The previous destination regions are still being drawn in the Display window because those cells in the matrix have not been cleared, and they are left unchanged if they're outside the new destination region.

This gives the effect of leaving "trails" of the previous image behind. We can potentially use these artifacts for their particular visual effect.



If the matrix is not cleared, old destination areas will be left behind if they're outside the new destination region. For continuous changes, this leaves a "trail" of past images.

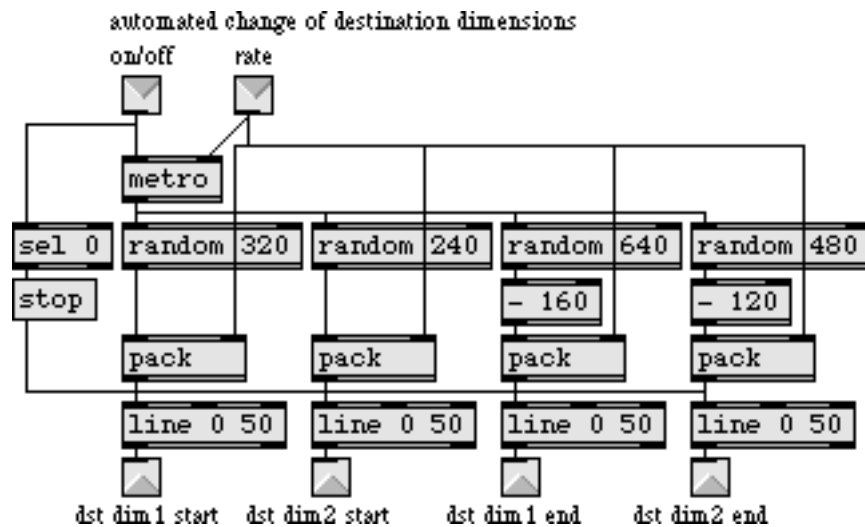
Moving the Image Data Around in the Matrix

By setting up an automated Max process that modifies the `dstdimstart` and `dstdimend` attributes, we can move the data around in the matrix, making the image seem to move around in the display.

- In the **preset** object, click on preset 6.

This starts an automated process inside the **patcher** `move_around` subpatch that provides a continuous stream of new arguments for the `dstdimstart` and `dstdimend` attributes. The **toggle** above the **patcher** turns on this process, and the **number box** gives a time, in milliseconds, for each move to a new destination.

- Double-click on the **patcher** move_around object to see the contents of the subpatch. So far, we're only using the right half of the subpatch.



The destination-moving process in the subpatch [move_around]

The "rate" value coming in the right inlet is a time interval for the **metro** object. The **metro** periodically bangs four **random** objects which choose new left, top, right, and bottom cell indices at random. These destination points are sent, along with the time value, to **line** objects. The **line** objects send out new values every 50 ms (the rate at which we're displaying the image) to gradually move the destination region to these new random points. Outside the subpatch, those values are used as arguments for the **dstdimstart** and **dstdimend** attributes of **jit.matrix**.

This subpatch contains a couple of tricks worth noting. The first trick is that we've made it so the arguments for **dstdimend** can potentially exceed the 320x240 range of the matrix. For example, we use a **random 640** object for the horizontal dimension, then subtract 160 from the result to give us an ending cell index from -160 to 479. We do this to increase the likelihood of a larger destination area so that we can see a larger view of the image as it moves around, and it also means that the image will more frequently move all the way to the edge of the window. It's noteworthy that we can specify destination boundaries that are beyond the limits of the actual cells in the matrix, and **jit.matrix** will place the image in that area to the best of its ability (clipping it off when it exceeds the limits of the matrix dimensions). The second trick is a trivial but useful detail: we use a **sel 0** object to detect when the **metro** gets turned off, and we use that to trigger a **stop** message to each of the **line** objects so that they don't continue sending out values after the user has turned off the process.

- Close the [move_around] subpatch window.

Changing, Resizing, and Moving the Source Image

Now we'll automate changes to the source image, as well.

- In the **preset** object, click on preset 7.

In much the same manner as we did for the destination area, we're now continually changing the source area of the image. In effect, we're now seeing a constantly changing view of some rectangular subset of the source matrix (using `srcdimstart` and `srcdimend`) while also constantly resizing that view and moving it around in the window (using `dstdimstart` and `dstdimend`). Because the source and destination rectangles are chosen randomly by the *[move_around]* subpatch, the image sometimes gets flipped, too. We have turned on the `interp` attribute in the **jit.matrix** object to smooth out the pixelation that would occur when the source image gets stretched.

- To get a slightly clearer view of what's going on, try turning on the **toggle** marked *Erase previous image*.

One More Word About Dimensions

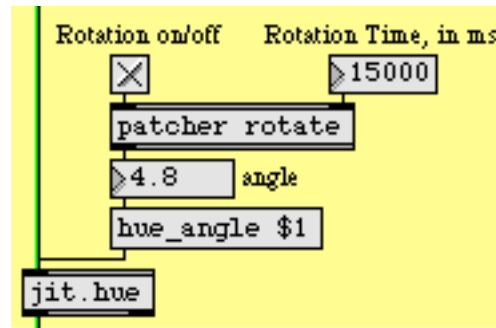
This tutorial has shown how to change the dimensions of a **jit.matrix** object, and how to specify source and destination regions within that object. For ease of discussion and visualization, we've used a two-dimensional matrix, and specified source and destination rectangles within the matrix. But we should point out that these ideas can also be employed with matrices that have any number of dimensions. (The number of arguments for `srcdimstart`, `srcdimend`, `dstdimstart`, and `dstdimend` should correspond to the number of dimensions in the **jit.matrix** object.) For example, if we have a three-dimensional matrix, these arguments can be used to specify a hexahedron in the virtual 3D space of the matrix.

Note: In certain Jitter objects that deal exclusively with 2D matrices, such as **jit.qt.movie**, source and destination regions will always be rectangular areas. So in those objects the source and destination areas are defined in single attributes called `srcrect` and `dstrect`, which take four arguments to specify the bounding (left-top and right-bottom corner) cells of the rectangles.

Hue Rotation

Just to add a little additional color variety, we've placed a **jit.hue** object between the two **jit.matrix** objects. (**jit.hue** is described in detail in *Tutorial 7*.)

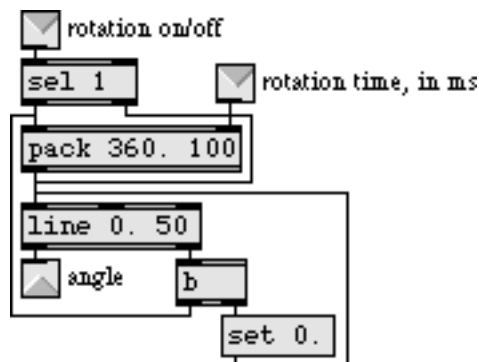
- In the **preset** object, click on preset 8 to see **jit.hue** in action.



Modify the hue angle

This preset turns off `usedstdim`, but keeps `usersrdim` on, and keeps interpolation on in **jit.matrix** to blur the expanded image. The automated process in the **patcher rotate** subpatch continually rotates the hue angle of **jit.hue**.

- Double-click on the **patcher rotate** object to see the contents of the subpatch.



The contents of the [rotate] subpatch

The value coming in the right inlet provides a time, in milliseconds, to complete a 360° hue rotation. When a 1 comes in the left inlet, the number 360 is combined with that time value to instruct the **line** object to go from 0 to 360 in that amount of time, sending out a new angle value once every 50 milliseconds. Note that the first typed-in argument of the **line** object contains a decimal point. This instructs **line** to send out float values rather than ints, for greater precision (and because the `hue_angle` message of **jit.hue** expects a float argument). When **line** reaches 360, its right outlet sends out a bang. We use that bang to set the internal value of **line** back to 0, then we re-bang the **pack** object to start the next rotation. When a 0 comes in the left inlet, the **sel 1** object passes it directly to **line** to stop **line** and reset the hue angle to 0.

- Close the *[rotate]* subpatch window.

- In the **preset** object, click on preset 9. This combines virtually all of the automation and image -manipulation techniques of the patch. The changes of destination dimensions of **jit.matrix** are set to 200 ms this time, creating a more rapid rhythmic effect.

Full Screen Display

When you've got your Max patch creating just the images that you want, and you want to display your results in a somewhat more elegant way, you can instruct **jit.window** to fill your entire screen. **jit.window** has an attribute called **fullscreen**; when **fullscreen** is on, the **jit.window** uses the entire screen as its display area. You can also use the **hidemenubar** message to Max to hide the menu bar (see "Messages to Max" in the *Topics* section of the Max documentation for details).

There are a few things to remember about using the fullscreen capability of **jit.window**.

First of all, once you have filled your screen with an image (and especially if you have also hidden the menubar), you will no longer be able to use your mouse to turn fullscreen off. So you will need to program into your Max patch some method of returning the fullscreen attribute to 0.

Secondly, only one **jit.window** can fill any one screen at any one time. If you have more than one **jit.window** object vying for access to the full screen, the **jit.window** object that has most recently had its fullscreen attribute set to 1 will fill the screen.

Also, even when a **jit.window** is fullscreen, its resolution is determined by its actual dimensions (that is, by the arguments of its **rect** attribute). So if the **rect** attribute describes a 320x240 rectangle, that will be the resolution of your image, even though your screen dimensions are much greater than that.

At the bottom of our patch, we've included the capability to turn the fullscreen attribute of **jit.window** on and off (and hide/show the menu bar) with the space bar of your keyboard.



Using the space bar to switch the window to full screen display

- Try toggling fullscreen on and off with the space bar.

- For a more abstract visual effect, try importing the *colorswatch.pict* image into the **jit.matrix** at the top of the patch, then try the different presets.

In this tutorial, we've used a still image as our source material so that you could easily see the effects being demonstrated, but there's no reason that you couldn't use a video (from **jit.qt.movie** or some other video source) as your basic material. (You might want to copy the contents of this patch to a new Patcher window and modify the top-left part of it to try that out.)

Summary

There are several ways to isolate and reposition certain data in a matrix. The **dim** attribute of **jit.matrix** sets the actual dimensions and size of the matrix. By turning on the **usesrcdim** and **usedstdim** attributes of **jit.matrix**, you can instruct it to use a particular portion of its input and output matrices, which are referred to as the "source" and "destination" regions of the matrix. You specify the cell boundaries of those regions with the **srcdimstart** and **srcdimend** attributes (to set starting and ending cells as the boundaries of the source region) and the **dstdimstart** and **dstdimend** attributes (for the destination region). These attributes do not change the actual size of the matrix, but they specify what part of the input matrix will be passed out in what part of the output matrix (when **usesrcdim** and **usedstdim** are on). If the source and destination regions are different in shape or size, **jit.matrix** will either expand or contract the source region to fit it in the destination region. This results in either duplication or loss of data, but can provide interesting stretching or *pixelation* effects. The source and destination regions can be altered dynamically with numbers provided by some other part of your Max patch, for interactive or automated modification of the size, shape, and position of the image.

When the **interp** attribute is on, **jit.matrix** interpolates (provides intermediate values) between values when a dimension of the destination region is greater than that of the source region. This smooths out pixelation effects, and blurs the changes between values in adjacent cells.

The **jit.window** object displays whatever size matrix it receives, using whatever display rectangle has been specified for it in its **rect** attribute. If the size of the incoming matrix differs from the size of the display area, the image will be expanded, or contracted, or distorted by **jit.window**. This, too, can be used for stretching and pixelation effects. **jit.window** also has an **interp** attribute which, when turned on, smooths out the pixelation caused by this expansion and stretching.

To fill the entire screen with an image, you can turn on **jit.window**'s **fullscreen** attribute, and you can hide the menu bar with a `;max hidemenubar` message. (Just remember to leave yourself some way to get your Patcher window back in the foreground.)

We've demonstrated the techniques of resizing, repositioning, flipping, and interpolating matrix data to create visual effects such as stretching, distorting, blurring, and pixelation.

Tutorial 15: Image Rotation

Rotating and Zooming with **jit.rota**

Jitter provides an easy way to rotate an image, and/or zoom it in or out, with an object called **jit.rota**. Rotation and zoom are common and useful video effects, and by combining them in different ways in **jit.rota** you can also achieve a variety of kaleidoscopic effects. **jit.rota** takes a matrix of video data (or any other sort of image) in its inlet, and sends out a version that has been zoomed, rotated, and otherwise distorted based on the settings of the object's attributes.

Basic Rotation

- Open the tutorial patch *15jImageRotation.pat* in the Jitter Tutorial folder. The QuickTime video *dishes.mov* is read into the **jit.qt.movie** object automatically by a bang from **loadbang**. To see the video, click on the *Display toggle* to start the **metro**.

The video is a three-second left-to-right camera pan over a set of dishes. However, the *loop* attribute of the **jit.qt.movie** object has been initialized to 2, so the movie loops back and forth, giving the illusion of a back-and-forth pan.

Note: Many attributes of Jitter objects use only the arguments 1 and 0 to mean "on" and "off", so it's reasonable to assume that the *loop* attribute of **jit.qt.movie** is the same. While it's true that *loop 0* turns looping off and *loop 1* turns it on, *loop 2* causes the video to play forward and then play backward when it reaches the *loopend* point, rather than leaping to the *loopstart* point.

The *theta* attribute of **jit.rota** determines the angle of rotation around a central *anchor point*.

- Drag on the *Rotation Angle number box* to rotate the video. Positive (or increasing) values cause counter-clockwise rotation, and negative (or decreasing) values cause clockwise rotation. The angle of rotation—a.k.a. the angle θ (*theta*)—is stated in *radians*. A value of 0—or any multiple of 2π (i.e., 6.283185)—is the normal upright positioning. A value of π (i.e. 3.141593)—or any odd multiple of π —is the fully upside-down position. Experiment until you understand the relationship between the *theta* values and the behavior of **jit.rota**.

Technical Detail: `jit.rota` does a lot of internal calculation using trigonometry to determine how to rotate the image. If you're not a trigonometry buff, you might not be used to thinking of angles in terms of radians. In everyday conversation we more commonly use *degrees*, with a full rotation being 360° . In trigonometry, it's more common to use *radians*, where a full rotation equals 2π radians. That's because a circle with a radius of 1 has a circumference of exactly 2π , so you can refer to an angle by referencing the point where it would intersect the unit circle. (For example if you started at a point on the unit circle and traveled a distance of exactly $\pi/2$ around the circumference, you would end up at a 90° angle—i.e. an angle of $\pi/2$ radians—from where you started, in reference to the circle's center.)

Also, in trigonometry we consider a positive change in angle to be a counter-clockwise rotation around the unit circle, whereas in everyday life you might more commonly think of a clockwise motion as being intuitively "positive" or "increasing" (like the passage of time).

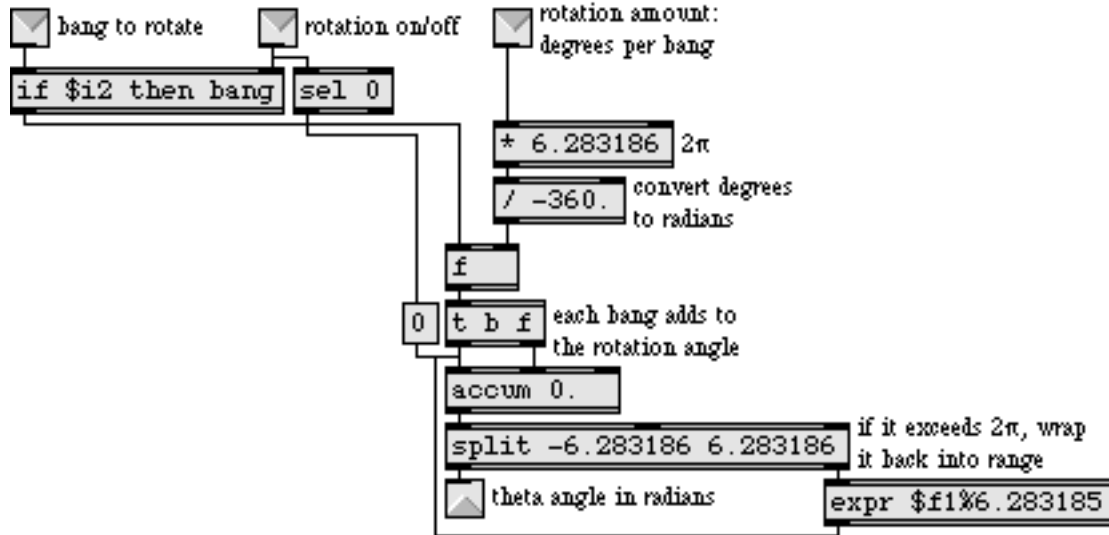
So, to convert a clockwise rotation in degrees into the same rotation in radians, you would need to multiply the degree angle by 2π , then divide by -360 .

Automated Rotation

Besides rotating the image by hand, you can also write an automated process in Max that will supply continually changing rotation angles. In the previous chapter we wrote a subpatch called *rotate* that used the **line** object to increase the angle of hue rotation continually from 0° to 360° . In this chapter we do something similar, but this time we use the **bang** from the **metro** that's displaying the movie to increase the angle of rotation. To keep it "user-friendly" we show the user *degrees* of angle rotation rather than radians (we convert degrees to radians inside the subpatch), and we also display the rotation speed as "rotations per second."

- In the **number box** labeled *Degrees per bang*, enter the number 6. This will cause the rotation angle to increase by 6 degrees with each **bang** from the **metro**. Since the **metro** sends out a **bang** 20 times per second (once every 50 ms), we know that we can calculate the number of rotations per second by the formula $d*20/360$ —that is, $d/18$ —where d is the degrees of angle increase per **bang**. Now click on the **toggle** marked *On/Off* to begin the automated rotation.

- Double click on the **patcher** rotate object to see the contents of the subpatch.



Automated rotation in the [rotate] subpatch

We convert what the user specifies as "degrees per bang" into an amount in radians, by multiplying the degrees by 2π and dividing by -360 . (See the **Technical Detail** sidebar above.) When a **bang** comes in the left inlet, if rotation is turned on then the bang gets passed through and it causes an increase of angle rotation to be added into the **accum** object. Note that a negative "degrees per bang" amount works fine, too, and causes a counter-clockwise rotation of the image. When the total rotation angle exceeds 2π (or -2π), we use a modulo operation to bring it back into range (resetting the value in the **accum** object) before sending it to the **outlet**. When rotation gets turned off, we detect that fact with a **sel 0** object, and reset the theta angle to 0.

- Close the subpatch window. Click on the **On/Off toggle** to stop the automated rotation.

Zoom In or Out

The other main feature of **jit.rota** is its zooming capability. The amount of zoom is determined by **jit.rota**'s **zoom_x** and **zoom_y** attributes. These permit you to zoom in or out in the horizontal and vertical dimensions independently; or you can zoom both dimensions simultaneously by changing both attributes at once.

- Drag on the **number box** labeled **Zoom** to zoom in and out. Values greater than 1 expand the image (zoom in), and values less than 1 shrink the image (zoom out). You can change the zoom of the *x* and *y* dimensions independently by entering values

directly into those **number boxes**. (Negative zoom values flip the image as well as resize it.)

When we zoom in on the image—say, with a zoom value of 2—we still retain reasonably good image quality because we've turned **jit.rota**'s `interp` attribute on with an `interp 1` message. If you turn `interp` off, you will get pixelation when you zoom in. When you're zooming out, `interp` has no appreciable effect, so it's pretty much a waste of the computer's time. (See *Tutorial 14* for a discussion of pixelation and interpolation.) However, interpolation does improve the look of rotated images, even when they've been shrunk by zooming out.

Beyond the Edge

- Set the zoom of both dimensions to some small value, such as 0.25.

When the image does not fill the entire display area because of shrinking or rotation, **jit.rota** has to decide what to do with the rest of the matrix that lies outside the image area. At present **jit.rota** is setting all the cell values outside the image area to 0, making them all black. The way that **jit.rota** handles the cells that lie outside the image boundaries is determined by its `boundmode` attribute. The different available `boundmode` settings are presented in the popup menu labeled *Space outside the image* in the upper-right corner of the patch. We initialized the `boundmode` value to 1, which instructs **jit.rota** to clear all the outlying cells. Here is the meaning of each of the `boundmode` settings:

- 0 *Ignore*: Leave all outlying cells unchanged from their previous values.
 - 1 *Clear*: Set all outlying cell values to 0.
 - 2 *Wrap*: Begin the image again, as many times as necessary to fill the matrix.
 - 3 *Clip*: For all the outlying cells, continue to use the values of the boundary cells of the image.
 - 4 *Fold*: Repeat the image, flipped back in the opposite direction.
- For special effects when the image is zoomed out, try setting the `boundmode` attribute to 2 (wrap) for a "Warhol" duplicate image effect, or 4 (fold) for a kaleidoscope effect.
 - Now try turning the automated rotation back on, to combine rotation and zoom, and modify the different parameters (*Degrees per bang*, *Zoom*, and *Space outside the image*).
 - When you have finished experimenting, turn off the automated rotation and return the zoom attributes (`zoom_x` and `zoom_y`) to 1.

Some Adjustments—Anchor Point and Offset

Up to now we've been using the center of the image as the center point of the rotation. However, you can actually choose any point around which to rotate the image. The central *anchor point* of the rotation is set with the `anchor_x` and `anchor_y` attributes. Right now those attributes are set to 160 and 120, but you can change them in the **number boxes** labeled *Anchor point*.

- Try different anchor points, and drag on the *Rotation Angle* **number box** to see the effect. Some anchor point settings you might want to try are 0,0 or 40,30 or 160,-120 or 320,240. You might want to set the `boundmode` attribute to 1 so that you can see the effects of different rotations more clearly. Note that the `anchor_x` and `anchor_y` values are specified relative to the upper-left corner of the matrix, but they may exceed the bounds of the matrix's dimensions.

In addition, you can move the image to a different location in the output matrix after zooming and rotation take place, using the `offset_x` and `offset_y` attributes.

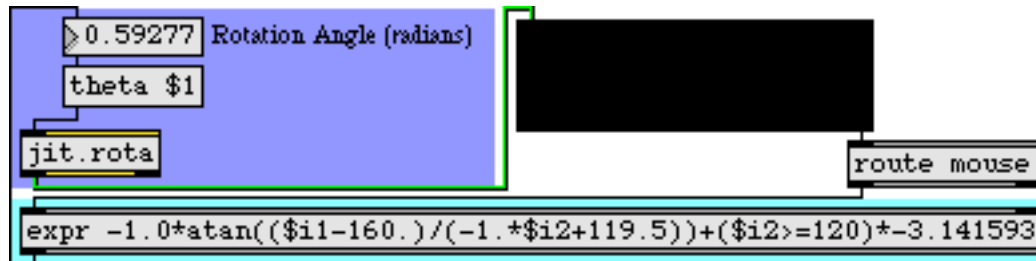
- To see this most effectively, first click on the **message** box above the **pvar** object in the lower right corner of the patch. This will set the rotation angle, boundary mode, zoom, and anchor points back to the settings we used at the outset of this chapter. (We have given names to the relevant user interface objects so that we can communicate with them via **pvar**.) Now set the *Zoom* number box to some value between 0 and 1, to zoom out on the image.
- Use the *Location offset* **number boxes** to move the image around by changing the `offset_x` and `offset_y` values. Try this in conjunction with `boundmode 4`, to see its utility in the "kaleidoscope" mode.
- When you have finished, reset the *Location offset* values to 0.

Rotary Control

We've devised one more way for you to rotate the image.

- Click in the **jit.pwindow** display object and, with the mouse button held down, drag in a small circular motion around the center of the object.

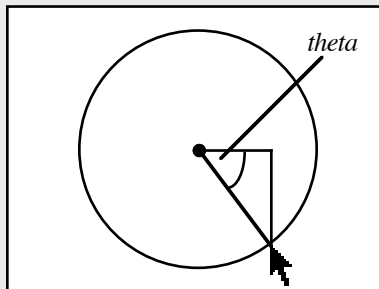
jit.pwindow tracks your mouse movements and, as long the mouse button is down, it sends coordinate information (and other mouse information) out its right outlet in the form of mouse messages. The first two arguments of the mouse message are the x and y coordinates of the mouse, relative to the upper left corner of **jit.pwindow**. We use those coordinates to calculate the angle of the mouse relative to the center of the **jit.pwindow**, and we send that angle to **jit.rota** as the argument to the theta attribute.



*You can use the mouse location in **jit.pwindow** as control information*

Technical Detail: Do you really want to know how we did that calculation? If so, read on.

If we think of the center point of the **jit.pwindow** as the origin point $0,0$, and we think of the current mouse location relative to that as being a point along a circle around the origin, then we can describe a right triangle based on those two points. By taking the arctangent of the mouse's coordinates y/x , we get the angle of the mouse relative to the center of **jit.pwindow**.



So we take the incoming x and y coordinates, and the first thing we do is convert them so that they're relative to the center of the **jit.pwindow**. We do that by subtracting 160 from the x dimension coordinate (so the x values will now go from -160 to 160) and multiplying the y coordinate by -1 (so values will increase as we go up, instead of down) then adding 119.5 to it. (If we added exactly 120, then every time we had a y coordinate of 120 from **jit.pwindow** we'd be trying to divide by 0 in **expr**, which is an undefined mathematical operation.) Once we have converted the x and y coordinates, we take the arctangent of y/x to get the angle in radians, then multiply that angle value by -1 to make clockwise rotation of the mouse cause clockwise rotation of the image.

This method only works within a 180° span, because the arctangent function can't tell the difference between a mouse location and its opposite point on the circle. (The calculation of y/x will be the same for both points.) So, every time the y coordinate of the mouse goes into the bottom half of the **jit.pwindow**, we add an offset of $-\pi$ to the θ angle to distinguish those locations from their counterparts on the opposite side. (That's the last part of the expression.)

Note that this expression only works relative to the point $160,120$ in the **jit.pwindow**. If we wanted to make an expression that works for the central point of *any* size **jit.pwindow**, we'd need to get the **jit.pwindow**'s dimensions with a `getsize` message, and use the size values as variables in our expression. As the math books say, "We'll leave that as an exercise for the reader."

Summary

The **jit.rota** object provides an easy way to rotate an image with its `theta` attribute, specifying an angle of rotation, in radians. It also provides an easy way to zoom in and out on an image with its `zoom_x` and `zoom_y` attributes. You can change the central point of the rotation with the `anchor_x` and `anchor_y` attributes, and you can move the resulting image in the output matrix with the `offset_x` and `offset_y` attributes. You can change the way that **jit.rota** treats the matrix cells that lie outside of the resulting image with the `boundmode` attribute. Using all of these capabilities in combination, you can get image-duplication and kaleidoscope effects in addition to simple zoom and rotation.

Zooming and rotation involve some rather intensive internal calculation by **jit.rota**, so these operations make substantial demands on the computer's processor. If you're curious, the formulae for these calculations are in the *jit.rota.help* file. There are additional attributes, not covered in this tutorial, that give you access to virtually every coefficient in the rotation formula, presenting you even more possibilities for distorting and rotating the image. These, too, are shown in *jit.rota.help*.

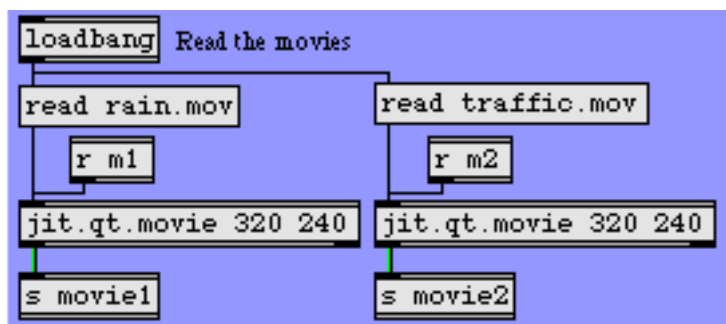
To manage the control of so many attributes at once, you can devise automated Max processes to generate attribute values, and/or interactive controls to change the values with gestures.

Tutorial 16: Using Named Jitter Matrices

In this tutorial we'll learn how to use the name attribute of the **jit.matrix** object to write matrix data from multiple sources into the same matrix. We'll also look at how to scale the size of matrices when they are copied into a new matrix, and how to use the Max low-priority queue to de-prioritize Max events in favor of more time-consuming tasks .

- Open the tutorial patch *16jNamedMatrices.pat* in the Jitter Tutorial folder.

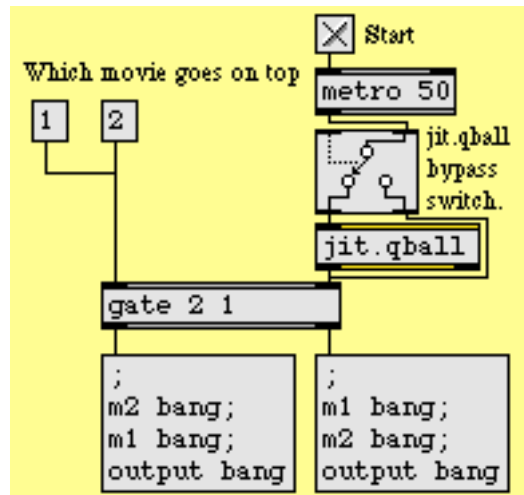
The tutorial patch is divided into five colored regions. The middle (light blue) region contains two **jit.qt.movie** objects. A **loadbang** object reads two movies (*rain.mov* and *traffic.mov*) into the **jit.qt.movie** objects when the patch is opened:



Reading in the movies

Unlike the tutorial patches we've looked at before, the **jit.qt.movie** objects in this patch use **send** and **receive** objects to communicate with the rest of the patch. The **receive** objects named m1 and m2 forward messages to the two **jit.qt.movie** objects. The output matrices of the two objects are then sent (using **send** objects) to **receive** objects named movie1 and movie2 elsewhere in the patch.

The yellow region at the top of the patch contains the **metro** object that drives the Jitter processes in the patch:



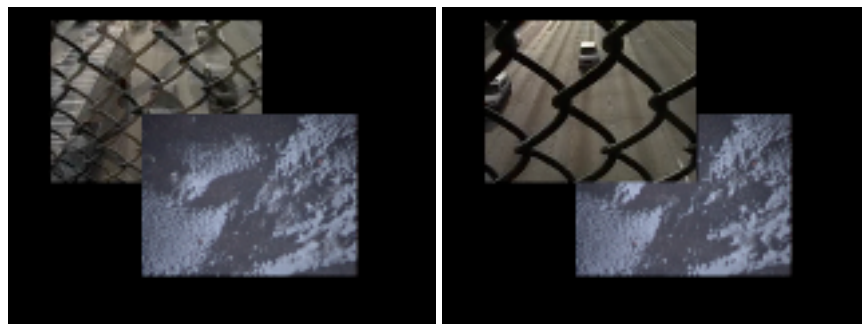
*The **metro** object drives one of the two **message** boxes*

- Click the toggle box to start the **metro**. The three **jit.pwindow** objects in the patch will start to display an image.

Order of Importance

The **metro** object goes through a **Ggate** object and an object called **jit.qball** (about which we will have something to say later) into a **gate** object. The bang messages sent by the **metro** are routed by the **gate** to one of two **message** boxes. Our final output matrix (in the **jit.pwindow** at the bottom of the patch) will change depending on which **message** box gets a bang.

- Click the two **message** boxes attached to the left inlet of the **gate** (1 and 2). Notice how the **jit.pwindow** at the bottom changes:



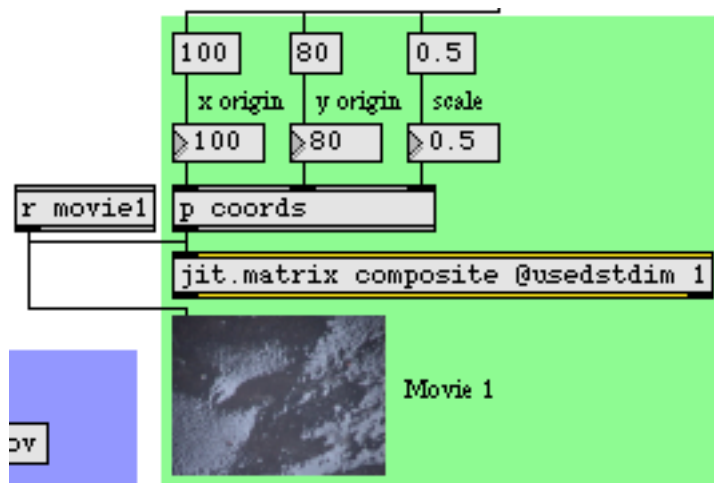
The final output matrix changes depending on the message ordering of the patch.

The two **message** boxes both send bang messages to the same three named receive objects (m1, m2, and output). The difference between the two is the order in which they send the messages. The lefthand **message** box (driven by the **metro** when the **gate** is set to 1) sends the first bang to the **receive** object labelled m2, then the **receive** object labelled m1, and then the **receive** object labelled output. This causes the right **jit.qt.movie** object (containing the traffic footage) to send out a matrix, followed by the left **jit.qt.movie** object (with the rain movie). Finally, the **jit.matrix** object at the bottom of the patch receives a bang, causing our final matrix to be sent out. The righthand **message** box (driven by the **metro** when the **gate** is set to 2) reverses the order of the two bang messages driving our **jit.qt.movie** objects (the left **jit.qt.movie** outputs a matrix first, followed by the right **jit.qt.movie** object).

The order in which these messages occur only becomes relevant when we look into what happens between the two **jit.qt.movie** objects and the final **jit.pwindow** object.

What's in a Name?

Once our **jit.qt.movie** objects receive their bang messages, they output a matrix to the **send** objects below them, which in turn pass their matrices over to **receive** objects named movie1 and movie2. The **receive** objects (in two identical regions at the right of the patch) are connected to **jit.pwindow** objects as well as two *named* **jit.matrix** objects:



*The named **jit.matrix** object*

Both of the **jit.matrix** objects at the right of the patch (as well as the **jit.matrix** object at the bottom of the patch above our final **jit.pwindow**) have a name. The name attached to all three of these objects is *composite*. The result of this is that all three of these **jit.matrix** objects share the *same* matrix data, which is contained in a Jitter matrix called *composite*.

Once we know that our two **jit.qt.movie** objects write data into the same Jitter matrix (via two separate **jit.matrix** objects sharing the same name) we can understand why the ordering of the bang messages is important. If the left **jit.qt.movie** sends out its matrix first, it writes

data into the composite matrix, followed by the right **jit.qt.movie**, which writes data into the same matrix. If the two matrices write to any cells in common (see below), the matrix that gets there last will *overwrite* any data that was in those cells before.

Important Note: If you are ever unsure about the order in which things are happening in your Max patch, you can do a Trace of the patch to see the way in which your patch executes. If you choose the **Enable** command from the Trace menu, and then turn on the **metro** object, you can step through the patch with the **Step** command (-T), to see how it executes. (See the "Debugging" chapter of the *Max Tutorials and Topics* manual for details about how to trace Max messages with the Trace feature.)

The Destination Dimension

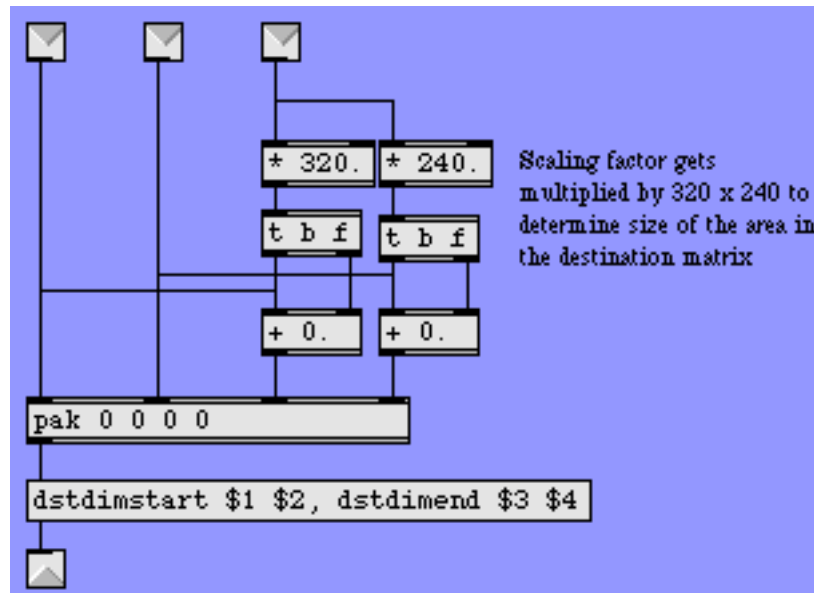
The two **jit.matrix** objects on the right of the tutorial patch have their `usedstdim` attribute set to 1. This allows us to scale the matrices sent by our **jit.qt.movie** objects so that they only write to a certain region of the composite Jitter matrix.

- Play around with the **number** boxes labelled `x origin`, `y origin`, and `scale` connected to the two subpatchers labelled **p coords**. Notice how you can move and resize the two images from the **jit.qt.movie** objects in the composite matrix.



Picture within a picture

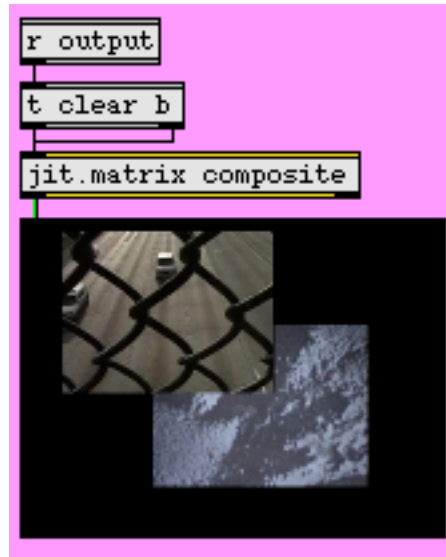
The subpatches **p coords** contain identical helper patches to format the `dstdimstart` and `dstdimend` attributes for the **jit.matrix** objects. These attributes specify the upper left and lower right coordinates, respectively, to use when copying data into our composite Jitter matrix. The `usedstdim` attribute simply tells the **jit.matrix** object to use those attributes when copying data. When `usedstdim` is set to 0, the incoming matrix is scaled to fill the entire matrix referred to by the **jit.matrix** object.



Scaling the input matrices before the are written into our shared matrix

The three numbers that we send into the subpatches get formatted by the Max objects inside to generate a list that describes the upper left and lower right areas of the output matrix which we want to fill with our input matrix. The **message** box before the outlet uses \$ substitution to replace the relevant arguments for the attributes with numbers from the list.

The last thing that happens after our two matrices have been written into the composite matrix is that a bang is sent to the **receive** object named output:



The final result

The region at the bottom of the tutorial patch contains a third named **jit.matrix** object. The bang sent by the **metro** goes through a **trigger** object that sends a bang to the **jit.matrix** (causing it to output its matrix to the **jit.pwindow**) followed immediately by a clear message. The clear message erases (zeroes) all the cells in the Jitter matrix named composite. If we didn't clear the matrix, changing the `dstdimstart` and `dstdimend` attributes of any of the **jit.matrix** objects could result in cells left over from a previous output location of our movies.

Jumping the Queue

The **jit.qball** object at the top of the patch provides an invaluable service in the event that Max can't keep up with our demands. The **metro** object (which is sending out bang messages every 50 milliseconds) is driving three separate operations (writing the two matrices from the **jit.qt.movie** objects into our named Jitter matrix, as well as displaying the data and clearing the matrix so we can start over). The **jit.matrix** object writes data into its internal Jitter matrix (in this case our named composite matrix) in a way that allows it to be *usurped* by a subsequent message. It also allows other Max events that are scheduled at a higher priority to happen while it works on a task. This makes it possible to display the matrix (or write more data into it) before the previous operation has finished, causing flicker and other unexpected results. The **jit.qball** object places messages input into the object at the back of Max's low priority queue where they too can be *usurped* by another message. This way, if **jit.qball** gets a bang from the **metro** object before all the current Jitter tasks are complete, it will wait until everything else in the low priority queue is finished

before sending out the bang. Similarly, if another bang comes along before that first bang has been sent (i.e. if it takes more than 50 milliseconds for the rest of the patch to do everything), the first bang will be usurped (jettisoned) in favor of the second. This allows you to set a maximum hypothetical rate of events in a Max patch without having to worry about events accumulating too rapidly for the objects in the patch to keep up.

- Click on the **Ggate** object labeled `jit.qball` bypass switch so that the output of the **metro** object bypasses the **jit.qball** object. The composite image in the **jit.pwindow** at the bottom will start to flicker, indicating that messages are arriving out of order.

Normally, sending a bang to a Jitter object will usurp an already pending event in that object (e.g. a bang that has already arrived but hasn't been dealt with yet by the object). However, the **jit.qball** object gives us this kind of control over multiple chains of Jitter objects, automatically usurping events ("dropframing") to guarantee that messages arrive in the right order.

Summary

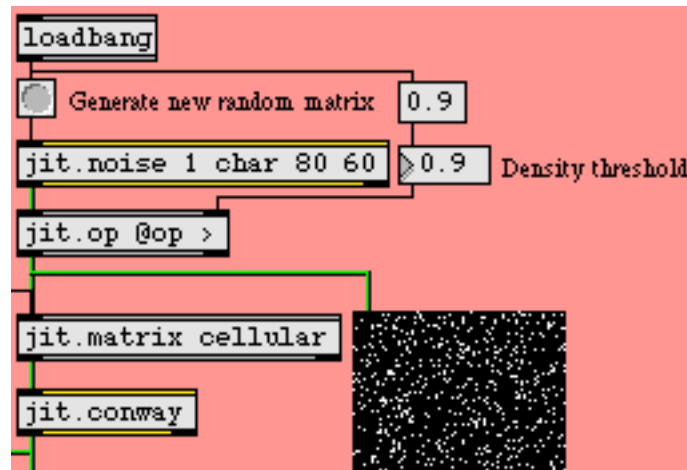
By giving a single name to multiple **jit.matrix** objects, you can write into and read from a common Jitter matrix in different parts of your patch. You can scale a Jitter matrix while copying it into the internal matrix of a **jit.matrix** object by using the `dstdimstart` and `dstdimend` attributes and by setting the `usedstdim` attribute to 1. The **jit.qball** object allows you to de-prioritize Max events by placing them in the low-priority queue where they can be usurped by subsequent events if there isn't enough time for them to execute.

Tutorial 17: Feedback Using Named Matrices

This tutorial shows a simple example of using named **jit.matrix** objects in a feedback loop. We'll use a matrix of random values to seed an iterative process (in this case, Conway's Game of Life).

- Open the tutorial patch *17jMatrixFeedback.pat* in the Jitter Tutorial folder.

The tutorial patch generates an initial matrix of randomized values with the **jit.noise** object:



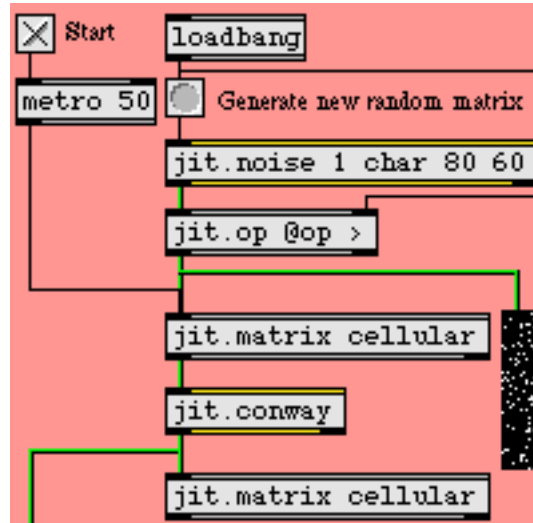
*The **jit.noise** object*

The **jit.noise** object generates a Jitter matrix full of random values. The dim, plane count, and type attributes of the object determine its output matrix (in this instance, we want an 80 x 60 cell matrix of one-plane char data). Our random cell values (which are initially in the range 0 -255) are then set to true (0) or false (255) by the **jit.op** object. The > operator to **jit.op** takes the value from the **number** box (arriving at the right inlet of the object) and uses it as a comparison operator. If a cell value is below that value the cell's value is set to 0. Otherwise the cell is set to 255. Sending a bang to **jit.noise** will generate a new random matrix.

- Try changing the number box attached to the **jit.op** object. Click the **button** attached to the **jit.noise** object to generate a new matrix each time. Notice how higher comparison values yield fewer white (255) cells. The small **jit.pwindow** below the **jit.op** object shows you the random matrix. The one-plane matrix data is correctly interpreted by the **jit.pwindow** object as grayscale video.

Jitter Matrix Feedback

The quantized noise we've generated at the top of our patch goes from the **jit.op** object into a **jit.matrix** object with the name of cellular:



*Two named **jit.matrix** objects in a feedback loop*

This **jit.matrix** object, which receives bang messages from a **metro** object at the top of the patch, is connected to an object called **jit.conway**, the output of which is hooked up to another **jit.matrix** with the same name (cellular) as the first. The result of this is that the output of the **jit.conway** object (whatever it does) is written into the *same* matrix that its input came from, creating a feedback loop.

- Start the **metro** object by clicking the **toggle** box. The **jit.pwindow** at the bottom of the patch will show you the output of the **jit.conway** object.

If you want to start with a fresh random matrix, you can always copy a new matrix into the feedback loop by clicking the **button** attached to the **jit.noise** object. The matrix from the **jit.op** object will go into our shared cellular matrix and will be used in the feedback loop.

The Game of Life

The **jit.conway** object performs a very simple cellular automata algorithm called the 'Game of Life' on an input matrix. Developed by John Conway at Princeton University, the algorithm simulates cycles of organic survival in an environment with a finite food supply. The cells in the matrix are considered either alive (non-0) or dead (0). Each cell is compared with the cells surrounding it in space. If a live cell has two or three live neighbors, it stays alive. If it has more or less than that number, it dies (i.e. is set to 0). If a dead cell has exactly three live neighbors it becomes alive (i.e. is set to 255). It's that simple.

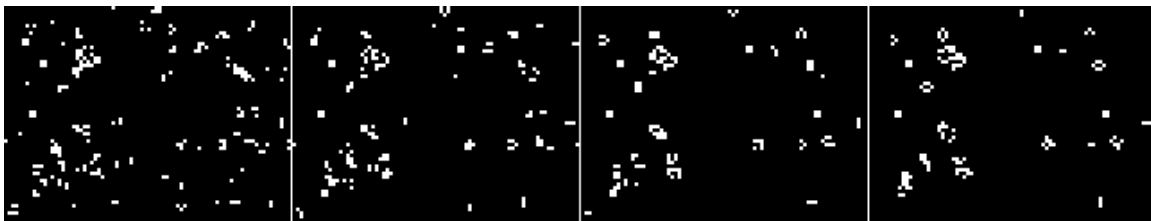
Every time the **jit.conway** object receives an input matrix it performs one generation of the Game of Life on that matrix. Therefore, it makes sense to use the object inside of a feedback loop, so we can see multiple generations of the algorithm performed on the same initial set of data.

For example, the initial random matrix:



Some random matrix values

Generates the following matrices in the first four iterations through the **jit.conway** object:



The first four generations of the Game of Life performed on the dataset above

After seeding the feedback loop with a random matrix, you can turn on the **metro** object and watch the algorithm run! The Game of Life is designed in such a way that the matrix will eventually stabilize to either a group of self-oscillating cell units or an empty matrix (a dead world). In either case you can just bang in a new set of numbers and start all over again.

Summary

You can use the name attribute of the **jit.matrix** object to create feedback loops in your Jitter processing. By using two **jit.matrix** objects with the same name at either end of an object chain, you create a patch where the output of the chain gets written to the same Jitter matrix as the input comes from. The **jit.noise** object generates matrices of random numbers of any type, dim, or plane count. The **jit.conway** object, which works best within such a feedback loop, performs simple cellular automata on an input matrix.

Tutorial 18: Iterative Processes and Matrix Re-Sampling

This tutorial demonstrates a more complicated example of when to use named **jit.matrix** objects, as well as how to use **jit.matrix** objects to upsample and downsample an image.

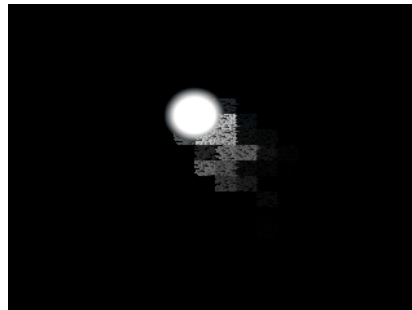
- Open the tutorial patch *18jMatrixIteration.pat* in the Jitter Tutorial folder.

The upper left-hand corner of the patch contains a **jit.qt.movie** object that has a still image (the file *fuzz_circle.jpg*) loaded into it when the patch opens.



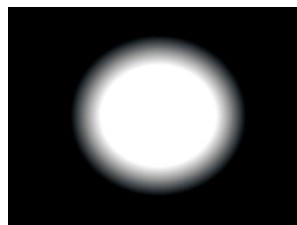
Read the image

- Start the **metro** object by clicking the **toggle** box above it. You should see an image appear in the **jit.pwindow** in the lower right of the tutorial patch:



Our little comet

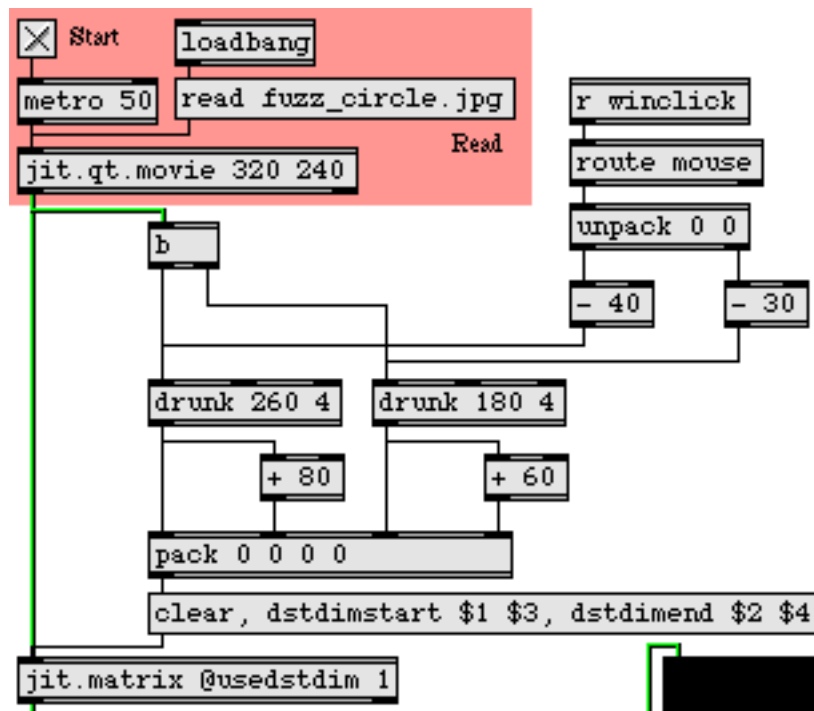
The *fuzz_circle.jpg* file contains an image of a white circle with a black background, which is being scaled in size to appear as a small circle inside of our final matrix.



The real fuzz circle

Getting Drunk

The top part of the patch writes the image from the **jit.qt.movie** into the first **jit.matrix** object in the chain. A bang generated by the **bangbang** object changes the **dstdimstart** and **dstdimend** attributes of the **jit.matrix** object with each frame, randomly varying the coordinates using Max **drunk** objects. Note that our first **jit.matrix** object has its **usedstdim** attribute set to 1, so that it will scale the input matrix:



The drunk part of the patch

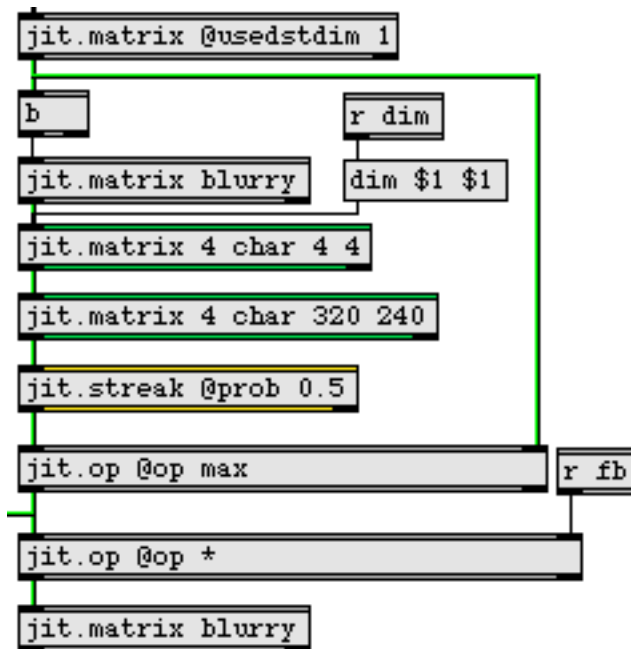
This first **jit.matrix**, therefore, simply serves to scale the circle image to fit in a small (80 by 60) region of our output matrix. Note that the **message** box that formats the coordinates for the scaled image also clears the matrix with every frame (with a **clear** message), so that there are no artifacts from a previously written image. The Max **drunk** objects vary the placing of the circle, causing it to jitter around (no pun intended).

- Click somewhere in the **jit.pwindow** in the lower right corner of the patch. The circle will jump to the position you clicked, and begin to move from there.

The result of a mouse click in the **jit.pwindow** is sent to the **receive** object with the name **winclick**. This message is then stripped of its selector (**mouse**) and the first two elements (the **x** and **y** position of the mouse click) are extracted by the **unpack** object. These coordinates are then used to set the new origin for the **drunk** objects.

The Feedback Network

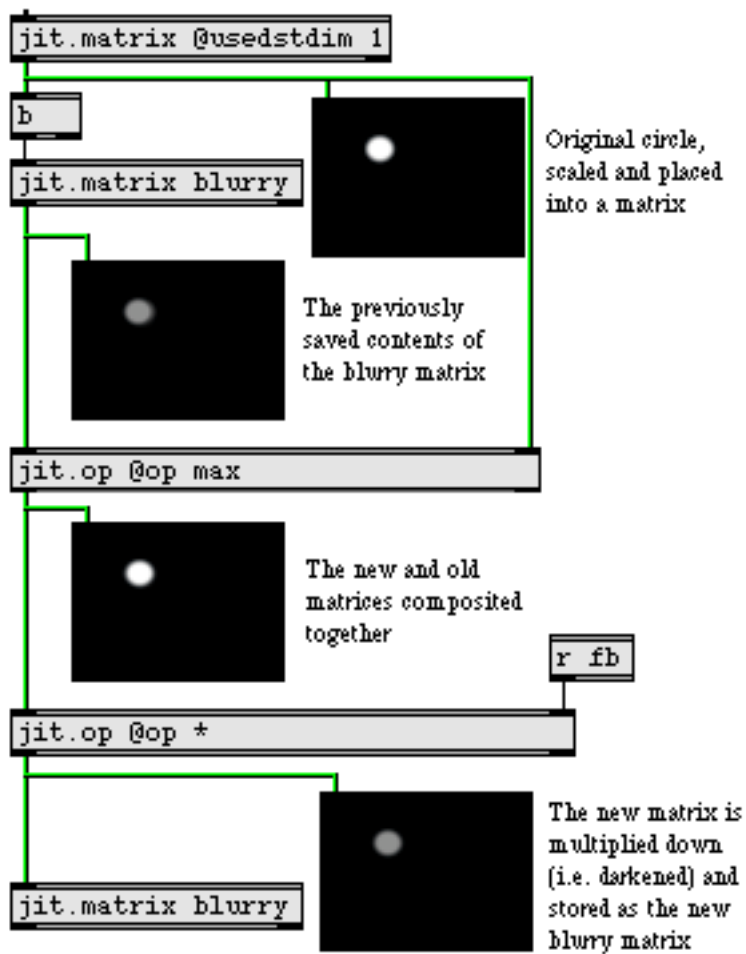
Once our circle image has been scaled and placed appropriately by the **jit.matrix** object, our patch enters a feedback chain that centers around a pair of **jit.matrix** objects sharing a matrix named **blurry**:



The feedback loop in our patch

This section of the patch contains four **jit.matrix** objects (not including the one at the top which scales down the circle image). Two of the objects share a name (**blurry**) and are used simply to store and retrieve previous matrices generated by the rest of the patch. The topmost **jit.matrix** object sends its matrix to the rightmost inlet of the first **jit.op** object in the patch. In addition, it sends a bang to the first named **jit.matrix** object using a **bangbang** object, causing it to output its stored matrix (called **blurry**). This matrix eventually ends up in the left inlet of the **jit.op**, where it is then displayed (by the **jit.pwindow**) and multiplied by a scalar (the second **jit.op** object). It eventually overwrites the previous **blurry** matrix (by going into the bottom named **jit.matrix** object).

Without worrying about what the intermediate Jitter objects do, you can see that the blurry matrix will hold some version of the previous 'frame' of our circle image:



A scaled-down and illustrated map of our patch

The new and old images are combined by the first **jit.op** object using the max operator. The max operator compares each cell in the two matrices and retains the cell with the highest value. The second **jit.op** object (with the * operator) serves to darken our image by multiplying it by a scalar (set by the **number box** on the right of the patch that is sent to the **receive** object named fb):

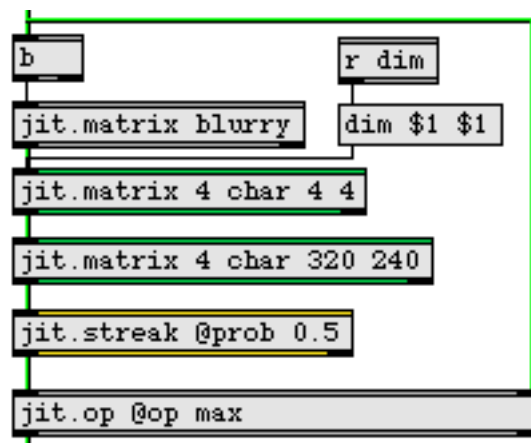


The feedback amount sets how much the image is darkened before being stored in the blurry matrix.

- Change the feedback amount of the patch by playing with the **number box** labeled *Feedback* on the right of the patch. Notice how the trails after the circle increase or decrease when you move the circle by clicking in the **jit.pwindow**, depending on how the feedback amount is set.

Downsampling and Upsampling

The final step in our image processing algorithm concerns the part of the patch in between the first named **jit.matrix**, which sends out the matrix saved there during the previous frame by the **jit.matrix** at the bottom, and the first **jit.op** object, which composites the previous matrix with the new one:



Using jit.matrix objects for resampling of an image

The two **jit.matrix** objects colored green in the tutorial patch are used to *resample* the blurry image matrix coming out of the **jit.matrix** object above them. The first of the two **jit.matrix** objects has its dim attribute set to 4 x 4 cells. This size can be changed by setting the attribute with the **number** box at the top left with the caption *Pixelation*. This number gets sent to the **receive** object named dim above the **jit.matrix** object.



Change the pixelation of the trails

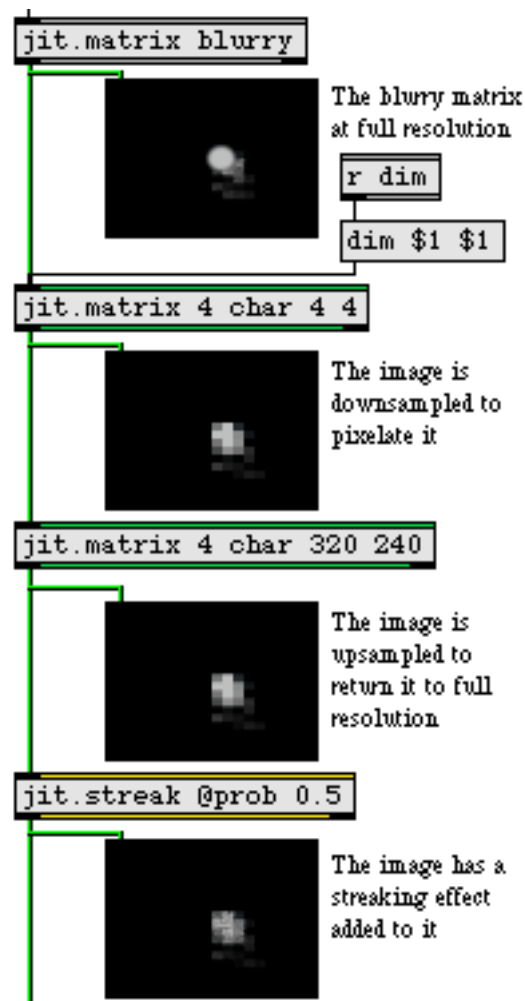
- Change the **number box** labeled *Pixelation* in the blue region of the tutorial patch. Notice how the circle trails change.

By downsampling the image matrix, the **jit.matrix** object copies the 320x 240 matrix from its input into a much smaller matrix, jettisoning excess data. The result is a pixelation of the image that you can control with the dim of the matrix.

The second **jit.matrix** object upsamples the matrix back to a 320x 240 matrix size. This is so that when subsequent Jitter objects process the matrix, they will have a full resolution image to work with and will output a full resolution matrix.

The **jit.streak** object adds a nice effect to the pixelated trails by randomly 'streaking' cells into their neighbors. The prob attribute of **jit.streak** controls the likeliness that any given cell in the matrix will be copied onto a neighboring cell. Our **jit.streak** object has a prob attribute of 0.5, so there's a 50% chance of this happening with any given cell.

Technical Detail: By default, **jit.streak** copies cells towards the left. Changing the direction attribute will alter this behavior. There is also a scale attribute that determines the brightness of the 'streaked' cells as compared to their original values. The help patch for **jit.streak** explains more about how the object works.



*Our effects chain with intermediate **jit.pwindow** objects to show the processing*

Summary

Pairs of named **jit.matrix** objects can be used effectively to store previous iterations of a Jitter process. These techniques can be used to generate video delay effects by combining the previous matrix with the current one using matrix compositing objects such as **jit.op**. You can also use **jit.matrix** objects to resample an image (using the **dim** attribute), both to perform an algorithm more efficiently (the smaller the matrix, the faster it will be

processed by subsequent images) and to create pixelation effects. The **jit.streak** object performs random cell streaking on an input matrix by copying cells over to their neighbors according to a probability factor (set by the `prob` attribute).

Tutorial 19: Recording QuickTime movies

This tutorial shows how you can record a single matrix or a sequence of matrices to disk as a QuickTime movie. We'll demonstrate the use of the **jit.qt.record** object for recording in real time and non-real time. Along the way, we'll also show you how to adjust some of the settings of the output movie.

Like most Jitter objects, the **jit.qt.record** object operates according to an event-driven model rather than a time-driven one—frames sent to a record-enabled **jit.qt.record** object are appended to the movie in progress as "the next frame" based on the timing characteristics of the output movie, without regard for the relative time between the arriving frames. This results in very smooth movies with consistent timing between frames, but it requires some preparation before recording.

The **jit.qt.record** object also offers a time-driven (realtime) method of recording in addition to the default mode described in this tutorial. Although the realtime mode might seem simpler to use, the recorded output won't be as smooth because frames of the incoming movie will usually be dropped fairly regularly during recording. You can set the time-driven mode for the **jit.qt.record** object by using the `realtime` message, but it is quite different from the realtime operation described in this tutorial.

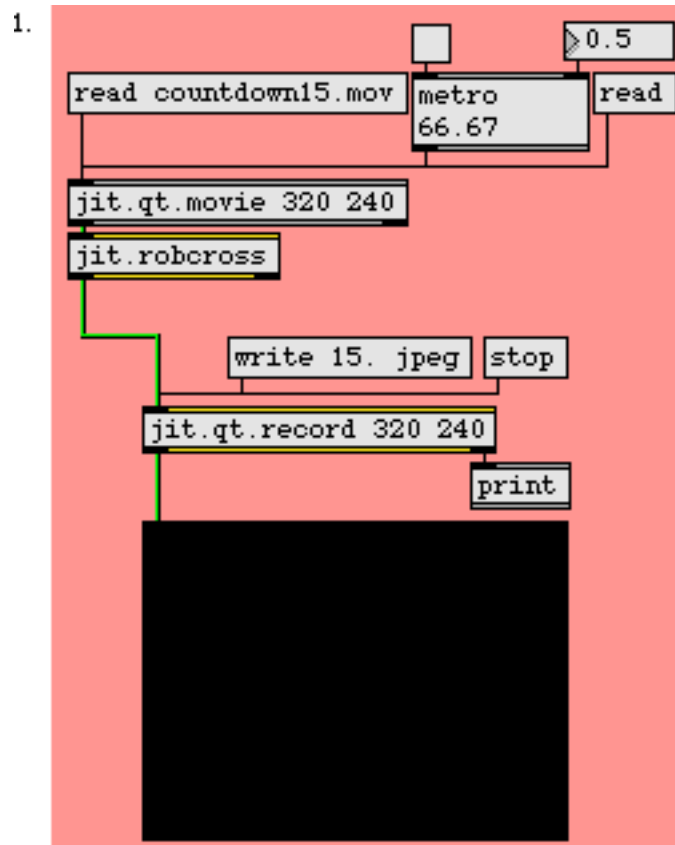
Your mileage may vary

Each and every matrix that arrives at a recording **jit.qt.record** object must be compressed before becoming part of your output movie. The speed of this compression depends on several factors: the speed of your processor and your disk drive, and—most importantly—the type of compression used (referred to as the *codec*, short for *compression/decompression*). The examples in this tutorial will use the Photo-JPEG codec. It should yield fairly consistent results from machine to machine without a lot of high CPU usage and disk access. On older or slower systems, it's possible that you'll notice some timing inaccuracy with the first set of examples for this tutorial. If you do, don't worry; the second part of this tutorial will show you how to achieve excellent results on even the most modest system by working in non-realtime mode.

On the clock

- Open the tutorial patch *19jRecording.pat* in the Jitter Tutorial folder.

The first example shows a very basic recording setup. Notice that the **jit.qt.record** object takes two arguments (320 and 240) that specify the width and height of any movies we record with this object.



A simple recording patch

- Click the **message** box that says read countdown 15.mov to load a movie into the **jit.qt.movie** object, and start the **metro** object by clicking on the **toggle** box connected to its inlet. You will see the movie's image appear in the **jit.pwindow** object, since the **jit.qt.record** object passes any matrices it receives out its left outlet. (The **jit.robcross** object in this patch is a simple edge detector that we're using to process the matrices passing through it).
- Click the **message** box that says write 15. jpeg. to start writing the movie (we'll look at the contents of the **message** box in more detail in a moment). A file Dialog Box will appear, prompting you for a file name. When you enter a valid name and click the

Save button, recording begins to the filename you've chosen (For the purposes of example, let's assume you've named your movie *myfilename.mov*). Click the stop **message** box to stop recording.

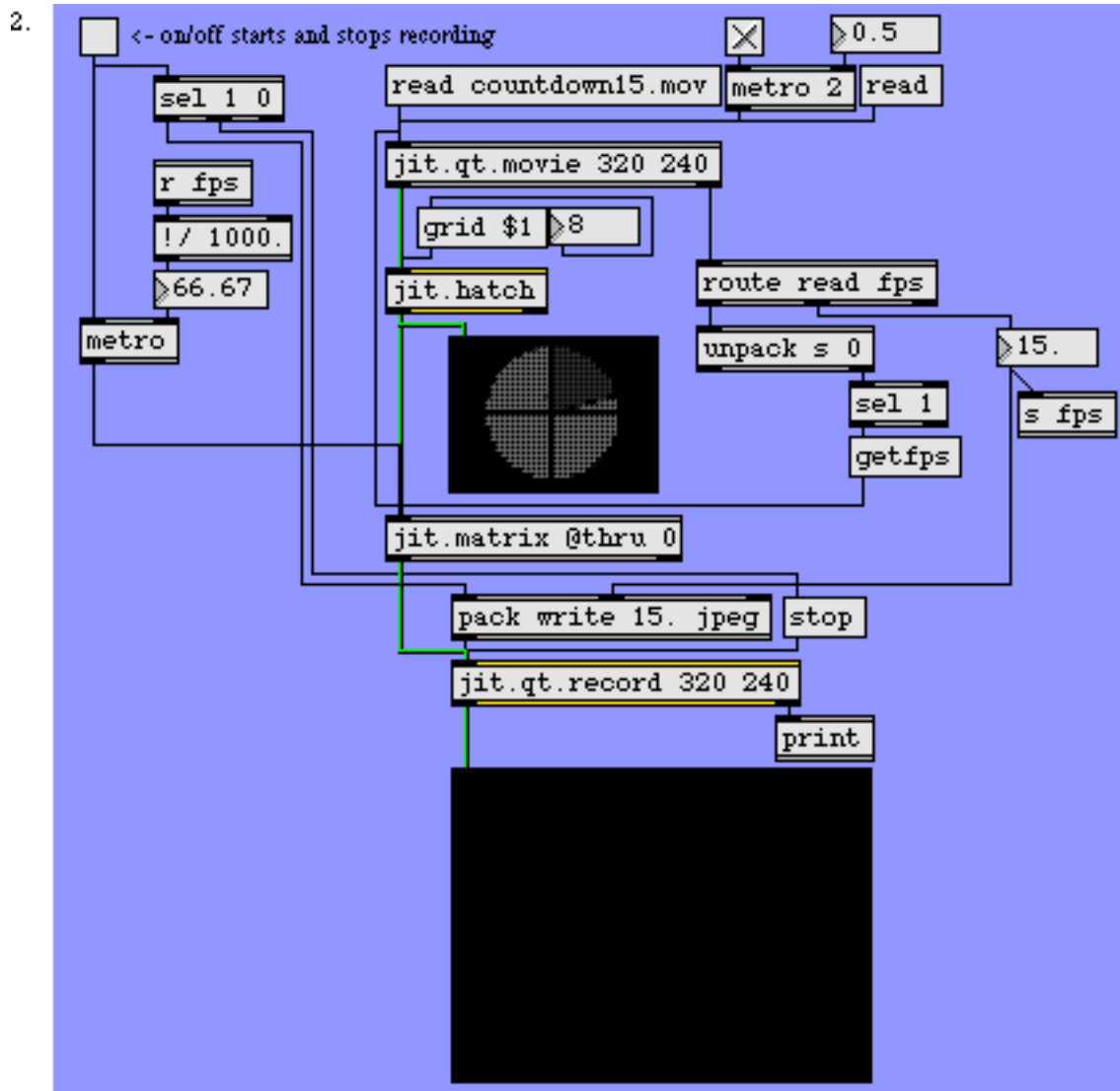
The **jit.qt.record** object sends a message out its right outlet after a write operation to confirm that the movie was successfully written. Since we've connected that outlet to a **print** object in our patch, we can see the results by looking at the Max Window. If everything went as expected, you should see `print: write myfilename.mov 1` in the Max Window. If there was a problem, the number after the filename will not be a 1.

- Click the read **message** box attached to the **jit.qt.movie** object and read in the movie you just recorded. How does it look?

Let's examine the write message to the **jit.qt.record** object for a moment. The write message puts **jit.qt.record** into a record-enabled mode. The actual recording doesn't begin until the object receives a matrix in its inlet. The write message can take several optional arguments to specify a file name, frame rate, the codec type, the codec quality and the timescale for the output movie. In this example we're only using two arguments (15. and jpeg) to specify the frame rate (15. indicates 15 frames per second) and the codec type (jpeg indicates Photo-JPEG) of the output movie. For a full description of all the arguments for the write message, see the Object Reference entry for the **jit.qt.record** object.

Note that we have the **metro** object set to send a bang message every 66.67 milliseconds. This is done to match the input movie's frame rate. If you change the **metro** object's rate and record a few more movies, you will see that a faster **metro** rate causes the movie to be longer and slower, while a slower **metro** rate results in a shorter, faster movie. This happens because of the event-driven model being used by **jit.qt.record**.

The second example patch is a little more complicated, but it has two important additions (and several minor ones).



Same idea, different patch

First, we're automatically detecting the frame rate of the original movie (using the `getfps` message), and using it to set both the rate of the **metro** object driving **jit.qt.record** and the output movie's frame rate, as well. Let's look at that portion of the patch in detail:

- When a movie is successfully loaded into **jit.qt.movie**, the object sends the message `read filename.mov 1` out its right outlet (if the read operation is unsuccessful, the number will not be 1). We're using the **route** object to retrieve this message.

- The **unpack** s 0 object breaks up the remaining elements of the message. We're not interested in the file name, only in the success or failure of the read operation, so we've attached nothing to the left outlet of **unpack**, and attached a **select** object, which tests for the integer 1, to the right outlet. If our read operation was successful, the **select** object will output a bang.
- The bang is used to send the message `getfps` to **jit.qt.movie**, which will respond by sending the message `fps [fps]` out its right outlet. The `[fps]` message element is a floating-point number representing the movie's frame rate. This value is sent from the **route** object's middle outlet, and used to set the **metro** object's speed and the output movie's frame rate.

This patch includes a second improvement. Rather than locking the entire patch to the frame rate of the original movie, we've added a **jit.matrix** object with `thru` turned off to "collect" the output of the patch, and we're using a second **metro** to send it bang messages at the correct frame rate for **jit.qt.record**. This permits the rest of our patch to operate independently, at as fast or slow a rate as we want, with only the very last part of the patch specifically timed to the recording process.

Finally, we've added a neat crosshatching filter (the **jit.hatch** object) with its own editing control (the **number box** and the grid \$1 **message** box). You can use this to modify your movie while you're recording it and watch the effects on playback. You can substitute any kind of Jitter object or patch you want at this point in the recording patch and modify what you record in real time.

Off the clock

Both of the previous patches have a problem: under high processor loads, these patches will drop frames. In many cases, this is not a problem, and the patches we've looked at are well suited to recording in any context, including live performance.

But how can we record a sequence in which *every* frame in an original movie has a corresponding frame in the output movie? How do you render movies in Jitter so that, even under heavy processor or disk load, every processed frame is captured?

There is a way. The **jit.qt.movie** object's `framedump` message offers a non-realtime playback mode that works perfectly with **jit.qt.record**.

While this patch runs slowly in real time, it does a great job of creating a QuickTime movie in non-realtime. Using the **jit.qt.movie** object's **framedump** message and the **jit.qt.record** object, the patch will capture every frame of the original movie, process it, and produce an output movie of identical length.

Here's how it works: the **framedump** message tells the **jit.qt.movie** object to stop playing its movie, and then sends each frame out, one at a time. (**N.B.** You don't need to send a **bang** or **outputmatrix** message to the **jit.qt.movie** object for the frames to be sent.) Because of the way the Max scheduler works, a new frame will not be sent until the previous frame has been fully processed and recorded. Let's try it out.

- Click on the **toggle** box connected to the inlet of the **metro** object to turn it off. (Sending a **bang** to **jit.qt.movie** during the **framedump** operation will cause an extra frame to be sent, which is not what we want.) We've connected the **button** on the top left corner of the patch to both the **pack** object that sends the **write** message, and the **framedump** message. Press the **button**, enter a file name when the File Dialog appears, and wait for the movie to render. When the **framedump** is finished, the **jit.qt.movie** object sends the message **framedump done** from its right outlet. The example patch uses that message to trigger a **stop** message to **jit.qt.record**.

The example patch also takes advantage of the fact that the **jit.qt.record** object sends a matrix out its left outlet after recording a frame. We're using that frame to trigger a **bang** message to a **counter** object that is driving the **grid** attribute of the **jit.hatch** filter. This creates a repeatable sequence of processing changes that happen as the QuickTime movie is captured.

- Click on the "read" **message box** to open the movie you've just created. Turn on the **metro** to display the movie. Click on the **toggle** box connected to the **gate** object to see your new movie. This switches the **jit.qt.movie** object's output directly to the **jit.pwindow**, instead of through the processing objects.

Summary

The **jit.qt.record** object provides event-driven ways to record Jitter matrices as QuickTime movies. You can record the output of a Jitter patch in real time as you manipulate your patch, or out of real time, in render-like operation. You can use the **jit.qt.movie** object's **framedump** message to process an entire movie file without dropping any frames. This method lets you record high quality movies regardless of the load being placed on your computer's processor.

Tutorial 20: Importing and Exporting Single Matrices

This tutorial shows how you can export a single matrix to disk from Jitter. We'll demonstrate the variety of options available, including QuickTime still picture formats, text and ASCII formats and Jitter's own .jxf file format.

In the previous Tutorial, we learned how to save a sequence of matrices as a QuickTime movie—and we can save a single matrix using the same techniques. Since the data that Jitter works with can describe much more than series of images, it makes sense that there should be several additional options for exporting individual matrices.

Import and Export from the `jit.matrix` object

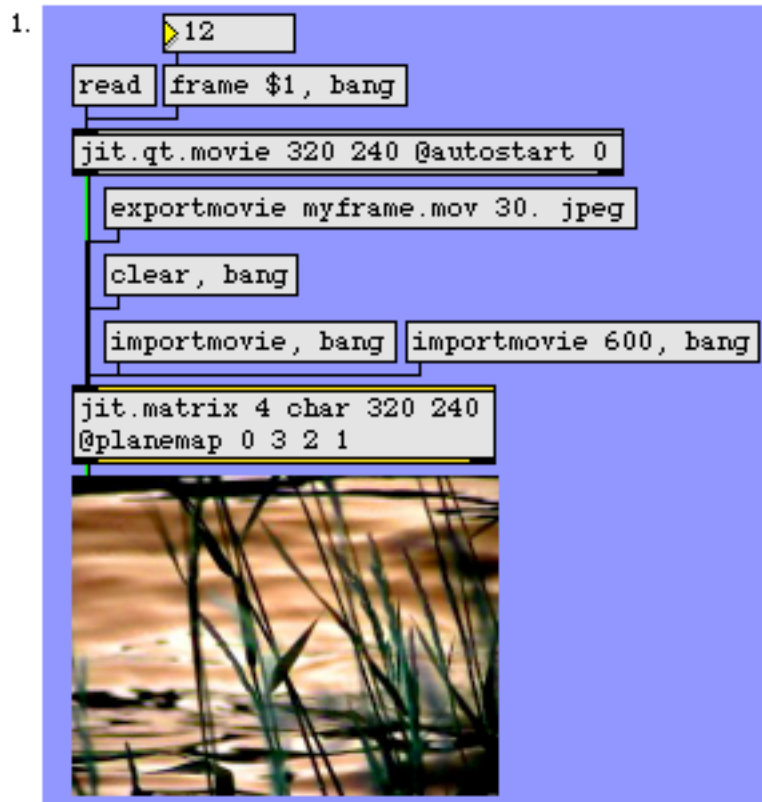
The **`jit.matrix`** object offers two types of single-matrix import/export: QuickTime movie and Jitter binary (.jxf). Both formats permit import and export of a single matrix. We'll discuss both, starting with the movie format.

QuickTime export and import

The QuickTime movie format is identical to the format we learned about in the last tutorial—**Recording QuickTime Movies**. The only difference is that, in this case, an exported output movie will be exactly one frame long. We can also import a single frame of a movie, regardless of the movie's length. The messages `importmovie` and `exportmovie` are used to import and export a single matrix from the **`jit.matrix`** object. The `exportmovie` message conveniently uses the same format as the **`jit.qt.record`** object's `write` message.

In fact, when you use the **`jit.matrix`** object's `exportmovie` message, the **`jit.matrix`** object is briefly creating an internal instance of a **`jit.qt.record`** object, and sending a `write` message to it with the arguments you've specified for `exportmovie`. Although that shouldn't change the way you think about it, it's pretty nifty.

- Open the tutorial patch *20jImportExport.pat* in the Jitter Tutorial folder.



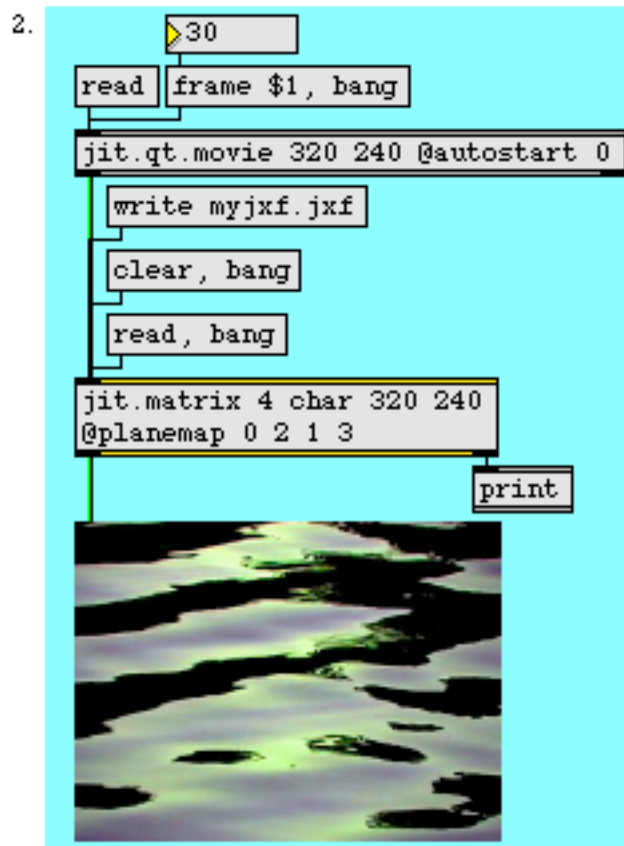
*the **jit.matrix** object's importmovie and exportmovie messages*

- Click the **message** box that says read to load a movie into the **jit.qt.movie** object. Since we're only interested in a single frame, there's no **metro** in the patch. You'll also notice that there is a new attribute set for the **jit.qt.movie** object— **@autostart 0**. Setting this attribute means that our movie will not begin playing automatically. We can choose a frame of the loaded movie we want to export, using the **number box** connected to the **message** box containing the message frame \$1, bang.
- Why are the colors all messed up? They're messed up because we're using the planemap attribute of **jit.matrix** to remap the planes of the incoming movie from 0 1 2 3 (alpha, red, green, blue) to 0 3 2 1 (alpha, blue, red, green). We'll see why in a few moments.
- Click the **message** box exportmovie myframe.mov 30. jpeg to export the frame as a QuickTime movie. As with **jit.qt.record**, we're specifying the frame rate and codec (30. and jpeg). We're also specifying a file name, myframe.mov. The exportmovie message automatically closes the file after writing the frame to it.

- To load the single frame we've just exported , we'll use the `importmovie` message. Since we want to be certain that we're really importing the frame we just exported, we'll clear the **jit.matrix** object first. Click on the **message** box that says `clear, bang`. This message clears the matrix and then outputs it to the **jit.pwindow**, which should now appear entirely black.
- Now, click the **message** box that says `importmovie, bang`. A file Dialog Box should appear, and you should locate and read *myframe.mov*. The movie should have been saved in the same folder as the Tutorial patch. If you can't find it there, use the Finder's Find... command to locate the file on your disk drive. The frame you just exported should now be back in the **jit.pwindow**. The `importmovie` message takes an optional first argument which specifies a file name.
- You're probably wondering what that second `importmovie` message is for. If you try to import a multi-frame movie into **jit.matrix**, the object assumes that you want the first frame, unless you specify a time value that you'd prefer. In this example, we've asked for time value 600 which is one second in from the beginning of a normal QuickTime movie—remember from Tutorial 4 that most QuickTime movies use a timescale of 600, which means they have 600 time values per second.
- Try reading another QuickTime movie by clicking on the **message** box `importmovie 600, bang`. The **jit.pwindow** should now be showing a frame from one second into that movie. You'll notice that the colors in the frame are not switched around this time. This is because we are importing an image matrix *directly into the object*. Since the **jit.matrix** object's `planemap` attribute only affects matrices that arrive via its inlet, no plane remapping occurred.

Jitter binary export and import

Jitter offers its own binary format for single-matrix export, called .jxf. The Jitter binary format is simpler to use than the QuickTime format, and .jxf stores your data without any of the data losses associated with the various codecs you need to use with the QuickTime format. Additionally, the .jxf format supports matrices of all types, dimensions and plane counts, whereas the QuickTime format is only capable of storing 4-plane char matrices (images).



*using .jxf format with the **jit.matrix** object*

The patch shown above uses the Jitter binary format, which is selected using the write and read messages. While it is very similar to the previous example patch, there are some important differences:

- The write message only takes a single argument that specifies the name of the output file. Since the .jxf format is always a single matrix of uncompressed data, we don't need any other arguments.

- The read message doesn't need a time argument (since a .jxf file will only have a single matrix in it). Like the importmovie message, read will take an optional argument for the file name.
- The read and write messages cause the **jit.matrix** object to send confirmation messages out the object's right outlet. We've connected that outlet to a **print** object, so you can see those messages in the Max Window.

Both techniques described in this section are available to the **jit.matrixset** object as well, and function in similar ways.

Import and Export from the **jit.qt.movie** object

The **jit.qt.movie** object offers two export methods you can use to save movie frames as QuickTime still images, rather than movies. The first method exports single frames as image files. The second method exports an entire movie as a sequence of image files, called an *Image Sequence*.

You may find this way of working preferable if you're using images you generate with Jitter for page layout, or if you're using the images in conjunction with other graphic processing software like Photoshop or GraphicConverter.

The **jit.qt.movie** object can export and import QuickTime movies in a number of non-movie formats. For the purposes of this Tutorial, we're only going to cover still image export. For a more comprehensive listing of import and export options available in **jit.qt.movie**, please consult the object's Object Reference entry and help file.

Exportimage

Let's start with the simpler of the two methods: the `exportimage` message to **jit.qt.movie**. Using `exportimage`, we can save the current movie frame as an image file. This method allows you to save your image in one of several standard graphic formats, including JPEG, PNG and TIFF.



*the **jit.qt.movie** object's `exportimage` message*

- Click the **message** box `read` to read in a QuickTime movie. Use the **number box** to navigate to a frame you'd like to export as a still image.
- Click on the **ubumenu** object. The object contains a list of the still image file types available. Choosing one of these items will send the `exportimage` message to **jit.qt.movie** (by way of the **pak** object), with your chosen file type as an argument. A file Dialog Box will appear, where you can enter a file name for the image file you're about to create. The **jit.qt.movie** object will automatically append the correct file name extension (.png, .jpg, .tif, etc.) to the filename when it exports the image, so you don't need to add them. Click Save to continue.
- That's it. To verify that your export was successful, look in the Max Window. The **jit.qt.movie** object sends the message `exportimage myfile 1` from its right outlet after a successful `exportimage` operation (`myfile` is the file name you chose). If `exportimage` was unsuccessful, the number will not be 1. You can either reopen the file in Max, by

reading it into **jit.qt.movie**, or switch to the Finder and open it in your favorite image viewing application.

- The `exportimage` message takes an optional int argument to call up the export Dialog Box. Click on the **message** box `exportimage 1` to see it. First, you'll be prompted for a file name. Then, before exporting the image file, the Dialog will appear, where you have the option to change the file type, and set file type-specific options for the `exportimage` operation.

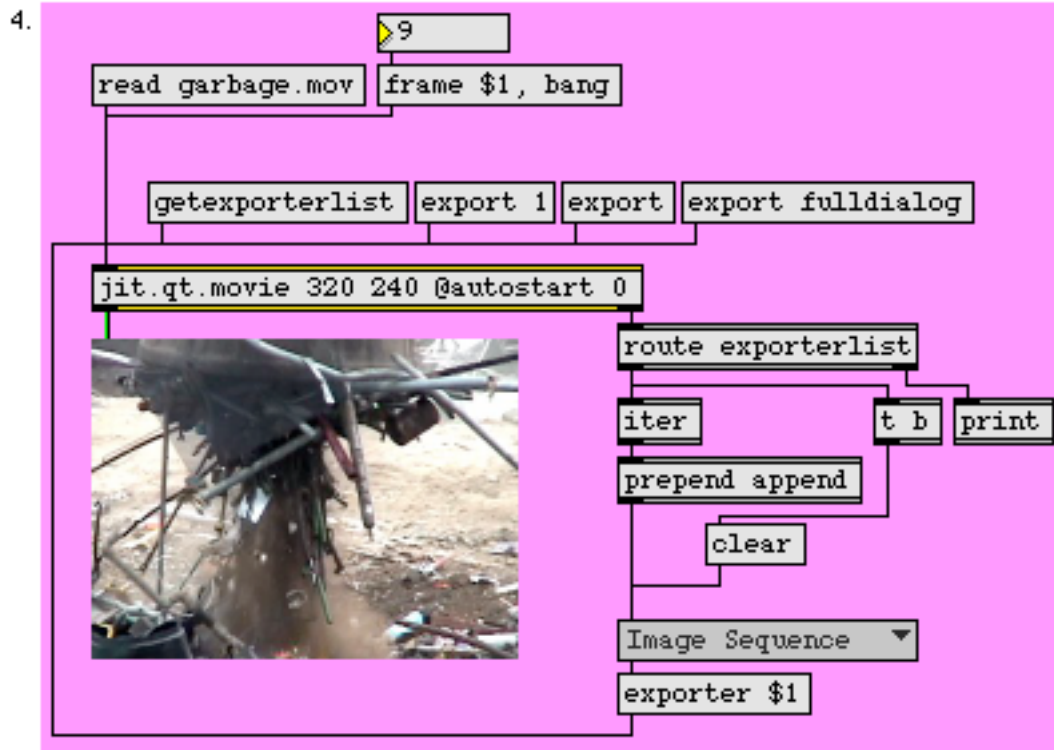
The full format of the `exportimage` message is `exportimage [file name] [file type] [dialog flag]`. All of the arguments are optional. If you omit the file type and dialog flag, the `exportimage` operation will use the last file type you specified. PNG is the default if there is no saved file type.

General Export from **jit.qt.movie**

We can also use **jit.qt.movie** to save an entire movie file as an *Image Sequence*—a series of still images, one for each frame. This process relies on a slightly more complicated export mechanism.

In QuickTime, every movie can potentially have a different set of available exporters—for example, you might want to export the sound tracks in a QuickTime movies as CD audio or AIFF files. Or you may be able to export your Photo-JPEG video tracks as a DV stream. Some movies might permit several image export formats, while others are limited to just a few.

Because of this potentially shifting landscape of options, we've attempted to make exporting in **jit.qt.movie** as flexible as possible. And that added flexibility will require a little extra explanation.



*generalized export from **jit.qt.movie***

- Click the **message** box `read garbage.mov`, to read in that file. We're using this movie because it's only one second long. You may use the **number box** to navigate around the movie, if you like.
- Click on the **message** box `getexporterlist`. This causes **jit.qt.movie** to send a list of available exporter components from its right outlet, preceded by the word `exporterlist`. Our patch breaks that list up and places the list into an **ubumenu** object for easy access.
- Click on the **ubumenu** to view the list. Our list contains entries like AIFF, BMP, FLC, HEURIS MPEG and several others. You probably recognize several of the formats offered. Choose the Image Sequence item. This causes the **message** box `exporter $1` to send the Image Sequence exporter's index number (which is the same as its position in the **ubumenu** item list) to **jit.qt.movie**.

Important Detail: Since each movie could potentially support different exporters, this index number might change from movie to movie for the same exporter component.

- Click on the **message** box export 1. A file Dialog Box will appear, prompting you for a file name. We'd suggest that you create a new folder for saving the sequence, since the operation generates several files. Enter a valid file name and click Save.
- You should now see a special Dialog Box appear, permitting you to adjust the options specific to the exporter you chose, such as file type and frame rate, in the case of the Image Sequence exporter. (Notice that the same file type options are available as with the exportimage operation.) If you leave the frame rate field empty, the export operation will use the movie's native frame rate to generate the images.

Each exporter may have its own options. This Dialog Box will only appear when there is a 1 (which we call the *options flag*) at the end of the export message to **jit.qt.movie**. The export message also accepts an optional file name before the options flag.

You should only need to use the export 1 message once, whenever you start using a new exporter. While you're using the same exporter, the **jit.qt.movie** object will remember the options after you've specified them the first time. Try clicking the **message** box export now. No Dialog Box will appear for the exporter options, and the file will be exported with the same options as before.

- You should see a progress Dialog Box appear briefly. When it disappears, you can check in the Max Window to verify that the operation was successful—**jit.qt.movie** will send the message export myfile 1 if it was. (The number will not be 1 if there was a problem.) If you switch to the Finder and look in the folder where you chose to save the Image Sequence, you should see several files, named sequentially. You've successfully exported your movie as an Image Sequence. Open a couple of the files up and verify that they show different frames.
- Returning to Max, click on the **message** box export fulldialog. The fulldialog argument is another flag to the export message, telling **jit.qt.movie** to open the full QuickTime export dialog, which permits you to set file name, exporter and exporter options all at once.

Technical Detail: When you use the fulldialog flag, **jit.qt.movie** won't know which exporter you chose while inside the Dialog Box. In order for **jit.qt.movie** to remember what your current exporter is, you must use the exporter message.

The jit.textfile object

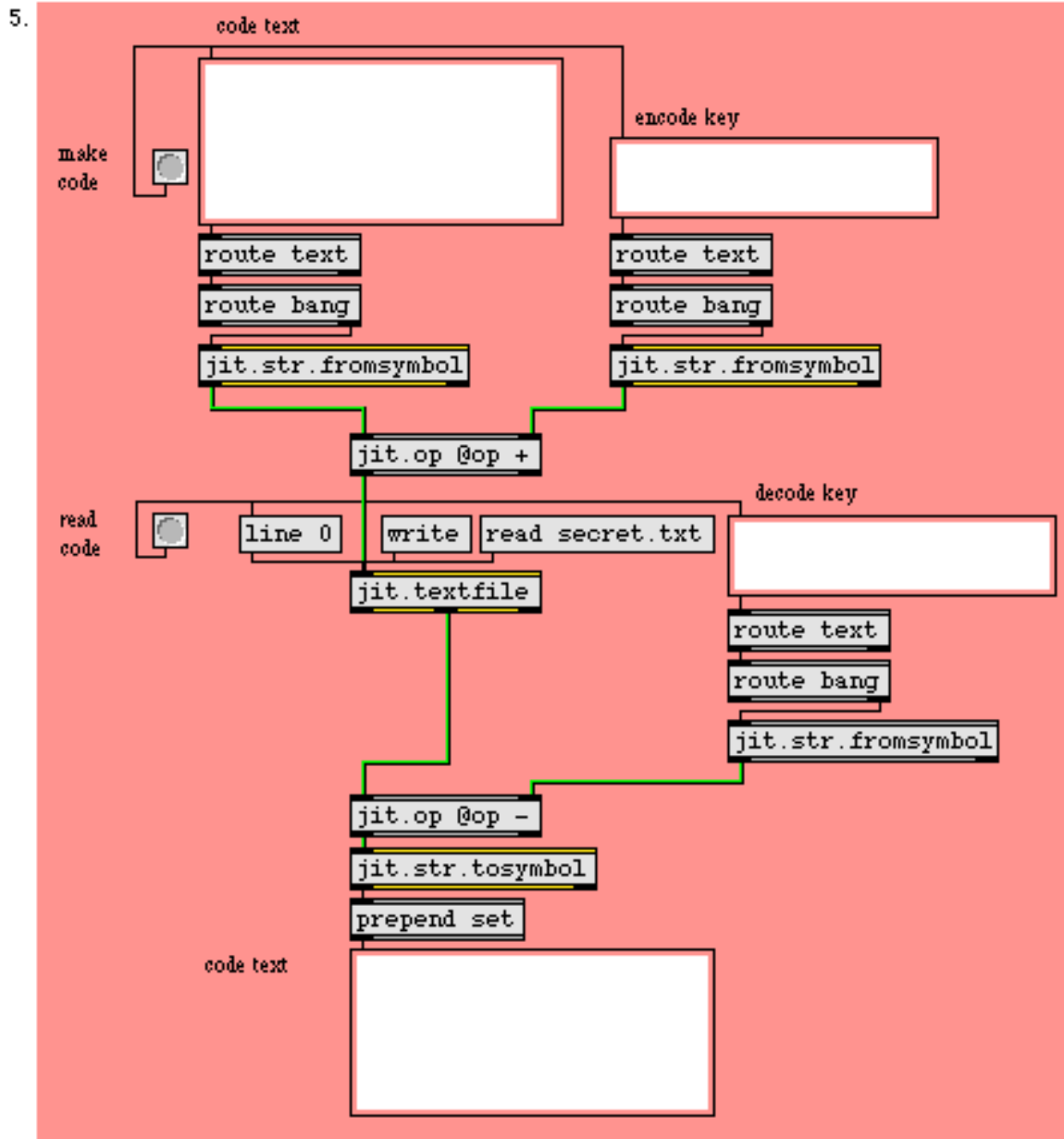
Had enough? Sorry—there's more, but let's take a little break from image data.

Jitter matrices can be used to manage all sorts of data, including text. The **jit.textfile** object provides a convenient interface for importing and exporting text files to and from Jitter. Once you've imported a text file, you can edit it, or send it to other objects for further processing or display as you would with any Jitter matrix. Or you might export a Jitter-generated text file, and continue to work on it in your word processor. In this section, we're going to take a quick look at the **jit.textfile** object's import and export abilities.

You may recall from the *What Is A Matrix?* chapter that one of Jitter's data types, *char*, is the size of a single character. With the notable exception of certain complex character sets (such as Japanese), most languages use the *char* type for computer representation. The correspondence between the numbers and letters has been standardized in the American Standard Code for Information Interchange, usually referred to as *ASCII*.

One of the interesting things about the ASCII code is that the regular characters used in the English alphabet can be represented in less than half of the space available to the *char* type (26 uppercase plus 26 lowercase plus 10 numerals is only 62 characters, compared to the 256 different values that *chars* can represent). Furthermore, all of these regular characters fall in the first half of the *char* range (specifically, between 32 and 126).

The following example patch will take advantage of that fact.



*String processing in Jitter with **jit.textfile***

This patch sends encoded text messages via Jitter matrices. Here's how to use it:

- Let's begin by encoding a message. Choose an encode key—a password that will be used for encoding (and later, decoding) your message. Enter your encode key in the **textedit** object labeled encode key. Enter the message you want to encode in the larger **textedit** object labeled code text, at the top of the patch. For the purposes of this example, you should limit your message to regular English language characters—no accented characters, please.

- Press the **button** labeled make code to encode your text. This causes the contents of the two **textedit** objects to be sent to two **jit.str.fromsymbol** objects. These two objects convert the contents of their respective **textedit** objects from Max symbols into matrices. The resulting two matrices are then added using the addition mode of the **jit.op** object, producing a new encoded matrix that is then sent to the **jit.textfile** object,

It doesn't matter that the two pieces of text (and therefore the two matrices) are different sizes. Jitter automatically scales the right-hand matrix of **jit.op** to the size of the left-hand matrix, stretching it or shrinking it as necessary. You might modify the test patch and use a **jit.print** object to view the output of each object, and then compare it against the output of **jit.op**, to see exactly what's happening.

- Double-click on the **jit.textfile** object. This causes the object's text editor window to open, displaying the encoded message.
- To verify that the encoding was successful, enter your encode key into the **textedit** object labeled decode key, and click on the **button** labeled read code. Your original message will appear in the **textedit** object at the bottom of the patch. We'll look at this process in a moment.
- Click the **message** box that says write to save the message to disk so you can retrieve it later to send to your secret message buddy (don't forget to give them the encode key!). A file Dialog Box will open, where you can enter a file name and click Save to save your message to a text file. You can open this text file with any text editor application. The write message takes an optional argument to specify a file name.
- There's a secret message waiting for you, too. Click the **message** box read secret.txt. The read message reads a text file from disk. In this case, we're using its optional file name argument. Without that argument, a file Dialog Box would open, allowing you to locate a text file on your disk drive.
- Double-click on the **jit.textfile** object again, and verify that there's a new encoded message in the editor window. Close the editor window when you're ready. You can decode this message by entering the decode key *Jitter* in the **textedit** object labeled decode key, and clicking on the **button** labeled read code. Did you get our message?
- Like Max's **Text** object, **jit.textfile** will output a single line in response to the line message. In Jitter, the line will be sent as a *string matrix*. A string matrix is a matrix using the char data type with one dimension and one plane. The matrix is composed of a sequence of characters ending with a zero (0). This is the format generated by **jit.str.fromsymbol**, and the format expected by **jit.str.tosymbol**. We're using the line 0

message to output the first line of **jit.textfile** (our encoded text) and send it to the **jit.op** object for decoding.

- The decoder simply reverses the encoding process—we use the subtraction mode of the **jit.op** object to subtract the key matrix from the encoded message matrix, leaving the original text. The decoded matrix, containing all the letters of the decoded text, is then sent to the **jit.str.tosymbol** object, where it is converted back into the form of an ordinary Max message, suitable for display in the **textedit** object.
- If we wanted this patch to work properly with extended Roman characters, such as accented letters, we'd need to make a couple of modifications. For extra credit, you might try to figure out a way to do that. Hint: the secret is in the **op** attribute to **jit.op**.

Summary

Jitter offers several methods for importing and exporting single matrices from and to disk. The **jit.matrix** object (and the **jit.matrixset** object, not demonstrated in this chapter) allow single matrices to be saved as single-frame QuickTime movies, using the same parameters as the **jit.qt.record** object's **write** message. The **jit.matrix** and **jit.matrixset** objects also support the Jitter binary (.jxf) format— an uncompressed format specially suited to Jitter matrices. The **jit.qt.movie** object's **exportimage** message lets you export a matrix as a single frame image in any of a number of image formats, and the **export** message lets you store video frames as an Image Sequence.

In addition to images, Jitter matrices can be used to manage other kinds of data, such as text. The **jit.textfile** object provides a convenient interface for importing and exporting text files to and from Jitter matrices. The **jit.str.tosymbol** and **jit.str.fromsymbol** objects provide conversion between Max symbols and string matrices. These objects provide one of the means by which data can be translated to and from Jitter's matrix data format.

Tutorial 21: Working With Live Video and Audio Input

This tutorial demonstrates how to use Jitter in conjunction with a QuickTime-compatible image capture device—such as a webcam, a DV camera, or a PCI video input card—to grab video sequences and use them as matrices, or record them directly to disk. We can also use Jitter to record sound directly to disk as a QuickTime movie.

The patch examples in this Tutorial assume that you have a QuickTime-compatible image capture device powered on and attached to your computer.

A Note For Windows Users: The **jit.dx.grab** object offers many of the same capabilities of the **jit.qt.grab** object. The main difference is that the object works natively with DirectX-compatible devices and doesn't require a QuickTime VDIG. Please refer to the Reference and help patch for the **jit.dx.grab** object for more information

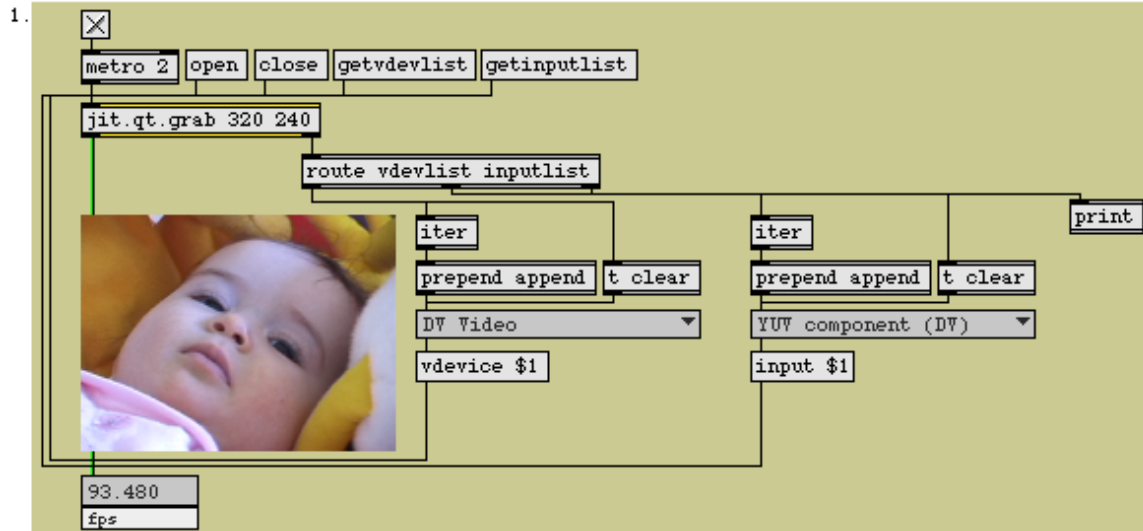
The Basics of Sequence Grabbing

When we capture video images from an input device, we are actually working with a sequence of images. In Tutorial 20, we created a sequence of images using the **export** operation of the **jit.qt.movie** object. This time, we'll use the **jit.qt.grab** object to grab a sequence of images.

The **jit.qt.grab** object provides an interface to one of QuickTime's *components*—the general mechanism used by QuickTime to extend its functionality. Components can include software modules that provide interfaces for new video codecs, or an interface between software and various kinds of hardware, or tasks like image capture and sound output. We've already used a QuickTime component in Tutorial 20 when we used the **export** message, and we'll be looking at other QuickTime component-based Jitter objects in the next several tutorials.

First grab

- Open the tutorial patch *21jSequenceGrabbers.pat* in the Jitter Tutorial folder.



grab it like you want it

Notice that the **jit.qt.grab** object takes two arguments (320 and 240) that specify the width and height of the matrix output by the object. They also represent the dimensions of the internal buffer into which any captured data will be placed. The matrix will always be a 4-plane *char* matrix).

- Click the **message** box that says `getvdevlist`. This message causes the **jit.qt.grab** object to search for available video capture devices. The **jit.qt.grab** object will then send the names of any devices it finds out the right outlet in the form of a list preceded by the symbol `vdevlist`. In our patch, we're using **iter** to break the list up, and placing the device names in an **ubumenu**. Our list only has one item, *DV Video*, but yours may include different items.
- Click on the **ubumenu** and select the device you'd like to use. This causes the **message** box that says `vdevice $1` to send the index of the selected video capture device (which is the same as its position in the **ubumenu**) to **jit.qt.grab**. If you don't explicitly choose a `vdevice`, **jit.qt.grab** defaults to the first selection in the list (equivalent to sending a `vdevice 0` message) when it opens the component connection.
- Click on the **message** box that says `open`. The `open` message tells **jit.qt.grab** to open a connection to the sequence grabber component—to create a component instance—for the video capture device you've selected. Until you send the `open`

message, **jit.qt.grab** will simply output its last matrix when you send it a bang or outputmatrix message—no sequence grabbing will occur.

- Click on the **toggle** attached to the **metro** object, to start sending bangs to **jit.qt.grab**. You should now see your captured video signal in the **jit.pwindow**.

If you don't see your video signal, check the Max Window and see if **jit.qt.grab** reported any errors. It may be possible that your device is in use by another application, or that it has gone to sleep (our video camera sleeps after about 3 minutes of non-use, for example). If your device supports multiple inputs, read on.

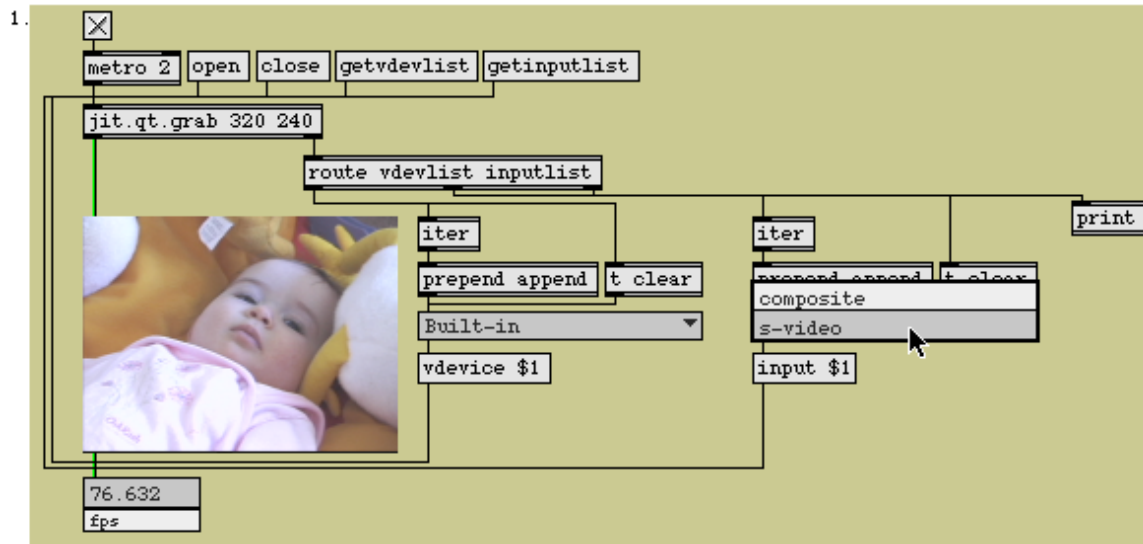
- If you have multiple devices connected to your computer, you can use the **vdevice** attribute to switch between them, even after sending the open message to **jit.qt.grab**.
- To close the component connection, click on the **message** box that says close. Leave this patch open; we'll return to it in a moment.

Switching between inputs

The video capture device you're using may support several different inputs. For instance, a CapSure Card supports s-video and composite inputs, and an ATI Rage 128 Pro supports s-video, composite and tuner inputs. FireWire DV generally supports a single input. In this section, we'll learn how to list those inputs and switch between them.

- Re-open the **jit.qt.grab** object's component connection by clicking on the **message** box that says open.
- Click on the **message** box that says **getinputlist**. This causes **jit.qt.grab** to send a list of available inputs for your chosen device out the right outlet preceded by the symbol **inputlist**. The patch breaks this list up using the **iter** object and sends it to the **ubumenu** on the right of the patch.
- Click on the rightmost **ubumenu** to see a list of inputs available to your device. Our DV camera reports that it has one input: YUV component (DV).

Our CapSure card reports composite and s-video inputs, as shown in the screen shot:



setting device inputs

- To change your input, select an item from the **ubumenu**. This causes the **message** box that says input \$1 to send the index of the selected input (which is the same as its position in the **ubumenu**) to **jit.qt.grab**, which will switch to your selection. If you don't explicitly choose an input, **jit.qt.grab** defaults to the first selection in the list (equivalent to sending an input 0 message) when it opens the component connection.
- You should now see your captured video signal (from your chosen input) in the **jit.pwindow**.
- When you're done, click the **message** box that says close to close the component connection.

Some video capture devices don't work properly in the **jit.qt.grab** object's default mode (vmode 0, or *sequence grabber* mode). If you find that **jit.qt.grab** is acting erratically, try sending it the message vmode 1 (*vdig* mode). This enables a method of video capture that is more reliable on certain capture devices.

Grabbing for quality

For webcams, the normal operating mode of **jit.qt.grab** provides speed and good quality. For other higher resolution capture devices such as analog video-capture boards or DV cameras, we'd like to be able to adjust the quality of the sequence grabber to get the most out of them.



high quality mode, with source and destination rects

- Double-click on the **p device_input** subpatcher. The Patcher window that opens contains everything you need to set your vdevice and input settings. Once you've done this, close the subpatch window.
- Click the **message** box that says open to start the component connection. Click on the **toggle** attached to the **metro** object. You should now see your captured video signal (from your chosen input) in the **jit.pwindow**.

- To enable high quality mode, we need to send the message `vmode 2` to **jit.qt.grab**. Enter the numeral 2 in the **number box** attached to the **message** box that says `vmode $1`. If your capture device supports a high quality mode, you'll probably see the image in the **jit.pwindow** change slightly.

If your device does not support high quality mode (`vmode 2`), you may see no change, or some undesirable result, like distorted image proportions or noise. Should this occur, return to sequence grabber or `vdig` mode (`vmode 0` or `vmode 1`).

- To compare the different quality modes and settings, let's zoom in on a detail of the captured signal. Click on the **toggle** attached to the **message** box that says `usesrrect $1`. Sending the message `usesrrect 1` causes **jit.qt.grab** to capture only the portion of the input signal specified by the *source rectangle* attribute, `srrect`. By default, the `srrect` attribute is set to the full frame of the input signal.
- Change the values of the **number boxes** attached to the **pak srrect 0 0 320 240** object. This will generate a message to set the `srrect` attribute. The `srrect` message arguments refer to the *left x*, *top y*, *right x* and *bottom y* coordinates of the input frame, respectively. You should see the image changing to reveal only the portion of the input frame you specify. For best results, choose two numbers that have the same 4:3 aspect ratio as the input frame.
- Let's do some comparisons. First, switch the `vmode` setting back to 0 or 1, depending on what you were using before (reminder: `vmode 1` (`vdig` mode) is only necessary and supported on specific capture devices). Compare the image quality to `vmode 2` (high quality mode). On our system, using a DV camera, the image becomes noticeably crisper, and aliased edges are tightened up.
- Click on the **ubumenu** object. The object contains a list of available quality settings for `vmode 2` (the settings have no effect in sequence grabber and `vdig` modes). Choosing an item from the **ubumenu** will cause the item's index number to be sent to **jit.qt.grab**, via the **message** box that says `codequality $1`. Select some different quality settings, and observe the changes—both in terms of image quality and frame rate (displayed on the **jit.fpsgui** object). The default setting for `codequality` is `max` (equivalent to sending a `codequality max` or `codequality 4` message to **jit.qt.grab**).

You should experiment with your hardware to determine the best settings for your system. For example, using the `vmode 2` in conjunction with the `codequality min` and `codequality low` modes actually looks *worse* than `vmode 0` on our system. While `vmode 2` offers control over the capture quality, it will not necessarily improve the image quality over the **jit.qt.grab** object's default capture mode.

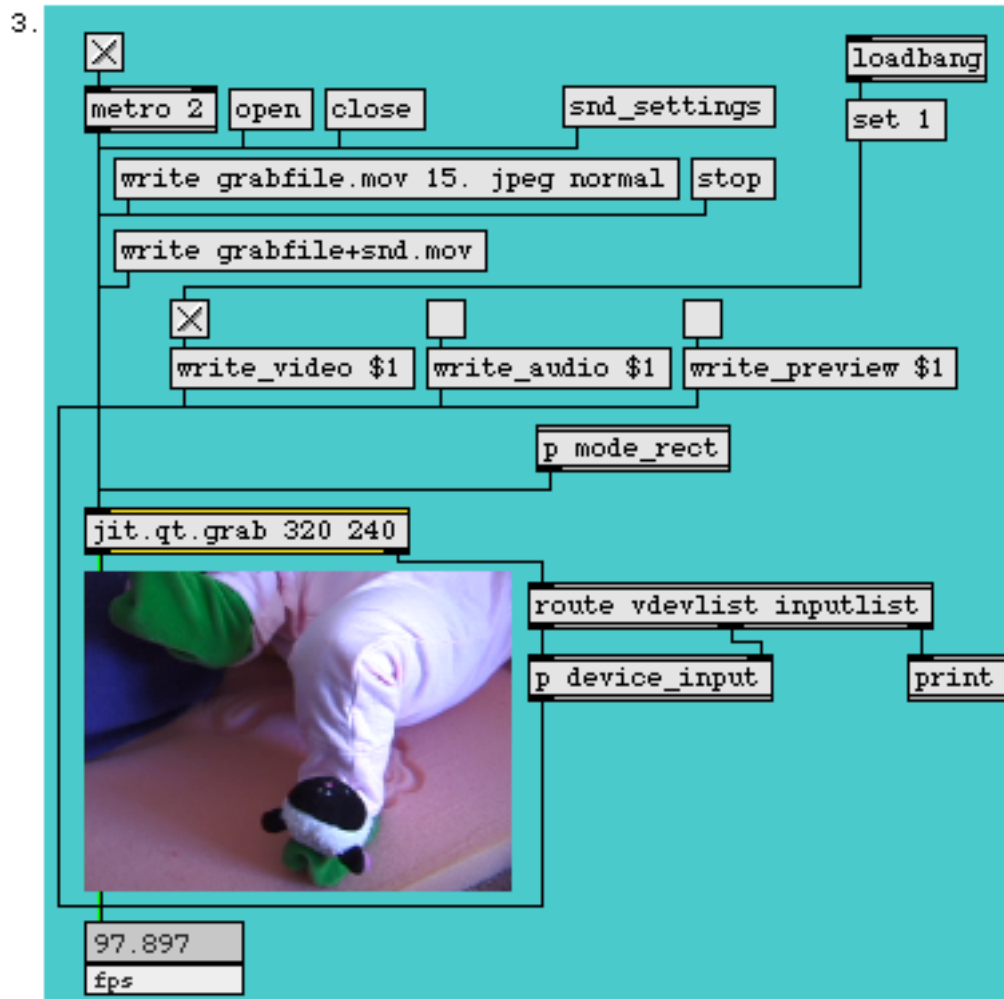
- You can also specify the portion of the output matrix the captured frame will occupy—a *destination rectangle*—with the `dstrect` attribute. To enable the destination rectangle, click on the **toggle** attached to the **message** box that says `usedstrect $1`. By default, the destination rectangle is set to the dimensions of the **jit.qt.grab** object's output matrix.
- Change the values of the **number boxes** attached to the **pak** `dstrect 0 0 320 240` object. As with the `srcrect` message, the `dstrect` message arguments refer to the *left x*, *top y*, *right x* and *bottom y* coordinates of the output matrix. You should see the image changing to occupy only the portion of the output matrix you specify. Notice that the **pak** object is connected to a **tl clear** object, which causes a `clear` message to be sent to **jit.qt.grab** before sending the `dstrect` message. You could modify the patch to see what happens if you don't send the `clear` message.
- When you're done, click the **message** box that says `close` to close the component connection.

Grabbing to disk

You can save grabbed sequences directly to disk as a QuickTime movie, if you prefer. When you do this, **jit.qt.grab** writes straight to the hard disk without sending out any matrix output. With only a few minor differences, this process is similar to the one described in *Tutorial 19*. But this time, we'll be recording audio, too.

You should quit or disable any applications or settings that interrupt processing on your machine while recording to disk, such as Internet applications or AppleTalk.

Grabbing video to disk



grabbing to disk

We've encapsulated the `vmode`, `srcrect` and `dstrect` settings we just looked at in the `p mode_rect` subpatch. You can access them by double-clicking on the object if you'd like to change them. Note, however, that the `vmode` attribute does not affect the grab to disk operation.

- Double-click on the `p device_input` patcher. The Patcher window that opens contains everything you need to set your `vdevice` and input settings. Once you've done this, close the subpatch window.
- Click the open **message** box to start the component connection. Click on the **toggle** attached to the `metro` object. You should now see the captured video signal from your chosen input in the `jit.pwindow`.

- The **jit.qt.grab** object, by default, has its `write_video` attribute set to on (1). To reflect this, we've used **loadbang** to turn on the **toggle** attached to the **message** box that says `write_video $1`. If you wanted to disable writing video and only write audio, you would send a `write_video 0` message and a `write_audio 1` message to **jit.qt.grab**. For now, leave the **toggle** connected to the **message** box that says `write_audio $1` turned off (by default, the `write_audio` attribute is set to off (0)).
- Click on the `write grabfile.mov 15. jpeg normal` **message** box. You've probably noticed that this message looks like the write message that we send to **jit.qt.record** object to start recording. Except for the lack of a timescale argument, the write message to **jit.qt.grab** is formatted identically—`grabfile.mov` is the file name, `15.` refers to the output movie's frame rate, `jpeg` specifies the Photo-JPEG codec, and `normal` specifies normal codec quality. All of the arguments are optional; if we omit the file name, a file Dialog Box will appear. The remaining settings, if omitted, will retain their previous values. If there are no previous values, the default values for the capture device will be used.

Technical detail: There is an important difference in the formatting of the write message to **jit.qt.grab** if you plan to record separate video and audio (split) files. For more information, consult the Object Reference entry and help file for the **jit.qt.grab** object.

- Recording begins immediately after you send the write message. While we are recording, the **jit.qt.grab** object doesn't output any matrices. We can tell because the **jit.fpsgui** object's display isn't changing. You can use the `write_preview` attribute to switch matrix output on and off while recording. The `write_preview` attribute is off (set to 0) by default. Click on the **toggle** attached to the **message** box that says `write_preview $1` to enable matrix output while recording. Your recordings will be smoother if you leave `write_preview` off.
- Click the **message** box that says `stop` to stop the grab to disk operation, but leave the patch open.

The **jit.qt.grab** object sends a message out its right outlet after a write operation to confirm that the movie was successfully recorded. Since we've connected that outlet to a **print** object in our patch, we can see the results by looking at the Max Window. If everything went as expected, you should see `print: write grabfile.mov 1` in the Max Window.

We can also record sound to disk using the technique just described. There are two ways to choose the sound device for **jit.qt.grab**.

1. Use the `snddevice` and `sndinput` messages to set sound device and sound device input. This method is very similar to the method we used to list the video devices and inputs,

except that we use the `getsnddevlist` and `getsndinputlist` messages instead of the `getvdevlist` and `getinputlist` messages.

2. Use the MacOS *Sound Settings* Dialog Box to adjust all of the device and input settings at once.

We're going to use the *Sound Settings* Dialog Box for this tutorial, but you might find it instructive to try making an altered copy of the example patch for the message-based method. You can use the `p device_input` subpatcher as a model.

There is a *Video Settings* Dialog Box available for changing video device and input settings, as well. You can access it by sending the settings message to **jit.qt.grab**.

- Click the **message** box that says `snd_settings`. The *Sound Settings* Dialog Box will appear. Using the pop-up menus and other controls, adjust the sound input settings to your preference. Close the Dialog Box by clicking OK.
- Click on the **toggle** attached to the **message** box that says `write_audio $1` to enable audio for the grab to disk operation.
- Click the `write grabfile+snd.mov 15. jpeg normal` **message** box. The recording will begin immediately as before, but sound will also be recorded to the movie.
- Click the **message** box that says `stop` to end the recording. Locate the movie on your hard drive, and open it in QuickTime Player (or Max, if you prefer) to verify that sound was recorded with the movie.
- Click the close **message** box to close the component connection.

If you investigate the Sound and Video Settings Dialog Boxes, you will notice that both offer numerous options for controlling the input settings of each device—brightness, saturation, gain, etc. All of these parameters may be modified directly from Max/Jitter, although it's beyond the scope of this Tutorial to describe their operation. Please refer to the Object Reference entry and help file for **jit.qt.grab** for more information.

Summary

The **jit.qt.grab** object allows you to grab images from any QuickTime-compatible video input device. The object offers both Max-based and Dialog Box interfaces for listing devices and inputs, switching between them, and controlling image quality. The `srcrect` and `dstrect` attributes can be used to crop and position a captured image within a Jitter matrix.

The **jit.qt.grab** object also provides a mechanism for recording video and sound directly to disk as a QuickTime movie.

Tutorial 22: Working With Video Output Components

This tutorial demonstrates how to use Jitter in conjunction with a QuickTime-compatible video output device—such as a DV camera—to send matrices and video sequences directly to hardware, bypassing your computer’s analog video output. We’ll explore the **jit.qt.videoout** object, and return to the **jit.qt.movie** object to explore its video output capabilities.

The patch examples in this tutorial assume that you have a DV camera attached to your computer’s FireWire port, powered on and in VTR mode. However, the techniques described will work with any device that has a QuickTime video output component.

End of the Line

You can place the **jit.qt.videoout** as the final object in a Jitter patch and send processed Jitter matrices directly to a video output device. You’ll find that this process is very similar in operation to the **jit.qt.grab** object that we used in the previous tutorial.

The **jit.qt.grab** and **jit.qt.videoout** objects both require that you create a component connection before they can do their work. In the case of the **jit.qt.videoout** object, that means a connection to a video output device. And, as in the previous tutorial, we’ll need to specify the settings we want to use by creating a listing of all our available devices and output modes and choose the settings we want before we use the **jit.qt.videoout** object. Since we’re already familiar with that procedure from working with the **jit.qt.grab** object in the last tutorial, using the **jit.qt.videoout** object will be easy.

simply passes any matrix received via its inlet directly to its outlet. Don't click the open **message** box just yet—we still have some setup work to do.

- Click the **message** box that says `getvoclist`. Sending a `getvoclist` message to the **jit.qt.videoout** object causes it to send out its right outlet a list of messages preceded by the symbol `voclist`. The items in this list refer to each available video output component — *voc*, for short. This list of available video outputs is routed through an **iter** object and from there into the **ubumenu** object on the left, from which we can easily make our selection. On our system, the list contains a single item: FireWire.

If the video output component is registered with the MacOS, it will appear in the `voclist` and can be opened *even if the hardware isn't available*. If there is no hardware available, the **jit.qt.videoout** object will report a -200 error in the Max window when you send matrices to it.

- Select the FireWire output component from the **ubumenu**. This causes the item's index number (which is the same as its position in the **ubumenu**) to be sent to the **jit.qt.videoout** object, via the `voc$1` **message**. If you don't select anything, the first item in the list will be used as the default component. This is equivalent to sending the message `voc 0` to the **jit.qt.videoout** object.
- Click on the **message box** that says `getvocmodes` to retrieve a list of available modes for the output component you chose. The resulting list, preceded by the symbol `vocmodes`, is routed through **iter** and into the righthand **ubumenu** object. Our list contains two items: Apple FireWire NTSC and Apple FireWire PAL.
- Select the output mode you'd like to use from the **ubumenu**. Your selection will depend on your FireWire hardware—our camera is NTSC, so we're using Apple FireWire NTSC mode. This causes the item's index number (which is the same as its position in the **ubumenu**), to be sent to the **jit.qt.videoout** object, via the `vocmode$1` **message**. If you don't select anything, the first item in the list will be used as the default mode. This is equivalent to sending the message `vocmode 0` to the **jit.qt.videoout** object.
- Now we're ready to start sending data to our output device. Click the open **message** box to start the component connection. After a few moments, you should see the same image on your device's display as you see in the **jit.pwindow** in the patch.

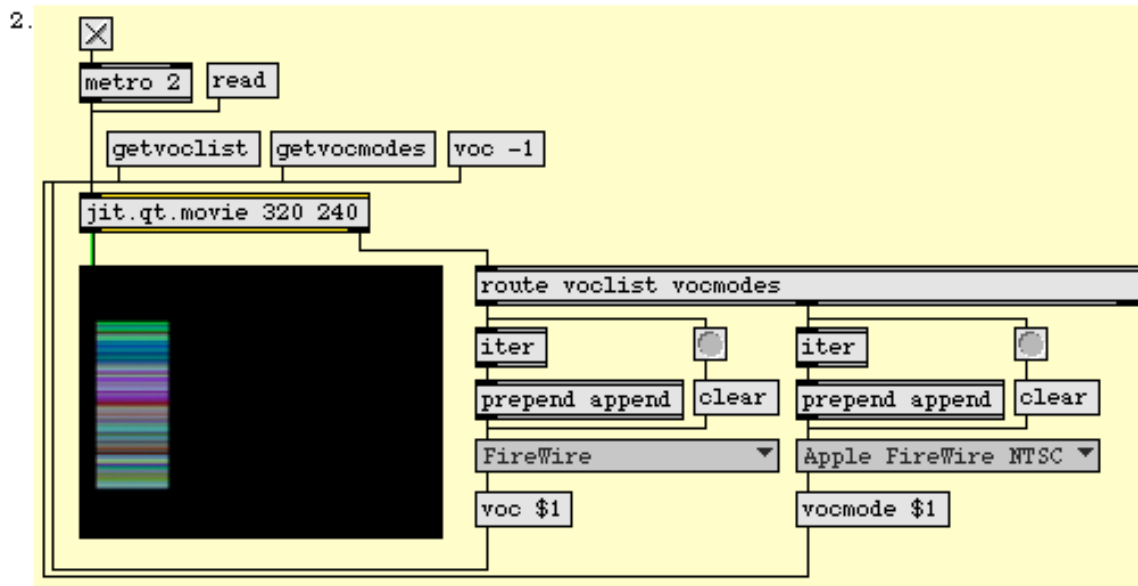
If you are getting errors in the Max window, and not seeing any video on your hardware device, check to make sure that your device hasn't gone to sleep.

- We have control over other aspects of the output, too. While the specific codec we use is hardware-specific (Apple FireWire always uses the Apple DV codec for output, for

example), we do have control over the codec quality of our output. Try selecting different values in the **ubumenu** object connected to the `codequality $1` message and compare the results on your output device. You should be able to improve speed at the expense of a small loss in image quality by using the Minimum Quality (`codequality 0`) or Low Quality (`codequality 1`) settings. The default is Normal Quality (`codequality 2`).

Just Passing Through

If you want to send a movie directly from your disk drive to an output device and you don't need to process any of your data using Jitter, you can use the **jit.qt.movie** object. You would typically want to do this only when the movie on your disk is already in the proper format for the video output component you're using—for example, if you're using a FireWire output device, the file should be already be compressed with the Apple DV codec at the standard image resolution and frame rate).



*using **jit.qt.movie** with output components*

While using **jit.qt.movie** with output components doesn't look very different from using the **jit.qt.videoout** object, there is one important difference:

The **jit.qt.movie** object doesn't use the open or close messages. You send the **jit.qt.movie** object a `voc` message with an argument of 0 or greater to open a component connection, and close the connection and return the **jit.qt.movie** to normal operation by sending the message `voc -1`. The argument to the `voc` message specifies the output component, just as with **jit.qt.videoout**.

Other than that, the two objects access video output components identically.

- Click the **message** box that says read to load a DV-encoded QuickTime movie from your disk drive. These usually have the file extension *.dv* at the end of their file name. Because these movies tend to be large, we've chosen to not include one with the tutorial package. If your computer came with Apple iMovie, there are sample DV movies in the iMovie Tutorial folder.
- Click on the **toggle** object attached to the **metro**. Your movie will be shown in the **jit.pwindow**. You may notice that the display is skipping frames; that's because DV movies contain a lot of data. Don't worry—the playback will improve when we switch to the video output component.

Technical note: Playback improves when we switch to the video output component because the movie is no longer being decompressed in software before it's displayed—it's passing directly to the hardware where it is processed (as necessary) there.

- Use the **getvocl** and **getvocmodes** **message** boxes and the **ubumenu** objects to select your preferred device and output mode as we did in the previous chapter. When you set the output device (via the **voc \$1 message** box), the movie will stop playing in the **jit.pwindow**, and should begin playing on the display of your output device. If your DV movie includes sound, the sound will play on the output device, as well. If your movie is correctly encoded for your output device, playback will be smooth.
- Click the message box that says **voc -1** to disconnect from the output component. Playback will return to normal mode, and the movie should display in the **jit.pwindow** object again.

Important: You may have noticed that the **jit.qt.movie** object that we've created has dimensions of 320x240, even though DV movies are generally 720x480. When we switch to the output component, the movie is sent to your selected output device at the movie's native size, without regard for the dimensions of the **jit.qt.movie** object's output matrix.

Summary

The **jit.qt.videoout** object provides a way to send processed Jitter data directly to a device supported by QuickTime video output components—such as a DV camera—over FireWire. If you don't require Jitter processing and would like to send a properly encoded QuickTime movie from your disk drive to a supported output device, use the **voc** message to the **jit.qt.movie** object.

Tutorial 23: Controlling Your FireWire Camera

This Tutorial describes how to use Jitter to control the transport of a FireWire-based DV camera or deck using the **jit.avc** object. This object provides a simple interface for communicating between Jitter and your DV device, and lets you control and automate your FireWire DV device using the Max interface.

AV/C (an abbreviation for Audio Video Control) is the official name for the protocol that communicates between a computer and an external device such as a DV camera. The exact specification used is called the AV/C Tape Recorder / Player Subunit. You can find a copy of the entire specification online at <http://1394ta.org/Technology/Specifications/specifications.htm>.

The specification breaks the different types of transport controls you can use into three main groups:

1. The *WIND* group is used to move the media forward and backward, without playing it. WIND group commands include stop, fast forward and rewind.
2. The *PLAY* group contains commands used to play media while moving forward or backward at various speeds. Examples include play, pause, slow forward and slow reverse.
3. The *RECORD* group is used to insert data on media.

These three groups are fully implemented in the **jit.avc** object. Additionally, the object supports a custom message, which lets you send *any* command to your hardware, even those not directly supported by the **jit.avc** object.

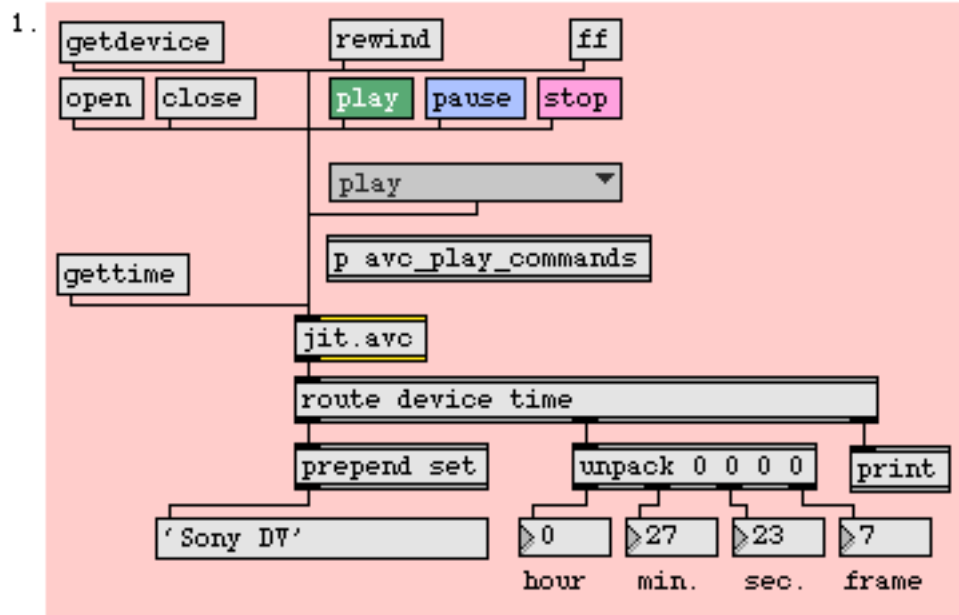
The patch examples in this Tutorial assume that you have a DV camera or deck attached to your computer's FireWire port and powered on.

Plug and Play

The **jit.avc** object's hardware connection has to be explicitly opened and closed, as is the case for the **jit.qt.grab** and **jit.qt.videoout** objects we've used in the previous tutorials. Using the **jit.avc** object is a little simpler, though—it communicates only with a single type of device.

Basics

- Open the tutorial patch *23jFWCameraControl.pat* in the Jitter Tutorial folder.



jit.avc and friends

- Click the **getdevice** message box. This causes the **jit.avc** object to look for a compatible device attached to your computer. If it finds one, the object sends the device's name from its outlet, preceded by the symbol **device**. With our Sony DV camera attached, **jit.avc** sends the message **device Sony DV**.
- In the patch, we're using the **route** and **prepend** objects to set the contents of the blank message box to the name of the located device. If **jit.avc** cannot find any compatible devices, an error message will be posted to the Max Window.

If you receive an error message, double-check your cable connections between your camera/deck and computer, and make sure that the device is powered on.

- Click the **message box** that says open. This causes **jit.avc** to open a connection to the device it found.

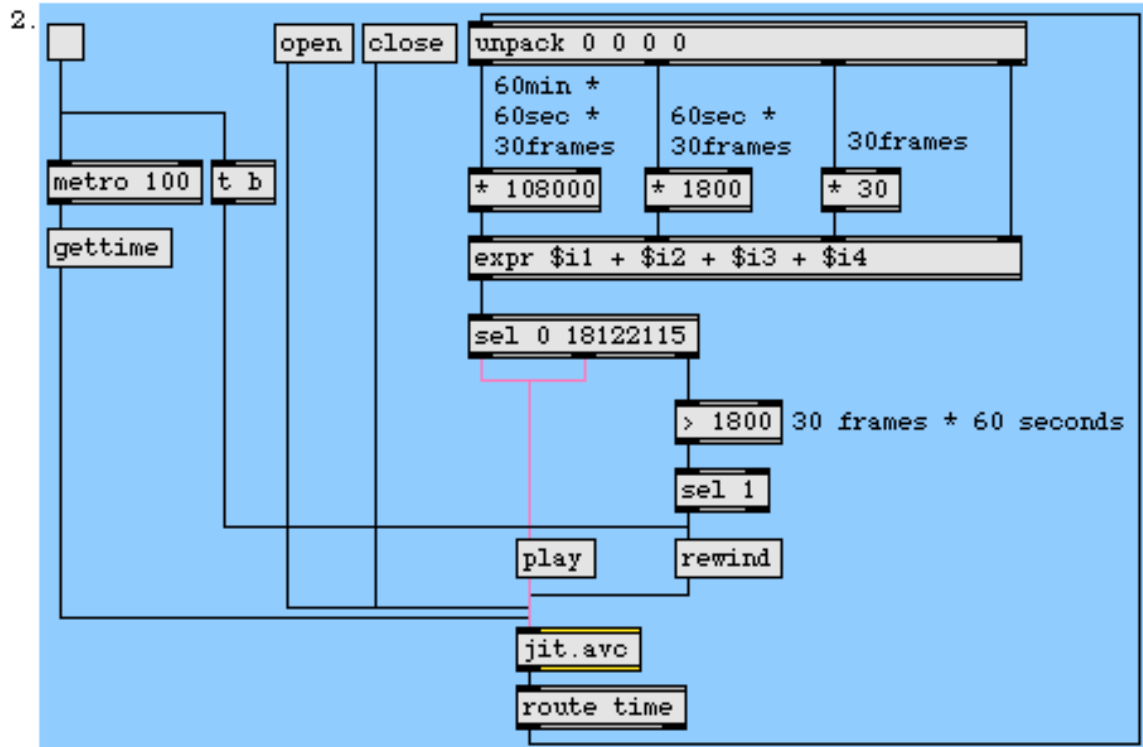
Technical note: We don't need to do any configuration prior to sending the open message to the **jit.avc** object. Unlike objects like **jit.qt.grab** and **jit.qt.videoout**, which can switch between multiple input or output devices, **jit.avc** communicates with a single “device”—the DV component—which supports a single piece of hardware (this is a MacOS limitation).

PLAY and WIND groups (VTR mode)

This portion of the tutorial assumes that you have a DV cassette (with data already recorded on it) loaded in your device, and that you are in VTR mode.

- Use the play, stop, pause, rewind and ff **message boxes** (one at a time) and observe the behavior of your FireWire device. It should be responding as if you'd pressed those buttons on the unit itself.
- Click on the **ubumenu** object to view a partial list of PLAY group messages. These messages consist of the word play followed by an additional symbol describing the specific type of PLAY group operation. Double-click on the **p avc_play_commands** subpatch to see brief descriptions of the messages we've placed in the **ubumenu**. A full list of supported messages can be found in the Object Reference entry for **jit.avc**. Use the **ubumenu** to send a few of these messages to **jit.avc** to check their effect.
- To check the current tape time, click the **gettime message box**. This causes the **jit.avc** object to send the current time from its outlet as a list of four integers preceded by the symbol time. We're using **route** and **unpack** objects to display those integers in the **number boxes** at the bottom of the patch. The numbers represent *hours*, *minutes*, *seconds* and *frames*, respectively.
- Click the **message box** that says close to turn off the connection between **jit.avc** and your DV device.

The following patch uses the `gettime` message to do something useful—it causes an attached DV device to play the first minute of a loaded tape, and then rewinds and starts over. You might use a patch like this to automate a video installation, for example.



- 197

Actually, our frame count is a little inaccurate. DV time has 29.97 frames per second, not 30. For simplicity, we've rounded up. For the purposes of this exercise, the math is close enough.

- Next, we test the frame count against the numbers 0 and 1812215 using the **select** object. Why? When a DV device can't find time code (which is the case when it's fully rewound and stopped), it reports the current time as either 0000 or 165 165 165 165, depending on the device model. Since $(0 * 10800) + (0 * 1800) + (0 * 30) + 0 = 0$, and $(165 * 10800) + (165 * 1800) + (165 * 30) + 165 = 1812215$, we can use these numbers to determine if the tape has been fully rewound. If our frame count does equal 0 or 1812215, then we send a bang to the play **message box** to start playback.

The **jit.avc** object also offers an advanced message called **gettransport** that reports the exact state of the device's transport using hexadecimal numbers, which we might have used in preference to the method above. For our purposes, though, this is fine, and a little bit cleaner. For more information on **gettransport**, refer to the Object Reference entry and help file for **jit.avc**.

- If our frame count does not equal 0 or 1812215, then we are either rewinding or playing. Now, we want to make sure that we stop playing after approximately one minute, or 1800 frames (60 seconds * 30 frames). We test the frame count against 1800 with the **> 1800** object and, if it sends out a 1, the **sel 1** object will bang the **rewind message box** and start the process all over again—that's all there is to it.
- When you're ready, turn off the **toggle**. This will both disable the **metro** and send a final rewind message to your DV device, just for neatness. Click on the **message box** that says close to close the connection between **jit.avc** and your device.

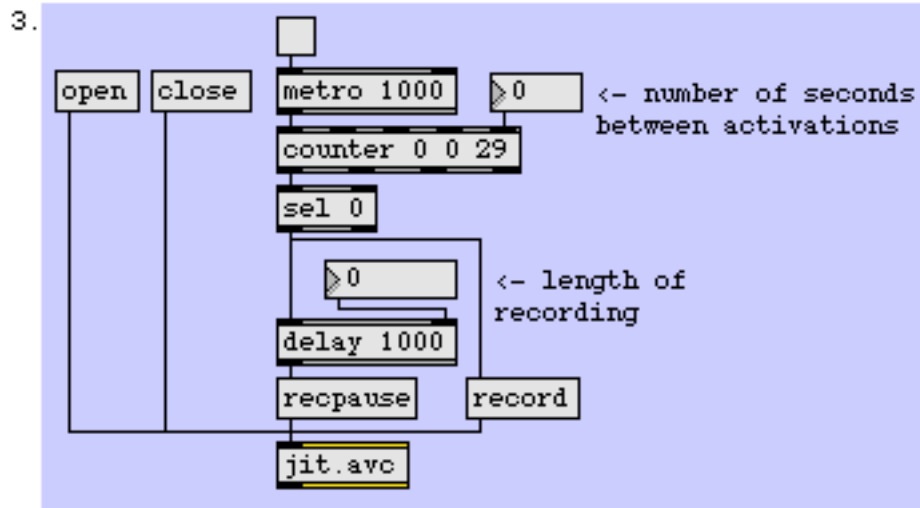
Some FireWire DV devices support an additional message, **time**, which causes the device to wind to the specified location on a loaded tape (the location is described by four integers, in the same format as noted above). DV cameras *do not* support this message, generally speaking. Please refer to the help file for **jit.avc** for an example of building an auto-wind function in Max.

RECORD group

If your DV device is a camera, it likely has two modes of operation: VTR mode and Camera mode. VTR mode is used for tape playback and non-camera (line-level) recording. Camera mode is used for recording directly from the camera portion of the unit. In VTR mode, all of the PLAY and WIND group commands are available, plus the RECORD group. In Camera mode, only the RECORD group commands are available

(since you can't rewind while recording!). You can use either mode for this portion of the tutorial.

The RECORD group provides two primary messages: record and recpause. The record message starts recording, while the recpause message places the unit into record pause mode.



*time-lapse with **jit.avc***

This simple little patch is all that's necessary to set up time-lapse recording with **jit.avc**. It records 1 second of material every 30 seconds. Let's go through it:

- Click the **message box** that says open to start the connection between **jit.avc** and your DV device.
- Before we turn it on, let's examine the patch to see what's going to happen. Every second, the **metro** object will send a bang to the **counter**. When the **counter** hits 0, the **sel 0** object will output a bang, first to the record message, and then to the **delay** object. After the specified delay (we've chosen 1000 ms.), the recpause message will be sent to **jit.avc**. We've supplied some **number boxes** so that you can change the counter and delay values.
- Make sure that your DV device is receiving input (either from its built-in camera, or from a line-level source). Then, turn on the **toggle** attached to the **metro**, and watch your DV device for a few minutes—time-lapse videography was never so easy!
- When you're ready, turn off the **toggle** and click the **message box** that says close to close the connection between **jit.avc** and your DV device.

Summary

The **jit.avc** object provides a way to control and automate your FireWire DV device, such as a DV camera or deck, using the Max interface. It offers complete control over the WIND, PLAY and RECORD group commands.

Tutorial 24: QuickTime Effects

This tutorial describes the basic operation of the **jit.qt.effect** object—Jitter’s interface to Apple’s QuickTime Effects architecture. We’ll learn how to set up and configure the object, discuss the difference and use of tweenable and non-tweenable parameters and apply a few effects to Jitter matrices. We’ll also learn how to create QuickTime Effects tracks inside of movies.

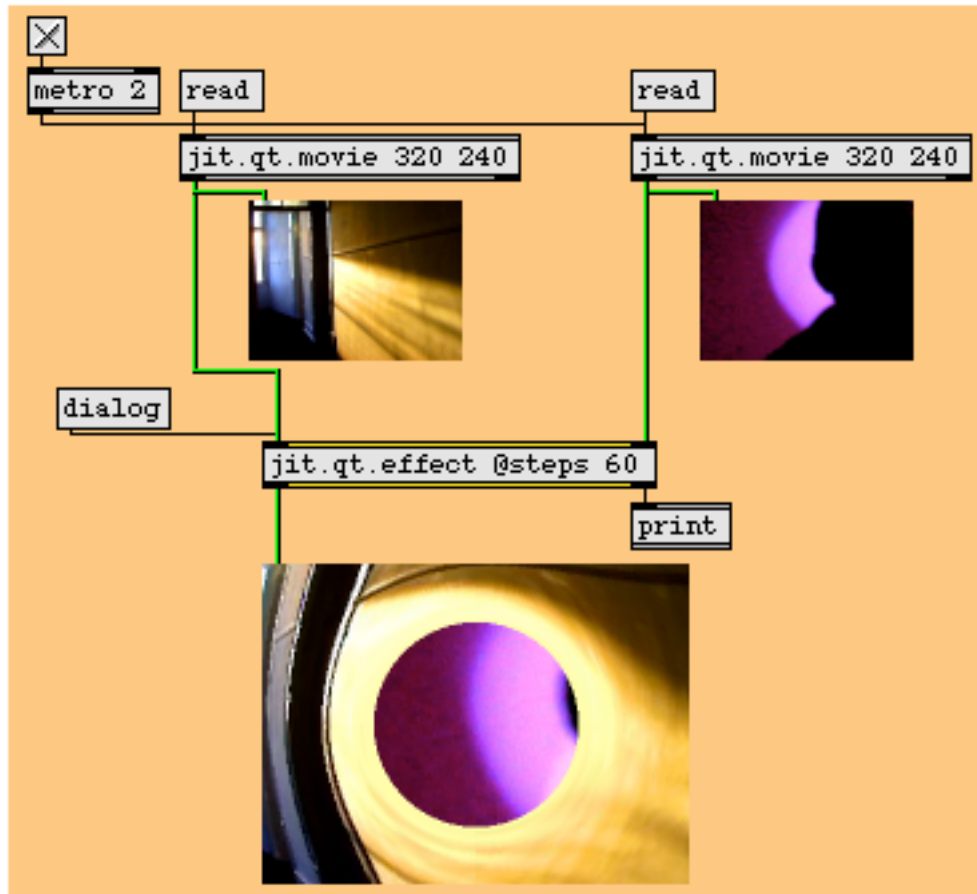
QuickTime contains a large number of built-in effects, including transitions (wipes and fades), filters and standalone (render) effects, collected under the general system known as QuickTime Effects. While QuickTime Effects are slow compared to many of Jitter’s transitions and filters, there are some unique effects available that Jitter does not include. The other difference between using QuickTime Effects and Jitter filters is that QuickTime Effects can be embedded as tracks inside a QuickTime movie, in a similar way as audio or video tracks.

The QuickTime Effects architecture uses a single interface to support a wide variety of different effects, each of which may require different numbers of parameters and parameter types—QuickTime Effects use the same interface to describe effects which don't use any input sources at all (such as the Fire or Cloud effects), effects which require a single input source (such as Blur or Gain) and effects that require two inputs (such as the Chromakey effect). So it should come as no surprise that the **jit.qt.effect** object is one of the most complex objects in the Jitter package. But don't worry—we'll be taking it one step at a time in this tutorial.

The Dialog Box Interface

The quickest and easiest way to access QuickTime Effects and to become acquainted with them is to use QuickTime’s standard Dialog Interface. The following patch will let you do just that.

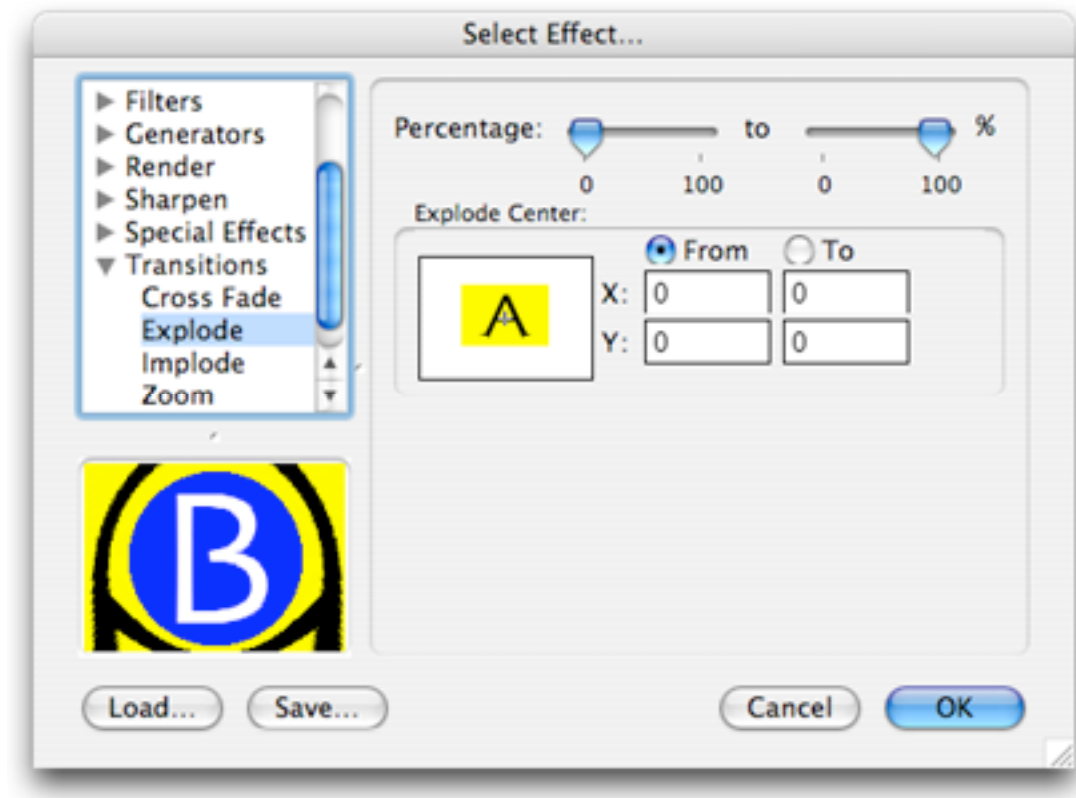
- Open the tutorial patch *24jQTEffectsDialogInterface.pat* in the Jitter Tutorial folder.



exploding QuickTime Effects

- By clicking on both of the **message** box objects that say read, load QuickTime movies into the two **jit.qt.movie** objects.
- Turn on the **toggle** attached to the **metro** object to start the patch. You will see your movies in the two smaller **jit.pwindow** objects. On the larger **jit.pwindow** object, you should see the left-hand movie. Since we haven't actually selected a QuickTime effect, the **jit.qt.effect** object is just passing the matrix received in its left inlet through its outlet.

- Click on the dialog **message** box. This causes the **jit.qt.effect** object to display the standard QuickTime Effects Dialog Box. You can select any available effect, and set any of the parameters for it. Here's an example of what the Dialog box looks like under QuickTime—we've selected the Explode effect:



the QuickTime Effects Standard Dialog Box interface

- All available QuickTime Effects are shown in the scrollable window on the left. Your list may look slightly different from this example—especially if you've added extra effects to your system. As you select different items from the list, the preview window on the bottom-left of the patch changes to show you an example of what the effect does. The adjustable parameters for the effect are shown on the right part of the window.
- Go ahead and try a few different effects, and experiment with different parameters. Choose an effect, set the parameters, and click on the OK button to see your changes in the patch. To try another effect, just click on the dialog **message** box again and choose another effect. Note that some of the effects use both matrix inputs to **jit.qt.effect**, some only use one, and some don't use the incoming matrix at all, except for timing purposes.

- You'll notice that a number of effects offer two values for each parameter. For instance, in the Explode effect's parameter panel shown above, all three parameters have two values. These are known as *tweenable* parameters, because the effect will interpolate *between* the two values entered. If you look at the **jit.qt.effect** object in our patch, you'll notice that it has a typed-in value of 60 for the steps attribute, indicating that any tweened parameters should interpolate over 60 steps (60 incoming matrices). We'll discuss tweened parameters in more detail later on in this tutorial.
- When you've had your fill of the QuickTime Effects Dialog Box, click on the **toggle** to turn off the patch.

To the Max

Every effect and option that you saw in the QuickTime Effects Dialog Box is available when using Jitter, too. Here's where things become a little bit complicated, and here's where we can help you out. While you were experimenting with the last patch, you probably noticed that each effect had a different number of parameters, each with completely different parameter types. For example, let's compare the Blur and Cloud effects. The Blur filter effect has one parameter in the form of an enumerated list of values (the menu), while the Cloud render effect has three parameters—two of which specify RGB colors, and the third of which is a floating-point number representing the degree of rotation. How will you know how many QuickTime Effects parameters you need to control, and what kind of data you need to send for each one?

We've created a patch we hope you'll use if you want to use QuickTime Effects from with Jitter. Our patch handles all the QuickTime Effects information by listing all the variables on the fly, as you make selections in Jitter. We've created an interface in the Max patch that interprets this information in a clear way that you can use and adapt for your own purposes. You should feel free to use our patch—at least until you're comfortable enough to make something similar for yourself. We promise it will save you a lot of hassle and frustration.

In the *jitter-examples* folder installed with Jitter, you'll find a whole set of QuickTime Effects helper patches, in the folder *jit.qtx.helpers*. Each of these patches isolates and documents the functionality of a single QuickTime Effect—and we've made helper patches for all of them. You can use these helper patches as though they were normal Jitter objects, and ease your entry into the rich, but complex, world of QuickTime Effects.

Listing and loading effects

To generate a list of available effects using **jit.qt.effect**, we use the `geteffectlist` message. This causes a series of messages to be sent out the object's right outlet, one message for each

available effect. Each of these messages is in the format `effectlist [index] [name] [code]`, where `index` is an index number (starting at 0), `name` is the plain English name for the effect, and `code` is a 4-letter code used to describe the effect.

- Open the tutorial patch *24jQTEffectsMaxInterface.pat* in the Jitter Tutorial folder.
- Don't worry about the **jit.qt.movie** objects for now. Let's start by looking at the other parts of the patch first.
- Click on the **message** box that says `geteffectlist`. We're using a **route** object to separate out messages beginning with the symbol `effectlist`, formatting the messages with **sprintf**, and then appending them to the **ubumenu**.
- Click on the **ubumenu** to see the list of available effects. It should contain the same items as the list that appeared on the left side of the QuickTime Effects Dialog Box.
- To load an effect for use in **jit.qt.effect**, we use the `loadeffect` message. The `loadeffect` message takes a single argument, which can be either the index number or the 4-letter code of the desired effect. In this example, we're using the effect's index number (which is the same as the item's position in the **ubumenu**) to load effects into **jit.qt.effect** via the `loadeffect $1` message.

Now that we've loaded our effect, it's time to list and specify the parameters for our effect.

Parameter types

Let's take a moment and look over the various parameter types used by QuickTime Effects. When the **jit.qt.effect** object reports available parameters, it describes the type of each parameter and any limitations placed on the values. You'll need this information to set the parameters properly.

The types are:

- *long*: (long integer) an integer value
- *fixed*: (fixed-point) floating-point value, usually between 0. and 1.0.
- *double*: (double-precision floating point) floating-point value
- *rgb*: (RGB color) 3 integers between 0 and 255, describing an RGB color
- *bool*: (Boolean) an integer value, 0 or 1

- *enum*: (enumerated list) a list of integer indices, in which each index is associated with a specific value (the indices are not necessarily consecutive)
- *text*: (text) This type is not currently supported from Max (you can use effects with text parameters from the Dialog Box)
- *imag*: (image) This type is not currently supported from Max (you can use effects with image parameters from the Dialog Box)

Listing parameters

To generate a list of available parameters for your selected effect, we use the `getparamlist` message. This causes a series of messages to be sent out the **jit.qt.effect** object's right outlet, one message for each available parameter, plus a "header message".

The header message is sent first. It is in the format `paramhead [name] [code] [params] [sources]`, where `name` is the plain English name for the effect, `code` is the 4-letter code, `params` is the number of parameters the effect has, and `sources` is the number of input sources used by the effect.

Although some QuickTime Effects report the ability to use 3 sources, the **jit.qt.effect** object only supports up to 2-source effects at present.

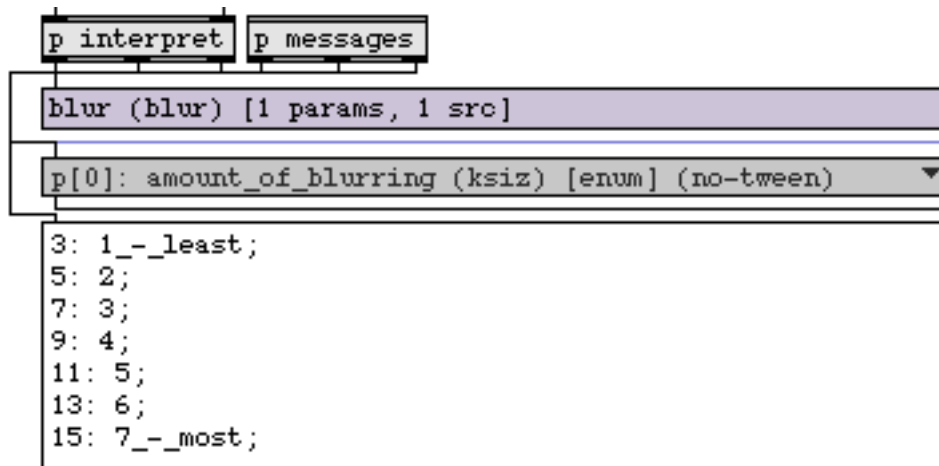
After the header message comes the series of parameter messages, in the format `paramlist [index] [name] [code] [tween] [type] [variable]`. The `index` argument is the index number of the parameter (starting at 0), `name` is the plain English name of the parameter, `code` is the 4-letter code for the parameter, `tween` is an integer describing the parameter's tweenability (0 = always tween, 1 = never tween, 2 = optionally tween), `type` is the type of parameter (from the list above), and `variable` is zero or more additional arguments which indicate permitted values, depending on the parameter type. They are:

- *long*: [min (optional int)] [max (optional int)]
- *fixed*: [min (optional float)] [max (optional float)]
- *double*: [min (optional float)] [max (optional float)]
- *rgb*: `rgb_range` (which is just a reminder that RGB values are from 0 - 255)
- *bool*: none
- *enum*: [index (int)] [value (int /symbol)] ... (these repeat in pairs until all enumerated values are described)

- *text*: [max characters (optional int)] [max lines (optional int)]
- *imag*: none

In practice

- In our patch, when you select an effect from the **ubumenu**, a bang is automatically sent to the getparamlist **message** box. The subpatcher **p interpret** contains an algorithm that automatically takes the information we've just described and formats it for readability (If you'd like to see how it works, double-click on the **patcher** object).

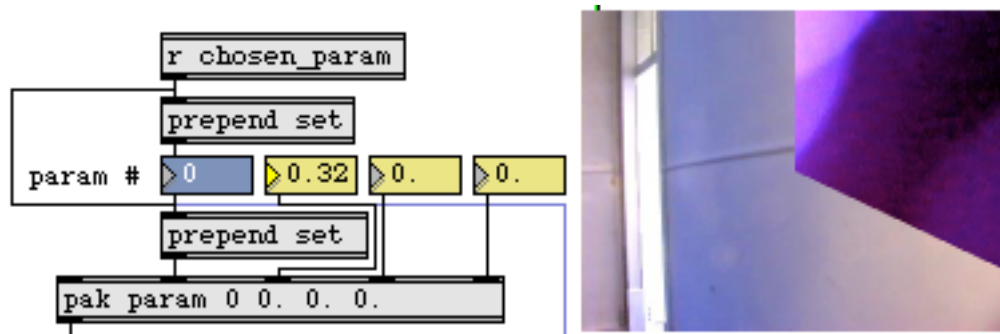


- Take a look at the portion of the patch shown above. The three outlets from the **p interpret** patcher are attached to the inlets of the three objects below—the **message** box, **ubumenu** and **textedit** object. The **message** box contains the formatted paramhead message. The **ubumenu** contains the formatted paramlist messages. The **textedit** object displays the additional parameter data. Above, you see the information reported for the Blur effect. We see that it's called 'blur', has a 4-letter code of 'blur', takes only one parameter and one source. Parameter 0, the first parameter, represents the amount of blurring, and has a 4-letter code of 'ksiz'. This parameter is an enumerated list, and cannot be tweened. In the **textedit** object, we display the correspondence between the indices and their actual values.
- Choose a few other effects from the left-hand **ubumenu**, and see how the information displays change. Choose different parameters from the right-hand **ubumenu** to get a feel for how the different parameters are described.

Making changes to parameters

- Load some movies into the two **jit.qt.movie** objects by clicking on the read **message** box objects and turn on the **toggle** attached to the **metro** to start the patch.

- In the left-hand **ubumenu**, choose the transition effect called ‘radial’ (in our **ubumenu**, it’s at index 24).
- Looking at the parameter display, we see that ‘radial’ has seven parameters and takes 2 sources. In the **jit.pwindow**, you should see your left-hand movie playing.
- The effect’s first parameter is called ‘percentage’. It’s a fixed-point parameter, and, by consulting the **textedit** object, we see that it takes a minimum value of 0, and a maximum value of 1.

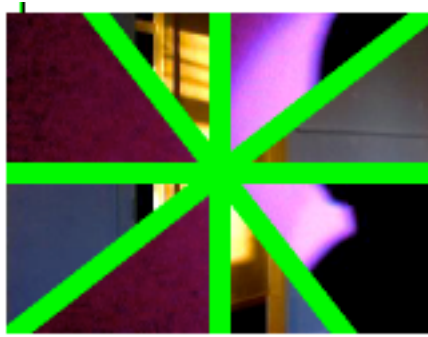


- In the example patch, find the display pictured in the screen shot above. It’s located toward the upper right of the patch. This is our parameter editor. The value in the blue **number box** represents the parameter number. We’ll use the yellow **number boxes** to set values for the chosen parameter. Each time you change one of the yellow number boxes, the **pak** object sends a param message to **jit.qt.effect** to set the parameter values.

We’ve set up the patch so that when you change the blue **number box**, the parameter information display automatically changes to display the parameter you’ve chosen. Behind the scenes, we’re using another message to **jit.qt.effect** called **getparam** to retrieve the current values of the parameter.

- For now, let’s just edit the first parameter. Make sure the blue **number box** is set to 0. Now, adjust the first yellow **number box** to values between 0 and 1. You should see the image in the **jit.pwindow** change as you do. The right-hand screen shot, above, shows an example.
- Change the parameter number to 1. From the information display, we see that parameter 1 is an enumerated list that controls the wipe type. You can scroll down in the **textedit** object to see the full list of types. Try entering different indices in the leftmost yellow **number box** to try them out. You can always switch back to parameter 0 to see the effect at different percentage values.

- Let's skip to parameters 4 and 5 (parameters 2 and 3 control horizontal and vertical repeat values—feel free to try them out). Change to parameter 4 (border width, which takes a fixed-point number between 0 and 20) and set the parameter value so that you see a border appear at the edge of the wipe. Now, change to parameter 5. This parameter sets the RGB color of the border. Using all three yellow **number boxes**, set the parameter to 0 255 0. The border should appear bright green, as pictured below.



- Finally, change to parameter 6, (soft edges). Parameter 6 is a Boolean value (the equivalent of a check box in the Dialog Box). If you set this parameter to 1 (on), the border will be drawn anti-aliased.
- You've now successfully negotiated **jit.qt.effect** and worked with every major parameter type.

Tweening

Like any other Jitter object, we use messages to control **jit.qt.effect** parameters using messages. Additionally, we can take advantage of QuickTime Effect's built-in support automatic interpolation between parameter values, or tweening. As we saw in the first part of this tutorial, using tweened parameters is fairly straightforward. Now, we're going to control them from Max.

You'll recall that tweened parameters change from an initial value to a target value over a specified number of steps. Setting these values from a patch differs only slightly from the method we just used.

Instead of setting parameters using the param message, we use a pair of messages, param_a and param_b, which represent the initial and target values, respectively.

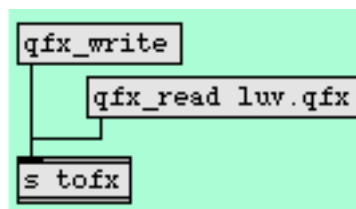


- Locate the portion of the patch shown in the screen shot, above.
- Select the 'cloud' render effect from the left-hand **ubumenu**. The parameter information display should indicate that 'cloud' has 3 parameters and takes no sources. If you quickly browse through the parameters, you'll see that they are all marked 'tween-optional', which means that we can tween them all if we like.
- Let's set our initial values. Click on the blue **ubumenu**, and change the selection to param_a. This sets the first item in the **pak** object, so that any parameter changes are now sent to the **jitter.qt.effect** object as a param_a message.
- The 'cloud' effect's first parameter, parameter 0, sets the RGB color of the cloud. Let's go for red. Using the yellow **number box** objects, set this parameter's value to 255 0 0.
- Parameter 1 sets the RGB color of the background. Anyone for black? Set this parameter to 0 0 0.
- Parameter 2 sets the cloud's rotation. We'll set up a full rotation for this example. Change this parameter to 0.
- Click on the blue **ubumenu** again, and select param_b. We'll set target values now.
- Set the cloud color to blue (0 0 255), the background color to yellow (255 255 0) and the rotation to 360.
- To actually see our tweened parameters, change the number box connected to the **message** box that says steps \$1. Look at that sky change!

Saving and Loading Parameter Files

QuickTime Effects supports a special file type that we use to save and retrieve effect parameters, called .qfx. We can use .qfx files to back up parameter configurations that we'd like to recall later, or to move a parameter set from one patch to another (or from one application to another—.qfx files use a standard file type that any program that utilizes QuickTime Effects should know how to read). In Jitter, we also use .qfx files to create QuickTime Effects tracks in the **jit.qt.movie** object.

- Locate the **message** boxes shown in the screen shot below in the lower right of the patch. Since we're confident that you'll want to hold on to the settings we just entered for the 'cloud' effect, let's save them to disk.



- Click on the **qfx_write** message box. A File Dialog box will appear. Enter a filename and click on the Save button to write your parameter file to disk. The **qfx_write** message also accepts an optional filename as an argument, if you'd like to bypass the File Dialog box.
- We use the **qfx_read** message to retrieve a .qfx file. Click on the **message** box that says **qfx_read luv.qfx**. Make sure you have two movies running in the patch, and that your steps attribute is set to a value greater than 0. It's a little present from us to you, because we love you so much.

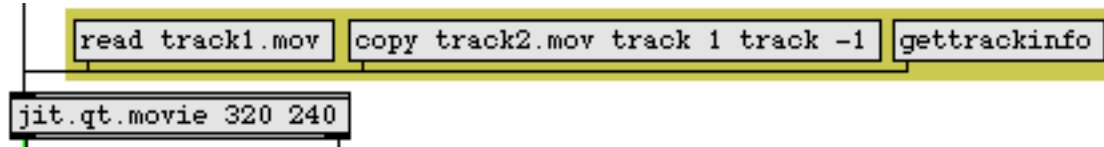
If you are using the QuickTime Effects Dialog Box interface to set your effect parameters, you can access the functionality of the **qfx_read** and **qfx_write** messages with the Load and Save buttons at the bottom left of the Dialog Box.

Using QuickTime Effects in QuickTime Movies

QuickTime Effects can be used as special tracks inside of movies. A QuickTime Effects track uses existing video tracks as sources (if applicable) and renders the effect in real time, while the movie plays.

The **jit.qt.movie** object lets you add effect tracks to an existing movie by using .qfx files to describe the effect and its parameters.

- Open the tutorial patch *24jQTEffectMovieTracks.pat* in the Jitter Tutorial folder.



- Turn on the **toggle** attached to the **metro** object to start the patch.
- Click on the read track1.mov **message** box to read in an example movie.
- To determine the number of tracks this movie contains, click the **message** box that says gettrackinfo. This causes one or more messages to be sent out the **jit.qt.movie** object's right outlet, beginning with the symbol trackinfo, followed by several arguments. The format of the message is trackinfo [track index] [track name] [track type] [track enabled] [track layer].

Let's take a moment to look at this message. It reads trackinfo 1 'unnamed video' video 1 0. Since there is only one message, we know there is one track in track1.mov, at index 1. It is an unnamed video track. It's enabled, and at layer 0.

QuickTime layers are numbered from -32768 to 32767. Tracks at lower layers are displayed in front of tracks at higher layers. If two tracks are at the same layer, the track with the higher track index is displayed in front. When you create a new track, QuickTime assigns it a layer of 0.

- Click on the **message** box that says copy track2.mov track 1 track -1. This message finds the movie *track2.mov*, and copies its first track (which we happen to know is a video track) to a new track in our current movie (the -1 tells **jit.qt.movie** to create a new track). Because the new track has the same layer as the existing track (0, in this case), it is displayed in front.

If we wanted to know, without a doubt, what the track layout of *track2.mov* is, we could send the message gettrackinfo track2.mov to **jit.qt.movie**. Most track query commands—messages beginning with gettrack—accept a remote source (a file on disk, a URL or data on the clipboard) as an initial argument.

- Click on the gettrackinfo **message** box again. You should see two messages printed in the Max Window, indicating that we now have two tracks in our movie:

```
print: trackinfo 1 'unnamed video' video 1 0
print: trackinfo 2 'unnamed video' video 1 0
```


- Let's insert *luv.qfx* as an effect track in this movie. Take a look at the `addfxtrack dialog 0 0 1 2` message:

```
addfxtrack dialog 0 0 1 2 deletefxtrack
```

This message causes **jit.qt.movie** to present a File Dialog Box, in which you can choose a *.qfx* file to import as an effect track (if we replaced the reserved symbol `dialog` with the name of a file, the object would load the file specified). The message format is `addfxtrack [filename] [offset] [duration] [source track A] [source track B]`. The offset argument refers to the starting position of the effect in the movie. The duration argument specifies the length of the effect (in QuickTime time values). The source track A and source track B arguments specify the source tracks, by track index.

A duration argument of 0 causes **jit.qt.movie** to calculate a maximum duration for the effect. For a 0-source effect, this value is the length of the movie (minus any offset). For a 1-source effect, it's the length of the source track (minus any offset). For a 2-source effect, the value is the length of the shortest of the two source tracks (minus any offset).

- Click the **message** box that says `addfxtrack dialog 0 0 1 2`. Find *luv.qfx* and click the Open button in the File Dialog Box.
- You should immediately see the *luv.qfx* effect applied to the two movie tracks as the movie plays.
- Click the `gettrackinfo` **message** box one last time. You should see the following in your Max window:

```
print: trackinfo 1 'unnamed video' video 1 0
print: trackinfo 2 'unnamed video' video 1 0
print: trackinfo 3 ___effect_src2 video 1 0
print: trackinfo 4 ___effect_src1 video 1 0
print: trackinfo 5 ___effect_track video 1 -32768
```

Apparently, our movie has acquired some new tracks, hasn't it? Applying an effect track to a movie will create between one and three new tracks—one track for the effect itself (named `___effect_track`), and one track for each of the source tracks (named `___effect_src1` and `___effect_src2`).

You shouldn't change the names of these tracks— the **jit.qt.movie** object relies on these track names for the `deletefxtrack` message, as we'll see.

- We use the `deletefxtrack` message to remove effects tracks from a movie. Click on the **message** box that says `deletefxtrack`. This will remove any tracks with the reserved names discussed above.

Summary

The **jit.qt.effect** object provides an interface to Apple's QuickTime Effects architecture. The object permits control over QuickTime Effects using a standard Dialog Box, or with Max messages. The **jit.qt.effect** object permits control over tweenable parameters, and offers import and export with the .qxf file format. The **jit.qt.movie** object can use .qxf files to generate effect tracks for QuickTime movies.

Tutorial 25: Tracking the Position of a Color in a Movie

Color Tracking

There are many ways to analyze the contents of a matrix. In this tutorial chapter we demonstrate one very simple way to look at the color content of an image. We'll consider the problem of how to find a particular color (or range of colors) in an image, and then how to track that color as its position changes from one video frame to the next. This is useful for obtaining information about the movement of a particular object in a video or for tracking a physical gesture. In a more general sense, this technique is useful for finding the location of a particular numerical value (or range of values) in any matrix of data.

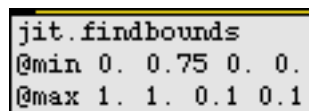
jit.findbounds

The object that we'll use to find a particular color in an image is called **jit.findbounds**. Since we're tracking color in a video, we'll be analyzing—as you might expect—a 4-plane 2-dimensional matrix of *char* data, but you can use **jit.findbounds** for matrices of any data type and any number of planes.

Here's how **jit.findbounds** works. You specify a minimum value and a maximum value you want to look for in each plane, using **jit.findbounds**'s `min` and `max` attributes. When **jit.findbounds** receives a matrix, it looks through the entire matrix for values that fall within the range you specified for each plane. It sends out the cell indices that describe the region where it found the designated values. In effect, it sends out the indices of the *bounding* region within which the values appear. In the case of a 2D matrix, the bounding region will be a rectangle, so **jit.findbounds** will send out the indices for the left-top and bottom-right cells of the region in which it found the specified values.

- Open the tutorial patch *25jColorTracking.pat* in the Jitter Tutorial folder.

In this example we use the **jit.qt.movie** object to play a movie (actually an animation) of a red ball moving around. This is obviously a simpler situation than you will find in most videos, but it gives us a clear setting in which to see how **jit.findbounds** works. Notice that we've used typed-in arguments to initialize the `min` and `max` attributes of **jit.findbounds**.

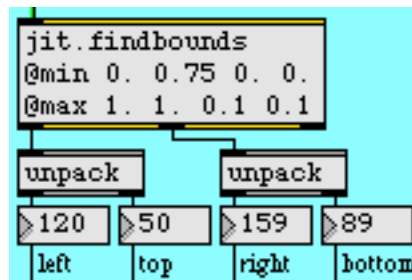


```
jit.findbounds
@min 0. 0.75 0. 0.
@max 1. 1. 0.1 0.1
```

Minimum and maximum values specified for each of the four planes

There are four arguments for these attributes—one value for each of the four planes of the matrix that **jit.findbounds** will be receiving. The **min** attribute sets the minimum acceptable value for each plane, and the **max** attribute sets the maximum acceptable value. These arguments cause **jit.findbounds** to look for any value from 0 to 1 in the alpha plane, any value from 0.75 to 1 in the red plane, and any value from 0 to 0.1 in the green and blue planes. Since the data in the matrix will be of type *char*, we must specify the values we want to look for in terms of a decimal number from 0 to 1. (See *Tutorials 5* and *6* for a discussion of how *char* values are used to represent colors.) We want to track the location of a red ball, so we ask **jit.findbounds** to look for cells that contain very high values in the red plane and very low values in the green and blue planes. (We'll accept any value in the alpha plane.)

- Click on the **toggle** to start the **metro**. As the red ball moves around, **jit.findbounds** reports the cell indices of the ball's bounding rectangle. Stop the **metro**, and examine the numbers that came out of **jit.findbounds**. You'll see something like this:



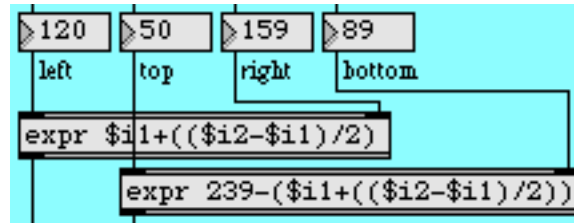
jit.findbounds reports the region where the specified color appears

The **jit.findbounds** object will report the region where it finds the desired values *in all planes of the same cell*. In this picture, the **jit.findbounds** object found the values we asked for somewhere in columns 120 through 159 and somewhere in rows 50 through 89 inclusive. This makes sense, since the red ball is exactly 40 pixels in diameter. Those cell indices describe the 40x40 square region of cells where the ball is located in this particular frame of the video.

Note that the output of **jit.findbounds** from its first two outlets is in the form of two lists. The first outlet reports the starting cell where the values were found in each dimension and the second outlet reports the ending cell of the region in each dimension. (Since it's a 2D matrix, there are only two values in each list, and we use the **unpack** objects to view them individually.)

If we wanted to know a *single point* that describes the location of the ball in the video frame, we could take the center point of that rectangular region reported by **jit.findbounds** and call that the "location" of the ball. That's what we do with the **expr** objects.

For each dimension, we take the difference between the starting cell and the ending cell, divide that in half to find the center between the two, and then add that to the starting cell index to get our single "location" point.



Calculating the center point of the rectangle

Notice that for the vertical dimension we actually subtract the vertical location coordinate from 239. That's because the cell indices go from top to bottom, but we would like to think of the "height" of the object going from bottom to top. (That's also how the **uslider** object behaves, so since we're going to display the vertical coordinate with the **uslider**, we need to express the coordinate as increasing from bottom-to-top.)

We send the results of our "location" calculation to a **hslider** and a **uslider** to demonstrate that we are successfully tracking the center of the ball, and we show the coordinates in the **number boxes**. We also scale the coordinates into the range 0 to 1, to show how easily the horizontal and vertical location of the ball could potentially be used to modify some activity or attribute elsewhere in a Max patch. For example, we could use the vertical location to control the volume of a video or an MSP sound, or we could use the horizontal coordinate to affect the rotation of an image.



Scale the location coordinates into the range 0-1, for use elsewhere in the patch.

Tracking a Color in a Complex Image

Well, that all worked quite nicely for the simple example of a plain red ball on a plain white background. But tracking a single object in a "real life" video is a good deal tougher. We'll show some of the problems you might encounter, and some tricks for dealing with them.

- Make sure the *redball* movie is stopped. Now double-click on the **patcher** *bballtracking* object to see *A More Detailed Example*. Click on the **toggle** labeled *Start/Stop* in the upper-left corner of the *[bballtracking]* subpatch to start the video.

This movie has objects with distinct colors: a red shirt, green pants, and a yellow-and-blue ball. Potentially it could be useful for color tracking. However, there are a few factors that make tracking this ball a bit harder than in the previous example.

First of all, the top few scan lines of the video (the top few rows of the matrix) contain some garbage that we really don't want to analyze. This garbage is an unfortunate artifact of the imperfect digitization of this particular video. Such imperfections are common, and can complicate the analysis process. Secondly, the image is not highly saturated with color, so the different colors are not as distinct as we might like. Thirdly, the ball actually leaves the frame entirely at the end of the four-second clip. (When **jit.findbounds** can't find any instance of the values being sought, it reports starting and ending cell indices of -1.) Fourthly, if we want to track the color yellow to find the location of the ball in the frame, we need to recognize that the ball is not all one shade of yellow. Because of the texture of the ball and the lighting, it actually shows up as a range of yellows, so we'll need to identify that range carefully to **jit.findbounds**.

Let's try to solve some of these problems. As we demonstrated in *Tutorial 14*, some Jitter objects allow us to designate a "source" rectangle of an image that we want to view that's different from the full matrix. In *Tutorial 14* we demonstrated the `srcdimstart`, `srcdimend`, and `userrrect` attributes of **jit.matrix**, and we mentioned that **jit.qt.movie** has comparable attributes called `srrect` and `userrrect`. Let's use those attributes of **jit.qt.movie** to crop the video image, getting rid of some parts we don't want to see.

- Click on the **message** box labeled *Crop Source Image*. This sends to **jit.qt.movie** the cell indices of a new source rectangle that we want to view, and tells **jit.qt.movie** to use that source rectangle instead of the full matrix. Notice that by starting at row 4 (that is, starting with the fifth row of the matrix), we crop out the garbage at the top of the image. We also chop 20 pixels off of the left side of the source image, so that the first bounce of the ball occurs exactly in the lower-left corner. Now we've focused on the part of the video we want to analyze.
- Next we'll deal with our other problems. Click on the small **preset** object labeled *Setup* in the lower-right corner of the window. This sets all the user interface objects to just the settings we desire.

This sets the `loop` attribute of **jit.qt.movie** to 2 for back-and-forth playback, and it sets a loop endpoint at time 2160 (just at the moment when the 54th frame would occur) so that the movie now plays back and forth from frame 0 to frame 53 and back. The movie now plays just up to the moment of the first bounce of the ball on the pavement, then reverses direction.

We have also sent some values to the **jit.brcosa** object (discussed in detail in *Tutorial 7*) to set its brightness, contrast, and saturation attributes just the way we want them. This doesn't exactly result in the best-looking image, but it does make the different colors more

distinctive, and compresses them into a smaller range of values, making them easier for **jit.findbounds** to track.

And we've turned on the **usesrcdim** attribute of the **jit.matrix** object (in the center of the patch) so that it is now using the output of **jit.findbounds** to determine its source rectangle. You can see the tracked region displayed in the **jit.pwindow** labeled *Show Tracked Region*.



*Using the output of **jit.findbounds** to set the **srcdimstart** and **srcdimend** attributes of **jit.matrix***

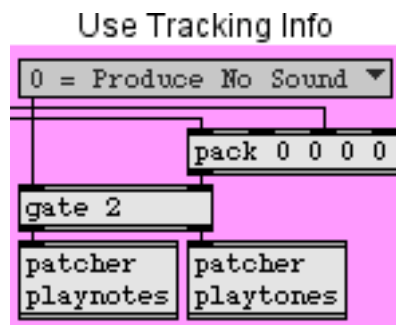
The basic yellow of the ball has nearly equal amounts of red and green in it, so we set the min and max attributes of **jit.findbounds** to look for cells containing high values in the red and green planes and a low value in the blue plane. You can see that with careful settings of **jit.brcosa** and careful settings of the min and max attributes of **jit.findbounds**, we've managed to get very reliable tracking of the yellow part of the ball.

Note: One fairly important detail that we haven't really discussed here is how to set the min and max attributes of **jit.findbounds** most effectively to track a particular color in a video. A certain amount of trial-and-error adjustment is needed, but you can get some pretty specific information about the color of a particular pixel by using the **jit.suckah** object demonstrated in *Tutorial 10*. You can place the **jit.suckah** object over the **jit.pwindow** of the video you want to analyze, click on the color you want to track, and use the output of **jit.suckah** to get the RGB information of that cell. (The values from **jit.suckah** are in the range 0-255, but you can divide them by 255.0 to bring them into the 0-1 range.)

Using the Location of an Object

So, at least in this particular situation, we've managed to overcome the difficulties of tracking a single object in a video. But now that we've accomplished that, what are we going to do with the information we've derived? We'll show a couple of ways to use object location to control sound: by playing MIDI notes or by playing MSP tones. Neither example is very sophisticated musically, but they should serve to demonstrate the basic issue of mapping location information to sound information.

We'll send the location data to two subpatches located in the part of the patch marked *Use Tracking Info*. We use a **pack** object to pack all of the output of **jit.findbounds** together into a single 4-item list, and then we use a **gate** object to route that information to the **patcher** playnotes subpatch (to play MIDI notes) or the **patcher** playtones subpatch (to play MSP tones) or neither (to produce no sound).



Send the location information to one of two subpatches

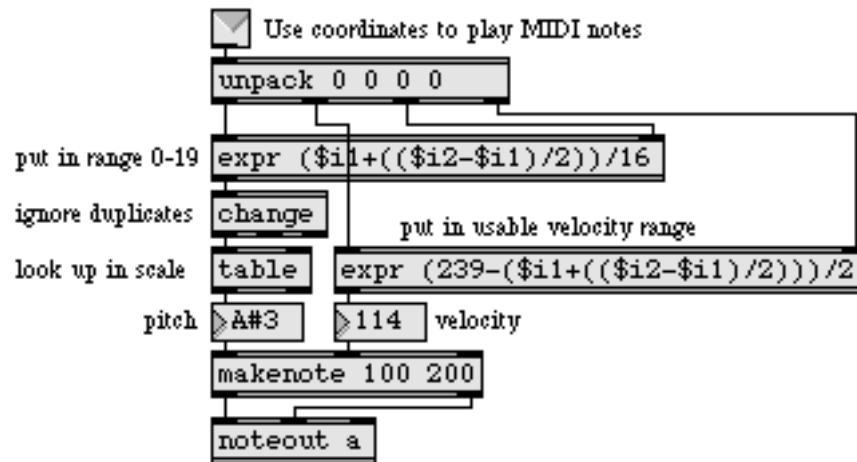
Note: In order for any of the tutorial examples involving MIDI or MSP to work, you'll need to have your equipment configured properly. The MIDI examples assume that you have OMS installed and that you have a multi-timbral synthesizer keyboard connected to Max's virtual MIDI port a. The MSP examples assume that you have MSP installed and that you have the *Driver* set to the proper output device in the DSP Status window.

For more information on how to configure your equipment, consult the "Setup" section of the *Max Getting Started* manual, the "Introduction" section of the *Max Tutorials and Topics* manual, and the "Audio I/O" section of the *MSP* manual.

Playing Notes

- In the **ubumenu** labeled *Use Tracking Info*, choose the menu item 1 = Play MIDI Notes. Double-click on the **patcher** playnotes object to see the contents of the *[playnotes]* subpatch. If you are not hearing any notes being played (and you've verified that the

movie is still playing), try double-clicking on the **noteout** a object and choosing a different MIDI synthesizer in the device dialog box.



The contents of the [playnotes] subpatch

In the *[playnotes]* subpatch we use the same sort of mapping formulae as we used in the first example to calculate the location coordinates of the ball and place the information in a usable range. We calculate the horizontal location and divide by 16 to get numbers that will potentially range from 0 to 19. We use the **change** object to ignore duplicate numbers (i.e. repeated notes), and then we look up the note we want to play in the **table**.

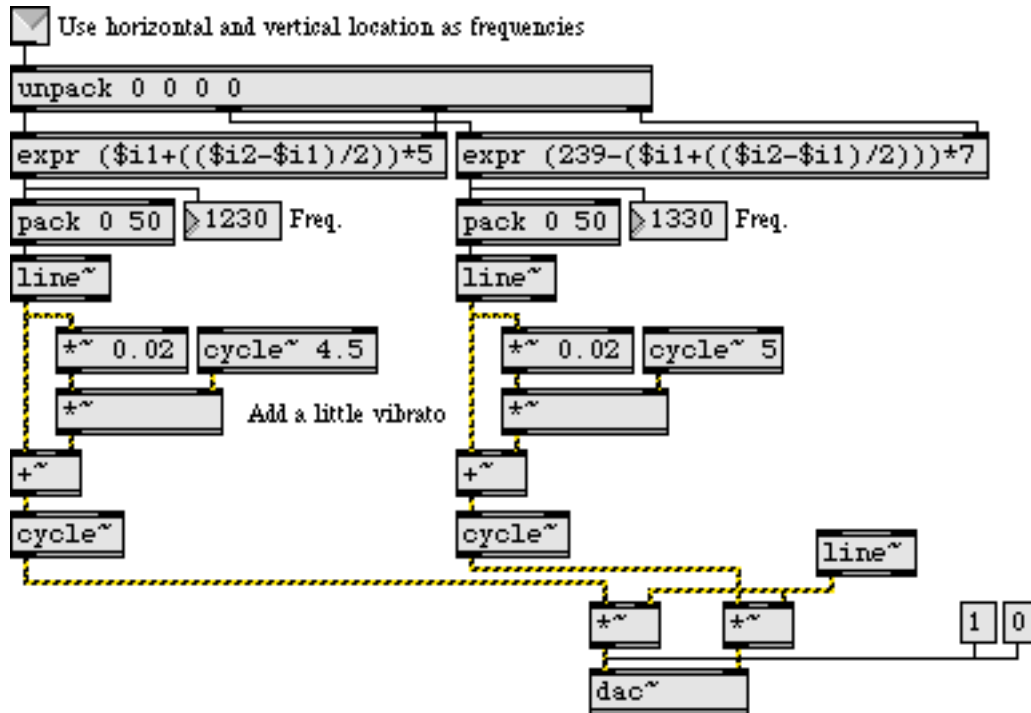
Note: The basketball player's motion has no relationship to any particular musical scale, so taking the raw location data as MIDI key numbers would result in an atonal improvisation. (Not that there's anything wrong with that!) If we want to impart a tonal implication to the pitch choices, we can use the numbers generated by the horizontal motion of the ball as index numbers to look up notes of the scale in a lookup **table**. If you want to see (or even alter) the contents of the **table**, just double-click on the **table** object to open its graphic editing window.

We use the vertical location of the ball—which we've mapped into the range 0-119—to determine the velocity values. The **makenote** object assigns the duration (200ms) to the notes and takes care of providing the MIDI note-off messages. The underlying pulse of the music (20 pulses per second) is determined by the speed of the **metro** that's playing the movie, but because the **change** object suppresses repeated notes, not every pulse gets iterated as a MIDI note.

- Close the *[playnotes]* subpatch window. Click on the **message** box labeled *Crop and Flip Source Image*. This sends a new source rectangle to **jit.qt.movie** to flip the image horizontally, which reverses the high-low musical effect of the *[playnotes]* subpatch.

Playing Tones

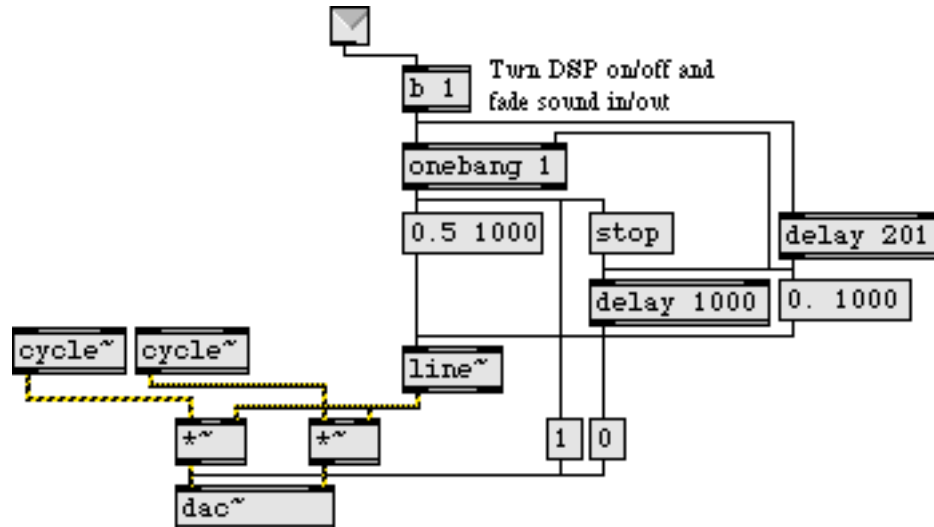
- In the **ubumenu** labeled *Use Tracking Info*, choose the menu item 2 = Play MSP Tones. Double-click on the **patcher** playtones object to see the contents of the *[playtones]* subpatch.



Use location of a color as frequency control information for MSP oscillators

Here we're using the horizontal and vertical location coordinates of the basketball as frequency values for MSP oscillators. The equations we use to calculate those values are somewhat arbitrary, but they've been devised so as to map both coordinates into similar frequency ranges. The horizontal coordinate is used to control the oscillator in the left audio channel, and the vertical coordinate controls the frequency of the oscillator in the right channel.

We use the presence of incoming messages to turn MSP audio on (and fade the sound up), and if the messages are absent for more than 200 ms, we fade the sound down and turn the audio off.



First message starts and fades in audio; lack of message for 201ms fades out and turns off audio.

- Close the *[playtones]* subpatch window. Flip the video image horizontally by clicking on the **message** boxes labeled *Crop Source Image* and *Crop and Flip Source Image* to hear the difference in the effect on the MSP oscillators.

Deriving More Information

In this tutorial we've shown a pretty straightforward implementation in which we use the location coordinates of a color region directly to control parameters of sound synthesis or MIDI performance. With a little additional Max programming, we could potentially derive further information about the motion of an object.

For example, by comparing an object's location in one video frame with its location in the preceding frame, we could use the Pythagorean theorem to calculate the distance the object traveled from one frame to the next, and thus calculate its velocity. We could also calculate the *slope* of its movement ($\Delta y / \Delta x$), and thus (with the arctangent trig function) figure out its angle of movement. By comparing one velocity value to the previous one, we can calculate acceleration, and so on. By comparing an object's apparent size from one frame to the next, we can even make some crude guesses about its movement toward or away from the camera in the "z axis" (depth).

Summary

The **jit.findbounds** object detects values within a certain range in each plane of a matrix, and it reports the region in the matrix where it finds values within the specified range of each plane. This is useful for finding the location of any range of numerical data in any type of matrix. In particular, it can be used to find the location of a particular color in a 4-plane matrix, and thus can be used to track the movement of an object in a video.

Cropping the video image with the `srcrect` attribute of **jit.qt.movie** helps to focus on the desired part of the source image. The **jit.brcosa** object is useful for adjusting the color values in the source video, making it easier to isolate and detect a specific color or range of colors.

We can use the output of **jit.findbounds** to track the location of an object, and from that we can calculate other information about the object's motion such as its velocity, direction, etc. We can use the derived information to control parameters of a MIDI performance, MSP synthesis, or other Jitter objects.

Tutorial 26: MIDI Control of Video

The MIDI–Video Relationship

When Max was first developed it was mainly for the interactive control of musical instruments via MIDI. As computer processor speeds increased, it became practical to use Max for processing audio signals directly with MSP, and to process large matrices of data such as video images with Jitter. The great power of Max is that it gives you access to all of the digital information in each of these domains —MIDI, audio, and video—and helps you program interesting correlations between them. This tutorial will focus on using incoming MIDI data to control aspects of video playback in Jitter.

The two main benefits of using MIDI for controlling video are 1) the advantage of using a physical interface—a slider, wheel, etc.—rather than a mouse dragging an onscreen control, and b) the potential to make interesting relationships between music and video in realtime performance. There are many available MIDI controllers that provide useful physical interfaces for video: banks of multiple sliders on fader boxes or digital mixers, jog wheels, etc. Even most synthesizer keyboards have buttons, sliders, wheels, and foot pedals in addition to the obvious piano-like keys. For this tutorial we'll limit ourselves to the controls that are commonly available on most MIDI keyboards.

This tutorial makes the same assumptions about your MIDI setup as are made in the *Max Tutorial* and the *MSP Tutorial*, namely: a) that you have OMS installed, have used the OMS Setup application to describe your setup to OMS, and have used Max's **Midi Setup...** command in the File menu to assign Max's *virtual ports* to devices in your OMS Setup, b) that you have a 61-key MIDI synthesizer keyboard with a modulation wheel and a pitchbend wheel, and c) that the keyboard is connected to your MIDI interface and is assigned to Max's virtual port a.

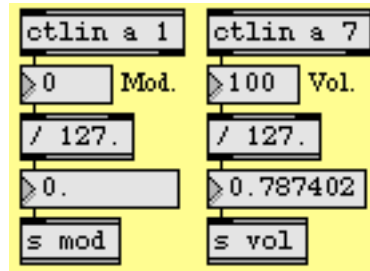
In this tutorial patch we'll play a movie, we'll use the notes from the MIDI keyboard to move around in the movie, and we'll use various types of MIDI messages to apply effects to the video and modify it in real time.

Mapping MIDI Data for Use as Video Control Parameters

The data in MIDI channel messages (notes, pitchbend, aftertouch, etc.) is in the range 0 to 127. For the most part, the attributes of Jitter objects expect arguments in the range 0 to 1—especially when handling 4-plane 2D matrices of *char* data, as with video. So one of the first tasks we need to do is map the MIDI data into the appropriate range for controlling parameters of the Jitter objects. In the tutorial patch we show some examples of how to do this.

- Open the tutorial patch *26jMIDIControl.pat* in the Jitter Tutorial folder.

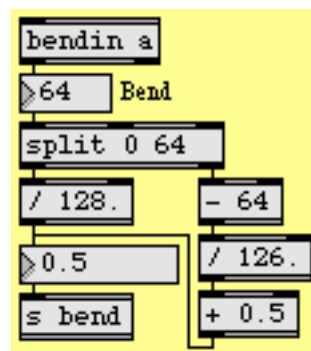
In the yellow portion of the patch we have various MIDI input objects, for gathering the data generated by a keyboard controller attached to port a: **ctlin a 1** for the modulation wheel, **ctlin a 7** for the volume pedal, **bendin a** for the pitchbend wheel, and **notein a** for the keyboard. The most straightforward controls for our use are continuous controllers like the mod wheel and the volume pedal. It's a simple matter to map their values into the 0 to 1 range, just by dividing by 127.



Map MIDI control data into a more useful 0-to-1 range

Even though both controllers have a range from 0 to 127, the mod wheel's "normal" resting position is at 0 (modulation off), while the volume pedal's usual resting position is at some non-zero position such as 100 or 127 (volume on). Thus, they might be useful to us in slightly different ways for controlling video effects.

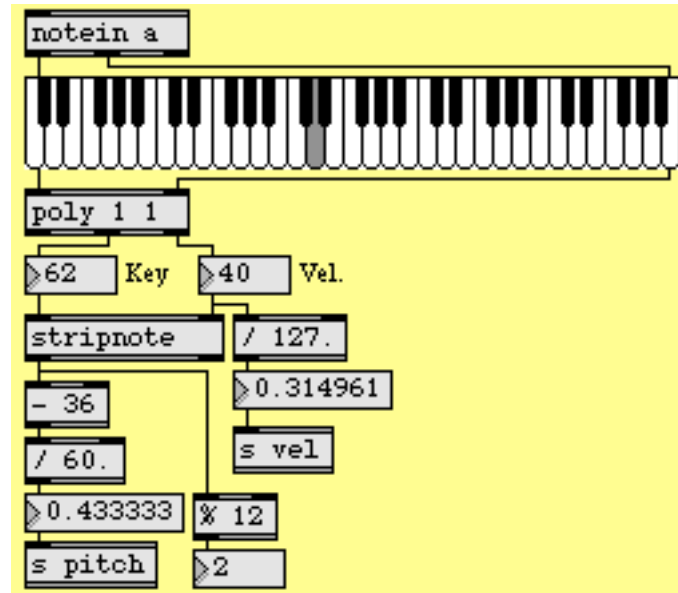
The pitchbend wheel uses still another normal position. It's resting position is at 64, and it springs back to that position when released by the user. So we want it to give us a value of 0.5 when at rest. The problem is that 64 is not *exactly* half way between 0 and 127; if we simply divide by 127, a bend value of 64 will give us a result of about 0.504. So we have to treat the "downward bend" and "upward bend" values differently, as shown in the following example.



There are 64 pitchbend values below the central value, and 63 above the center.

For the pitch information from the keyboard, the problem is a bit more complicated. First of all, most keyboards do not have keys for all pitches 0 to 127; the normal 5-octave keyboard sends out MIDI key numbers 36 to 96. But even more importantly, we're usually concerned not only with pitch "height" (where it lies in the 0 to 127 range), but

also the musical significance of the pitch *class* (C, C#, D, etc.). In our patch, we use both ways of viewing pitch. We map the key range 36 to 96 into the 0-to-1 parameter range, and we derive the pitch class with a % 12 object. (All Cs will be 0, all C#s will be 1, etc.)



Using the note-on key value to derive pitch height and pitch class

In the above example we use a couple of other handy objects to thin out the incoming note data. The **poly 1 1** object allows only one note-on message to go through at a time; it turns off the preceding note (sends it out with a velocity of 0) before passing the new note through. That's because we only want to try to track one key number at a time. If the user plays with *legato* technique (plays a note before releasing the previous one) or plays several notes in a chord "simultaneously" (i.e., nearly simultaneously; nothing is really simultaneous in MIDI) it might be hard to tell which note our patch is actually tracking. The **poly 1 1** object ensures that all notes except the most recently played one will be turned off in our patch, even if the notes are still being held down on the actual MIDI keyboard. The **stripnote** object suppresses note-off messages, so only the note-on key numbers will get through. We don't want to track the pitches as the notes are being turned off, we only want the get the key number when the note is first played.

Note: We've set up the patch so that you don't *really* need a MIDI keyboard to try it out. You can play (silent) pseudo-notes by clicking on the keys of the **kslider** object, and you can generate other values by dragging on the **number boxes** labeled *Mod.*, *Vol.*, *Bend*, *Key*, and *Vel.* Needless to say, the mouse will be a bit less gratifying than a MIDI keyboard as a physical interface, but you can at least test the patch and try out the things that are explained in this chapter.

- Try out your MIDI keyboard (and wheels) to verify that MIDI messages are getting into Max OK. If not, double-click on the MIDI input objects to select the proper input device.

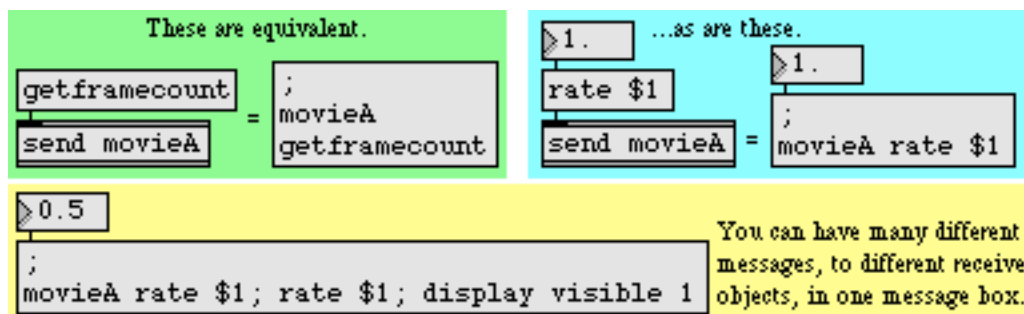
Using send and receive

The workings of this patch may be a bit difficult to follow because we make liberal use of the **send** and **receive** objects. We do this mainly to avoid what would otherwise be a ludicrous mess of patch cords. It's particularly appropriate here because we'll be sending so many messages into and out of the **jit.qt.movie** object from/to so many other places in the patch. So we use **receive** and **send** objects for the input and output of the **jit.qt.movie** object, and all other objects in the patch can now communicate with it remotely.



*We can communicate with **jit.qt.movie** no matter where it's located.*

Just in case you're not familiar with the use of a semicolon (;) in a **message** box, we'll take a moment to point out that you can put a semicolon, the name of a **receive** object, and a message in a **message** box, and when that **message** box is triggered it will function exactly as if you had sent that message into a **send** object. See the following example.



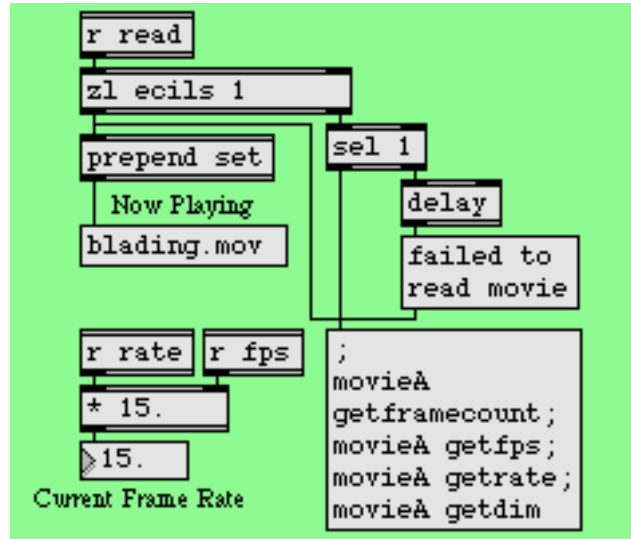
*A semicolon in a **message** box is like using a **send** object.*

So, let's trace what happens when we send a read message to **jit.qt.movie**.

- Click on the red **message** box that says ; movieA read blading.mov; movieA vol 0.

This opens the movie *blading.mov*. When **jit.qt.movie** has completed the movie-opening operation, it sends a read message out its right outlet. If it opened the file successfully, the full message will be read blading.mov 1. (If it was unsuccessful, the last argument will not be 1.) This message gets sent to the **receive** Arightoutlet object in the purple region in the

bottom-left corner of the patch. We use the **route** objects to detect all the messages we expect to get out of that outlet and route the messages to the proper places elsewhere in the patch. The arguments of the read message get sent to the **r read** object in the green region in the bottom-right corner of the patch. With the **zl ecils 1** and **sel 1** objects we check to see if the last argument of the message was a 1. If so, that means that the read was successful, so we then go ahead and get the movie's attributes.



If the movie was read in successfully, get its framecount, fps, rate, and dim attributes.

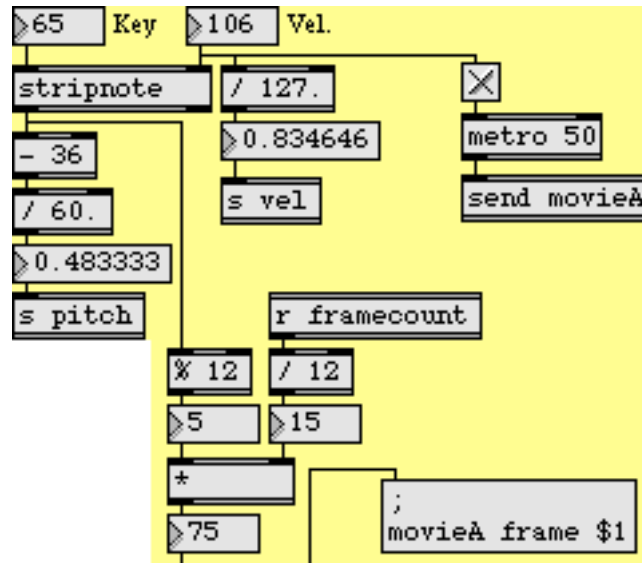
The remainder of the read message will be the name of the movie file, so we put that into a **message** box to show the user what file is now playing. If the read was unsuccessful, the **sel 1** object will trigger the message failed to read movie instead. (The **delay** object is there to ensure that the failure message gets put into the message box *after* the filename.

Using MIDI Notes to Trigger Video Clips

We have chosen to use the pitch class of the note played on the MIDI keyboard to decide where to go in the video. (There are, of course, many ways you could choose to use MIDI to navigate through a movie or select different video segments. This just happens to be the method we've picked for this tutorial. In a later tutorial chapter we demonstrate how to switch from one video to another.) So, we take the total number of frames in the movie (the framecount attribute of **jit.qt.movie**) and divide that by 12. We then use each note's pitch class ($key \% 12$) to leap to a certain twelfth of the movie.

The movie *blading.mov* is a 12-second long video consisting of twelve 1-second edits. So in this case each different pitch class takes us to a different scene of this short movie. (Of course, that's all very neat and convenient since we planned it that way.

But by using the movie's actual framecount, we've made it so our patch will successfully divide a movie of *any* length into twelfths.)



The pitch class (5) of F above middle C takes us to frame 75, 5/12 into the movie.

The note-on velocity will turn on the **toggle** that starts the **metro** that bangs the **jit.qt.movie** object, and the note-off velocity will stop the **metro**.

- Click on the **toggle** labeled *Show/Hide Display Window* make the *Display* window visible. Play some notes on your MIDI keyboard (or click on the **kslider**) to leap to different points in the movie.

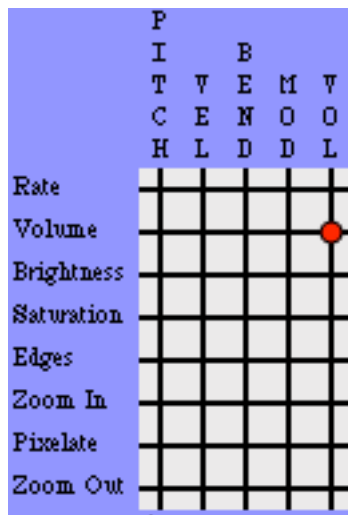
The **jit.qt.movie** object's matrices go to a **send** Aleftoutlet object, and eventually get to the **jit.pwindow** via a **receive** display object. Where do the `jit_matrix` messages go between **send** Aleftoutlet and **receive** display? They actually go into a subpatch for video effects processing. But before we examine that subpatch, we'll discuss how we intend to control those effects.

Routing Control Information

Earlier in this chapter we saw the various ways that the incoming MIDI data gets mapped into the 0-to-1 range for use in controlling Jitter attributes. If you look in the yellow region of the patch, you can see that that control information goes to five `σenvδ` objects: **s pitch**, **s vel**, **s bend**, **s mod**, and **s vol**. These are five different sources of MIDI control, and we will use them to control up to eight different video effects linked in a series. The effects link is something like this:

movie -> *rate control* -> *volume control* -> *brightness control* -> *saturation control* -> *edge-detection* -> *zoom-in control* -> *pixelation* -> *zoom-out control* -> **display window**

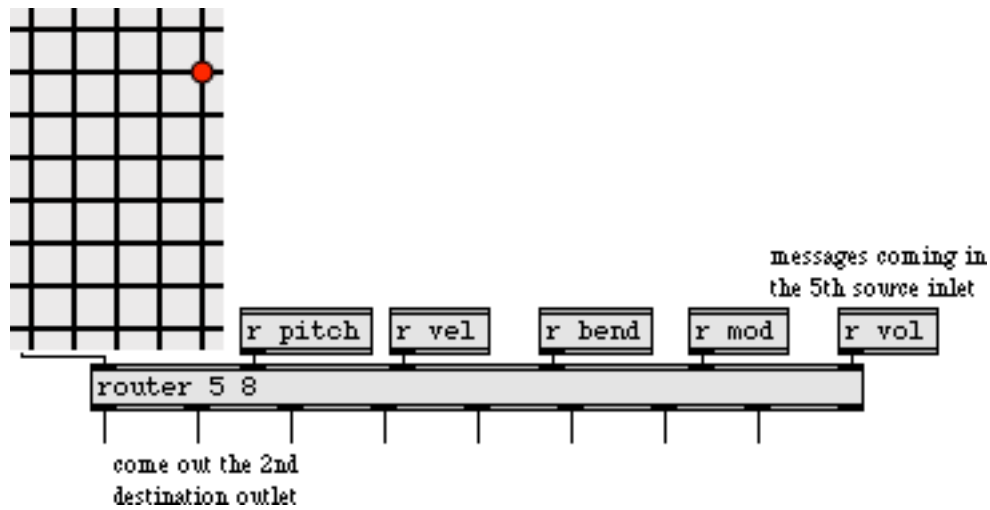
For maximum versatility, we'd like to be able to control any of those effects with any of our MIDI sources. To accomplish that, we use a combination of objects designed for just such a purpose: **matrixctrl** and **router**. The **router** object takes messages in its various inlets and routes those messages internally to any of its outlets that you specify. The **matrixctrl** object (designed for controlling the MSP **matrix~** object and the Max **router** object, not for controlling Jitter matrices *per se*) provides a user-interface for specifying those routings. Take a look at the **matrixctrl** object in the blue region of the patch.



Route fifth input (VOL) to second output (Volume)

matrixctrl shows input sources on vertical grid lines and output destinations on horizontal grid lines. So, if we want to route the messages from the fifth source inlet of a **router** object to the second destination outlet, we need to click on the grid point where those meet. In this example, we're asking to route the *vol* data to control the *volume* effect. Clicking at that point on the **matrixctrl** grid sends a message to a **router** object that tells it to make this source–destination connection internally. (Clicking again erases the red dot and breaks the connection in **router**.)

In our program the **router** object is inside the **patcher** effects subpatch, but if they were in the same patch, their connection would look like the following example.



router is the "patchbay" for Max messages, and **matrixctrl** is its user interface

Routing Around (Bypassing) Parts of the Patch

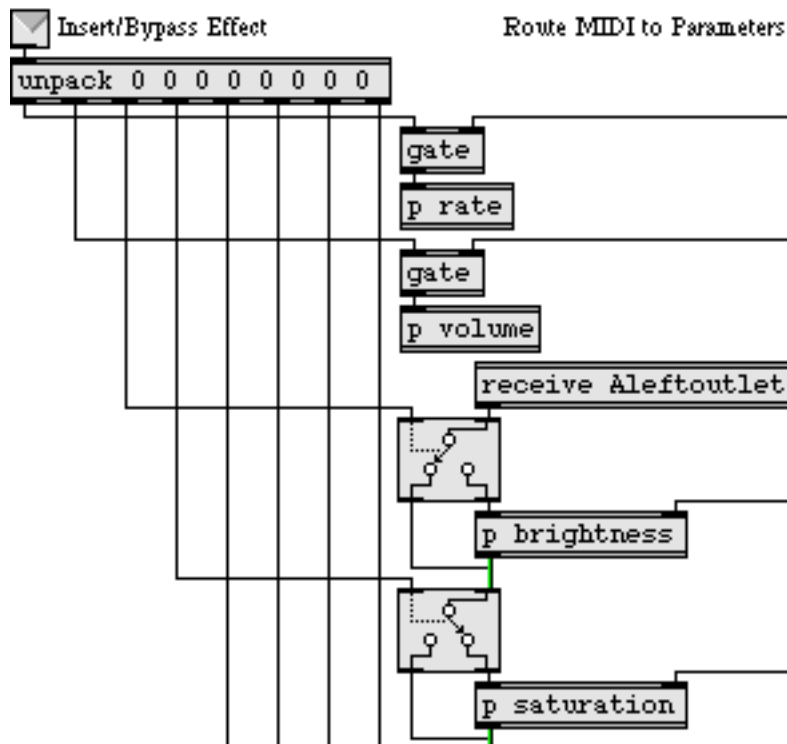
If we aren't using some of the video effects at a certain time (for example, maybe we don't want any zooming or pixelation), we'll want to bypass those particular effects. In the *effects* subpatch we'll use **Ggate** and **gate** objects to bypass some of the effects. To give the user easy control over which effects to use and which to bypass, we've set up checkboxes in the main patch, using the **radiogroup** object.

When the user clicks on one of the checkboxes, **radiogroup** sends the on/off status of all of the checkboxes out its outlet, and we can use that information to switch the routing of the **Ggates** in the subpatch.



Zoom In, Pixelate, and Zoom Out effects are completely bypassed.

- Double-click on the **patcher** effects object to see the contents of the *effects* subpatch.



*The output of **radiogroup** is used to switch **gate** and **Ggate** objects in the subpatch.*

In the subpatch, the **receive** Aleftoutlet object receives jit_matrix messages from the **jit.qt.movie** in the main patch. In the example above, the **Ggate** object routes the jit_matrix message around the **p** brightness subpatch—bypassing that effect—and the next **Ggate** object routes the message through the **p** saturation subpatch. Thus, the **Ggate** objects serve as *Insert/Bypass* switches for each effect, and the checkboxes in the **radiogroup** provide the user interface for those switches. At the end of this chain of effects, the matrix is finally passed to a **send** display object, which sends the matrix to the *Display* window.

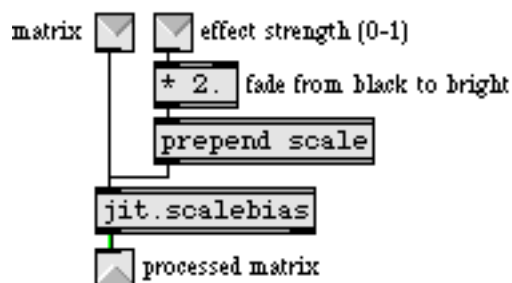
So, in the main patch we have two separate controls for the user to set up the routing of control data and effects. With the **matrixctrl** object, MIDI data from a source (or more than one source) can be routed to any (or many) of the effects. With the **radiogroup** checkboxes, the user can elect to insert effects or bypass one or more effect entirely.

- Close the *[effects]* subpatch window. Use the checkboxes to select which video effects you want to insert, and use the **matrixctrl** to assign MIDI sources to the control of those effects. Play around with different combinations to see what types of control are most intuitive (and work in the context of a keyboard performance) for each effect.

User Control of Video Effects

Each video effect in this tutorial patch is pretty simple, so we won't describe each one in detail. We'll just point out a few traits that might be instructive as you decide how you want to apply realtime user control of video effects.

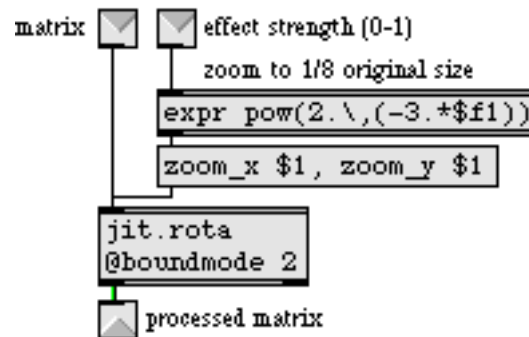
- Double-click on the **patcher** effects object once again to open the *effects* subpatch window. Double-click on the **p** brightness object to see a typical one of these video effect subpatches.



The 0 to 1 data is scaled to the range 0 to 2 for controlling the brightness.

The jit_matrix message comes in the left inlet and the control data (in the range 0 to 1) comes in the right inlet. The control data is scaled to an appropriate range, and is used to alter an attribute in a Jitter object, and the processed matrix is passed on out to the next effect in the chain. The **p** saturation and **p** zoomin subpatches work pretty similarly.

The **p** zoomout subpatch is also similar, but uses a slightly more involved mathematical expression, shown in the following example.



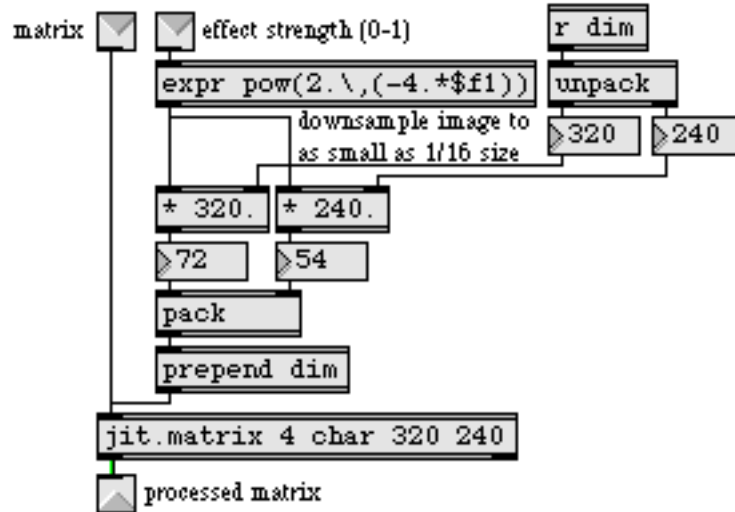
The 0 to 1 data is remapped as an exponential curve from 1 to 0.125.

In the above example the incoming control data (0 to 1) is used to calculate the exponent of a power of 2. When the control data is 0, the expression will be $2^0=1$. When the control data is 1, the expression will be $2^{-3}=0.125$. Thus, the actual control value is flipped to have reverse meaning and to describe an exponential curve instead of a linear change.

In the **p** edges subpatch, the object that's really creating the effect is a Sobel edge detection object called **jit.sobel**. What we're controlling is the *mix* between the original input image and the edge-detector's output. So we're really just controlling the *xfade* parameter of a **jit.xfade** object (described in detail in *Tutorial 8*).

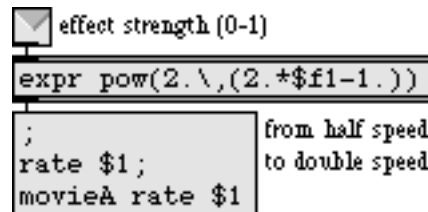
The **p** pixelate subpatch reduces the dimensions of the image matrix (a process known as *downsampling*), so that some of the data must be discarded and the image will be pixelated when it's displayed in the 320x240 *Display* window. (This method of pixelation is detailed in *Tutorial 14*.)

We got the dimensions of the original image by retrieving the `dim` attribute of `jit.qt.movie` (back when we first read in the movie), so we use our control data to scale those dimensions by some factor from 0 to 0.0625, and we use those new dimensions to set the `dim` attribute of a `jit.matrix` object, as shown in the following example.



Downsampling an image causes it to be pixelated when it's upsampled to its original dimensions.

The **p** rate and **p** volume subpatches are a bit different because we're not actually processing a matrix in those subpatches, we're just changing an attribute of `jit.qt.movie` in the main patch. The following example shows the contents of the **p** rate subpatch.



*Send the result to any **r** rate object, and also use it so set the rate attribute of `jit.qt.movie`*

Summary

The physical interface afforded by MIDI controllers gives you a good way to control video in real time in Jitter, and particularly to make correlations between music and video. Each type of controller—keyboard, pitchbend wheel, modulation wheel, volume pedal, etc.—implies a different type of control mapping. All the data of MIDI channel messages falls in the range 0 to 127, but the way you map that data to control Jitter attributes varies depending on the effect you're trying to produce. In this patch, as a starting point we mapped all the pertinent MIDI data into the range 0 to 1, then we scaled that range as necessary for each video effect.

Because Jitter objects receive so many different messages, it's often necessary to use a **message** box to construct the desired message. If you find yourself directing many different messages to the same place(s) from different parts of the patch, you might consider using the **message** box's remote message-sending capability—a semicolon plus the name of a **receive** object—to reduce the patchcord spaghetti in your patches.

If you need to send Max messages from many different sources to many different destinations, and you need the ability to reconfigure the routing of source messages to the desired destinations, the **router** object functions well as a configurable "patch bay" for Max messages. The **matrixctrl** object provides a readymade user interface for configuring the internal source–destination patching within a **router**. in this patch, we used a **matrixctrl** and **router** to allow the user to direct any of five types of MIDI control data to any of eight different video effects. We used a **radiogroup** object to create a bank of checkboxes that act as *Insert/Bypass* switches for the video effects.

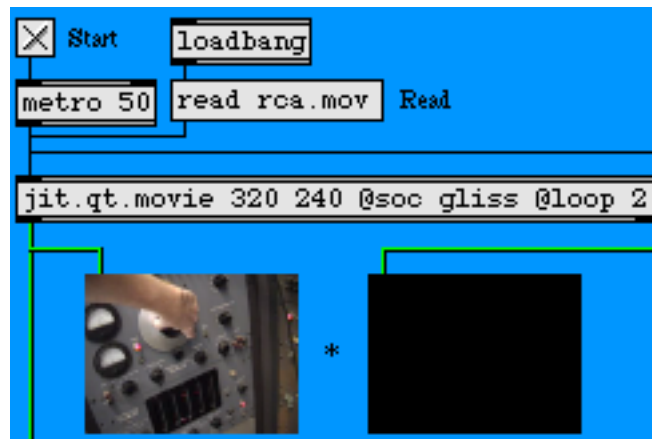
Tutorial 27: Using MSP Audio in a Jitter Matrix

This tutorial shows how to copy an MSP audio signal into a Jitter matrix using an object called **jit.poke~**. Along the way, we'll investigate how to use the soundtrack from a QuickTime movie in the MSP signal network using the sound output component attribute of the **jit.qt.movie** object and a new MSP object called **spigot~**.

This tutorial assumes familiarity with routing MSP signals using **send~** and **receive~**. It also uses a simple delay network using **tapin~/tapout~** objects. *Tutorial 4* and *Tutorial 27* in the *MSP* manual cover these topics.

- Open the tutorial patch *27jAudioIntoMatrix.pat* in the Jitter Tutorial folder.

The **jit.qt.movie** object at the top left of the tutorial patch reads a QuickTime movie called *rca.mov* upon opening.



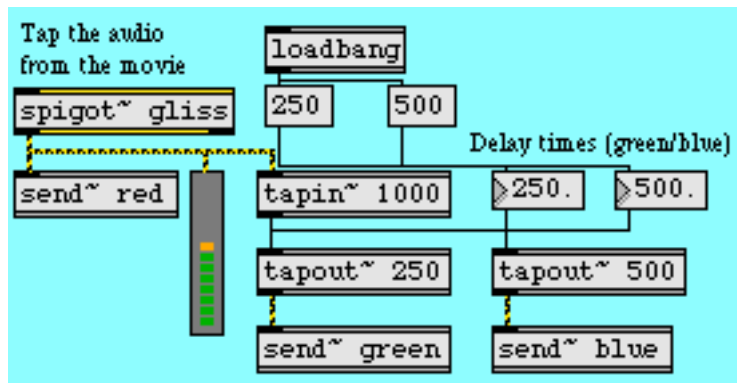
Our **jit.qt.movie** object

- Start the **metro** object at the top of the patch by clicking the **toggle** box. You will see an image in the lefthand **jit.pwindow** object below the **jit.qt.movie** object. You won't see anything in the other **jit.pwindow** objects yet, nor will you hear any sound.

Our **jit.qt.movie** object has two attributes set in its object box in addition to its dim attribute (320 by 240 cells). The loop attribute with a value of 2 tells the **jit.qt.movie** object to loop the movie as a *palindrome*. Once the playback of the movie reaches the end, it will play backwards to the beginning of the file, rather than looping around to the beginning and playing forward (the default behavior, when the loop attribute is set to 1). If you watch the movie, you'll see that the arm manipulating the oscillator control moves up and then down again in an endless loop. The movie actually only contains footage of the arm moving upward, but the loop attribute we've used reverses the playback in the second half of the loop.

The Sound Output Component

The second attribute we've set in our **jit.qt.movie** object sets the *Sound Output Component* (soc) for that instance of the **jit.qt.movie** object. The name specified as an argument to the soc attribute (in this case gliss) specifies a new sound output component that MSP can use to acquire the soundtrack of the movie loaded into the **jit.qt.movie** object. By default, the soc attribute is set to none, which routes the movie's audio output directly to the Sound Manager. A named soc attribute routes the audio to a **spigot~** object with the same name as the component, allowing you to access the audio signal in MSP:

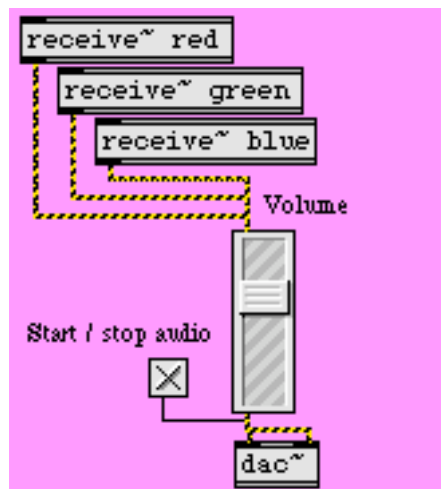


The **spigot~** object

The **spigot~** object in the upper-right hand corner of the tutorial patch has an argument (gliss) which matches the soc attribute of our **jit.qt.movie** object. If a movie file loaded into that **jit.qt.movie** object has a soundtrack (which the *rca.mov* file conveniently does), the audio from the movie is sent out as MSP signals from the **spigot~**. Note that the **spigot~** object has two outlets, which correspond to the left and right audio channels of the movie soundtrack. Our *rca.mov* file has a monaural soundtrack, so we only need to use one of the outlets in our patch.

Important: The `soc` attribute of the `jit.qt.movie` object allows you to create a separate sound output component for each `jit.qt.movie` object in your patch. You can use as many `spigot~` objects as you like, each with a unique name, to grab the audio from multiple QuickTime movies. It's important to note, however, that you can only have one `spigot~` object per sound output component, and each `jit.qt.movie` object must have a unique `soc` attribute (unless, of course, the `soc` is set to `none`—the Sound Manager can take the sound from as many movies as you wish). Once you have multiple movie audio tracks as MSP signals you can mix them as you please.

- Start the `dac~` object at the bottom of the patch by clicking the **toggle** box attached to it. You will see images appear in the remaining `jit.pwindow` objects and will see a signal level appear in the `meter~` object attached to the `spigot~`. If you turn up the `gain~` slider attached to the `dac~`, you should begin to hear sound out of whatever device you currently have selected as your MSP audio driver. For more information on how to setup your computer's audio system with MSP, consult the *Audio I/O Chapter* in the *MSP manual*.



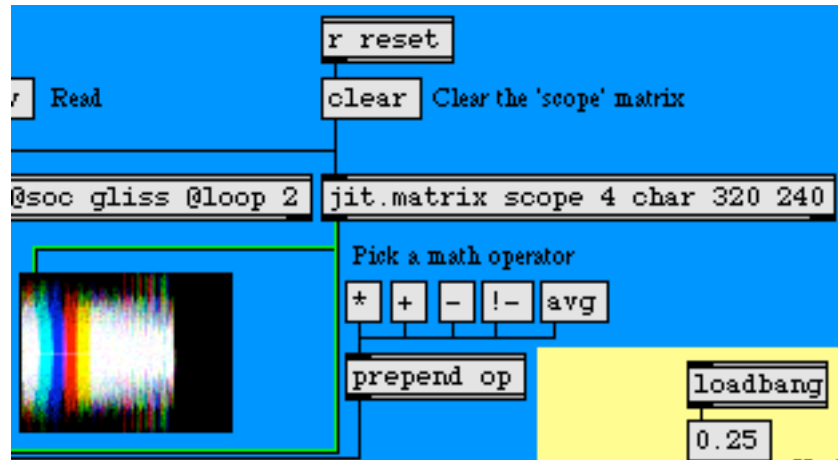
Receiving the audio signal from the `spigot~`

The soundtrack from the *rca.mov* file is sent as an MSP signal from the `spigot~` object into a two-tap delay line (generated by the `tapin~` and `tapout~` objects in the patch). The dry audio signal is sent to a `send~` object with the name `red` attached to it; the two delay taps are sent to `send~` objects named `green` and `blue`, respectively. The three audio signals are output by named `receive~` objects and summed into the `gain~` object at the bottom of the patch, allowing you to hear all of them at once.

- Adjust the delay times using the **number box** objects labeled *Delay times (green/blue)* attached to the `tapout~` objects. You can adjust the delays up to a maximum length of 1000 milliseconds (the maximum delay time allocated by our `tapin~` object).

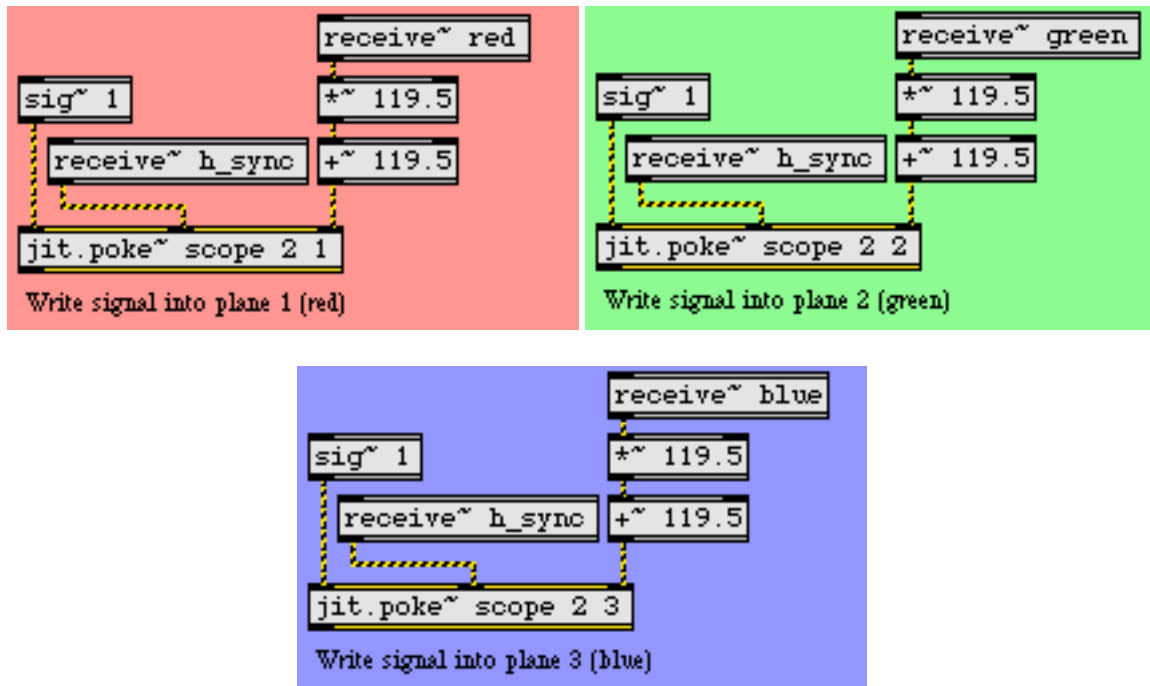
Poke~ing Around

The righthand **jit.pwindow** object at the top of the tutorial patch shows the output of a **jit.matrix** named **scope**, which also gets bang messages from the **metro** object at the top of the patch:



*The output of the scope **jit.matrix***

The scope Jitter matrix is generated by three **jit.poke~** objects at the right of the tutorial patch, which write MSP audio signals into cells in the matrix. These cells, when displayed in the **jit.pwindow** object, portray an oscilloscope view of the movie soundtrack, with the dry and two delayed signals appearing as the colors red, green, and blue, respectively.



*The three **jit.poke~** objects, writing into the scope matrix*

The three similar regions at the right of the screen use the **jit.poke~** object to write MSP signal data into our scope matrix. The **jit.poke~** object takes three arguments: the name of the Jitter matrix to write into, the number of dim inlets to use, and the plane of the destination matrix to write numbers to. All three **jit.poke~** objects in our patch write into the matrix scope. Since scope is a 2-dimensional matrix, we need 2 inlets to specify where to write the data (one inlet for the column and one inlet for the row). The three objects differ in that they each write to a different plane of the scope matrix.

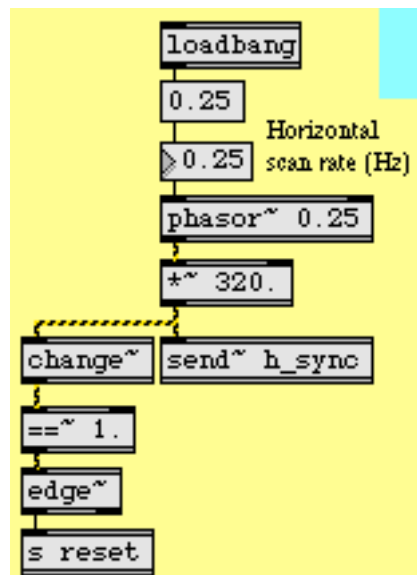
The first inlet of the **jit.poke~** object provides the value to write into the matrix cell specified by the other two inlets, which take signals to specify the cell location. We use a **sig~** object with a value of 1 to write a constant value into our current position in the scope matrix. The value of 1 gets interpreted as 255 when writing into a matrix containing *char* data (which is what we're doing in this case).

The other two inlets in our **jit.poke~** objects determine where in the output matrix they should write data (this set of coordinates defines the *write pointer* for the object—you could think of this as the location of the record head, only with two dimensions instead of one). The rightmost inlet receives the audio signal from our named **receive~** objects and

sets the vertical (dim 1) coordinate of the write pointer to correspond to the amplitude of the signal. The `*~` and `+~` objects in the patch scale the output of the audio signal from between -1 and 1 (the typical range for an audio signal) to between 0 and 239 (the range of the vertical dimension of our output matrix).

Sync or Swim

The middle inlet to our `jit.poke~` object receives a *sync* signal that specifies where along the horizontal axis of the matrix we write the current amplitude from the audio signal. This signal is unrelated to the audio data coming from the movie—you could think of it as the horizontal refresh rate of the virtual oscilloscope we've made in this patch. The sync signal is generated by a `phasor~` object in the middle of the tutorial patch:



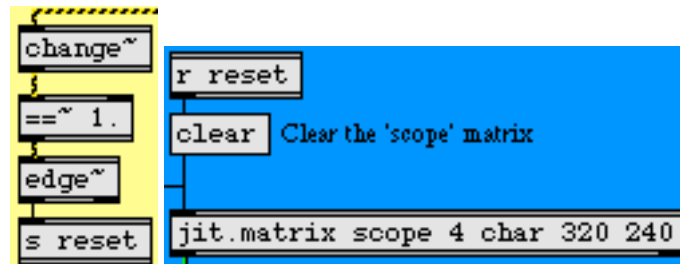
Generating the horizontal sync signal for our `jit.poke~` objects

Our `phasor~` object generates a repeating ramp signal from 0 to (nearly) 1. The `*~` below it rescales this signal to generate values appropriate to the width of our matrix (0 to 319). This signal is then passed to a `send~` object with the name `h_sync`, which forwards the signal to `receive~` objects connected to the middle inlets of our `jit.poke~` objects. The frequency of the `phasor~` (specified by the number box connected to its first inlet) determines the rate at which our `jit.poke~` objects scan from the left to right through the matrix.

- Try changing the frequency of the `phasor~` by changing the number box labeled *Horizontal scan rate (Hz)*. Notice how at higher frequencies you can see the waveform generated by the movie audio in more detail. If you set the rate to a negative value, the matrix will be written backwards (i.e. from right to left).

The dry audio signal and the two delayed outputs are visualized as the three visible planes of our scope matrix (1, 2, and 3, or red, green, and blue). When the cells written by the **jit.poke~** objects overlap, different color combinations will appear in the output matrix.

Now that we understand how the matrix is being written, we need to look into how the matrix clears itself every time a horizontal scan is completed. The relevant parts of the patch are shown below:

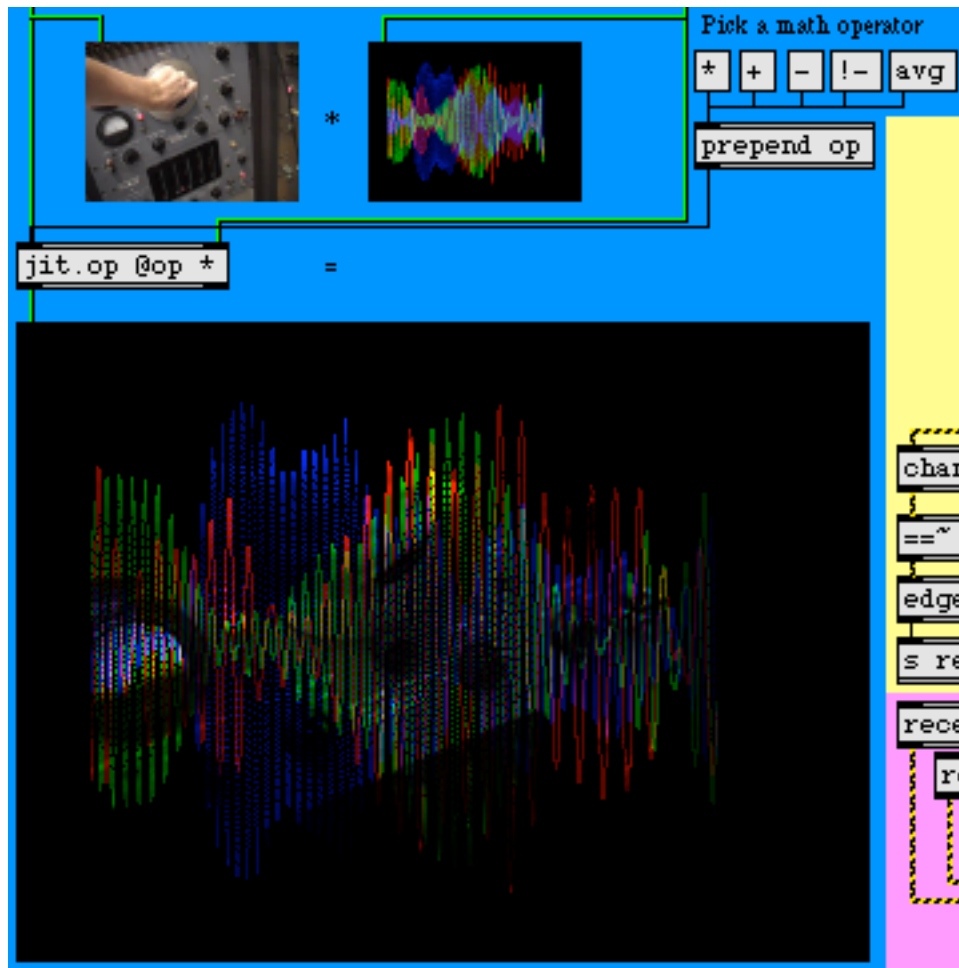


Detect when the horizontal sync resets and clear the matrix

The **change~** object outputs a value of 1 when the ramp generated by the **phasor~** object is on the increase. When the **phasor~** snaps back to 0 at the end of the ramp, **change~** will briefly output a value of -1. The **==~** operator, which outputs a 1 when the **change~** object does, will output a 0 at that point. When the **phasor~** begins to ramp again, the **==~** object will output a 1, triggering a bang from the **edge~** object (which detects a zero to non-zero transition in the last signal vector). The bang is then sent to a **receive** object named **reset**, which triggers a **clear** message to the **jit.matrix** object. As a result, our scope matrix is cleared every time the **phasor~** restarts its ramp.

Putting it all Together

Our two Jitter matrices (the image from the **jit.qt.movie** object and the oscilloscope drawn by our **jit.poke~** objects) are composited into a final matrix by the **jit.op** object:



*Compositing the two matrices using **jit.op***

The **op** attribute we've specified initially for our **jit.op** object is `*`. As a result, our composite is made from the multiplication of the two matrices. Since most of the cells in our scope matrix are 0 (black), you only see the movie image appear in those cells and planes where the **jit.poke~** objects have traced the waveform.

- Change the **op** attribute of the **jit.op** object by clicking on some of the **message** boxes attached to the **prepend** object to the right of the **jit.pwindow** showing the scope matrix. Notice how the different arithmetic operators change the compositing operation of the two matrices.

Summary

The `soc` attribute of the **jit.qt.movie** object lets you define a named *Sound Output Component*. The **spigot~** object lets you access the soundtrack of a QuickTime movie as an MSP signal by giving it an argument that matches the `soc` attribute of the **jit.qt.movie** object playing the movie.

You can use the **jit.poke~** object to write data from MSP signals into a named Jitter matrix. The **jit.poke~** object takes arguments in the form of the name of the matrix to write to, the number of inlets with which to specify cell coordinates, and the plane to write to in the matrix. The first inlet of **jit.poke~** takes the value to be written into the matrix. Subsequent inlets take MSP signals that specify the cell location in the matrix in which data should be written.

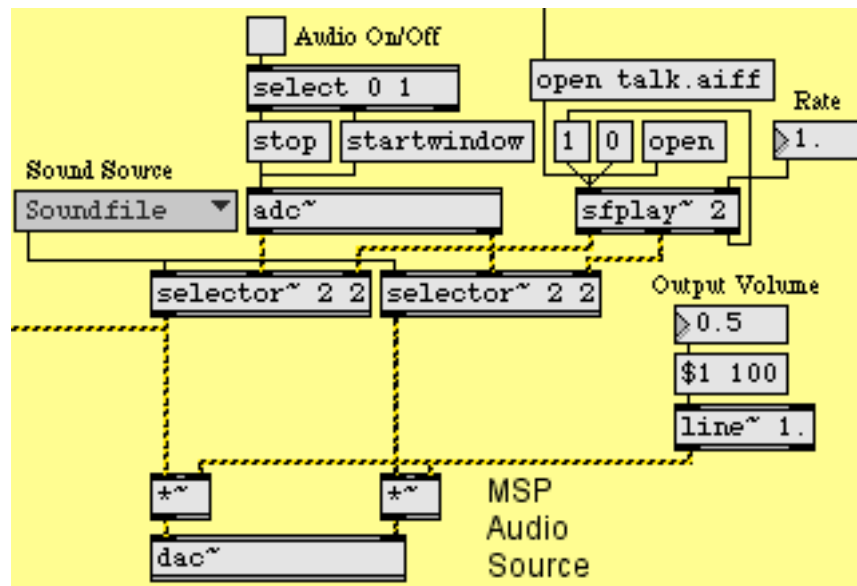
Tutorial 28: Audio Control of Video

Audio as a Control Source

This tutorial demonstrates how to track the amplitude of an MSP audio signal, how to use the tracked amplitude to detect discrete events in the sound, and how to apply that information to trigger images and control video effects.

- Open the tutorial patch *28jAudioControl.pat* in the Jitter Tutorial folder.

In the upper-right corner of the patch we've made it easy for you to try out either of two video sources: the audio input of the computer or a pre-recorded soundfile.



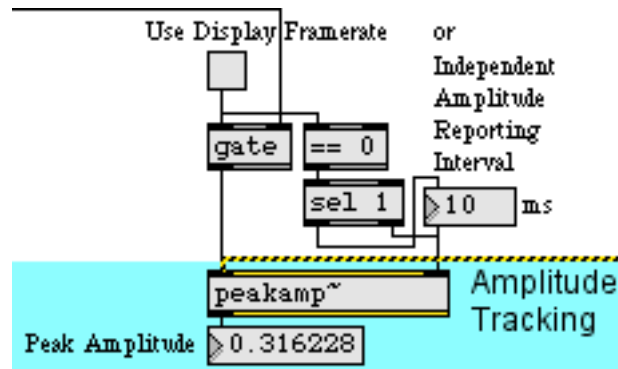
The popup menu lets you select one of two audio sources: **adc~** or **sfplay~**

We've used a **loadbang** object (in the upper-middle part of the patch) to open an AIFF soundfile *talk.aiff* and a QuickTime movie *dishes.mov*, and to initialize the settings of the user interface objects with a **preset**. So, in the above example, the **ubumenu** has already selected the **sfplay~** object as the sound source, the soundfile has already been opened by the **open talk.aiff** message, the rate of **sfplay~** has been set to 1, and the output volume has been set to 0.5. The left channel of the sound source (the left outlet of the left **selector~** object) is connected to another part of the patch, which will track the sound's amplitude.

Tracking Peak Amplitude of an Audio Signal

To track the sound's amplitude for use as control data in Max, we could use the **snapshot~** object to obtain the instantaneous amplitude of the sound, or the **avg~** object to obtain the average magnitude of the signal since the last time it was checked, or the **peakamp~**

object to obtain the peak magnitude of the signal since the last time it was checked. We've elected to track the peak amplitude of the signal with **peakamp~**. Every time it receives a bang, **peakamp~** reports the absolute value of the peak amplitude of the signal it has received in its left inlet. Alternatively, you can set it to report the peak amplitude automatically at regular intervals, by sending a non-zero time interval (in milliseconds) in its right inlet, as shown in the following example.



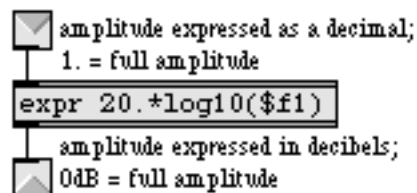
A non-zero number in the right inlet is a reporting time interval in milliseconds

Every 10 milliseconds, **peakamp~** will send out the peak signal amplitude it has received since the previous report. We've given ourselves the option of turning off **peakamp~**'s timer and using the **metro** that's controlling the video display rate to bang **peakamp~**, but the built-in timing capability of **peakamp~** allows us to set the audio tracking time independently of the video display rate.

Using Decibels

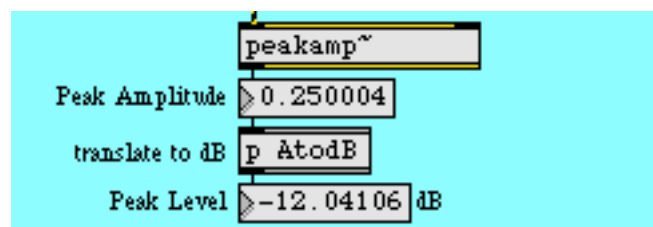
We actually perceive the intensity of a sound not so much as a linear function of its amplitude, but really more as a function of its relative level in *decibels*. This means that more than half the sound pressure level we're capable of hearing from MSP resides in the bottom 1% of its linear amplitude, in the range between 0 and 0.01! For that reason, it's often more appropriate to deal with sound levels on the logarithmic decibel scale, rather than as a straight amplitude value. So we convert the amplitude into decibels, using the **p AtodB** subpatch (which is identical to the **AtodB** subpatch used in the *MSP Tutorial 4*).

Convert a linear amplitude to amplitude in decibels. 0dB = 1. (full amplitude)



The contents of the [AtodB] subpatch

The *[AtodB]* subpatch takes the peak amplitude reported by **peakamp~** and converts it to decibels, with an amplitude of 1 being 0 dB and all lesser amplitudes having a negative decibel value .



Convert amplitude to a decibel value, relative to a reference amplitude of 1

Technical Detail: The formula for conversion of amplitude into decibels is:

$$dB = 20 \cdot \log_{10} \left(\frac{A}{A_0} \right)$$

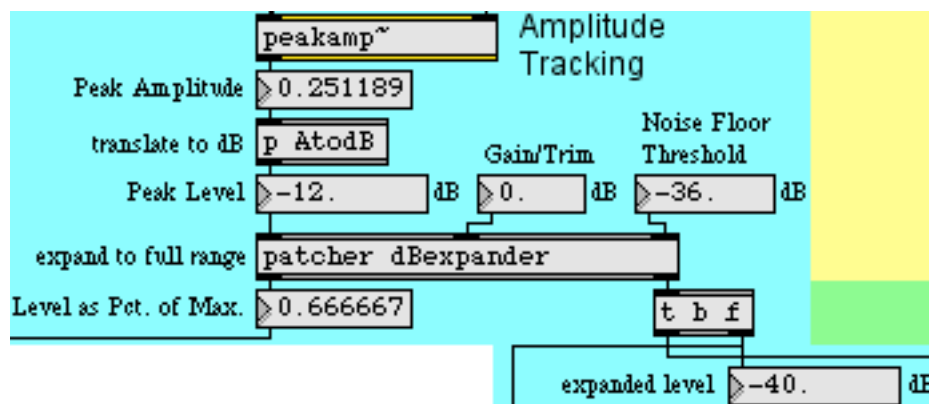
where A_0 is a reference amplitude and A is the amplitude being measured.

The *decibel* scale is discussed in the "Digital Audio" and "Tutorial 4" sections of the *MSP* manual.

Focusing on a Range of Amplitudes

In many recordings and live audio situations, there's quite a bit of low-level sound that we don't really consider to be part of what we're trying to analyze. The sound we really care about may only occupy a certain portion of the decibel range that MSP can cover. (In some recordings the music is compressed into an extremely small range to achieve a particular effect. Even in many uncompressed recordings, the most important sounds may all be in a small dynamic range.) The level of the soft unwanted sound is termed the *noise floor*. It would be nice if we could analyze only those sounds that are above the noise floor.

The **patcher** dBExpander subpatch lets us control the dB level of the tracked amplitude and set a *noise floor threshold* beneath which we want to ignore the signal. The subpatch takes the levels we *do* want to use, and expands them to fill the full range of the decibel scale from 0 dB down to -120 dB. In the following example, we have specified a noise floor threshold of -36 dB. The amplitude of the MSP signal at this moment is 0.251189, which is a level of -12 dB. The subpatch expands that level (originally -12 in the range from 0 down to -36) so that it occupies a comparable position in the range from 0 down to -120. The resulting level is -40 dB, which is sent out the right outlet of the subpatch. The level relative to the noise floor is sent out the left outlet expressed on a scale from 0 to 1, which is a useful control range in Jitter. In this example, the input level of -12 dB is 24 dB greater than the noise floor; that is, it's $\frac{2}{3}$ of the way to the maximum in the specified 36 dB range.



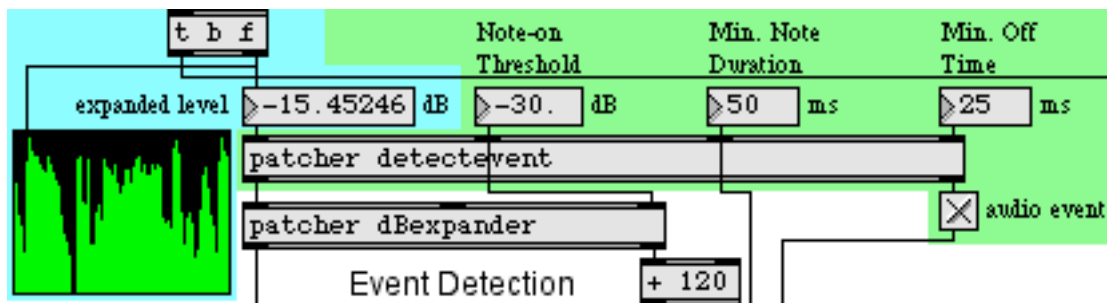
Convert linear amplitude in the region above -36dB into full range

- You can apply this value as control data for Jitter. Turn on the **toggle** labeled *Use Display Framerate*. This will temporarily turn off the internal timer of **peakamp~** and will use the bangs from the **metro** instead. Turn on the **toggle** labeled *Audio On/Off* to start MSP audio processing. Click on the **message** box containing the number 1 above the **sfplay~** object to start the playback of the sound file. Turn on the **toggle** labeled *Display Movie* to start the video playback. The peak amplitude of the audio is reported at the same rate as the movie matrix is displayed—every 25 milliseconds. The tracked

decibel level—40 values per second—is displayed in the green and black **multiSlider** labeled *expanded level*. The level, mapped into the range 0 to 1, is used to change the `val` attribute of the **jit.op** object, affecting the displayed video. You can scale the range of that value up or down with the **number box** labeled *Effect Strength*. Values in the range 0.5 to 1.5 have the most effect on the image.

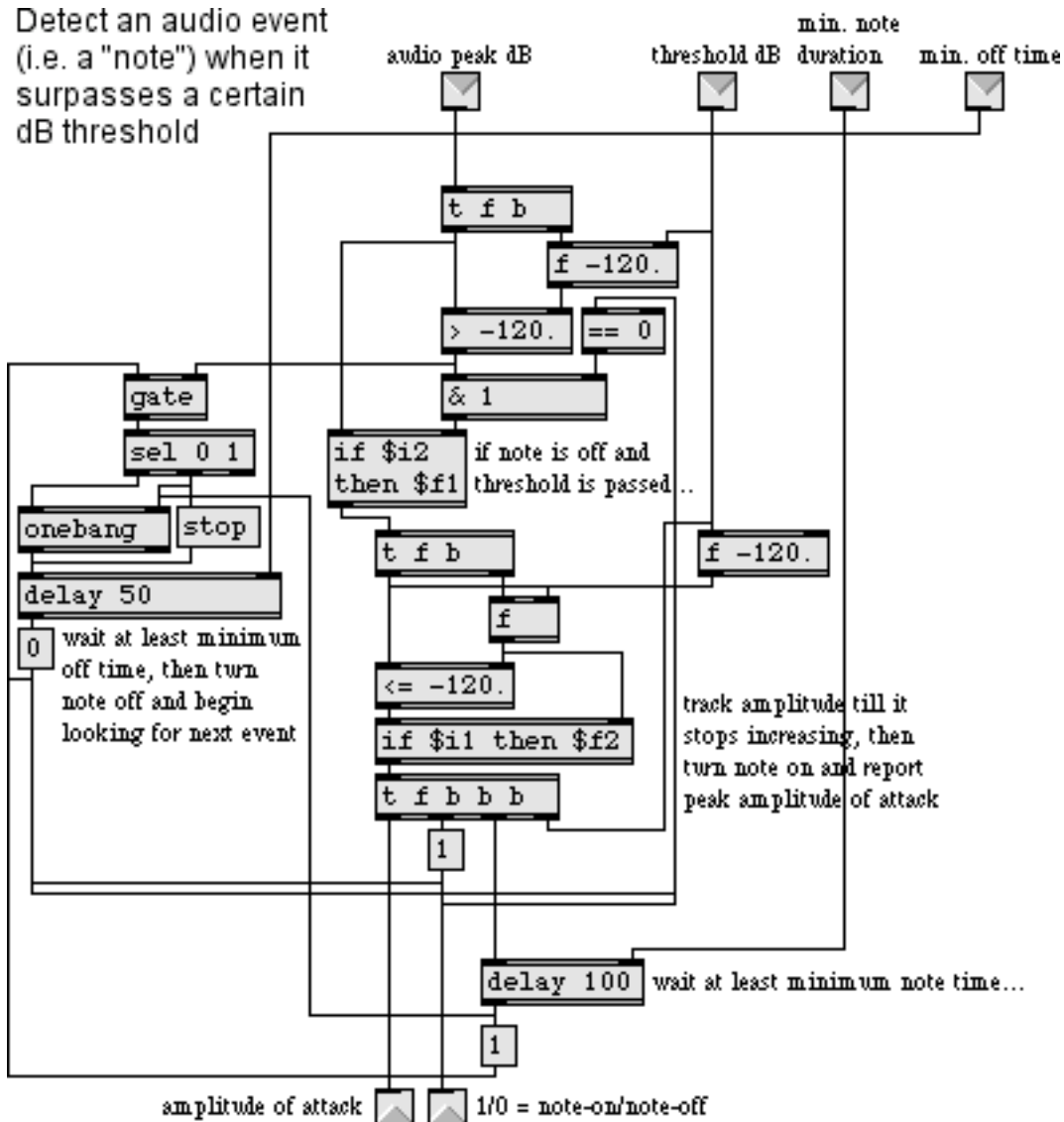
Audio Event Detection

In the preceding section we tracked the amplitude envelope of the sound and used the peak amplitude to get a new control value for every frame of the video. We can also analyze the sound on a different structural level, tracking the rhythm of individual events in the sound: notes in a piece of music, words in spoken text, etc. To do that, we'll need to detect when the amplitude increases past a particular threshold, signifying the attack of the sound, and when the sound has gone below the threshold for a sufficient time for the event to be considered over. We do this inside the **patcher detectevent** subpatch. In the main patch, we provide three parameters for the *[detectevent]* subpatch: the *Note-on Threshold* (the level above which the sound must rise to designate an "event" or "note"), the *Min. Note Duration* (a time the subpatch will wait before looking for a level that goes back below the threshold), and the *Min. Off Time* (the amount of time that the level must remain below the threshold for the note to be considered ended). In the following example a "note" event will be reported when the level exceeds –30 dB, and the note will only be considered off when the level stays below –30 dB for at least 25 milliseconds. Since the subpatch will wait at least 50 ms before it even begins looking for a note-off level, the total duration of each note will be at least 75 milliseconds.



When the level exceeds the threshold and reaches a local maximum, an audio event is reported.

- To see the contents of the subpatch, double-click on the **patcher** detectevent object.



Event-detection based on amplitude exceeding a threshold

The comments in the subpatch explain the procedure pretty succinctly. When a new level comes in the left inlet, two conditions must be satisfied: the level must be greater than the threshold and there must not already be a note on. If both those conditions are met, then we keep watching the amplitude until it stops increasing, at which point we consider the note to be fully on so we send the number 1 out the right outlet and send the peak level out the left outlet. We wait the "minimum note time", then open the **gate** to begin looking for indications (from the **>** object) that the level has gone below the threshold. Once such a level has been detected, we wait the "minimum off time" before deciding that the note is off. If another level above the threshold comes before the minimum off time has elapsed,

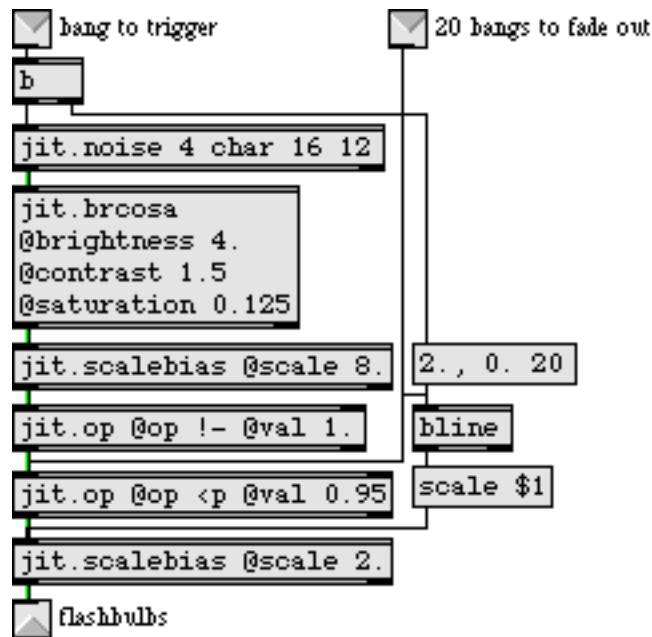
the **delay** object is stopped and a new note-off level must be detected. When the note is truly off, a 0 is sent out the right outlet, the fact that the note has been turned off is noted (in the **== 0** object), and the **gate** is closed again. It's now ready for the next time that the threshold is passed.

- Close the *[detectevent]* subpatch window. For this event-detector to work well on fast-changing sounds, the peak amplitude should usually be tracked at a fairly rapid rate. Turn off the *Use Display Rate* **toggle** so that the **peakamp~** object will use its internal timer at an interval of every 10 ms.

In the main patch you can see three demonstrations of ways to use the output of the *[detectevent]* subpatch. In the bottom right corner of the patch we use the 1 from the right outlet of **patcher detectevent** to trigger another subpatch, **patcher flashbulbs**, which places random colored dots in a display window. We take the value out of the left outlet of **patcher detectevent** and expand its range just the way we did for the original audio level, so that the value signifying the "note amplitude" can cover the full available range. We use that to trigger MIDI notes, and also to choose different pictures to display. Let's look at each of those procedures briefly.

Using Audio Event Information

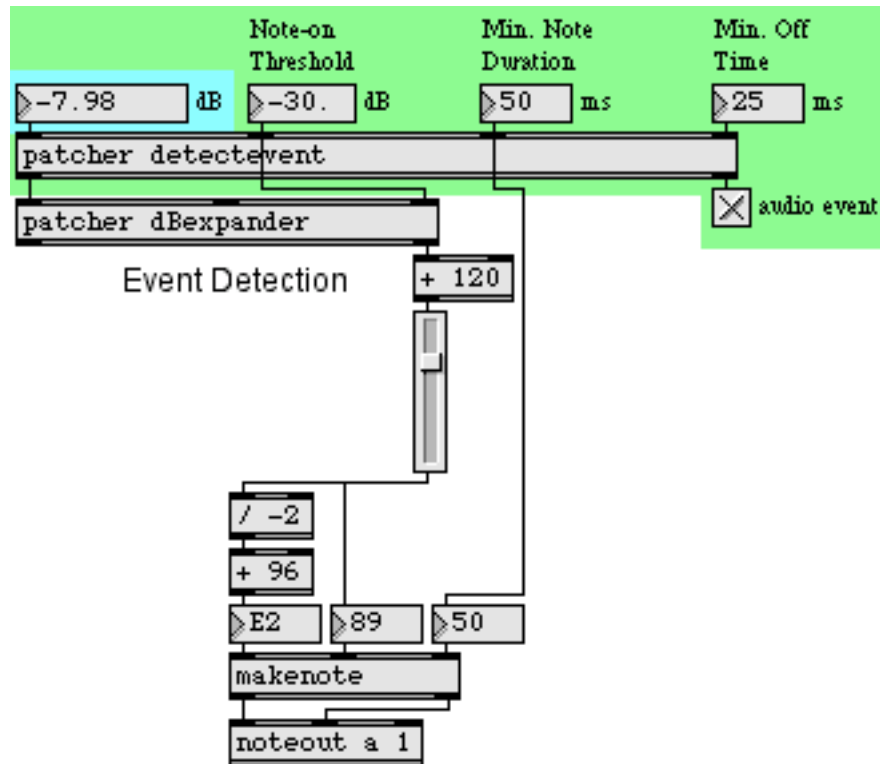
The simplest use of an audio event is just to trigger something else when an event occurs. Whenever an audio event is detected, we trigger the **patcher** flashbulbs subpatch. That subpatch generates a 16x12 matrix of random colors, then uses scaling to turn most of the colors to black, leaving only a few remaining cells with color. When that matrix goes out to the main patch, those cells are upsampled with interpolation in the **jit.pwindow** and look like flashes of colored light. Subsequent level values from **peakamp~** are used in the *[flashbulbs]* subpatch to bang a **bline** object, causing the colors to fade away after 20 bangs.



The [flashbulbs] subpatch

In the **patcher** pickpicture subpatch, we simply divide the event amplitudes up into five equal ranges, and use those values to trigger the display of one of five different pictures.

In the following example, you can see the use of audio information to trigger MIDI notes.



Peak level determines pitch and velocity of a MIDI note

We use the expanded decibel value coming out of the right outlet of the **patcher** expander to derive MIDI pitch and velocity values. We first put the values in the range 0 to 120, then use those values as MIDI velocities and also map them into the range 96 to 36 for use as MIDI key numbers. (Note that we invert the range so as to assign louder events to lower MIDI notes rather than higher ones, in order to give them more musical weight.) The note durations may be set by the *Min. Note Duration* **number box**, or they may be set independently by entering a duration in the **number box** just above **makenote**'s duration inlet.

- You can experiment further with this patch in a number of ways: by changing the rate of the audio file with the *Rate* **number box**, by opening different soundfiles and movies, by choosing *Sound Input* from the **ubumenu** to use live sound input, and by changing the various tracking parameters such as *Reporting Interval*, *Noise Floor Threshold*, *Note-On Threshold*, and *Min. Note Duration*.

Summary

We've demonstrated how to track the peak amplitude of a sound with **peakamp~**, how to convert linear amplitude to decibels, and how to detect audio events by checking to see if

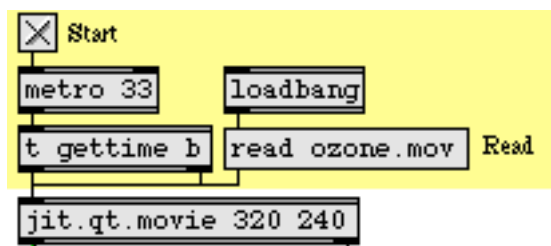
the amplitude level has exceeded a certain threshold. We used the information we derived about the amplitude and the and peak events to trigger images algorithmically, select from preloaded images, play MIDI notes, and alter video effects.

Tutorial 29: Using the Alpha Channel

In this tutorial we'll look at how to composite two images using the *alpha channel* of a 4-plane *char* Jitter matrix as a transparency mask. We'll explore this concept as a way to superimpose subtitles generated by the **jit.lcd** object over a movie image.

- Open the tutorial patch *29jUsingTheAlphaChannel.pat* in the Jitter Tutorial folder.

The upper-lefthand region of the tutorial patch contains a **jit.qt.movie** object that reads the file *ozone.mov* when the patch opens. The **metro** object outputs a new matrix from the **jit.qt.movie** object every 33 milliseconds and polls the time attribute of the object (the current playback position of the movie, in QuickTime time units) by using a **trigger** object:



Play back the movie, getting the current time position with each new matrix

- Start viewing the movie by clicking the **toggle** box attached to the **metro**. The **jit.qt.movie** object will start to output matrices as well as the current playback position within the movie. You should see the movie (with subtitles!) appear in the **jit.pwindow** at the bottom of the patch.

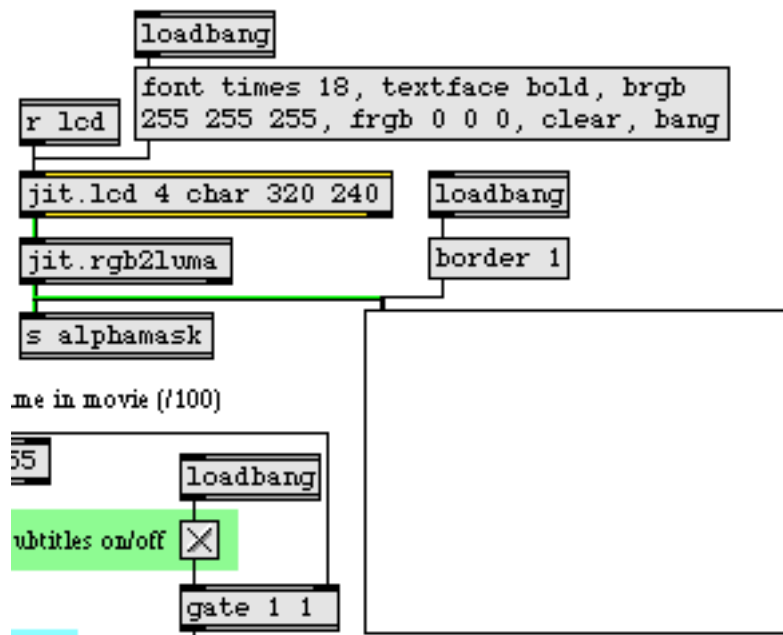


Nice kitty

First, we'll look at how the subtitles are being generated. Then we'll investigate how we composite them with the image from the movie using the alpha channel.

The jit.lcd object

The subtitles in our patch are generated by sending messages to the **jit.lcd** object (at the top of the patch). The arguments to **jit.lcd** specify the plane count, type, and dim of the matrix generated by the object (**jit.lcd** only supports 4-plane *char* matrices). The **jit.lcd** object takes messages in the form of QuickDraw commands, and draws them into an output matrix when it receives a bang. We initialize our **jit.lcd** object by giving it commands to set its font and textface for drawing text and its foreground (frgb) and background (brgb) color (in lists of RGB values). We then clear the **jit.lcd** object's internal image and send out an empty matrix with a bang:

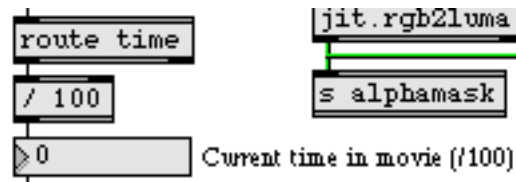


*An initialized **jit.lcd** object*

Note: The **jit.lcd** object has the same complete set of QuickDraw commands and functions that are available in the Max **lcd** object. Though we'll be using it in this patch for the purpose of generating text images, **jit.lcd** can be used to generate all manner of 2-dimensional vector graphics as well. *Tutorial 43* in the *Max Tutorials and Topics* manual demonstrates some of the features of the **lcd** object, all of which can be applied just as easily to **jit.lcd**.

The **jit.lcd** object outputs its matrix into a **jit.rgb2luma** object, which converts the 4-plane image output by **jit.lcd** into a 1-plane grayscale image. The **jit.rgb2luma** object generates a matrix containing the luminosity value of each cell in the input matrix. This 1-plane matrix is then sent to a **send** object with the name **alphamask** and to a **jit.pwindow** object so we can view it. Note that the **jit.pwindow** object has its **border** attribute set to 1. As a result, we can see a 1-pixel black border around the white image inside.

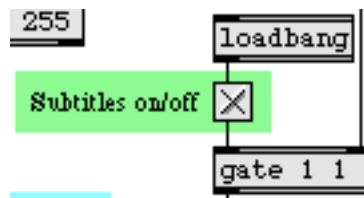
Our **jit.lcd** object also receives messages from elsewhere in the patch (via the **receive** object named **lcd** attached to it). The subtitles are generated automatically by looking for certain times in the movie playback:



*Parsing the time values from the **jit.qt.movie** object*

The **jit.qt.movie** object outputs its current playback position with every tick of the **metro** object, thanks to the **t gettime b** we have between the two. The time attribute is sent out the right outlet of the **jit.qt.movie** object, where we can use a **route** object to strip it of its message selector (time). We divide the value by 100 so that we can search for a specific time more accurately. Since the **metro** only queries the time every 33 milliseconds, it's entirely possible that we'll completely skip over a specific time—dividing the time value by 100 makes it easier to find the point in the movie we want.

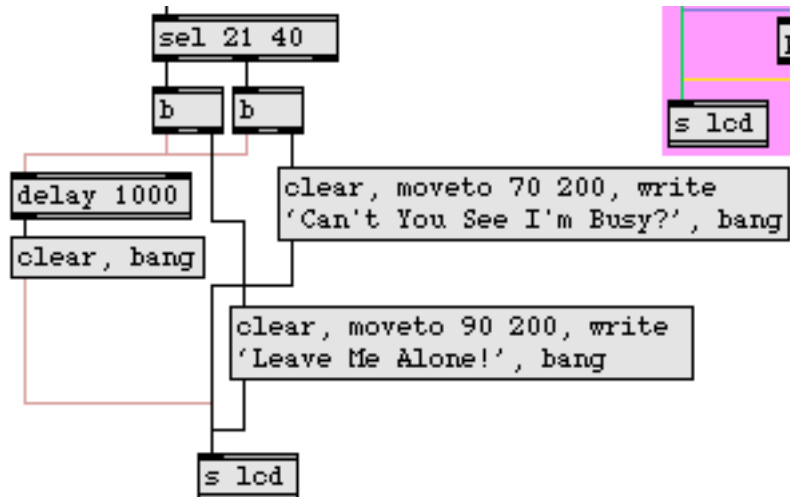
The time values are sent through a **gate** object where you can disable the subtitles if you so choose:



*Control the flow of the time values with a **gate** object*

- Click the **toggle** box attached to the **gate**. The subtitles should disappear. You can resume the subtitles by clicking the **toggle** box again.

The subtitles are finally generated when the time values 21 and 40 make it past the **gate** object. The select object sends out a bang when those values arrive. This triggers commands from the **message** boxes to **jit.lcd**:

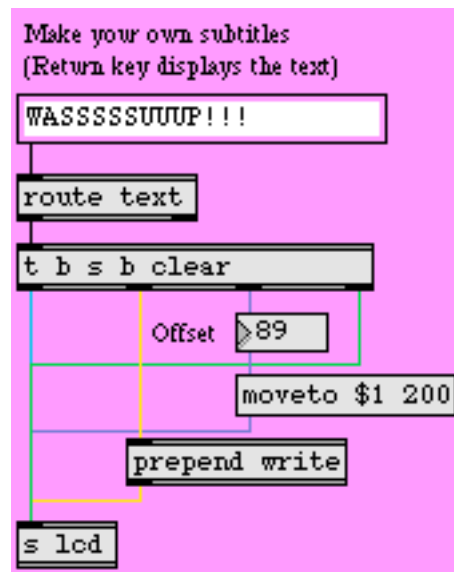


Performing the subtitling based on the time values

The clear message to **jit.lcd** erases the drawing canvas, filling all the pixels with white (our chosen background color). The moveto message moves the cursor of the **jit.lcd** object to a specific coordinate from which it will draw subsequent commands. The write message draws text into the matrix using the currently selected font and textface. Once we've written in our subtitles, we send the object a bang to make it output a new matrix. With every subtitle, we also send a bang to a **delay** object, which clears and resends the matrix 1000 milliseconds later, erasing the title.

Make Your Own Titles

The region of the tutorial patch to the right (with the magenta background) lets you use the **textedit** object to generate your own subtitles. The **number box** labelled *Offset* determines the horizontal offset for the text. The **trigger** object allows you to send all the necessary QuickDraw commands to the **jit.lcd** object in the correct order.



Make your own subtitles

- Turn off the automatic subtitling with the **toggle** box above the **gate**. Type some text into the **textedit** box and hit the return key. The text will appear superimposed over the image.



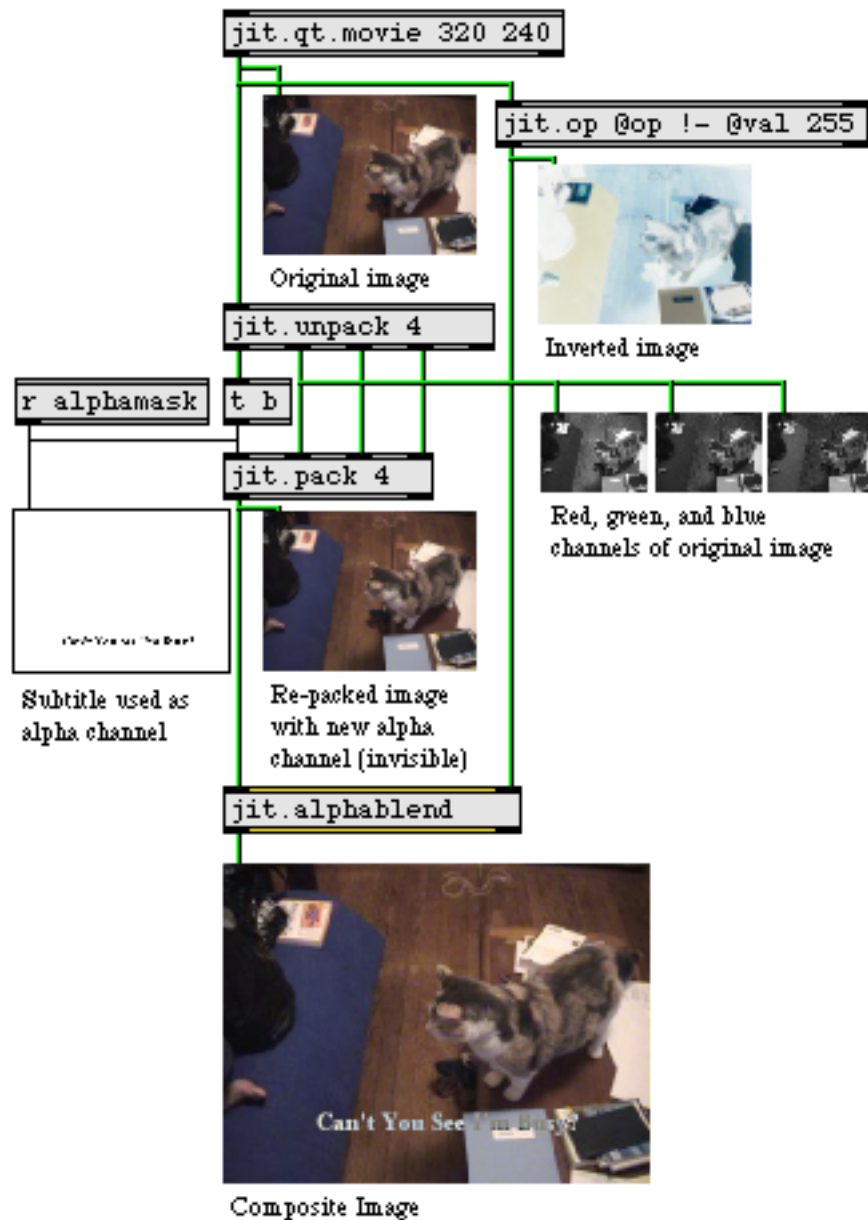
The new subtitle over the image

Now that we understand how the titles are generated, lets take a look at how they get composited over the movie.

in its leftmost inlet). The **jit.op** object creates a negative of the original matrix from the QuickTime movie (by subtracting the matrix values from 255 using the `!-` operator). The **jit.alphablend** object then uses our new alpha channel—white values in the subtitle matrix cause the original image to be retained, while black values bring in the inverted image from the righthand matrix.

Different techniques are often used for subtitling. The technique of superimposing white text over an image (sometimes with a black border around it) is far more common than the technique used here of filling an alpha mask with an inverted image. However, doing our subtitling this way gives us a perfect excuse to use the **jit.alphablend** object, and may give you more legible subtitles in situations where the background image has areas of high contrast.

The image below shows the compositing process with **jit.pwindow** objects showing intermediate steps:



The compositing process, showing intermediate steps

Summary

The **jit.lcd** object offers a complete set of QuickDraw commands to draw text and 2-dimensional graphics into a Jitter matrix. The **jit.rgb2luma** object converts a 4-plane ARGB matrix to a 1-plane grayscale matrix containing luminance data. You can replace the alpha channel (plane 0) of an image with a 1-plane matrix using the **jit.pack** object. The

jit.alphablend object crossfades two images on a cell-by-cell basis based on the alpha channel of the lefthand matrix.

Tutorial 30: Drawing 3D text

This tutorial shows you how to draw and position 3D text in a **jit.window** object using the **jit.gl.text3d** and **jit.gl.render** objects. Along the way, we will cover the basics of drawing OpenGL graphics using the **jit.gl.render** object.

The **jit.gl.text3d** object is one of the many Jitter OpenGL drawing objects that work in conjunction with the **jit.gl.render** object. OpenGL is a cross-platform standard for drawing 2D and 3D graphics, designed to describe images so that they can be drawn by graphics coprocessors. These coprocessors, also known Graphics Processor Units or GPUs, speed up drawing operations enormously, allowing complex scenes made of textured polygons to be animated in real time. OpenGL graphics can help you create faster visual displays or interfaces without bogging down your computer's CPU.

- Open the tutorial patch *30j3DText.pat* in the Jitter Tutorial folder.

In the lower left of the patch, there is a **jit.window** object named *hello*. This window will be the destination for our OpenGL drawing.

- Click on the **toggle** labeled *Start Rendering*.

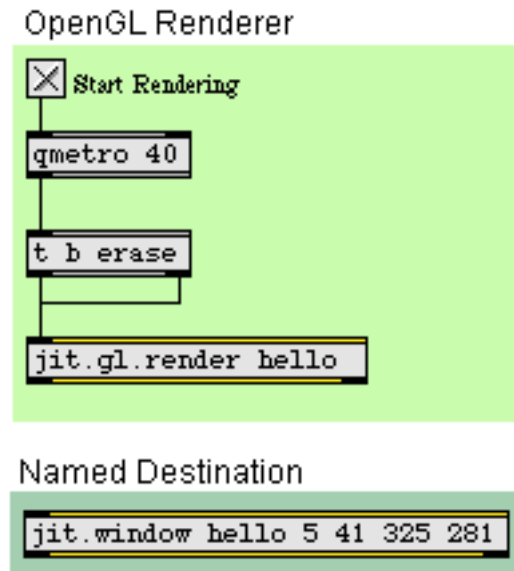
The toggle starts the **qmetro** object, which sends bang messages to a **trigger** object. For each bang received at its inlet, the **trigger** object sends the message *erase* out its right outlet, and then a bang message out its left outlet. These messages are sent to the **jit.gl.render** *hello* object.

Creating a Drawing Context

All OpenGL graphics in Jitter are drawn with the aid of **jit.gl.render** objects. Each **jit.gl.render** object must refer to a named *destination* for drawing. You can specify this destination using an initial argument to **jit.gl.render**. So in this case, **jit.gl.render** *hello* creates a **jit.gl.render** object that will draw to the destination window that we have named *hello*.

It's important to note that *hello* is not the name of the **jit.gl.render** object—only its destination.

We refer to this combination of a **jit.gl.render** object and a named destination as a *drawing context*. A drawing context is required for OpenGL rendering. The objects on the left side of this patch are sufficient to build a valid drawing context, and so when you click on the toggle, the message `jit.gl.render: building GL on window 'hello'` appears in the Max window. This tells you that the context is being created.



*A **jit.gl.render** object and a named destination create a drawing context*

GL Objects in the Context

A variety of Jitter objects are provided for drawing different things into OpenGL contexts. These Jitter objects have names that all start with “jit.gl”. Some examples are **jit.gl.model** (which draws 3D models), **jit.gl.plato** (which draws Platonic solids), and the **jit.gl.text3d** object, as seen in the upper right of the Tutorial patch. These objects all take the name of a drawing context as an initial argument, and are automatically drawn into that context by a **jit.gl.render** object each time it receives a bang message. So in this example, as long as the **metro** object is sending bang messages, the **jit.gl.render** object is performing the same set of operations: erase the screen, draw all objects, repeat. The only reason that the **jit.gl.text3d** object’s output is not visible in the window is that it has no text string assigned to it. Let’s remedy this.

- Click on the **message** box reading Hello\, Jitter!.

This sets the text contained in the **jit.gl.text3d** object. You should now see the word “Hello” in the **jit.window**. Note that the comma in the text is preceded by a backslash;

since commas normally separate sequential messages in a **message** box, the backslash is needed to tell Max to treat the comma as an ordinary text symbol.

Technical Detail: The **qmetro** object is needed in this patch because, unlike most previous tutorial patches, there are no **jit.matrix** objects present. In complex patches, the drawing or matrix calculations being repeatedly triggered by a **metro** object may not complete before the next bang is scheduled to arrive. In this example, this situation would occur if the text took more than 40 milliseconds to render. Normally, the Max scheduler would place this bang on its queue—a list of pending messages. Max is not normally allowed to drop messages from the queue. So in this patch, if each bang generated a sequence of events that take longer than 40 milliseconds, and no dropping of messages was allowed, the queue would eventually *overflow*, or run out of space to store additional pending messages. Then the scheduler would stop, and so would your show! This is probably not what you had in mind. The **qmetro** object (which is really just a combination of a **metro** object and the **jit.qball** object—see *Tutorial 16*) helps you avoid this situation by dropping any bang messages that are still pending during a calculation. If a bang is scheduled to occur while the rest of the patch is rendering, it will be placed in a queue. If that bang still hasn't been passed to the rest of the patch by the time the next bang occurs, the first bang will be *usurped* by the second, which will be placed next in the queue. And so on.

Let's say the output of a **metro** object set to output a bang every 10 milliseconds is sent to a **jit.qt.movie** object, followed by some effects that take 100 milliseconds per frame to draw. So the **jit.qt.movie** object will receive 10 bang messages during the time the effects are being calculated. The **jit.qt.movie** object knows in this case that the processing is not yet complete, and instead of sending out ten **jit_matrix** messages next time it has a chance, it drops all but one. While the **jit.qt.movie** and **jit.matrix** objects have this capability built in, the **jit.gl.render** object does not. This is because the **jit.gl.render** object will often need matching groups of messages in order to draw a frame properly. In this example, the erase message is needed to clear the screen, and the bang message draws the text. If an erase message was dropped to avoid overflowing the queue, multiple bang messages might be processed in a row, and multiple copies of the text would be drawn at once, which is not what we want to do. As patches get more complex, different kinds of visual artifacts might result. So, the **qmetro** object is provided to let the designer of the patch decide what messages to drop from the queue. In this example, as in most cases, simply substituting **qmetro** objects for **metro** objects insures that the drawing will always look correct, and the queue will never overflow.

Common 3D Attributes

All Jitter OpenGL objects (which all begin with "jit.gl") share a common set of attributes that allow them to be positioned in 3D space, colored, and otherwise modified—the *GL group*. Drawing 3D graphics can be a fairly complex task. The ob3d group simplifies this by insuring that the messages you learn to draw one 3D object will work with all of them, whenever possible. These common attributes are fully documented in the *GL group* section of the Jitter *Object Reference*. To introduce the 3D group here, we will demonstrate the position, rotation, scale and axes attributes by manipulating the **jit.text3d** object.

We can add a set of spatial axes to the **jit.text3d** object to make our spatial manipulations easier to see and understand—it's both easier to see how the text object is oriented, and also to see additional information about exactly where the object's origin is.

- Click on the **toggle** connected to the **message** box reading axes \$1 in order to see the 3d object's axes.



The 3D text at the origin

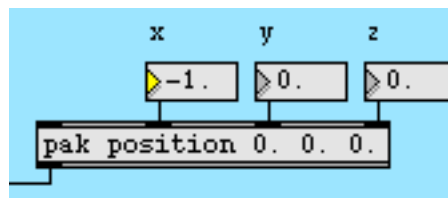
The *x axis*, drawn in red, points to the right. Zero is at the center of the screen, and increasing *x* values move in the direction of the right of the screen. The *y axis*, drawn in green, points upwards, and the *z axis*, drawn in blue, points out of the screen towards you (since it is pointed directly toward you, you will only see a small blue dot). These axes represent the object's local coordinate system as it moves through the world. When a GL

group object is first created, its local coordinate system is not rotated, translated, or scaled—it has the same coordinate system as the rest of the 3D world, by default

The text “Hello, Jitter!” is displayed in the **jit.window** object's display area, but it starts at the center of the screen, so the final part is cut off. Let's move the text to the left.

- Set the **number box** labeled *x* in the *Common 3D attributes* section of the patch to the value -1 .

The Number box sends a floating-point value to the **pak** object, which sends the message position followed by three numbers to the **jit.gl.text3d** object. As you change the value in the number box, you can see the text slide to the left until all of it is visible on the screen.



Changing the position attribute

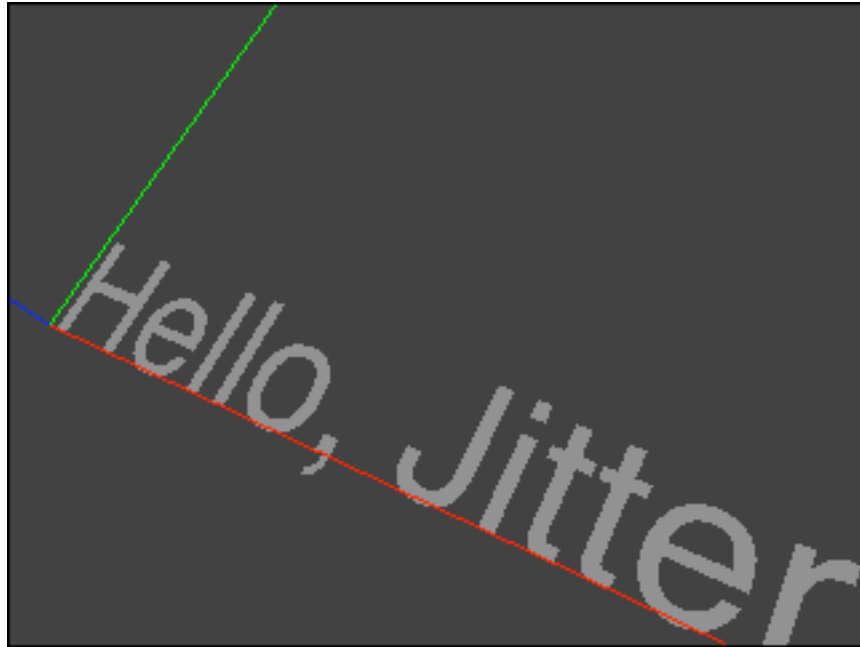
We have just changed the position attribute of the **jit.gl.text3d** object. The position message can be followed by three number arguments that set the position of the object in three dimensions. If fewer than three numbers follow the position message, the axes are filled in the order [x, y, z], and the position of the object on the unspecified axes is set to 0. For example, sending the message position 5. will set the position of a GL group object to the location [5, 0, 0].

The operation of changing an object's position is called *translation*.

Now, let's rotate the text.

- Set the **number boxes** labeled *x*, *y* and *z* directly above the **pak** rotation object in the *Common 3D attributes* section of the patch to the value 1. This sets the axis of rotation.

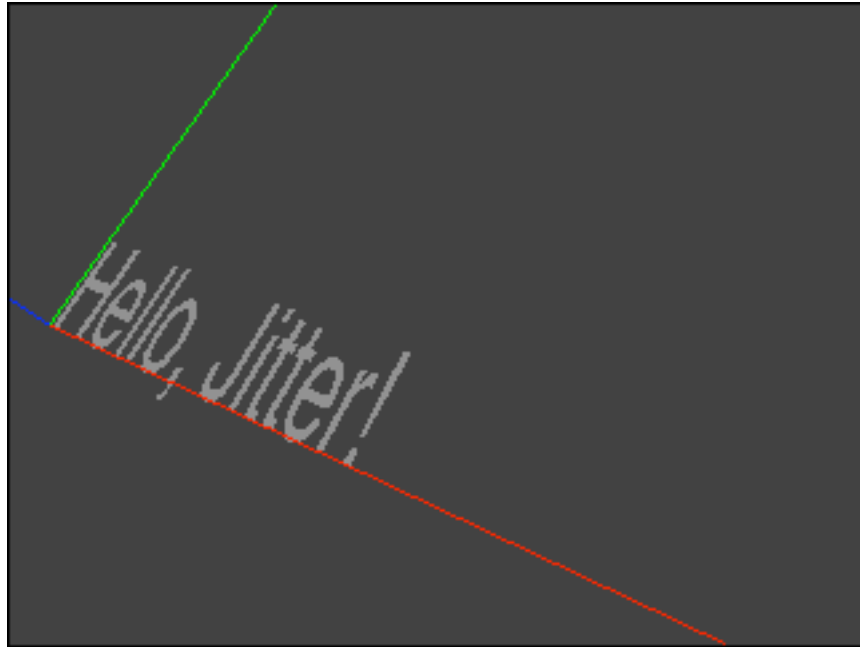
- Drag the number box labeled angle to the value 320. You will see the text rotate around the axis into the position shown in this screenshot.



The 3D text after translation and rotation

Sending the message `rotate` followed by from one to four numbers sets the rotation attribute of an GL group object. The first number is an amount of rotation in degrees, counterclockwise around the object's axis of rotation. The other three numbers specify this axis of rotation as an $[x, y, z]$ vector. As with the position attribute, some values following the rotation attribute can be dropped and default values will be understood. If the rotation message is followed by only one number, that number is understood to mean the angle in degrees, and the axis of rotation is $[0, 0, 1]$. This rotates the object around the z axis, or in other words, in the x - y plane of the screen. If two or more numbers accompany the rotation message, the first one always specifies the rotation angle, and following ones specify the vector of rotation in the order in which they appear.

- Set the **number box** labeled *x* directly above the **pak** scale object to 0.5. This scales the 3D text by half along its local *x* axis.



The 3D text after translation, rotation and scaling

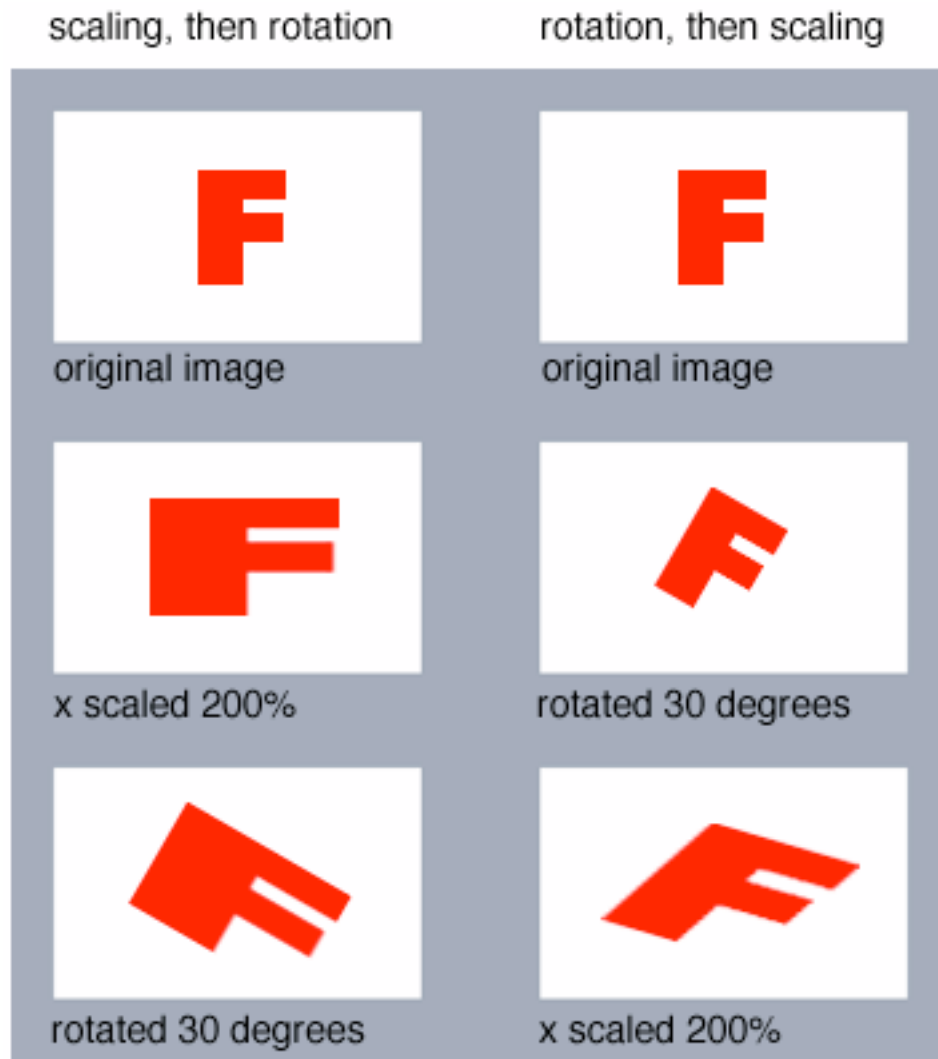
Note that the red axis scales along with the object. The dots along the line are now twice as close together as they were before the scale operation. The axes are always drawn in the GL group's local coordinate system, which in this case has been translated, rotated, and scaled with respect to the world coordinate system in the **jit.gl.render** object.

It's important to consider the order of operations when performing geometry transforms. You can set the rotate, position and scale attributes of an object independently in any order you wish. But each time it is drawn, the object is transformed in the following order:

1. Scaling
2. Rotation
3. Translation

Keeping this order is essential so that the attributes behave predictably. If rotation occurred before scaling, for example, a `scale 0.5` message would scale the object not just in its *x* coordinate, but in some combination of *x*, *y* and *z* depending on the rotation of the object.

The following example shows the difference that performing operations in different orders makes.



The order of operations makes all the difference

Summary

Creating a draw context is the first step to using OpenGL graphics in Jitter. A draw context consists of a named destination, such as a window, and a **jit.gl.render** object drawing to that destination.

A variety of Jitter objects exist which draw OpenGL graphics in cooperation with **jit.gl.render**; their names all start with “jit.gl.” The **jit.gl.text3d** object is one example. All the Jitter OpenGL objects share a common set of attributes for moving them in 3D space. This group of objects is referred to as the *GL group*.

Tutorial 31: Rendering Destinations

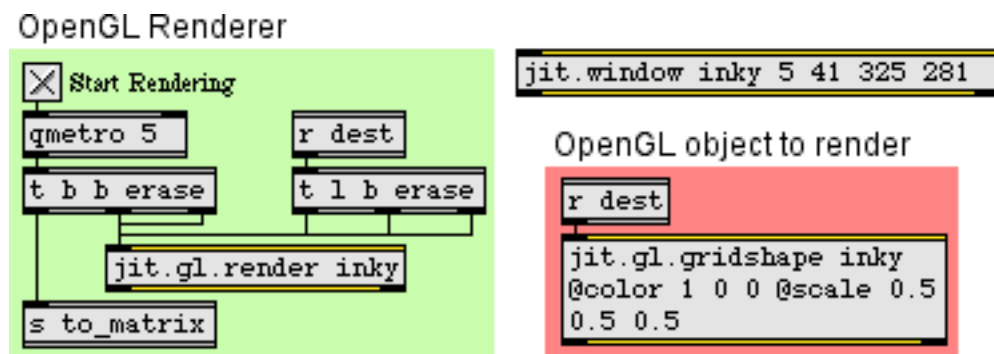
In the previous tutorial, we saw how to draw OpenGL graphics into a **jit.window** using the **jit.gl.render** object. Now we will look at the other kinds of destinations into which a **jit.gl.render** object can draw, what to use them for, and how to switch between destinations.

- Open the tutorial patch 31jRenderDestinations.pat in the Jitter Tutorial folder.

In the upper left is, among other objects, a **jit.gl.render** object with an argument of **inky**. We saw in *Tutorial 30* that the initial argument to **jit.gl.render** specifies a named *rendering destination*. In the upper right of the patch, we have such a destination: the **jit.window** object named **inky**.

- Click on the **toggle** in the upper left of the patch labeled *Start Rendering*.

You should see a red ball appear in the **jit.window** object. The **jit.gl.gridshape** object is drawing the ball. Its first argument, **inky**, specifies the drawing context currently being drawn by the **jit.gl.render** object to the **jit.window**. The other arguments set the color and scale attributes of the ball.



Draw a red ball in the window named inky

Drawing and Swapping Buffers

Clicking the **toggle** starts the **qmetro** object, which sends bang messages to the object **t b b erase**. This object first sends an erase message followed by a bang message to the **jit.gl.render** object, and then a bang message that is used elsewhere in the patch.

When the **jit.gl.render** object receives the bang message, it draws all of the GL group objects which share its destination, then copies its offscreen buffer, if any, to the screen. An *offscreen buffer* is an area of memory not visible on the screen, the same size as the drawing destination. By default, all drawing contexts have an offscreen buffer. Drawing is done into the buffer, which must then be copied to the screen in order to be seen.

The offscreen buffer makes flicker-free drawing and animation possible, as we see here. Even though the buffer is being erased before the red ball is drawn each time, we never see the buffer in its erased state. To see what drawing looks like without the offscreen buffer, click on the **message** box `doublebuffer 0` in the upper right of the patch. You will probably see the image start to flicker. This happens because the `erase` message now causes the window itself to be erased, not the offscreen buffer. The window remains blank for a short but indeterminate period of time before the red ball is drawn, so you can see the area under the ball flickering between red and the dark gray of the erased window. Click the **message** box `doublebuffer 1` to remake the offscreen buffer and stop the flickering.

Setting Fullscreen Mode

The **p fullscreen** subpatch contains a **key** object, a **select** object, a **toggle**, a **message** box and an **outlet** object that sends the results of the subpatch to the **jit.window** object. This is standard Max stuff, so we won't go over it in too much detail—the result is that the escape key toggles between sending the messages `fullscreen 0` and `fullscreen 1` to the **jit.window** object. (See the "Full Screen Display" section of *Tutorial 14*.)

- Press the 'esc' key to change the **jit.window** object to fullscreen mode and back.

The escape key seems to be a common way to toggle fullscreen mode, so we've used this setup in many of the example patches. It's important to note, however, that this is just a convention—there's nothing built into Jitter that forces you to use one key or another for this purpose.

You can use the `fsmenubar` message in conjunction with the `fullscreen` message to hide the menubar in addition to covering the screen.

Setting a jit.pwindow Destination

The **jit.gl.render** object can draw to three different kinds of destinations. The right side of the tutorial patch contains an object example of each destination: a **jit.window**, a **jit.pwindow**, and a **jit.matrix**. Right now we are rendering to the **jit.window** object. To change the destination to the **jit.pwindow** object, we need to first name the **jit.pwindow** object and then set the destinations of the **jit.gl.render** object and the **jit.gl.gridshape** object.

- Click on the **message** box name `blinky` above the topmost of the **jit.pwindow** objects.

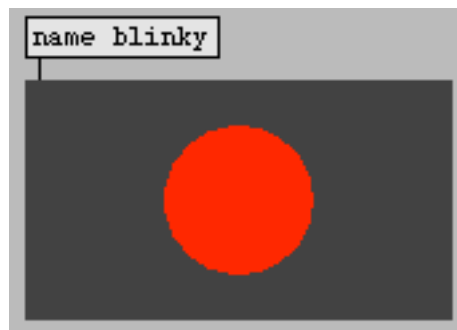
This names the **jit.pwindow** object, which allows it to be used as a rendering destination. To switch the drawing to this destination, we need to send messages to both the **jit.gl.render** object and the **jit.gl.gridshape** object, telling them about the new destination.

- Click on the **message** box `blinky` in the *Switch Destinations* section of the patch.

The symbol `drawto` is prepended to the message `blink`, and the result is sent to the `s dest` object. Two objects receive this message—`jit.gl.gridshape` and `tlb erase`. The `trigger` object sends a sequence of messages to the `jit.gl.render` object, which tell it to:

1. Erase its current destination's draw buffer
2. Swap that buffer to the screen, visibly clearing the old destination, and
3. Switch future drawing to the new destination.

The result is that the red ball is displayed on the `jit.pwindow` object at the right of the patch.



Viewing our rendered output in a `jit.pwindow` object

Setting a `jit.matrix` Destination

In addition to drawing to `jit.window` objects and `jit.pwindow` objects, we can draw to `jit.matrix` objects. We introduced offscreen buffers in the discussion of `jit.window` destinations, above. When a 2D `jit.matrix` object is a rendering destination, the `jit.matrix` data is used as the offscreen buffer. This puts an image of an OpenGL scene into the same format as video data, so that any of Jitter's video-processing effects can be applied to the image.

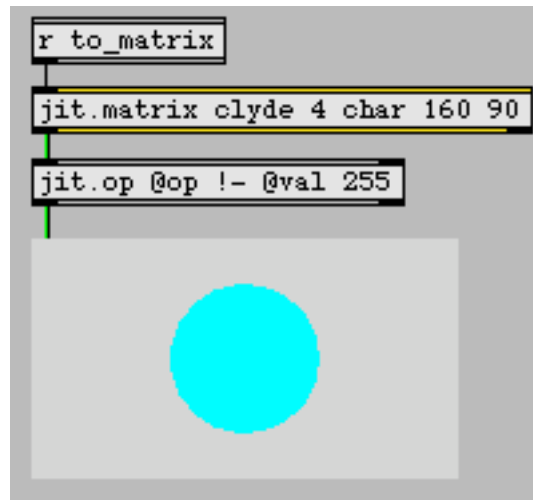
The `jit.matrix` object must meet a few criteria in order for OpenGL graphics to be drawn into it:

- It has to have four planes of *char* data.
- It has to have two dimensions.
- It has to be bigger than eight pixels in both width and height.

We have such a matrix in the bottom right corner of the tutorial patch. It is 160 pixels wide by 90 pixels high—the same dimensions as the `jit.pwindow` object below it.

- Click on the **message** box clyde in the *Switch Destinations* section of the patch to draw into the **jit.matrix** object.

You should see a cyan ball on a light gray background. This is because the red ball image generated by OpenGL is processed through the **jit.op** object, which subtracts each color component from 255, inverting the image.



*Rasterizing the output into a **jit.matrix***

There's one more important detail about drawing to **jit.matrix** objects. Note that underneath the **qmetro** object there's a trigger object **t b erase**. The leftmost (and therefore last) bang message from this object is sent to the **jit.matrix** object into which we are drawing. This is necessary to see the image in the **jit.pwindow**. When the **jit.gl.render** object receives a bang message, it finishes constructing the drawing in the offscreen buffer belonging to the **jit.matrix** object. But to send the resultant matrix out for viewing or further processing, it's necessary to send a bang message to the **jit.matrix** object.

Hardware vs. Software Rendering: One of the great advantages about using OpenGL for rendering graphics is that most of the work can be done by the graphics accelerator hardware in your computer, freeing the CPU up for other work such as generating audio. When drawing into **jit.window** objects or **jit.pwindow** objects, the hardware renderer can be used. Unfortunately, due to limitations of the current system software (Mac OS 9) the hardware renderer cannot draw directly into **jit.matrix** objects. This is not a limitation inherent in OpenGL, and may change in the future. Now, however, this means that drawing directly into Jitter matrices is significantly slower than drawing to **jit.window** or **jit.pwindow** objects, especially for complex scenes or those involving textures.

Multiple Renderers and Drawing Order

To move an OpenGL scene to a different destination, the **jit.gl.render** object as well as any GL group objects involved in drawing the scene must receive the `drawto` message. Why not just send a message to the renderer, telling it to move the entire scene? The reason is that each GL group object can have an independent destination, as well as each renderer. Objects in the GL group can be moved between multiple renderers. To see an example of why this might be useful, please look at the other patch for this tutorial.

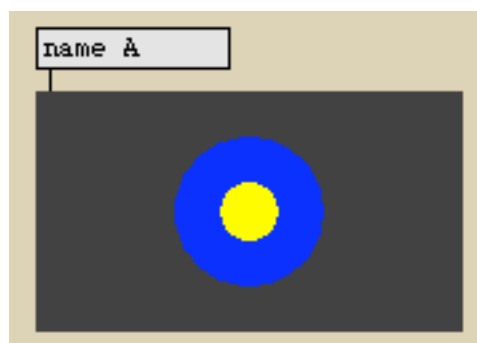
- Open the tutorial patch entitled *31jMoreRenderDestinations.pat*.

At the right are three **jit.pwindow** objects named A, B and C. The **message** box objects above them are not strictly necessary, because once a **jit.pwindow** object has been named, its name is stored with it in the patch. But they are useful here as labels and as a reminder of what messages to send to name the objects.

There are also three **jit.gl.render** objects in the patch. Each of them is pointed at a different destination.

- Click the **toggle** labeled *Start Rendering*.

This repeatedly sends the `erase` message followed by a `bang` message to each of the three renderers. You should see a yellow circle within a blue circle in the topmost drawing destination. This simple OpenGL scene is created by the two **jit.gl.gridshape** objects in the bottom left of the patch. We can move each of these objects to any of the three drawing destinations present.



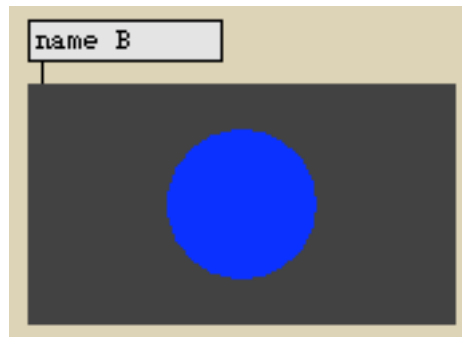
Rendering to **jit.pwindow A**

- Click the topmost **message** box reading B in the *Switch Destinations* section of the tutorial patch. This changes the destination of the blue circle to the draw context named "B"—it now appears on the center **jit.pwindow**.

- Click the lower **message** box reading B in the *Switch Destinations* section of the tutorial patch. This changes the destination of the yellow circle to the draw context named "B". The two objects are reunited.

Each time a **jit.gl.render** object receives a bang message, it draws all of the GL group objects that have been added to its context. It draws the objects in the order in which they were added to the context. In this case, the yellow circle was added to the draw context named "B" after the blue circle, so it appears on top. To change this order, we can send the message `drawto B` to the **jit.gl.gridshape** object drawing the blue circle again. This will remove it from its present place in the list of objects for the context named B, and add it again at the end of the list.

- Click the upper **message** box reading B in the *Switch Destinations* section of the tutorial patch again. The blue circle should now obscure the yellow circle.



The blue circle obscures the yellow one

Summary

We have introduced a flexible system for creating multiple OpenGL *renderers* and *drawing destinations*, and for moving objects in the GL group between them using `drawto` messages.

Three Jitter objects can function as drawing destinations: **jit.window**, **jit.pwindow**, and **jit.matrix**. Each kind of destination has different uses. A **jit.window** object can be moved to different monitors and quickly enlarged to cover the screen. A **jit.pwindow** object keeps its location in the patch. A **jit.matrix** object may be used as an offscreen buffer for rendering. The output of the **jit.matrix** object is a rasterized image of the 3D scene, to which further video processing may be applied.

Tutorial 32: Camera View

This tutorial shows you how to set up the camera view and how to position and rotate GL objects using **jit.gl.handle**. It will cover the group of components which together make up the *camera view*: the camera's position, the point at which the camera is looking, the "up" vector, the type of projection, the lens angle, and the clipping planes.

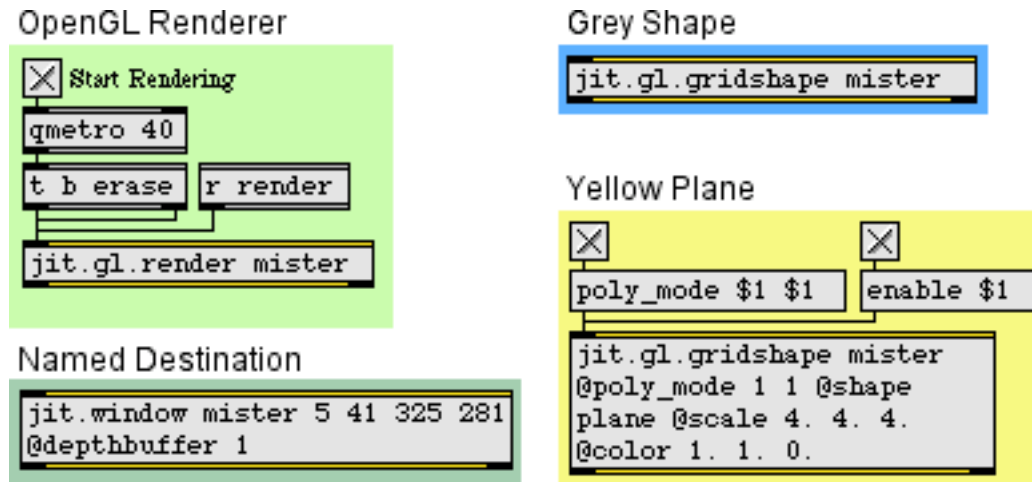
- Open the tutorial patch *32jCameraView.pat* in the Jitter Tutorial folder.

In the lower left of the patch, there is a **jit.window** object named *mister*. This window will be the destination for our OpenGL drawing. You will notice that the **jit.window** object has an attribute argument `@depthbuffer 1` to specify the creation of a depth buffer. A depth buffer allows the OpenGL renderer to determine which drawing command is visible at a given pixel based on the proximity of the geometry to the camera. Without depth buffering, OpenGL uses what is often referred to as the "Painter's Algorithm"--i.e. the visible results of drawing commands correspond to sequence in which they are performed.

- Click on the **toggle** object labeled *Start Rendering*.

We now see large gray circle and some yellow lines. These are being drawn by two instances of the **jit.gl.gridshape** object. The **jit.gl.gridshape** object can draw a variety of 3D shapes, including spheres, tori, cylinders, cubes, planes, and circles. The grey circle we see drawn in the window is actually a sphere and is being drawn by section of the patch labeled *Grey Shape*. The yellow lines are actually a plane and are being drawn by the section of the patch labeled *Yellow Plane*. The yellow plane is being rendered with `poly_mode 1 1`, which means that the shape is being drawn with outlined polygons rather than filled polygons for both the front and back faces of the plane.

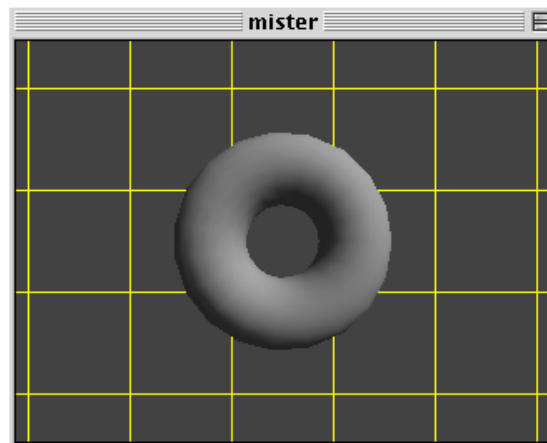
If you switch off the **toggle** object connected to the **message** box `poly_mode $1 $1`, you can see the plane rendered with filled polygons.



The mister drawing context

- In the *Grey Shape* section of the patch, click on the **message** box `scale 0.3 0.3 0.3` and then click on the message box containing shape `torus`. You should now see what looks like a grey doughnut.
- Click on the **toggle** object connected to the **message** box `lighting_enable $1` and then click on the **toggle** object connected to the **message** box `smooth_shading $1`. We are now staring at a lit, smoothly shaded, 3D gray torus.

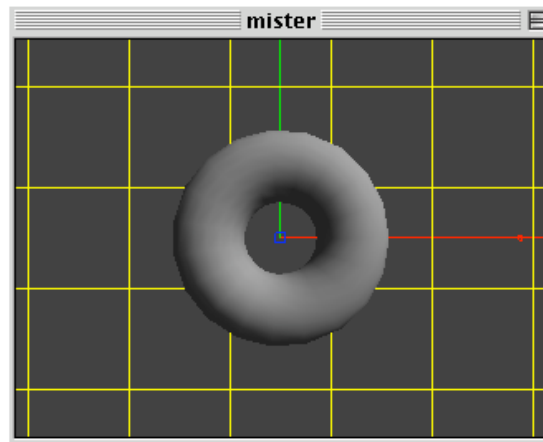
By default, Jitter's GL objects have lighting and smooth shading disabled, so it is necessary to turn these on. Lighting will be covered in detail in *Tutorial 36*.



Rendered shapes with smooth shading and lighting enabled

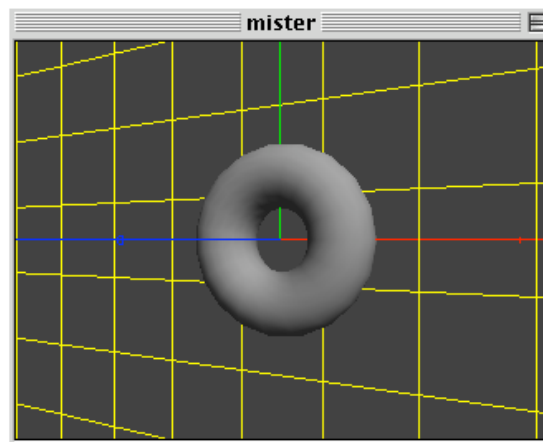
- In the *Camera View* section of the patch, click on the **toggle** object labeled *jit.gl.render axes*.

You should see a red line from the center of the window to the right of the window, a green line from the center of the window to the top of the window. These are the x and y axes, respectively. They help us to determine the origin of our scene. Since the default camera position is at $[0.,0.,2.]$, and the default lookat position is $[0.,0.,0.]$, the camera is looking directly at the origin of our scene along the z axis. Hence, we do not see the blue line which represents the z axis along which the camera is looking.



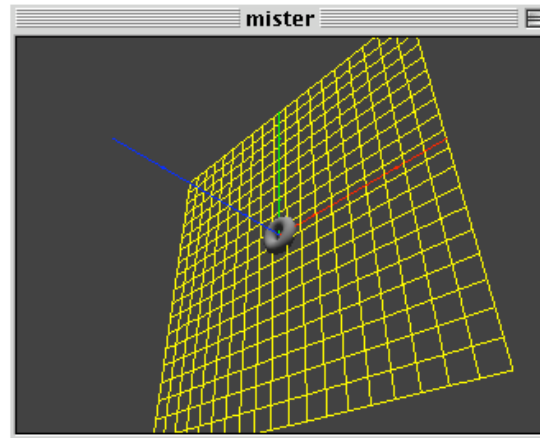
View the axes

- Under the *camera position* label, set the x value to be 1. Now the camera is at the position $[1.,0.,2.]$, it is still looking at the position $[0.,0.,0.]$, and the blue line of the z axis has become visible.



The axes with a different viewing position

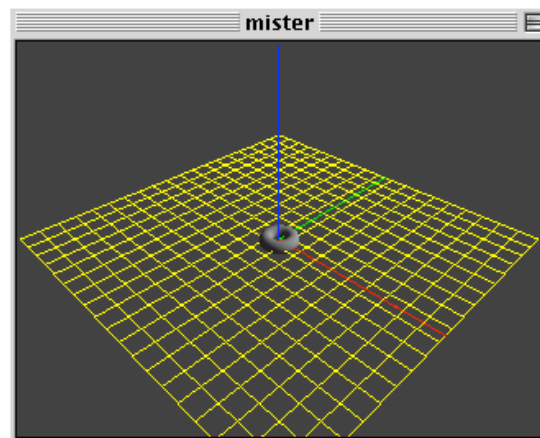
- Now let's set the camera position x value to 6., y value to -6., and z value to 6. so that the camera position is $[6,-6,6]$. You can see that the yellow plane and the axes are, in fact, finite.



Viewing the edge of the plane

So far, the y axis has always been pointing upwards with respect to the camera view. This is because the default "up" vector is $[0,1,0]$ —i.e. the unit y vector.

- Under the *up vector* label, let's set the y value to 0. and the z value to 1. We see that the view has rotated, and the blue line of the z axis is now pointing upwards.

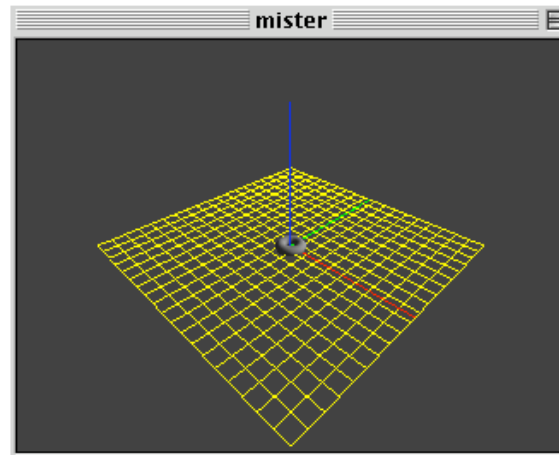


Using a different "up" vector

You may have noticed, as we've moved the camera further away from the origin, that the torus, plane, and axes have become smaller. This is because the default viewing mode uses a perspective projection, which is similar to the way we see things in the 3-dimensional world we inhabit. If you are familiar with camera lenses, you may also know that depending upon the angle of the lens, objects will be smaller as the lens angle increases to accommodate a larger field of view. Similarly, we can change the lens angle of our

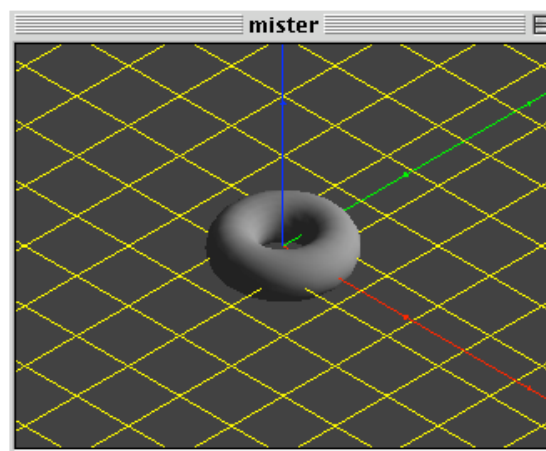
perspective transformation to increase the field of view, and in doing so the objects will become yet smaller.

- The default lens angle is 45 degrees, so let's change it to something like 60 degrees by changing the **number box** connected to the **message box** `lens_angle $1`.



Using a 60-degree lens angle

Another type of projection supported by OpenGL is the *orthographic projection*. This type of projection does not diminish the size of objects based on camera position. The orthographic projection is common to 3D CAD software used for tasks such as mechanical engineering. Many early video games like Q-Bert also used an orthographic projection. You can switch between the perspective projection and the orthographic projection by clicking on the **toggle box** labeled *orthographic projection*. The message `ortho 1` will turn on orthographic projection. If you try moving the camera with orthographic projection turned on, you should not see the objects become any smaller. However, changing the lens angle will still change the field of view, and the size of objects relative to the view.

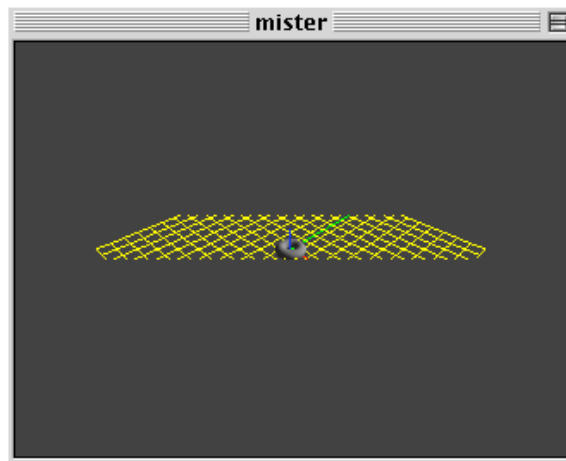


Viewing our scene using orthographic projection

- Click on the **toggle** again to turn off the orthographic projection with an `ortho 0` message.

Let's examine the *clipping planes* that determine the extent along the camera's view that will be rendered. OpenGL has a *near* clipping plane and a *far* clipping plane, and only geometry which lies in between these two planes will be rendered. These clipping planes are specified in units of distance from the camera along the viewing vector using the `clip_near` and `clip_far` messages. By default, the near clipping plane is set to 0.1 and the far clipping plane is set to 100.

- Try increasing the near clipping plane to 10 and decreasing the far clipping plane to 12. You should see the near and far edges of the yellow plane that fall outside the clipping planes disappear.

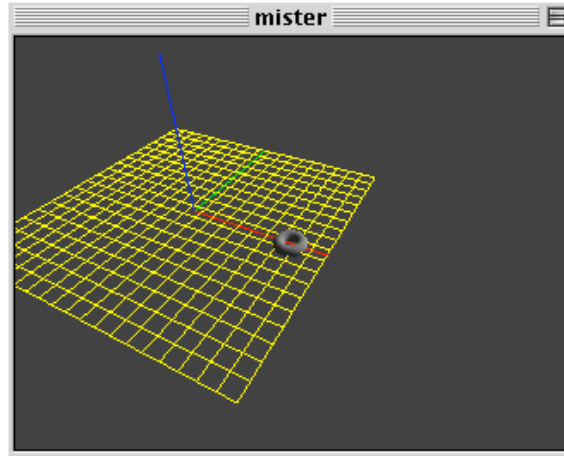


Using a more constrained clipping plane

- Set the near clipping plane back to the default of 0.1 and the far clipping plane back to the default of 100.

So far, the camera has always been looking at the origin $[0,0,0]$. If we change the `lookat` position's `x` value to 3., the camera is now looking at $[3,0,0]$.

- Let's move the torus to the position [3.,0.,0.], by clicking on the **message** box containing position 3. 0. 0. in the section of the patch labeled *UI Rotation and Position Control*. The torus is now again located at the center point of the view, [3.,0.,0.].



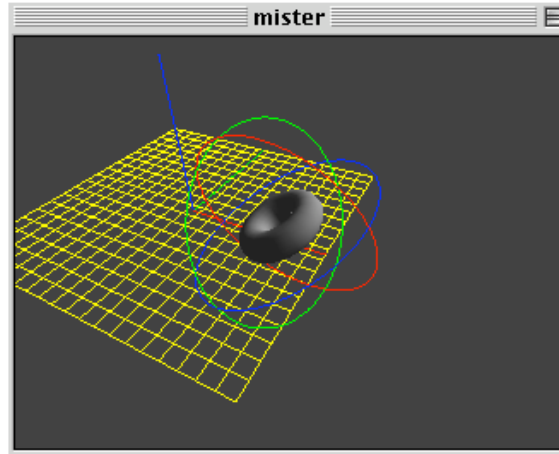
Changing the viewing position and the position of the shape

Not only did this send the position 3. 0. 0. message to the torus, but also to the **jit.gl.handle** object. The **jit.gl.handle** object is a GL group object that uses mouse information to move and rotate objects in the 3D scene. Like the **jit.gl.gridshape** object, the **jit.gl.handle** object requires a named draw context into which to draw. Unlike the **jit.gl.gridshape** object, it's also a user interface object that translates mouse activity in the draw context's destination to Max messages.

In this patch, messages from the **jit.gl.handle** object are sent to the **jit.gl.gridshape** object. They are also sent to the **route rotate position** object and formatted so you can see exactly what is being sent. These messages are the only communication from the **jit.gl.handle** object—there is nothing going on "behind the scenes."

If you click on the torus and drag the mouse, you will see the torus being rotated by the **jit.gl.handle** object as though it were a virtual trackball. If you hold down the command key while dragging, you can move the torus left, right, up, and down. If you hold the option key as you drag, you can move the torus towards you or away from you. Using the shift key as you perform any of the above mouse actions will constrain the action to a single axis.

- Try manipulating the orientation of the torus by clicking on it in the **jit.window** object. Get a feel for how the **jit.gl.handle** object translates the 2-dimensional mouse information into 3-dimensional rotation information.



*Using the **jit.gl.handle** object to manipulate the object's position*

As with the displayable axes of the **jit.gl.render** object, the **jit.gl.handle** object shows colored lines that correspond to the *x* (red), *y* (green), and *z* (blue) planes of the object being rotated. The lines appear as circles around the relevant object being "handled." The mouse controls the axes whose circles are nearest to the front of your current field of view. By manipulating the image so that those circles move to the back of the object, you can control a different pair of axes with the next mouse click. The modifier keys let you reposition the object by relocating it on the three axes. The **jit.gl.handle** object outputs the relevant messages to set the rotate and position attributes of the GL group object attached to it. Note that if you are displaying a GL context in a **jit.pwindow**, the **Help in Locked Patches** option of Max (which you can change under the Options menu) needs to be disabled in order for zooming to work using **jit.gl.handle**. Otherwise, the option key will cause the help patch for **jit.pwindow** to appear(!).

Summary

We have examined the several components which make up an OpenGL scene's camera view, and the necessary attributes of the **jit.gl.render** object which control them. The camera attribute specifies the camera position; up specifies the upwards vector; lookat specifies the position at which the camera is looking; ortho specifies whether to use an orthographic or perspective projection; and near_clip and far_clip specify the clipping planes. Lighting and smooth shading attributes can be enabled by setting the lighting_enable and smooth_shading

attributes of the GL group object handling the geometry (in this case the **jit.gl.gridshape** object).

The **jit.gl.handle** object lets us rotate and reposition OpenGL objects using the mouse and the modifier keys on the keyboard. The **jit.gl.handle** object takes the name of a valid draw context to attach itself to, and sends messages to any connected object that is also using that context, setting the rotation and position attributes of that object.

Tutorial 33: Polygon Modes, Colors and Blending

In the previous tutorial, you saw how to position the camera and objects in the GL group to construct an OpenGL scene in Jitter. After understanding this tutorial, you'll be able to hide selected polygons of an OpenGL object based on their spatial orientations, draw selected polygons in filled and wireframe modes, and add the results to the draw buffer using antialiasing and blending.

- Open the tutorial patch *33jPolyColorBlend.pat* in the Jitter Tutorial folder Click on the **toggle** box labeled "Start Rendering."

You should see a gray sphere in the **pwindow** in the tutorial patch. It is drawn by a **jit.gl.gridshape** object, connected to a **jit.gl.handle** object that allows you to control its rotation. The **jit.gl.handle** object's `auto_rotate` attribute is on, so once you rotate it the sphere will continue to rotate along the axis you set. If you like, give it a spin.

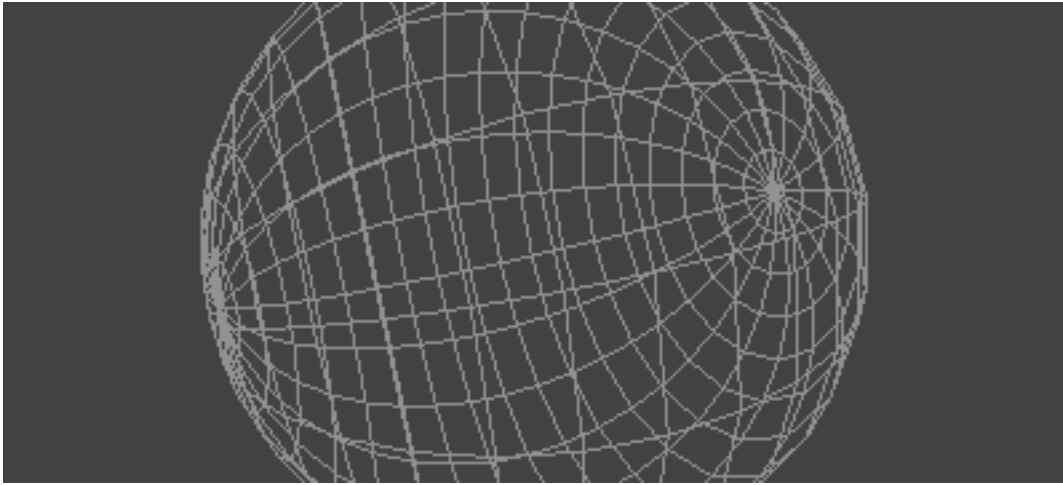


The gray sphere.

Wireframe Mode and Culling Faces

Just below the label "OpenGL Objects to Render" in the example patch is an object **pak** `poly_mode 1 1`. This object generates messages that set the polygon mode attribute of the **jit.gl.gridshape** object.

- Click both **toggle** objects on above the **pak poly_mode object**. You should see the gray sphere in wireframe mode.

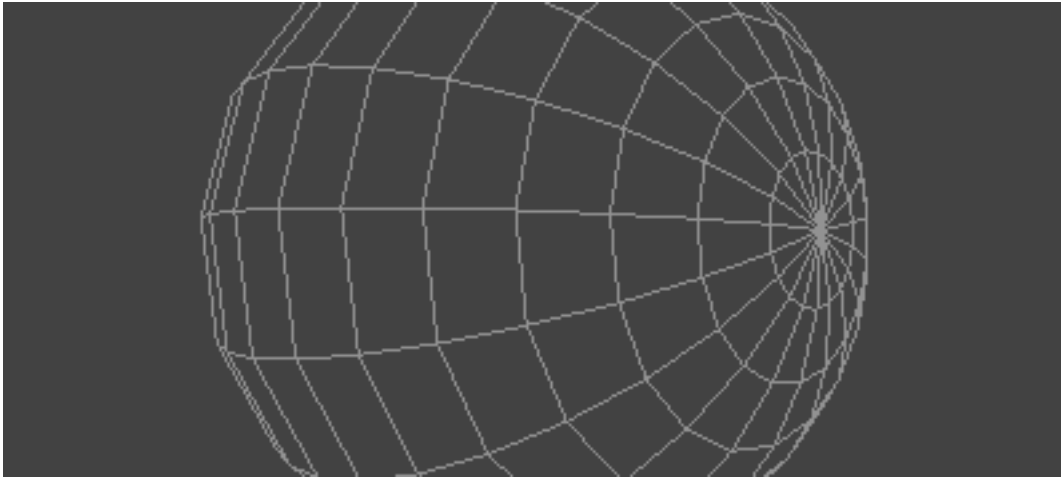


The sphere in wireframe mode.

Turning wireframe mode on allows you to see clearly that the **jit.gl.gridshape** object approximates a sphere using polygons—in this case, quadrilaterals. Every polygon drawn in OpenGL has a front side, defined as the side from which its vertices appear to wrap in a clockwise direction. Each polygon in a scene can, therefore, be categorized as either front-facing or back-facing, depending on whether its front side is pointed towards or away from the camera.

OpenGL can automatically hide either front-facing or back-facing polygons, which can serve to speed up drawing greatly or highlight certain aspects of data being visualized. In Jitter, you can control polygon visibility on an object-by-object basis using the `cull_face` attribute.

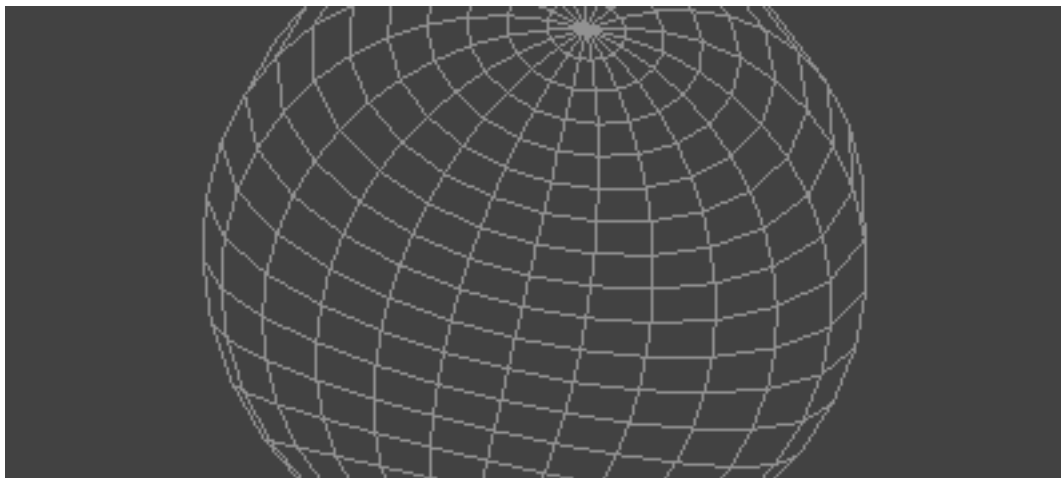
- Set the **number box** above the **prepend** cull_face object to 1.



The front-facing polygons of the sphere.

The cull_face attribute of a GL group object can be set to 0, 1 or 2. A setting of 0 shows all polygons in the object. A setting of 1 hides the back-facing polygons. A setting of 2 hides the front-facing polygons. With the current setting of 1, the wireframe sphere appears solid, because the *hidden lines*—polygon edges that would not be visible if the sphere were made of a solid material—are not drawn. The rotation (did you give it a spin?) convincingly depicts that of a real-world solid object.

- Set the number box above the **prepend** cull_face object to 2.



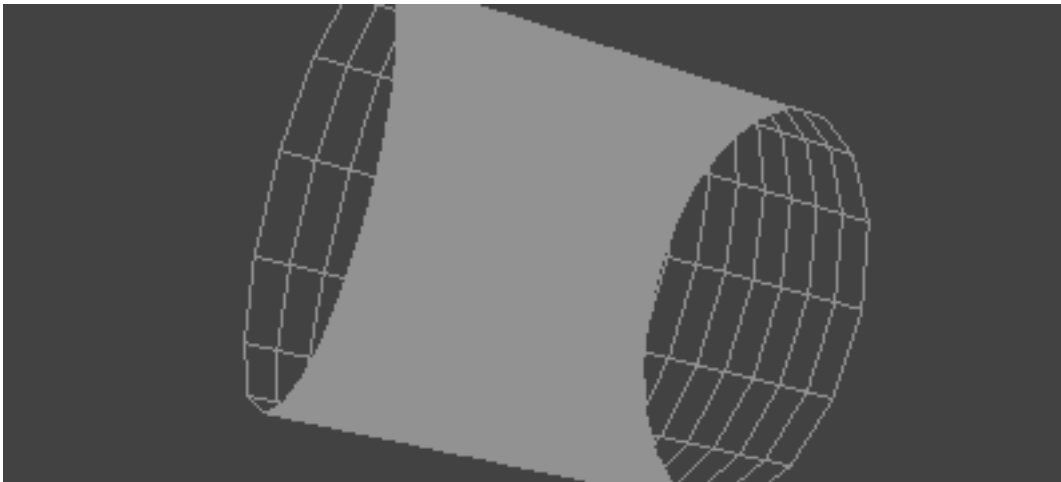
The back-facing polygons of the sphere.

Now the front-facing polygons are hidden. It's as if you are looking through the sphere and seeing only the inside part that faces towards you. You can see from this picture that

the perspective is somewhat strange, but watching the rotation in the patch makes it really obvious that the scene is not drawn “normally.”

In general, setting `cull_face 1` will do a good job of removing the polygons that should be hidden for solid, convex objects such as the sphere. But for objects that aren’t solid, a combination of the front-facing and back-facing polygons may be in view.

- Set the **number box** above the **prepend cull_face** object to 0 to show all the polygons. Set the left **toggle** box above the **pak poly_mode** object to 1 (on) and the right **toggle** box to 0 (off). Set the **ubumenu** above the **prepend shape** object to “opencylinder”.



Open cylinder, with solid front-facing polygons and wireframe back-facing polygons.

Using this new shape, we can see the distinction between front- and back-facing polygons. The message `poly_mode a b`, given two integers `a` and `b`, sets the front-facing polygons to wireframe mode if `a` equals 1, and the back-facing ones to wireframe if `b` equals 1.

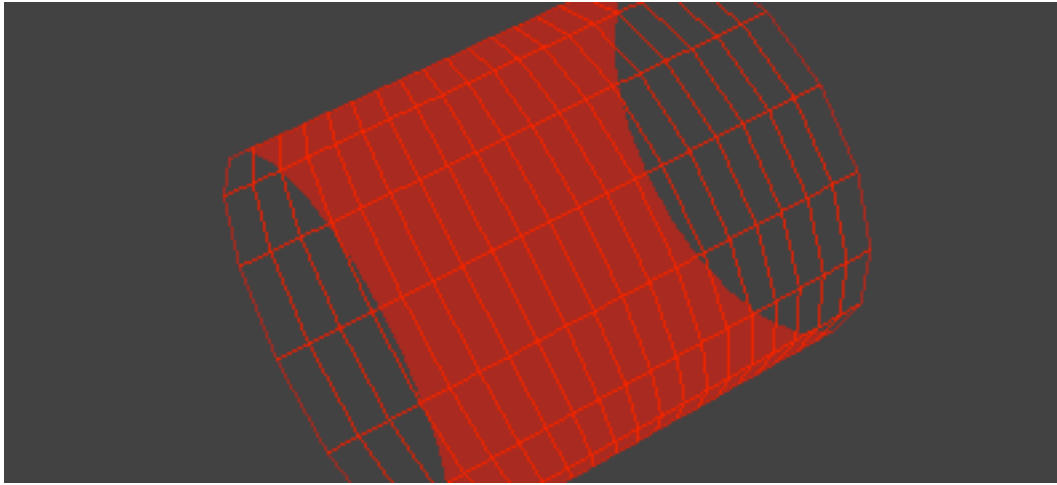
RGBA Colors

The colors of objects in the GL group are specified using the message `color R G B A`, where `R` is the red component of the color, `B` is blue, `G` is green, and `A` is the alpha or opacity component. All values range from 0 to 1.

- Set the **number box** objects above the **pak color...** object to the values 1,0,0,0.5. This specifies pure red at 50% opacity.

You will see the color of the cylinder turn red, but the opacity is not visible. This is because *blending*, the mixing of pixels with those previously in the draw buffer, is turned off by default.

- Click the **toggle** box above the `blend_enable $1` **message box** to turn blending on for the **jit.gl.gridshape** object. You should see something like this:



The red cylinder with blending enabled.

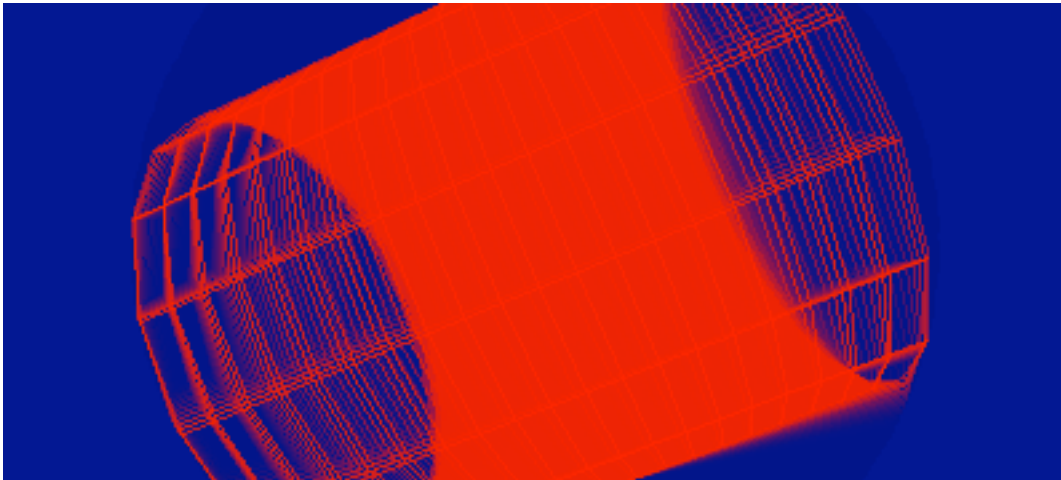
ARGB vs. RGBA If you've been using Jitter's video-manipulation objects, you know that colors in those objects are stored in planes and specified in arguments in the order A, R, G, B. In the GL group of objects, the order is RGBA, with alpha last, as we're seeing here. This may seem odd, so you are due a bit of explanation. Jitter objects are tied as closely to native formats as possible in both the OpenGL and video domains, to allow the fastest possible processing. OpenGL stores colors of objects and vertices in RGBA format, and QuickTime stores its images in ARGB format. So the Jitter objects reflect this. If you want to combine OpenGL and video matrix processing in your patch, the **pack**, **unpack**, **jit.pack** and **jit.unpack** objects provide an easy way to convert between the two systems. You can also convert a matrix of char values from ARGB to RGBA by sending the matrix through a **jit.matrix** object with the `planemap` attribute set to 1230 (effectively shifting all the planes by one). *Tutorial 6* shows more examples of using the `planemap` attribute of the **jit.matrix** object.

Erase Color and Trails

Right now, each time the **jit.gl.render** object receives the erase message, the draw buffer is filled with the dark gray that is the default erase color. You can set a different erase color using the RGBA number boxes above the renderer object.

- Set the number boxes above the **jit.gl.render** object to the values 0, 0, 0.5 and 0.1.

The background changes to a dark blue (Red = 0, Green = 0, Blue = 0.5). If the cylinder is spinning, you will also see trails in the image. This is because the background is being erased with an opacity of 0.1. When the dark blue pixels are drawn on top of the existing pixels in the draw buffer, they are overlaid such that the result is one-tenth dark blue and nine-tenths whatever color was at each pixel previously. As a result, previous frames linger for some time before being fully erased. Note that while the `blend_enable` attribute must be set in order to see draw colors with partial opacity, it is not necessary for erasing with partial opacity.



The spinning cylinder, leaving trails.

Blend Modes

When the `blend_enable` attribute of an object in the GL group is on, each pixel is applied to the draw buffer using a *blend function*. The blend function is the basic operation in image compositing. It controls the application of new pixels, the *source*, over existing pixels, the *destination*. The function has two parts: a source blending factor and a destination blending factor. The source blending factor specifies how the source's contribution to the finished image should be computed. The destination blending factor specifies the destination's contribution.

Jitter recognizes eleven possible modes for blending factors. Some can be applied only to the source, some only to the destination, and some to both. Each mode specifies a different set of multiplier values for red, green, blue and alpha. The message `blend_mode [src_factor] [dest_factor]` allows you to specify both factors for any of the drawing objects in the GL group.

The Blend Modes The source and destination blend factors are RGBA quadruplets which are multiplied componentwise by the RGBA values of the source and destination pixels, respectively. The corresponding components of the source and destination are added and then clamped to the range [0, 1] to produce the output pixel.

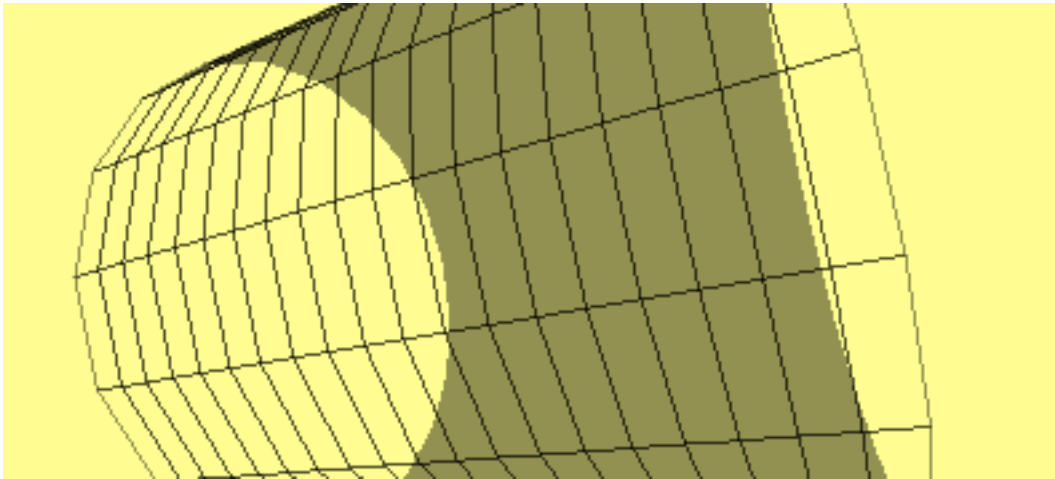
This table shows how the blend factors are calculated. The “Mode” column lists the number passed in the Jitter message `blend_mode [src_factor] [dest_factor]`. The “OpenGL name” column lists the name of the mode. The “Relevant to” column lists whether the mode can apply to the source factor, the destination factor, or both. Finally, the “Blend Factor Equation” column specifies the actual formula that is used to calculate the pixel. The subscripts *s* and *d* refer to source and destination components, respectively. For example, R_s refers to the red component of the source pixel.

Mode	OpenGL name	Relevant to	Blend Factor Equation
0	GL_ZERO	both	$(0, 0, 0, 0)$
1	GL_ONE	both	$(1, 1, 1, 1)$
2	GL_DST_COLOR	source	(R_d, G_d, B_d, A_d)
3	GL_SRC_COLOR	destination	(R_s, G_s, B_s, A_s)
4	GL_ONE_MINUS_DST_COLOR	source	$(1, 1, 1, 1) - (R_d, G_d, B_d, A_d)$
5	GL_ONE_MINUS_SRC_COLOR	destination	$(1, 1, 1, 1) - (R_s, G_s, B_s, A_s)$
6	GL_SRC_ALPHA	both	(A_s, A_s, A_s, A_s)
7	GL_ONE_MINUS_SRC_ALPHA	both	$(1, 1, 1, 1) - (A_s, A_s, A_s, A_s)$
8	GL_DST_ALPHA	both	(A_d, A_d, A_d, A_d)
9	GL_ONE_MINUS_DST_ALPHA	both	$(1, 1, 1, 1) - (A_d, A_d, A_d, A_d)$
10	GL_SRC_ALPHA_SATURATE	source	$(f, f, f, 1); f = \min(A_s, 1 - A_d)$

The default source and destination blend modes for all objects in the GL group are 6 and 7, respectively. These correspond to the GL blend factors `GL_SRC_ALPHA` and `GL_ONE_MINUS_SRC_ALPHA`. This is a very commonly used blending operation, and you may never need another one. It allows you to crossfade intuitively between the source and destination by changing the source’s alpha value.

Other values are useful for simulating various real-world lighting situations, as well as special effects that have no physical counterpart.

- Set the RGBA number boxes above the **jit.gl.render** object, which control the `erase_color` attribute, to the values 1.0, 1.0, 0.5 and 1.0. This will set the background color to yellow and remove the trails, making the blend effect more visible.
- Set the left and right number boxes above the object **pak** `blend_mode` to 0 and 7, respectively. This specifies a source blend factor of `GL_ZERO` and a destination blend factor of `GL_SRC_ALPHA`.



The cylinder with a setting of `blend_mode 0 7`.

Let's examine how this `blend_mode` produces the image we see here. The source factor is `GL_ZERO`. This means that all the components of the source pixel are multiplied by zero—the source pixel has no effect. You can verify this by trying different RGB values for the cylinder. They all produce the same colors.

The destination factor is `GL_SRC_ALPHA`. Looking in the table above, we can find the blend factor equation this corresponds to: (A_s, A_s, A_s, A_s) . Each component of the destination pixel is multiplied by the source's alpha, in this case 0.5, before being added to the source pixel times the source factor, which in this case is 0. So, each time a pixel is drawn, it has its brightness reduced by one half.

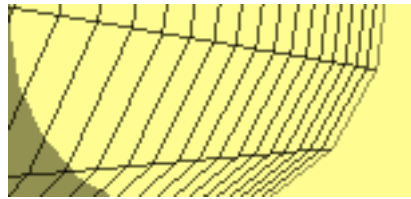
Antialiasing

When OpenGL draws polygons and lines, it approximates their ideal geometric shapes by filling in pixels on a raster grid. This process is prone to difficulties that parallel those in reconstructing ideal waveforms with digital audio. Inevitably, *aliasing*, spatial frequencies that are not included in the ideal image, will be introduced once it is reconstructed from

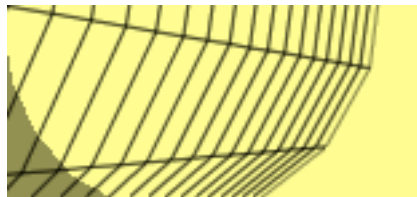
discrete pixels. This aliasing is visible as what are commonly known as “jaggies,” especially in near-horizontal or near-vertical lines or edges.

OpenGL has some antialiasing techniques for reducing the jaggies. We’ve made these available in Jitter through attributes of the GL group of objects. Using these attributes, you can specify whether any given object in the GL group will use antialiasing when it is drawn.

- Turn on the **toggle** box above the **message box** antialiasing \$1 to send the message antialias 1 to the **jit.gl.gridshape** object.



Antialiasing off



Antialiasing on

The antialiased lines have a smoother appearance, and also a fatter one. They may also draw more slowly, so if you’re concerned about drawing speed, you have to decide whether the improved appearance is worth the extra time.

The behavior of antialiasing in OpenGL is implementation-dependent. This means that makers of OpenGL hardware and drivers have some leeway in deciding what exactly happens when you request antialiasing. In the pictures above, for example, note that while the jaggies on the lines are reduced, the polygon edges in the lower left appear just the same. When you turn antialiasing on, Jitter requests that polygon edges be antialiased. But the particular OpenGL implementation that generated these pictures (an ATI Rage 128 accelerator with version 5.9.8 drivers) does not offer any help in this regard. Your implementation may differ.

Summary

We have defined front-facing and back-facing polygons, and seen how to draw them in both solid and wireframe modes (using the `poly_mode` and `cull_face` attributes). The

renderer's `erase_color` attribute and its use to draw trails have been introduced. We defined in detail what happens when a source pixel is applied to a draw buffer destination, taking opacity and blend modes into account. And finally, the handy if somewhat unpredictable antialiasing feature of OpenGL was introduced.

Tutorial 34: Using Textures

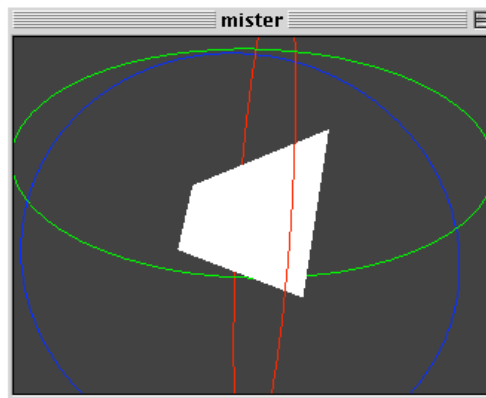
This tutorial shows you how to create and apply textures to 3D geometry data generated by the GL group. It will cover the creation of a named texture with **jit.gl.render**, assigning a named texture to a GL object, the use of colors in conjunction with textures, the conversion of image/video data to a texture, and various ways to wrap the geometry with a texture.

- Open the tutorial patch *34jUsingTextures.pat* in the Jitter Tutorial folder, and click on the **toggle** object labeled *Start Rendering*.

You will see a white parallelogram, but it is actually a tetrahedron being drawn by the **jit.gl.plato** object. The **jit.gl.plato** object is capable of rendering several platonic solids including tetrahedrons, hexahedrons (also known as cubes), octahedrons, dodecahedrons, and icosahedrons. Since lighting is not turned on and there is no texture being applied to the tetrahedron, it is difficult to tell that it is actually a 3D shape.

- Use the mouse to rotate the tetrahedron with the **jit.gl.handle** object, as covered in *Tutorial 32*.

This should illustrate that it is actually a 3D shape, but by applying a texture to the **jit.gl.plato** object, this will become even more apparent.



Rotating the platonic solid.

What is a Texture?

A texture is essentially an image that is overlaid upon geometry. Just like other images in Jitter, textures have an alpha, red, green, and blue component. In order to make use of the alpha component, blending must be enabled. Blending is covered in detail in *Tutorial 33*.

In Jitter, a texture has a name and belongs to the **jit.gl.render** object. Other objects that are attached to the drawing context associated with a given **jit.gl.render** object may make use of any of the named textures owned by that **jit.gl.render** object.

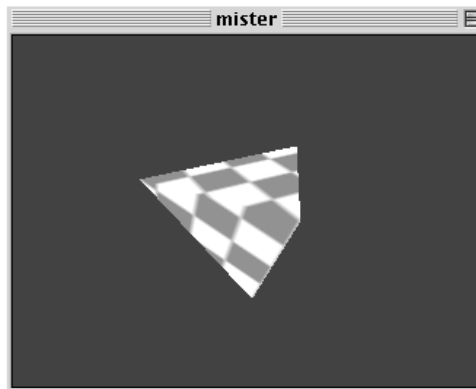
For efficiency reasons, most OpenGL implementations require that the dimensions of texture data must be integer powers of two—16, 32, 64, and so on. While there are some implementations that permit arbitrarily sized textures, this is not supported in the current version of Jitter. There is a minimum size of 1 by 1 and a maximum size dependent upon the OpenGL implementation, usually 1024 by 1024.

Creating a Texture

- Create a texture named **grid** by clicking the **message** box labeled **texture grid 64 64** in the section of the patch labeled *Textures*. This message is being sent to the **jit.gl.render** object.

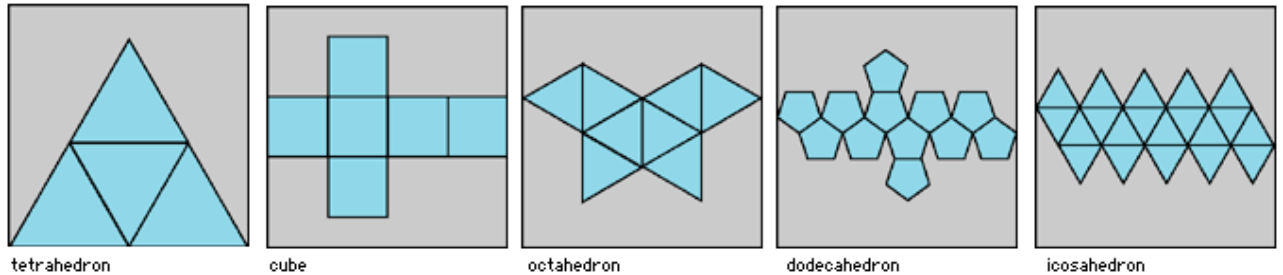
The first argument to the texture message is the texture name, and the two following arguments specify the width and height of the texture. This creates a 64 by 64 texture named **grid**, and fills it with the default pattern (a white and grey checkerboard). You will not see any of the results yet, because the texture has not yet been applied to the geometry.

- Apply the texture to the tetrahedron by clicking on the **message** box labeled **texture grid** in the section of the patch labeled *Platonic Solid*. This sets the **jit.gl.plato** object's texture attribute, and when drawing, it will use the texture named **grid**. You should now see a checkered tetrahedron.



Tetrahedron with a checkerboard texture applied to it.

The **jit.gl.plato** object uses a "gift-wrapping" strategy to apply the texture to the tetrahedron. In the **jit.gl.plato** help file, the *texture_maps* subpatch illustrates exactly how the different platonic solids are wrapped.



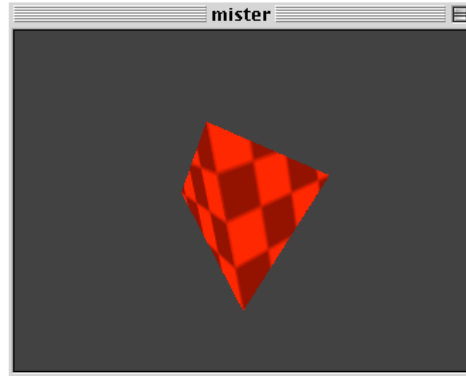
How texture maps are applied to different platonic solids.

Important Note: You will notice that both the **jit.gl.render** and **jit.gl.plato** objects use the texture message in different ways. The **jit.gl.render** object uses this message to *create* a texture, while the **jit.gl.plato** and other GL objects use this message to *apply* a texture. The **jit.gl.render** object also has a message to apply one of its named textures to raw geometry data passed as Jitter matrices. That message is *usetexture*. The **jit.gl.render** object's ability to render raw geometry data passed as Jitter matrices will be covered in *Tutorial 37*.

Textures and Color

When applying a texture to geometry, OpenGL also takes into account color and lighting information, so the current color and lighting values will be multiplied with the texture image when drawn. If the color is white and lighting is turned off, the texture colors will be unaltered.

- In the section of the patch labeled *Platonic Solid*, set the color of the tetrahedron to red by setting the **number box** labeled *red* to 1, the **number box** labeled *green* to 0, and the **number box** labeled *blue* to 0.



Manipulating the color of the rendered object.

- Set the color of the tetrahedron back to white (1. 1. 1.) for our next section.

Converting an Image or Video to a Texture

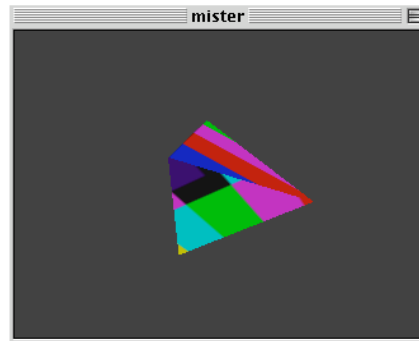
While illustrative, typically you will want to make use of textures other than the default grey and white checkerboard. This can be accomplished by loading an image or a movie into the **jit.qt.movie** or **jit.matrix** objects and sending the message texture [texture-name] jit_matrix [matrix-name] to the **jit.gl.render** object. If the texture specified by [texture-name] already exists, the incoming matrix will be resampled to the current dimensions of the texture. If no texture with that name exists, a new texture will be created. Its dimensions will be the nearest power of two greater than or equal to the dimensions of the Jitter matrix.

- Click the message box containing texture picture 128 128 in the section of the patch labeled *Textures*. This creates a 128 by 128 texture named picture, and like before, fills it with the default white and grey checkerboard pattern.
- Click the **message** box containing read colorbars.pict, bang to load the *colorbars.pict* image into the **jit.qt.movie** object, and send it on its way to the texture named picture.

You still won't see any of the results yet, because the **jit.gl.plato** object is still using the texture named grid.

- Click the message box containing texture picture in the section of the patch labeled *Platonic Solid*.

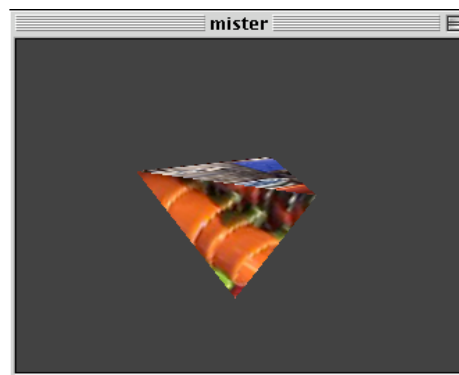
Now you should see the colorbars image wrapped around the tetrahedron.



Using an image as a texture.

In many instances you will only need to use still images as textures, but Jitter also supports the use of moving video as textures by repeatedly copying the output of the **jit.qt.movie** object into the named texture.

- Click the **message** box containing read dishes.mov to load *dishes.mov* into the **jit.qt.movie** object.
- Click on the **toggle** object connected to the **metro** object to start copying the video to the texture named picture.



Texture-mapping using a movie.

Interpolation and Texture size

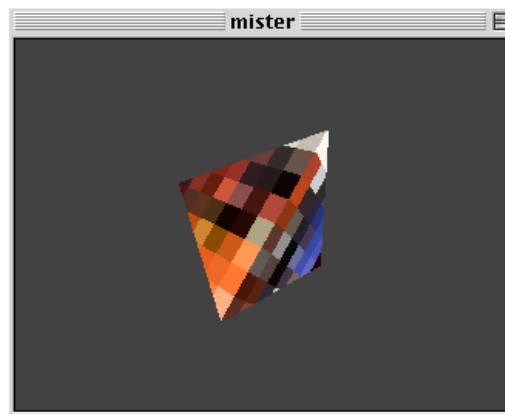
By default, texture interpolation is turned on, so screen pixels which are between texture pixels will use an interpolated value of its neighbors within the texture image. This has the effect of blurring or smoothing out the texture. To apply textures without interpolation, the interpolation may be turned off using the **jit.gl.render** object's **interp** message. This message only affects the **jit.gl.render** object's current texture, so prior to sending the **interp**

message, you can send the message `usetexture [texture-name]` to make the current texture the one specified by `[texture-name]` argument.

- Click the **toggle** object connected to the **message** box containing `usetexture picture, interp $1` to have the **jit.gl.render** object use the texture named `picture` and then to turn interpolation on and off for the texture named `picture`.

Once a texture has been created, the texture size can be changed, by sending the message `texture [texture-name] [width] [height]` where `[width]` and `[height]` specify the new dimensions.

- Set the **number box** labeled *Resize texture* to 16. This will send the **jit.gl.render** object the message `texture picture 16 16`, resizing the texture `picture` to be a 16 by 16 image.



Using an uninterpolated texture

Textures may be deleted in order to free up memory by sending **jit.gl.render** the message `deletetexture [texture-name]`.

Mapping Modes

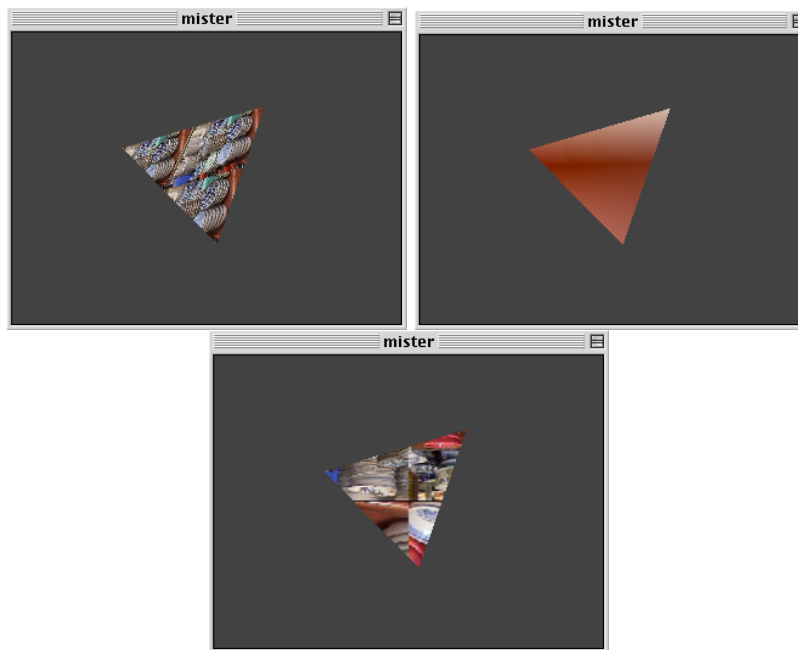
So far we have only addressed the explicit texture mapping that the **jit.gl.plato** object provides. OpenGL also provides a few other *implicit* texture mappings for applying textures to geometry data. These are the *object linear*, *eye linear*, and *sphere map* mapping modes.

The *object linear* mode applies the texture in a fixed manner relative to the object's coordinate system. As the object is rotated and positioned in the 3D scene, the texture mapping remains the same. In contrast, the *eye linear* mode applies the texture in a fixed manner relative to the eye's coordinate system. As the object is rotated and positioned in the 3D scene, the application of the texture to the object will change. Lastly, the *sphere map* mapping mode will produce the effect commonly called "environment mapping"; the object is rendered as though it is reflecting the surrounding environment, and assumes

that the texture contains a sphere mapped image of the surrounding environment. As the object is rotated and positioned in the 3D scene, the application of the texture to the object will change.

These implicit mapping modes may be used by setting the GL group `tex_map` attribute. A `tex_map` value of 0 is the default and will use the GL object's explicit texture coordinates. A `tex_map` value of `<m>1</m>` will use OpenGL's *object linear* mode. A `tex_map` value of 2 will use OpenGL's *sphere map* mode. A `tex_map` value of 3 will use OpenGL's *eye linear* mode.

- Try changing the **number box** connected to the message box containing `tex_map $1`. Position and rotate the tetrahedron with your mouse, and see how the various modes affect the texture mapping.



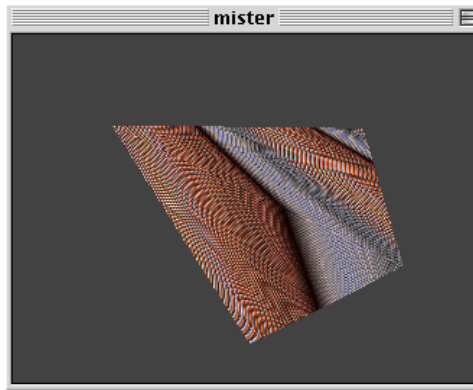
Using different implicit mapping modes: *object linear* (left), *sphere map* (middle), *eye linear* (right).

OpenGL's *object linear* and *eye linear* mapping modes have additional parameters that affect the way in which they apply the texture. These are set with the GL group `tex_plane_s` and `tex_plane_t` attributes. These attributes are each vectors in 4 dimensional homogenous coordinates. The scalar product of `tex_plane_s` and a given point in 4 dimensional homogenous coordinates determines the horizontal position of the texture image to apply. The scalar product of `tex_plane_t` and a given point in 4 dimensional homogenous coordinates determines the vertical position of the texture image to apply. By default, `tex_plane_s` is equal to (1. 0. 0. 0.) and `tex_plane_t` is equal to (0. 1. 0. 0.).

A more detailed description of how the `tex_plane_s` and `tex_plane_t` attributes affect the texture mapping are out of the scope of this tutorial, but that doesn't mean you can't play

with it anyway to generate interesting effects. For the curious, please consult the OpenGL *Red Book* or *Blue Book*.

- Experiment with the **number box** objects under the *texture plane s* and *texture plane t* labels (`tex_map` should be set to 1 or 3 in order to have any effect).



Experimenting with additional parameters.

Summary

We have established how to create textures and various ways to apply them to the geometry created by the GL group. The **jit.gl.render** object's texture message may be used to create, size, and copy image or video data to named textures. The GL group texture attribute specifies which named texture to use, and the GL group `tex_map` attribute selects either the explicit texture mapping mode or one of the three implicit OpenGL texture mapping modes: *object linear*, *eye linear*, or *sphere map*.

Tutorial 35: Lighting and Fog

Lighting—the illumination of objects in the real world—is a very complex subject. When we view objects, our eyes focus and detect photons throughout a range of energies that we call visible light. Between their origin in the Sun, lightning, fireflies or other light sources and our eyes, they can travel on a multitude of complex paths as they are reflected from or refracted through different materials, or scattered by the atmosphere. Our computers won't be dealing with this level of complexity anytime soon, so OpenGL simplifies it greatly.

The OpenGL Lighting Model

Lighting in OpenGL is based on a very rough model of what happens in the real world. Though very crude compared to the subtlety of nature, it is a good compromise, given today's technology, between the desire for realism and the cost of complexity.

We have already seen how colors in OpenGL are described as RGB (red, green and blue) values. The lighting model in OpenGL extends this idea to allow the specification of light in terms of various independent *components*, each described as RGB triples. Each component describes how light that's been scattered in a certain way is colored. The continuum of possible real-world paths is simplified into four components, listed here from most directional to least directional.

The *specular* light component is light that comes from a certain direction, and which also reflects off of surfaces primarily in a given direction. Shiny materials have a predominant specular component. A focused beam of light bouncing off of a mirror would be a situation where the specular component dominates.

Diffuse light comes from one direction, but scatters equally in all directions as it bounces off of a surface. If the surface is facing directly at the light source, the light radiation it receives from the source will be the greatest, and so the diffuse component reflected from the surface will be brightest. If the surface is pointing in another direction, it will present a smaller cross-section towards the light source, and so the diffuse component will be smaller.

Ambient light is direction-less. It is light that has been scattered so much that its source direction is indeterminate. So it appears equally bright from all directions. A room with white walls would be an environment with a high ambient lighting component, because so many photons bounce from wall to wall, scattering as they do so, before reaching your eye.

Finally, *emissive* lighting is another component that doesn't really fall anywhere on the directionality scale. It is light that only reaches your eye if you're looking directly at the object. This is used for modeling objects that are light sources themselves.

These components are used to describe *materials* and *light sources*. Materials made of specular, diffuse, ambient and emissive *components* are applied to polygons to determine how they will be colored. Polygons with materials applied are lit based on their positions and rotations relative to light sources and the camera, and the specular and diffuse properties of light sources, as well as the ambient light component of the scene.

Getting Started

- Open the tutorial patch *35LightingAndFog.pat* in the Jitter Tutorial folder. Click on the **toggle** (in the lower left, this time) labeled *Start Rendering*.
- Click the **message** box reading `name lt, depthbuffer 1` above the **jit.pwindow** object. This creates a depth buffer so that hidden-surface removal can be done.

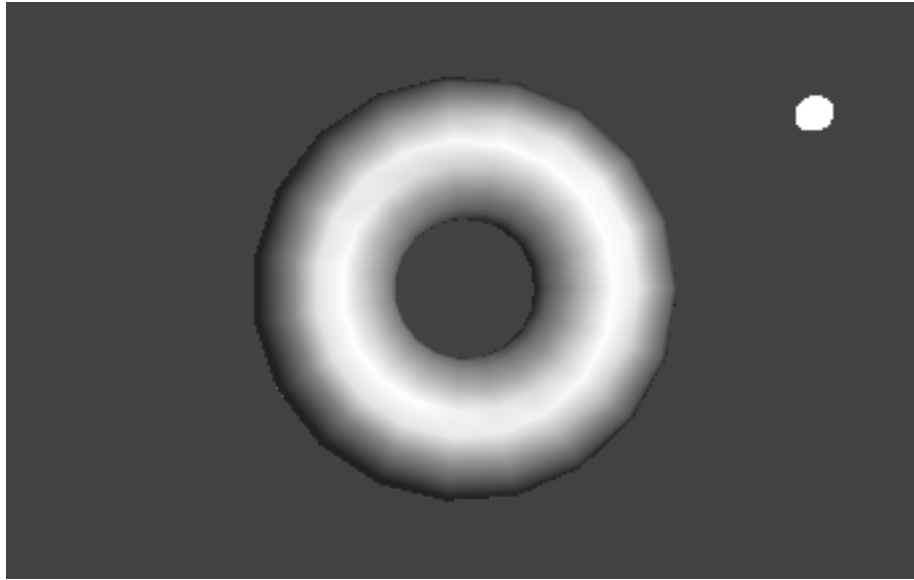
You will see a flat gray torus and a small white circle in the **jit.pwindow** object. The **jit.gl.gridshape** object in the center of the tutorial patch draws the torus. The other **jit.gl.gridshape** object in the patch, towards the right, draws the white circle. The scene is all presented in flat shades because lighting is not yet enabled.

- Click the **toggle** box objects above the top three **message box** objects reading `lighting_enable $1`, `smooth_shading $1`, and `auto_material $1` to set those attributes of the **jit.gl.gridshape** object drawing the torus to 1.

Lighting is off by default for each object, so you must enable it. The same goes for smooth shading. When you set these two attributes, you will see the torus go through the now-familiar progression (assuming you've been doing these tutorials in order) from flat shaded to solid faceted to solid and smooth. When you set the `auto_material` attribute to 1, you won't see any change, because its default value is already 1.

- Click the **toggle** objects above the **message** box labeled `auto_material $1` again, to set the `auto_material` attribute of the **jit.gl.gridshape** object to 0.

Now you should see a change in the lighting of the torus. Instead of the dull gray appearance it started with, you will see a shiny gray appearance like this:



The lit torus with the auto_material attribute off.

The `auto_material` attribute is provided in Jitter so that you don't always need to specify all the components of a material just to see an object lit. When the `auto_material` attribute of an object in the GL group is on and lighting is enabled for the object, the diffuse and ambient material components for the object will be set to the object's color, and the specular and emissive lighting components are disabled. This resulted in the flat gray torus we saw initially. In this tutorial, though, we want to see what happens when all the lighting components are specified explicitly, so we have turned the `auto_material` attribute off.

The image that results appears shinier, because the material applied to the torus now has a specular component.

Moving the Light

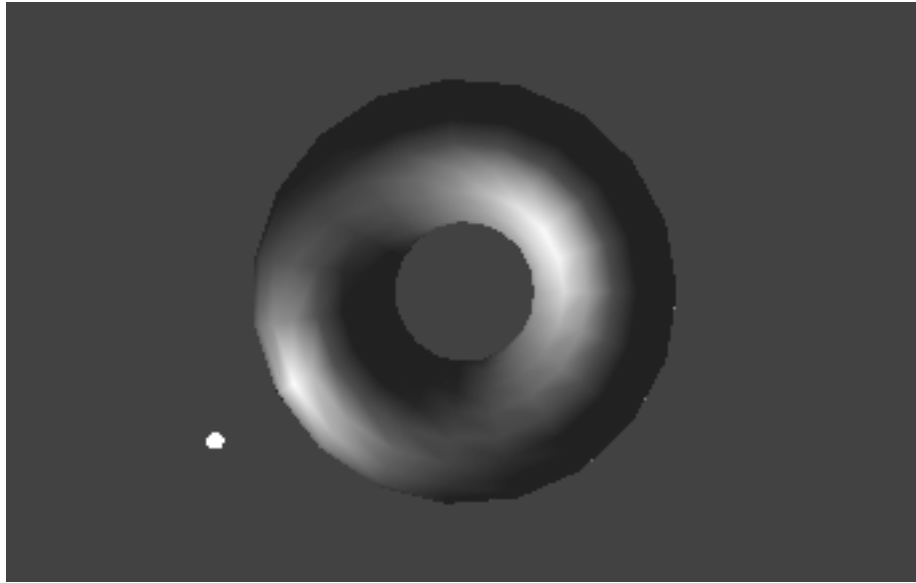
The `jit.gl.gridshape` object drawing the white circle has a `jit.gl.handle` object attached to it. As we've seen before this allows you to move the circle by clicking and dragging over it with the command key held down. By dragging with the option key held down, you can move the circle towards and away from the camera in the scene.

The x, y and z position values from the `jit.gl.handle` object are also routed to an `unpack` object, which sends them to `number box` objects for viewing. A `pak` object adds the symbol `light_position` to the beginning and another value to the end of the list. This message is sent to the `jit.gl.render` object to set the position of the light in the scene. Note that the light

itself is not visible, except in its effect on other objects. The white circle is a marker we can use to see the light's position.

- Click and drag the white circle with the command key held down to move it to the lower left corner of the scene. Then drag with the option key held down to move it away from the camera.

The light source moves with the white circle. If you move it to the right place, you can create an image like this.



The same scene with the light source moved.

The diffuse and specular components of the light are combined with the diffuse and specular components of the material at each vertex, according to the position of the light and the angle at which the light reflects toward the camera position. When you move the light, these relative positions change, so each vertex takes on a different color.

Normals: For lighting to take place, each vertex of a model must have a *normal* associated with it. The normal is a vector that is defined to be perpendicular to the surface of the object at the vertex. This is the value used to determine the contribution of the specular and diffuse lighting components to the image. Creating reasonable normals for a complex object can be a time-consuming process.

Jitter attempts to prevent you from worrying about this as much as possible, in a couple of ways. First, most objects in the GL group have normals associated with them. These are calculated by the object that does the drawing. The **jit.gl.gridshape** object is one example. If its shape is set to a sphere, it generates a normal at each vertex pointing outwards from the center of the sphere. Each shape has a different method of calculating normals, so that the surfaces of the various shapes are smooth where curved, yet the edges of shapes like the cube remain distinct and unsmoothed.

Secondly, if you send geometries directly to the **jit.gl.render** object in matrices, the **jit.gl.render** object will automatically create normals for you to the best of its ability. If you draw a connected grid geometry by sending a matrix followed by the `tri_grid` or `quad_grid` primitives, the generated normals will be smoothed across the surface of the grid. If you send a matrix using other primitives such as triangles, **jit.gl.render** will make no attempt to smooth the vertices, but will generate a normal for each distinct polygon in the geometry.

If you want to make your own normals, you can turn automatic normal generation off with by setting the attribute `auto_normals` of the **jit.gl.render** object to 0.

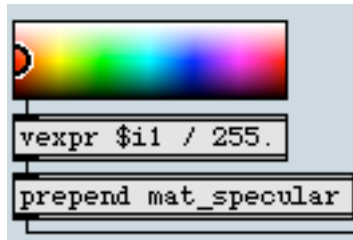
Tutorial37, "Geometry Under the Hood" describes how vertices, colors and normals can be passed in Jitter matrices to the **jit.gl.render** object. For more information on how to specify normals for geometry within a matrix, please refer to the *Jitter OpenGL Appendix* in this publication.

Specular Lighting

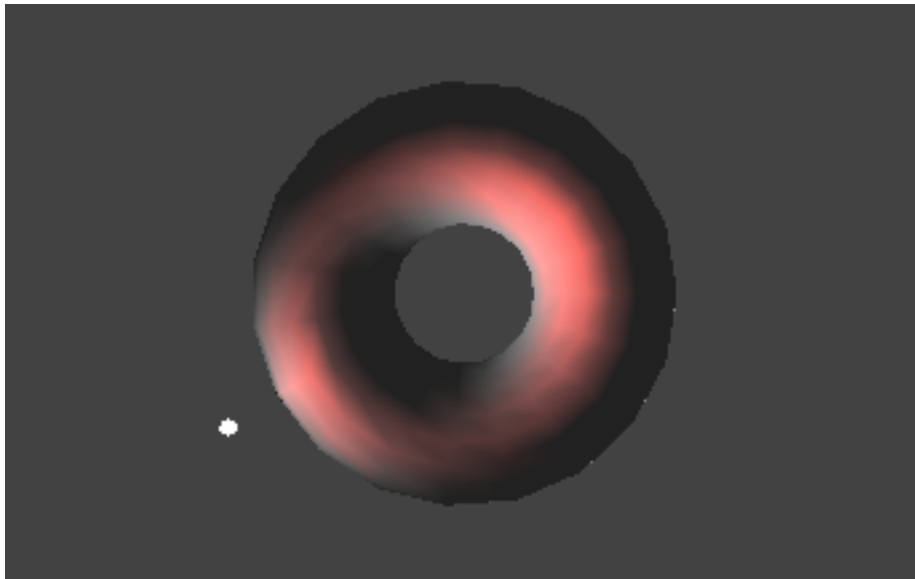
Let's change the specular components of the lighting to get a better feel for them.

- Locate the **swatch** object above the **prepend mat_specular** object. Move the circle in the swatch to the far left, centered vertically. This sends the `mat_specular` message to the **jit.gl.gridshape** object, followed by RGB color values that describe a pure red color.

The **swatch** object sends its output as a list of three integers from 0 to 255. The **vexpr** object divides each integer in the list by 255, to generate a floating-point value in the range 0.0 to 1.0, which is the range Jitter's OpenGL objects use to specify colors.



Setting the specular material component to red.



The resulting torus with red highlights.

The highlights of the image now have a red color. The specular component of the light source, which is currently white, is multiplied by the specular material component to produce the color of the highlights.

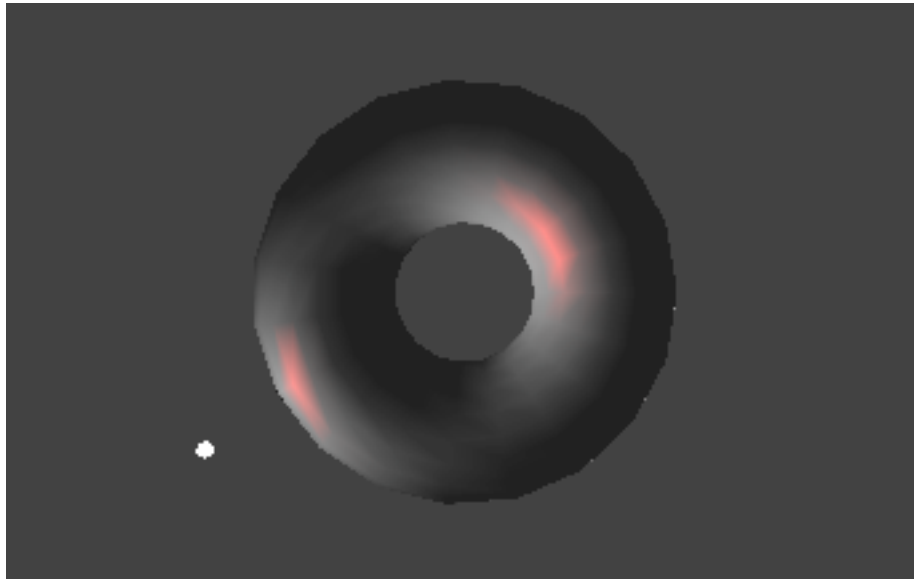
The red, green and blue values of the specular light source are multiplied by the red, green and blue values of the specular material component, respectively. So if the light source component and the material component are of different colors, it's possible that the result will be zero, and no highlights will be generated. This more or less models the real-world behavior of lights and materials: if you view a green object in a room with red light, the object will appear black.

- Try moving the circle in the **swatch** object above the **prepend** light_specular object to a green color. The highlights will disappear. Different colors with varying amounts of red will produce different brightnesses of red. When you are done, move the circle to the top of the swatch object so that the highlights are red again.

The shininess attribute of objects in the GL group is an important part of the material definition. It specifies to what extent light is diffused, or spread out, when it bounces off of the object. To model a mirror, you would use a very high shininess value. Values of approximately 2 to 50 are useful for making realistic objects.

- Set the **number box** above the **prepend** shininess object to 50.

This makes the contribution of the specular lighting components much smaller in area. Accordingly, you can see the specular lighting in red quite distinctly from the diffuse lighting, which is still gray.

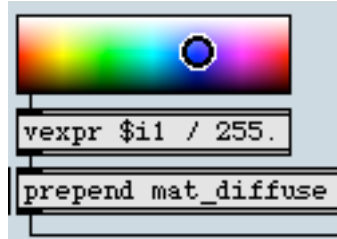


A very shiny torus.

Diffuse Lighting

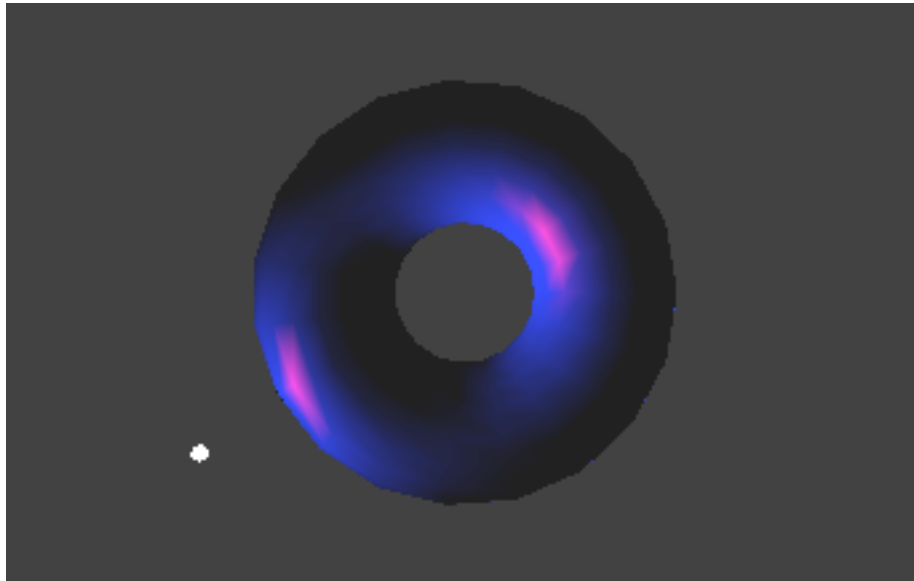
Let's manipulate the colors some more to see the effect of the diffuse component.

- Locate the **swatch** object above the **prepend mat_diffuse** object. Move the circle in the **swatch** approximately to the location in the illustration below, to produce a deep blue diffuse material component.



Setting the diffuse material to blue.

The diffuse reflections from the torus are now blue, and the highlights are magenta. This is because the color components of an object's material, after being multiplied with the lighting components depending on positions, are added together to produce the final color for the object at each vertex.

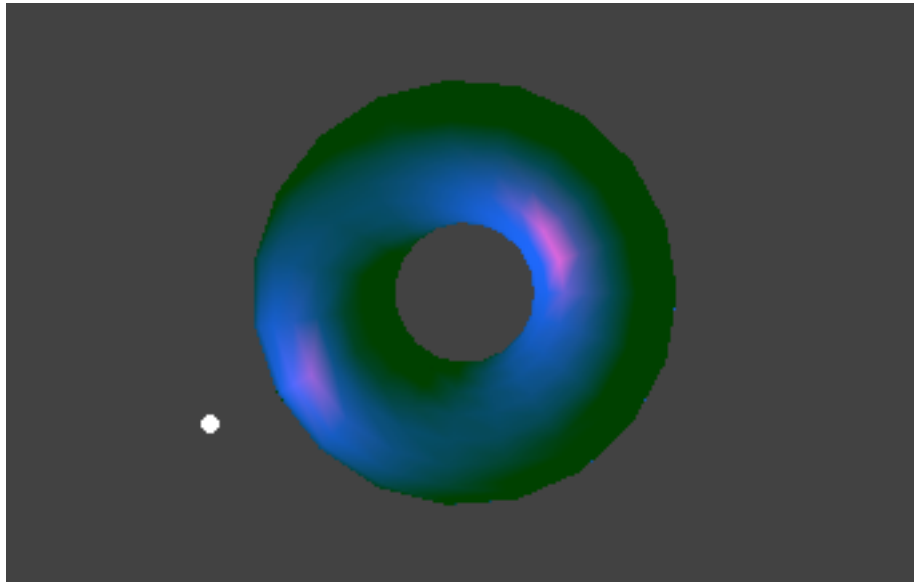


Red highlights added to blue diffuse lighting.

Ambient Lighting

We have yet to change the ambient component of the object's material. Currently, this is set to its default of medium gray, which is multiplied by the dark gray of the *global ambient component* to produce the dark gray areas of the torus in the picture above. The global ambient component is multiplied by all ambient material components in a scene, and added to each vertex of each object. You can set this with the `light_global_ambient` attribute of the **jit.gl.render** object.

- Set the circle in the **swatch** object above the **prepend** mat_ambient to a green color to produce a green ambient material component.



Green ambient illumination.

The ambient material component is multiplied by the global ambient illumination component to make the dark green areas that have replaced the gray ones.

The moveable light in the scene has an ambient component associated with it, which is added to the global ambient component. To change this, you can move the circle in the swatch object above the **prepend** light_ambient object. If you change this to a bright color, the whole object takes on a washed out appearance as the intensity of the ambient component gets higher than that of the diffuse component.

That's Ugly!

A green torus with blue diffuse lighting and magenta highlights is probably not something you want to look at for very long. If you haven't already, now might be a good time to play with the color swatches and come up with a more harmonious combination.

Note that control over saturation isn't provided in this patch for reasons of space. But it's certainly possible to specify less saturated colors for all the lighting and material components.

Directional vs. Positional Lighting

The moveable light in a scene can be either *directional* or *positional*. In the message `light_position [x] [y] [z] [w]` sent to **jit.gl.render**, the value `[w]` decides whether directional or

positional lighting is used. If w is zero, the light is a directional one, which means that the values x , y and z specify a direction vector from which the light is defined to come. If w is nonzero, the light is a positional one—it illuminates objects based on its particular location in the scene. The position of the light is specified by the homogeneous coordinates x/w , y/w and z/w .

Positional lights are good for simulating artificial light sources within the scene. Directional lights typically stand in for the Sun, which is so far away that moving objects within the scene doesn't change the angle of the lighting perceptibly.

- Turn on the **toggle** above the **p mover** subpatch to start the torus moving towards and away from the camera.

Notice how the lighting shifts across the surface of the torus as it moves, if the torus moves past the general vicinity of the light. You may have to move the light's position to see this clearly.

- To see the effects of directional lighting, change the last **number box** above the **pak light_position** 0.0.0.1. object to 1, then back to 0.

Now, notice that because directional lighting is on, the lighting no longer shifts when the object changes its position.

Fog

Like other aspects of lighting, the simulation of fog in OpenGL is primitive compared to the real-world phenomenon. Yet, it offers a convenient way to a richer character to an image. OpenGL fog simply blends the color of the fog with the color at each vertex after lighting calculations are complete, in an amount that increases with the object's distance from the camera. So faraway objects disappear into the fog.

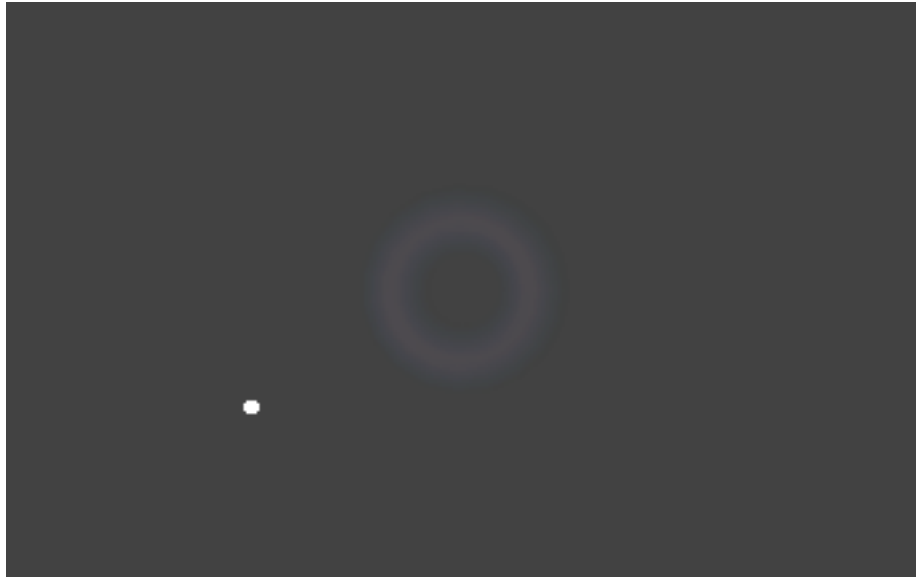
In Jitter, fog can be turned on or off for each object in the GL group by using the fog attribute. Some objects in a scene can have fog applied to them while others don't.

- Turn on the **toggle** above the “fog \$1” **message box**, which turns fog on for the **jit.gl.gridshape** object drawing the torus.
- Set the rightmost **number box** above the “**pak fog_params...**” object to the value of 10, which will send all the fog parameters listed in the same **jit.gl.gridshape** object.

You should see the torus receding into the fog as it gets farther from the camera, assuming the “**p mover**” subpatch is still active.

- Set the red **number box** above the “**pak fog_params**” object to 1. This specifies a fog color of 1., 0.2, 0.2.

Now, when the torus gets farther away, it doesn’t disappear. Rather, it turns bright red. Fog makes faraway objects tend toward the fog color, which may or may not be equal to the background color. Only if the fog color and the color of the background are nearly equal will realistic fog effects be achieved.



Torus receding into the fog.

If you like, try manipulating the other parameters of the fog with the **number box** objects above the **fog_params** message.

Implementation Dependent Like antialiasing, which was introduced in *Tutorial 33*, the effects of fog parameters may vary from system to system, depending on what OpenGL renderer is being used. The basic characteristics of fog discussed above should be basically the same, but details such as how the density parameter affects the fog may vary.

Summary

We have described OpenGL’s lighting model and its implementation in Jitter in some detail. We discussed the specular, diffuse and ambient components of the GL lighting model, how they approximate different aspects of a real world scene, and how they combine to make an image. The distinction between positional and directional lighting was introduced. Finally, we saw how to add fog to a scene on an object-by-object basis.

Tutorial 36: 3D Models

In this tutorial, we will show how to load 3D models using the **jit.gl.model** object and render them in Jitter with lighting, materials and texture mapping.

Review and Setup

- Open the tutorial patch *36j3Dmodels.pat* in the folder “Tutorial 36” in the Jitter Tutorial folder. Click on the **toggle** labeled “Start Rendering.”

You will see a brown background in the **jit.pwindow** object. The brown color comes from the `erase_color` attribute of the **jit.gl.render** object.

- Click on the message box reading “name ml, depthbuffer 1” above the **jit.pwindow** object.

The name attribute allows the **jit.pwindow** object to be used as a drawing destination, although this was already done and saved with the tutorial patch in this case. The message `depthbuffer 1` causes a depth buffer to be attached to the destination, so that automatic hidden-surface removal can be done.

- Click on the message box reading “texture grid 64 64” above the **jit.pwindow** object.

As we saw in the tutorial introducing textures, this message send to the **jit.gl.render** object causes it to build a texture 64 pixels in both width and height, filled in with a default checkerboard pattern. The texture can be used by any objects in the GL group which are drawing to its context. We are now ready to load some models and render them.

Reading a Model File

The **jit.gl.model** object reads descriptions of 3D objects as collections of polygons. These descriptions are stored in the .obj file format. The model files must have a .obj extension in order to load. This format is widely used, and has the advantage of being readable and modifiable in ordinary text editors.

- Click on the **toggle** object above the “verbose \$1” message box in the upper left of the “Draw a Model” section of the patch.

This sends the message `verbose 1` to the **jit.gl.model** object, enabling verbose mode. When this mode is on, you will see information about the model file you are loading printed in the Max window. It’s not good to leave verbose mode on while you are rendering real-time graphics, but the messages can be very helpful while you are setting up and debugging a patch.

- Click on the message box containing the message “read mushrooms.obj”. This object can be found above the **jit.gl.model object**.

The file mushrooms.obj should have been installed into Max’s search path when Jitter was installed. If for some reason this file has been moved, you will see the message “• error: jit.gl.model: can’t find mushrooms.obj” in the Max window. If the file is found correctly, you will see the following in the Max window:

```
reading mushrooms.obj .....  
done.  
jit.gl.model: rebuilding  
groups: 11  
materials: 0  
triangles: 10560  
vertices: 6199  
texture coords: 5589  
normals: 6046  
636408 bytes used
```

When the file you specify after the read message is found and determined to be a valid model file, the message “reading [file].obj” is printed. Then the dots are printed out every so often while the file loads. Reading large models from text files can take quite a while, so it’s nice to see the dots appearing in order to know that something hasn’t gone wrong.

When the model is done loading, “jit.gl.model: rebuilding” is printed. Some operations besides reading a new model cause the model’s representation to be rebuilt internally—all of them print this message so you know what’s going on.

The “groups: 11” message tells you that there are eleven polygon groups in the model. A model need not consist of one like an apple or mushroom, it can be a group of things or an entire scene. Assigning different objects in the model to different polygon groups allows them to be manipulated independently in your patch.

The “materials:0” message tells you how many material definitions there are in the model file. A material definition can specify the ambient, diffuse and specular components of an object’s surface color, as we covered in Tutorial 35. A model file can have multiple material definitions which can be applied to one or more polygon groups in the model. This model, however, has none.

We can also see that this model consists of 10,560 triangles joining 6,199 vertices, 5,589 texture coordinates and 6,046 normals. There are fewer texture coordinates and normals than vertices by a bit, because these numbers are referenced in a big index when the model is read, and can be reused when the triangles are constructed.

Finally, we see that approximately 636 kilobytes of memory have been used by Jitter to store the model internally. Though memory is getting more plentiful all the time, this amount is not insignificant. Memory usage is definitely something to keep an eye on when debugging larger patches.

.obj Model File Compatibility There are many extensions to the basic .obj file format in existence. Some of these are NURBS and parametric surfaces. Right now, only vertices, polygons, groups, materials and texture coordinates are supported.

Model Attributes

If you followed the directions, you've been reading all this dry text about messages in the Max window, while eleven mushrooms hover tantalizingly in the tutorial patch. Now let's examine the attributes which affect how the model is rendered.

Lighting and Shading

We see the mushrooms in space, but their outlines are simply filled in with the medium gray which is the default color for objects in the GL group.



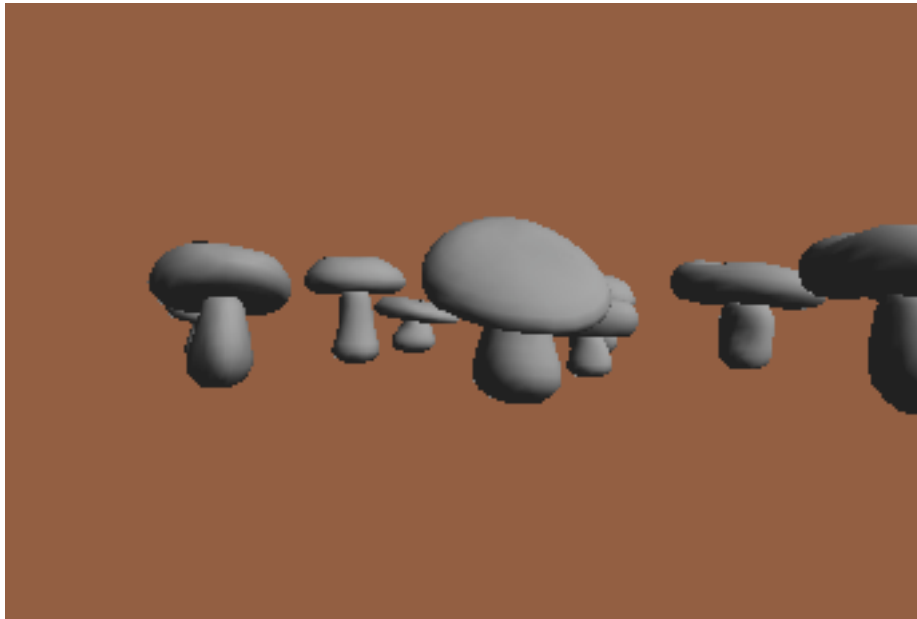
The flat-shaded mushrooms.

- Click on the toggle object above the message box reading "lighting_enable \$1" to send the message "lighting_enable 1" to the **jit.gl.model** object.

Now, the model has a more solid appearance because some polygonal areas are brighter and others darker. But each polygon has its own color uniformly over its entire area. The result is rather crude.

- Click on the toggle object above the message box reading “smooth_shading \$1” to send the message “smooth_shading 1” to the **jit.gl.model** object.

With smooth shading on, the shade of each polygon in the model is blended from vertex to vertex. This hides the polygon edges and gives a smoother appearance—we are beginning to get somewhere, if realism is our goal.



Mushrooms with lighting and smooth shading enabled.

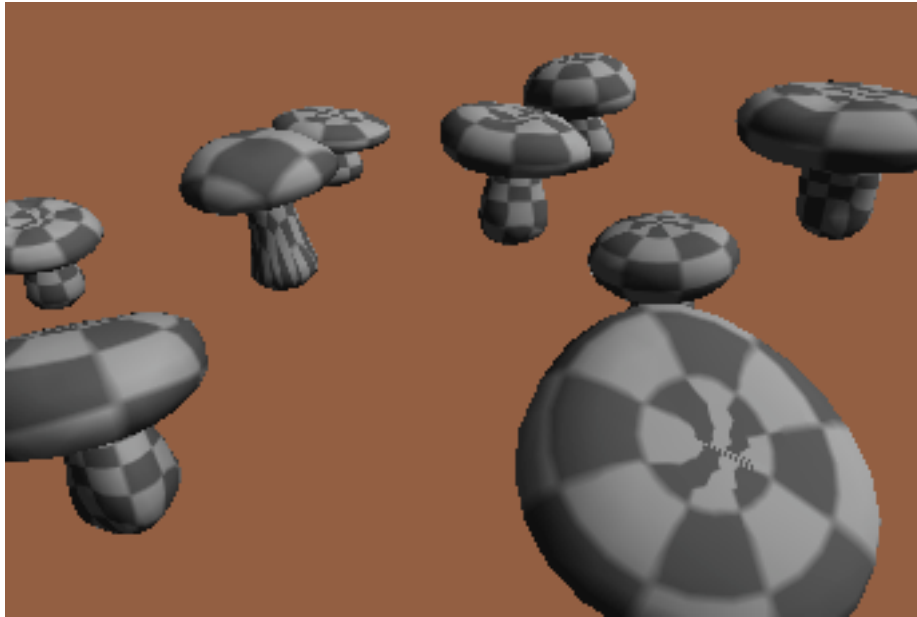
Texture Mapping

Like other objects in the GL group, you can set the texture applied to the entire model with the “texture” message.

- Click on the message box reading “grid” above the “prepend texture” object to send the message “texture grid” to the **jit.gl.model** object.

This sets the texture of the model. Since there is a **jit.gl.handle** object connected to the **jit.gl.model** object in this patch, we can rotate and move the model as discussed in Tutorial 32, in order to get a better look at how the texture is applied.

Notice the care with which the texture is wrapped around the mushrooms. The caps are covered radially, the stems cylindrically, and there are small variations in the look of each mushroom. These texture coordinates were created in a 3D modeling program by the maker of the model. The coordinates specify what point on the 2D texture to use at each vertex.



Mushrooms texture-mapped with care.

To see what the model looks like with automatically generated coordinates, you can use the “tex_map” message. As discussed in tutorial 34, tex_map modes of 1, 2 and 3 specify object linear, sphere map, and eye linear mappings respectively. “tex_map 0” uses the coordinates specified in the model.

- Experiment with these modes if you like, by changing the number box above the message box reading “tex_map \$1”. When you are done, restore the model to its built-in texture coordinates by setting the number box to 0.

In addition to the standard GL group facilities for applying textures, the **jit.gl.model** object can be used to apply a different texture to each polygon group in the model.

- Click on the message box reading “0” above the “prepend texture” object to turn off texturing for the model.
- Click on the message box reading “texgroup 1 grid”.

The `texgroup [group-number] [texture-name]` message sets the texture of polygon group [group-number] in the model to the texture [texture-name]. As a result of the message just sent, polygon group 1 should be set to the grid texture, and thus one of the mushrooms is grid-covered while the other ones remain solid gray. This message could be used in conjunction with the methods for updating textures covered in Tutorial 34 to apply a different picture or movie to each polygon group in the model.

If a `texgroup` message is received for a given polygon group, that group uses the texture named in the message, instead of any texture which may have been sent to the whole model using the “texture” message. If a `texgroup [group-number] 0` message is received, the named texture is no longer used for that group, and the group reverts to the model’s overall texture.

- Click on the message box reading “`texgroup 1 0`” to remove the grid texture from polygon group 1.

Drawing Groups

In addition to being texture differently, the polygon groups in a model can be drawn separately. You can control this drawing using the `drawgroup` message.

- Click on the bang object above the counter object on the right-hand side of the tutorial patch.

This increments the counter, which causes the message “`drawgroup 1`” to be sent to the **jit.gl.model** object. The polygon groups are indexed starting at the number 1. When the **jit.gl.model** object receives the message `drawgroup [n]`, it only draws polygon group n of the model until further notice. So now you should see a lone mushroom in the frame.

By clicking the bang object repeatedly, you can cycle through each mushroom in turn. You could use this technique to load a 3D animation, by storing one frame of the animated object in each polygon group of the model.

When it receives the message `drawgroup 0`, the **jit.gl.model** object turns the drawing of all polygon groups back on.

- Click on the message box reading “`drawgroup 0`” to turn all the polygon groups back on.

Material Modes

Models can contain material definitions, but this particular model doesn’t have any. Let’s load one that does.

- Click on the message box containing the text “read apple.obj” to load the apple model in the Tutorial folder.

This model is less complicated than the previous one, so it will load much faster. By reading the output of the **jit.gl.model** object in the Max window, we can see that it has 4,452 triangles. We can also see in the Max window that it has 3 materials. In the rendered image of the model, we can see the effects of these materials. Even though no textures have been applied, the apple is red with a brown stem.

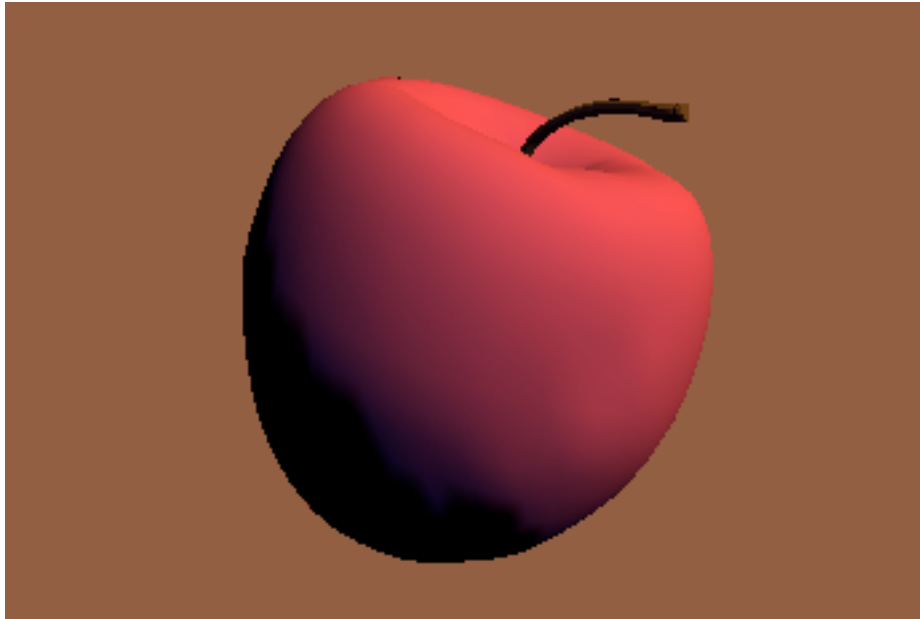


The apple model showing two materials.

The `material_mode` attribute of the **jit.gl.model** object determines to what extent the materials specified in the model file will be used to render the model. Right now the material mode is 1, the default.

- Set the number box above the message box reading “material_mode \$1” to the value 2.

You will notice that the skin of the apple takes on a shinier and slightly bluer appearance, while the stem remains the same flat brown. This is because the specular components of the materials are now being used in the rendering. When we first loaded the apple with the default setting for the material_mode attribute of 1, only the diffuse components of the materials were being used in the rendering.



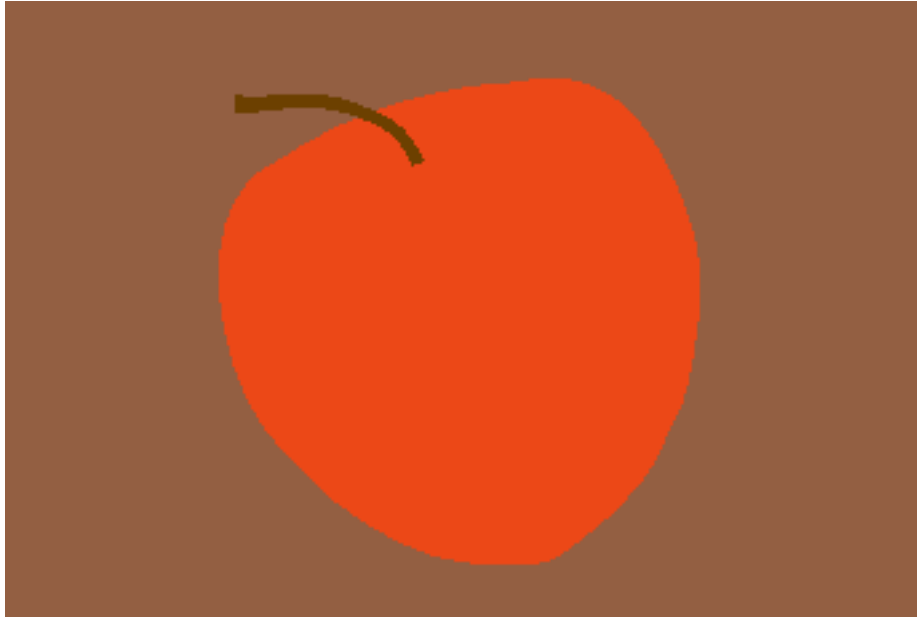
The apple with specular material components enabled.

- Set the same number box to the value 0.

Now, none of the material properties of the model are being used in the rendering. Instead, the color attribute of the object is used to determine the object's diffuse color. So, the model appears in the default gray. Let's try one more setting to see how the various material modes can be useful.

- Turn the toggle object above the "lighting_enable \$1" message box off. Then, try all the different settings for the material_mode attribute.

You will see that with lighting disabled, only in material mode 1 do the object's materials have any effect. Since the diffuse component of the material is being used to determine the object's color, the flat-shaded drawing takes on a different color for each polygon group.



The flat-shaded apple with a material mode of 1.

Here's a quick recap of the settings of the `material_mode` attribute:

- `material_mode 0:` don't use materials from model file. The object's color attribute determines the diffuse color, and the flat color if lighting is not enabled.
- `material_mode 1:` Use diffuse material component from model file to determine the diffuse color or flat color of the rendered model.
- `material_mode 2:` Use all material components from the model file.

Summary

We have seen how to load complex, multi-part 3D models from .obj format files on disk. The attributes of the **jit.gl.model** object can be used to affect the model's lighting and shading and to determine how its materials affect the rendering. The model file may contain different polygon groups which may be drawn and texture-mapped individually.

Thanks to Oliver Ffrench <<http://o.ffmpeg.free.fr/meshbank>> for the use of these elegant freeware models.

Tutorial 37: Geometry Under the Hood

This tutorial demonstrates the low-level support in Jitter to specify geometry data as matrices and render them with the **jit.gl.render** object. Since the data is contained in an ordinary Jitter matrix, this opens up a world of possibilities where arbitrary matrix operators may be used to generate and/or process the geometry matrices. The tutorial will cover the `matrixoutput` attribute of the GL group, the organization of data in geometry matrices, an example of how geometry matrices can be processed using matrix operators, and introduce various drawing primitives supported by the **jit.gl.render** object.

- Open the tutorial patch *37jGeometryUnderTheHood.pat* in the Jitter Tutorial folder, and click on the toggle object labeled *Start Rendering*.

Matrix Output

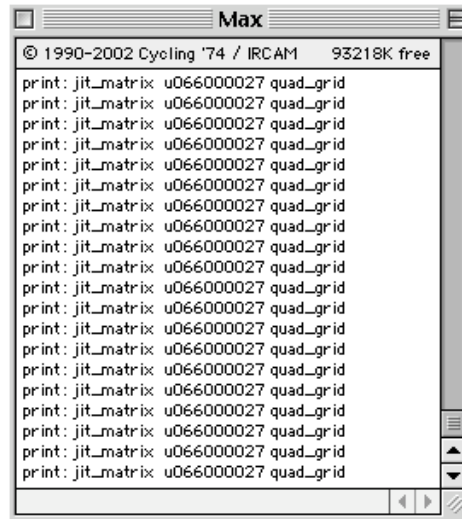
- Click on the **toggle** box labeled *Turn matrixoutput on/off*.

Wait a minute—the sphere that was just there a moment ago has disappeared. What's going on?

Some of the objects in the GL group support the `matrixoutput` attribute, and **jit.gl.gridshape** is one such object. This attribute determines whether or not the object's geometry is rendered directly in the object's associated drawing context or if the object sends a matrix containing geometry data out its left outlet. The geometry is not visible because it is being sent to a gate, which is closed.

- Select the menu item `print` from the **ubumenu** object labeled "Matrix Destination"

In the max window you should see a series of messages like "print: jit_matrix u26300000007 quad_grid". This is similar to what you've seen in previous tutorials as the output of objects, which pass matrix data, with the exception, that there is an extra element. Rather than sending the message `jit_matrix [matrix-name]`, as you should be familiar with, the **jit.gl.gridshape** object sends message `jit_matrix [matrix-name] [drawing-primitive]`. In this case the drawing primitive is `quad_grid`, which means interpret the matrix as a grid of quadrilaterals.



*The output of the **jit.gl.gridshape** object in the Max window*

At the time this tutorial was written, the only objects that support the `matrixoutput` attribute are the **jit.gl.gridshape**, **jit.gl.nurbs**, and **jit.gl.plato** objects. However, by the time you read this, there may be other objects, which support this mode of operation.

There is still no excitement in our window, but this is easily remedied.

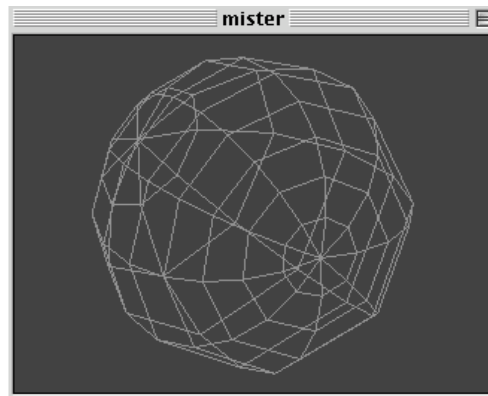
- Select the menu item direct from the **ubumenu** object labeled *Matrix Destination*.

The sphere is now visible again, because the message `jit_matrix [matrix-name] [drawing-primitive]` is being sent to the **jit.gl.render** object. In response to this message, the **jit.gl.render** object draws the supplied matrix using the specified drawing primitive. If no drawing primitive is specified, the **jit.gl.render** object's current drawing primitive will be used. Valid drawing primitives are: `points`, `lines`, `line_strip`, `line_loop`, `triangles`, `tri_strip`, `tri_fan`, `quads`, `quad_strip`, `polygon`, `tri_grid`, and `quad_grid`.

The **jit.gl.render** object's current drawing primitive may be set by sending the message `[drawing-primitive]` where `[drawing-primitive]` is any one of the valid drawing primitives mentioned above.

In the same fashion that you can change the dimensions of video data, you can change the dimensions of the matrix output by sending the **jit.gl.gridshape** object the `dim` message. By default the dimensions of the matrix output by the **jit.gl.gridshape** object is 20 by 20.

- Click on the **toggle** box labeled *Wireframe*.
- Change the **number box** labeled *Matrix Dimensions*.
- Try rotating the wireframe sphere with the mouse



“The Death Star plans are not in the main computer.”

You may have noticed that in this patch the **jit.gl.handle** object is communicating with the **jit.gl.render** object rather than the **jit.gl.gridshape** object. This has the effect of rotating and positioning the entire scene. The `@inherit_transform 1` argument is necessary in order to do this correctly, otherwise the rotation would be composited again with the scene's rotation, causing major confusion.

Geometry Matrix Details

Video in Jitter is typically represented by 4-plane char data, but how is the geometry data being represented?

Each vertex in the geometry is typically represented as float32 data with 3, 5, 8, 12, or 13 planes. Planes 0-2 specify the *x*, *y* and *z* position of the vertex. Planes 3 and 4 specify the texture co-ordinates *s* and *t*. Planes 5-7 specify the normal vector *nx*, *ny* and *nz* used to calculate the effects of lighting on the geometry. Planes 8-11 specify the *red*, *green*, *blue*, and *alpha* vertex color. Plane 12 specifies the edge flag *e*.

The output matrix of the **jit.gl.gridshape** object has 12 planes, but since we are not applying a texture to the geometry, and lighting is not enabled, the texture coordinates and normal vectors are ignored.

Processing the Geometry Matrix

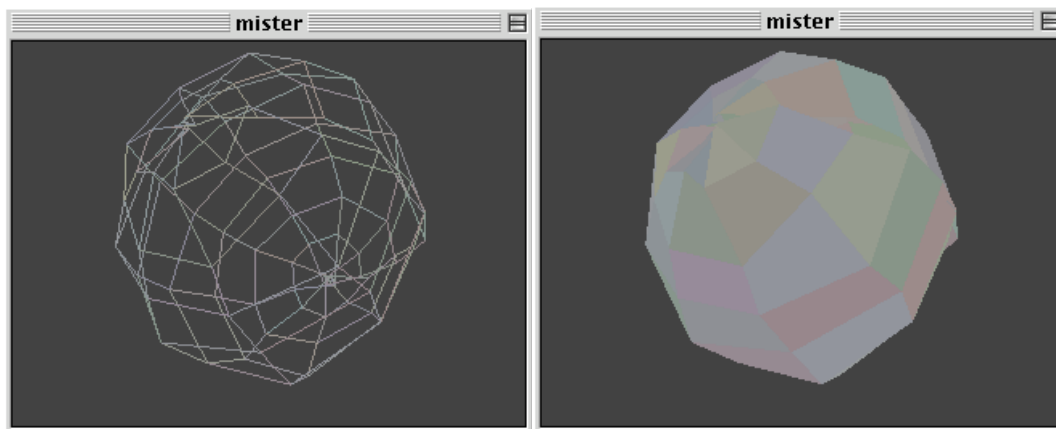
Instead of simply rendering the geometry unaltered, as you've done so far, it is possible to process this geometry with matrix operators.

- Select the menu item `xfade` from the **ubumenu** object labeled *Matrix Destination*.

Now the matrix is being sent through the **jit.xfade** object to cross fade the geometry matrix with a matrix of noise. Just as the **jit.xfade** object may be used to crossfade between video matrices, it may be used to crossfade between geometry matrices.

- Click on the **button** object labeled "Generate Noise"
- Gradually change the **number box** labeled "Crossfade Value" to 0.1.
- Turn wireframe rendering on and off by clicking the **toggle** box labeled *Wireframe*. Notice how the noise deforms both the geometry and the color of the shape.

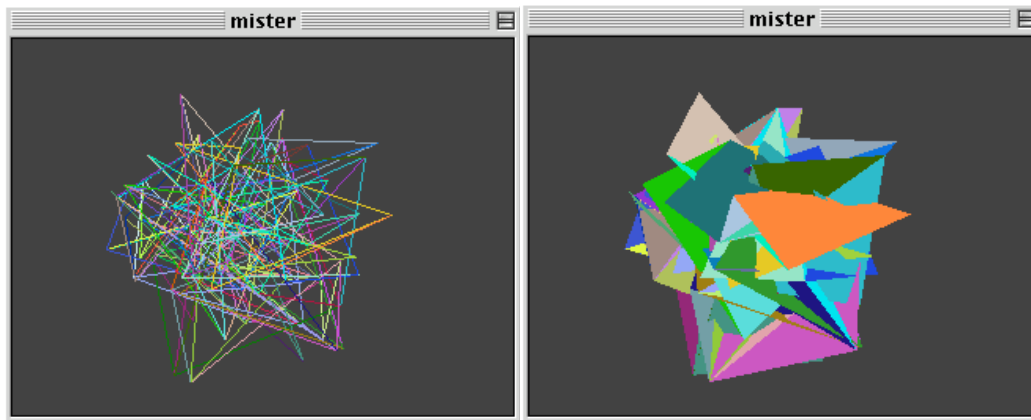
This will subtly deform the sphere with noise, as well as introduce random colors. The texture coordinates and normal vectors are also being affected, but this is not visible since there is no texture being applied, and lighting is not enabled.



A slightly distorted sphere (wireframe and filled)

- Click on the **button** object labeled "Generate Noise" several more times, or turn on the toggle box labeled "Rapid Noise" to generate a new matrix of noise for each time the geometry is drawn
- Gradually change the **number box** labeled "Crossfade Value" to 1.0. Again, turn wireframe rendering on and off to see how the noise is visualized.

Now the geometry being drawn is pure noise.



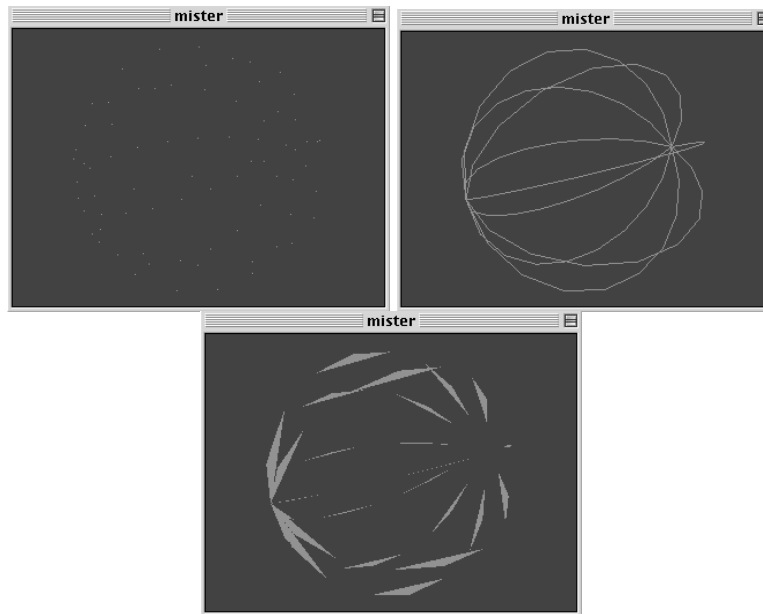
White noise expressed as a geometry (wireframe and filled)

- Set the **number box** labeled "Crossfade Value" back to 0.

Drawing Primitives

You will notice that despite the fact that **jit.gl.gridshape** is outputting the message `jit_matrix [matrix-name] quad_grid`, we are appending the output of `jit.xfade` with `quad_grid`. This is because most matrix operators will ignore the drawing primitive argument and simply output the message `jit_matrix [matrix-name]`. Hence we need to append the name of the drawing primitive to the message. This provides a good opportunity to experiment with various drawing primitives.

- Try selecting the various menu items available in the **ubumenu** labeled *Drawing Primitive*.



Using different drawing primitives: points (left), line_strip (center), and triangles (right)

Summary

In addition to drawing directly to their associated drawing contexts, some objects in the GL group support the `matrixoutput` attribute. When enabled, the geometry matrix is sent out the object's left output with the message `jit_matrix [matrix-name] [drawing-primitive]`.

Geometry matrices are typically float32 data with a plane count of 3, 5, 8, 12, or 13 planes, and may be processed using arbitrary matrix operators in a similar fashion to processing video matrices.

The **jit.gl.render** object supports various drawing primitives to render these geometry matrices: `points`, `lines`, `line_strip`, `line_loop`, `triangles`, `tri_strip`, `tri_fan`, `quads`, `quad_strip`, `polygon`, `tri_grid`, and `quad_grid`.

Tutorial 38: Basic Performance Setup

In this chapter, we will walk through the various steps involved in setting up your Jitter patch for live performance. Along the way, we'll look at using a second video monitor for fullscreen output.

- Open the tutorial patch *38jPerformanceSetup.pat* in the Jitter Tutorials folder.

Most of the patches we've looked at so far use the **jit.pwindow** object for displaying video data. While that object is extremely practical for looking at video inside of your patch, it's not so ideal for live performance (unless you want the audience to see your patch, too). In these situations, we use the **jit.window** object (which you may remember from way back in Tutorial 1: Play A Movie).

- Click on the **message box** labeled *read dishes.mov* and turn on the **toggle box** labeled *Start Movie* to begin movie playback. You should see the movie playing in the **jit.window** object's window.

What's that `@noaccel 1` attribute typed into the **jit.window** object? That tells the object that we want to disable OpenGL acceleration. For the purposes of this tutorial, we're going to ignore some of the more advanced features of **jit.window**—in particular, the ability to use Jitter's OpenGL support to improve the appearance and overall speed of video data. Don't worry, though. Those features will be covered in later Tutorial chapters.

Let's take a look at some of the capabilities of the **jit.window** object. In addition to moving and sizing the window with the mouse (clicking in the lower right hand corner lets us resize the window manually), we can send messages to the object and cause the window to move or resize, we can remove the window's border, make the window float above all other windows, and we can make the window fill the entire screen.



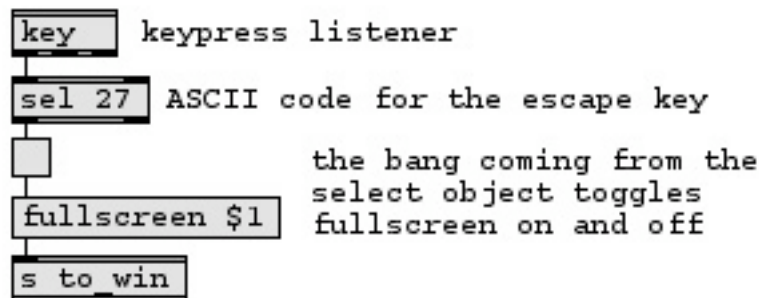
*Resizing the **jit.window** object's window with the mouse.*

- Click on the **message box** labeled size 160 120 and see what happens. Just like the **jit.pwindow** object, the **jit.window** object automatically scales incoming video to fit the size of the window when it receives the message size 160 120. Click on the **message box** labeled size 320 240 to return to the original size.
- Click on the **message box** labeled pos 300 400 and see what happens. We can move the window to any valid screen position, just by sending it the pos (position) message followed by coordinates. We'll use this feature in a little bit to create an automatic window-placement patch. Click on the message box labeled pos 800 100 to return to the original position.

- Turn off the **toggle** labeled *Border On/Off* and see what happens. The window's title bar and (on Windows) border should have disappeared. Turn the **toggle** on to bring the border back.
- Turn on the **toggle** labeled *Float On/Off* and see what happens. A floating window "floats" above all non-floating windows, and can never be sent to the back (although another floating window could cover it). Causing a window to float can help you to ensure that your video data is always visible if you have a complex patch with many windows. Turn the toggle off to turn off floating for this window.
- Finally, press the escape key (esc) on your computer keyboard and watch how the window automatically fills the entire screen of your primary monitor. Pressing the escape key again will collapse the window back to its previous size.

If you look in the patch, we can see clearly how this last trick functions. We've simply used the Max **key** object in combination with a **select** object to listen for the escape key (ASCII key code 27). When that key is pressed, a bang is sent from the **select** object to the **toggle**, causing it's state to change from 1 to 0 or from 0 to 1 (depending on what it was before it received the bang). This value is sent to the **message box** labeled fullscreen \$1, causing the messages fullscreen 1 and fullscreen 0 to be sent to the **jit.window** object.

Hooking this message up to a key is particularly clever, because once the window is fullscreen, it's sometimes a little tricky to get back to your patch! By the way, the **jit.window** object has a useful attribute—**fsmenubar**—which tells it whether or not to hide the system menu bar when the object is in fullscreen mode. By default, the **fsmenubar** attribute is set to 0, meaning that the menubar should stay where it is.

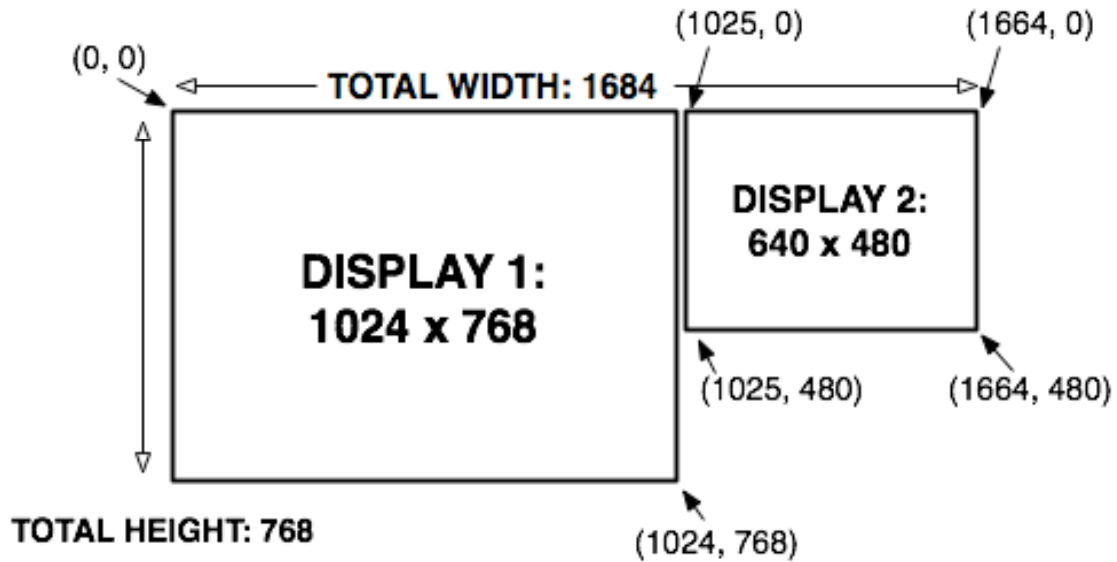


*Toggling the fullscreen state of the **jit.window** object with the escape key.*

Got It Covered

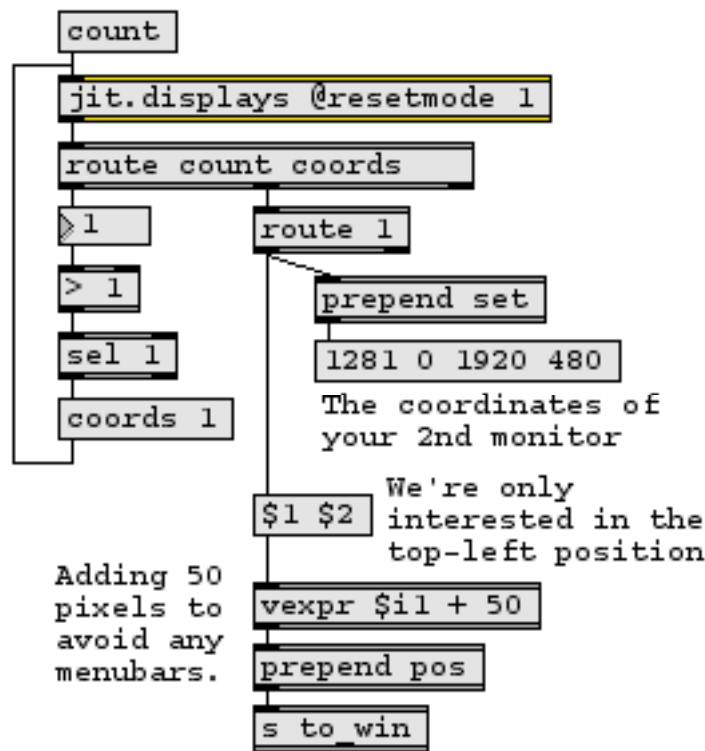
Most of the time, though, we want to be able to use our patch on one screen, while looking at the output on another. The **jit.window** object is capable of putting a window on any display attached to your computer (or display device, like a video projector).

As far as your computer is concerned, once you've attached additional displays in a spanning configuration (that means, the displays aren't mirroring one another), you simply have one large display area. Moving windows to a second screen is simply a matter of knowing the correct coordinates.



A view of a two-monitor system, with coordinates.

Of course, figuring out where your second monitor is located and doing the math is a little bit boring. Luckily, Jitter includes an object, **jit.displays**, which can figure it all out for you.



*The **jit.displays** object in action.*

Using the **jit.displays** object to enumerate the number and coordinates of attached monitors is simple. The `count` message causes the word `count`, followed by the number of attached displays to be sent from the object's left outlet. In our patch, we're testing to make sure that there are more than one attached display using the `>` (greater-than) object. If there are, the message `coords 1` is sent to the **jit.displays** object, requesting the coordinates of monitor 1 (monitors are numbered starting at 0—your primary display is typically monitor 0).

- Click on the **message box** labeled `count`. If you have a second monitor attached to your computer, the **jit.window** object's window should have appeared on it. By simply using the top-left coordinates of the second display as the basis for a simple calculation (we added 50 pixels to each coordinate to make sure that we avoided any menubars), we were able to send a `pos` message to the **jit.window** object and position the window squarely on it.

- Press the escape key again. The window should have gone into fullscreen mode, but this time, on the second display. The **jit.window** object figures out which display it's on before entering fullscreen mode, to ensure that the correct area of the screen gets covered.
- Press the escape key again to disable fullscreen mode.

Of course, the **jit.displays** object will work with systems comprising more than two displays.

Typically, you'll want to keep a variation of this portion of the patch as part of your performance setup. We've noticed that the position settings for a second (or third) display don't always remain the same from use to use, when connecting different kinds of devices to our computers. Using the **jit.displays** object to automatically determine the exact position of your attached hardware and move your output window eliminates some of the hassle and worry of working in new venues or with unfamiliar equipment.

A Little Performance Tip

Now that we've mastered the **jit.window** object, let's turn our attention back to the **jit.pwindow** object. Although the purpose of this tutorial isn't to offer tips and tricks for improving Jitter's performance, the **jit.pwindow** object is so often used as a small monitor in live performance settings, that a quick look at its effect on playback speed is worth our time.

- Click the cell in the **matrixctrl** object labeled *jit.pwindow* to route the video data to the **jit.pwindow** object. You should see video in both the **jit.window** object's window and the **jit.pwindow** object. Compare the framerates (as reported by the **jit.fpsgui** object) with the **jit.pwindow** object enabled and disabled. On our computer, we get around 300 frames per second with the **jit.pwindow** disable and about 110 frames per second with it enabled. With the **jit.pwindow** object enabled and the **jit.window** object turned off, we get around 260 frames per second. Your results will vary widely depending on your computer's operating system and hardware.

- Unlock the patch and open the **jit.pwindow** object's Inspector (on Macintosh, this is accomplished by selecting the object and pressing Command(Apple)-I. On Windows, Ctrl-I. You can also open the Inspector on both platforms by choosing **Get Info...** from the Object menu with the object selected).

Width, Height	80	60
Options	<input type="checkbox"/> Border <input type="checkbox"/> Use Onscreen <input type="checkbox"/> Idle Mouse Reporting <input type="checkbox"/> Interpolation <input checked="" type="checkbox"/> Doublebuffer <input type="checkbox"/> Depthbuffer	
Name	<none>	
<input type="button" value="Revert"/>		

*The **jit.pwindow** object's Inspector*

Do you see the checkbox labeled *Use Onscreen* in the **jit.pwindow** object's Inspector? Uncheck it, and compare framerate with both windows active. On our system, we get around 160 frames per second now—nearly 50% faster! *Anytime you are using a **jit.pwindow** object which is smaller than the video being displayed* (here, our video is 320x240, and our **jit.pwindow** is 80x60), *you should turn off onscreen mode*. It's a simple rule that will get the best possible speed from your performance patch.

Onscreen and offscreen modes use different drawing algorithms to do their work—data is drawn directly to the display in onscreen mode, while offscreen mode draws the data first to an offscreen buffer and then copies it to the display. The algorithm used in onscreen mode doesn't favor downsampling, so the performance is not as good.

The *Use Onscreen* setting in the **jit.pwindow** object's Inspector is saved with the patch, so it need only be set once. You can set the onscreen attribute directly from Max using messages, as well.

Another strategy for reducing the impact of **jit.pwindow** objects in your performance patch is to reduce the amount of data being drawn. The Max **speedlim** and **qlim** objects are ideal

for this purpose, although we're not going to demonstrate their use in this tutorial (the **speedlim** object is discussed in Max Tutorial 16).

Summary

In this tutorial, we learned a few strategies for overcoming typical challenges in a performance environment: the appearance settings for the **jit.window** object; the fullscreen mode; the use of the **jit.displays** object to sense additional display hardware attached to your computer and autoconfigure the position of a **jit.window** object's window; and the management of the **jit.pwindow** object's onscreen drawing mode to improve the performance of your in-patch screens.

Tutorial 39: Spatial Mapping

In this tutorial, we will look at ways to remap the spatial layout of cells in a Jitter matrix using a second matrix as a spatial map. Along the way, we'll investigate ways to generate complete maps through the interpolation of small datasets as well as ways to use mathematical expressions to fill a Jitter matrix with values.

Jitter matrices can be remapped spatially using lookup tables of arbitrary complexity. In Tutorial 12, we learned that the **jit.charmap** object maps the cell values (or color information) of one Jitter matrix based on a lookup table provided by a second Jitter matrix. In a similar fashion, the **jit.repos** object takes a Jitter matrix containing spatial values that it then uses as a lookup table to rearrange the spatial positions of cells in another matrix.

- Open the tutorial patch *39jSpatialMapping.pat* in the Jitter Tutorials folder.

The tutorial patch shows the matrix output of a **jit.qt.movie** object being processed by a new object, called **jit.repos**. The right inlet of **jit.repos** receives matrices from the output of a **jit.xfade** object fed by two named matrices, each of which has been given a name (stretch and cartopol) so that they share data with **jit.matrix** objects elsewhere in the patch. To the right of the main processing chain are two separate areas where we construct those two matrices, which will serve as our spatial maps for this tutorial.

- Click the message box that reads `read colorswatch.pict` to load an image into the **jit.qt.movie** object. Click the **toggle** labeled *Display* to start the **metro** object.

Notice that at this point, we only see a solid white frame appear in the **jit.pwindow** object at the bottom-left of the patch. This is because we haven't loaded a spatial map into our **jit.repos** object.

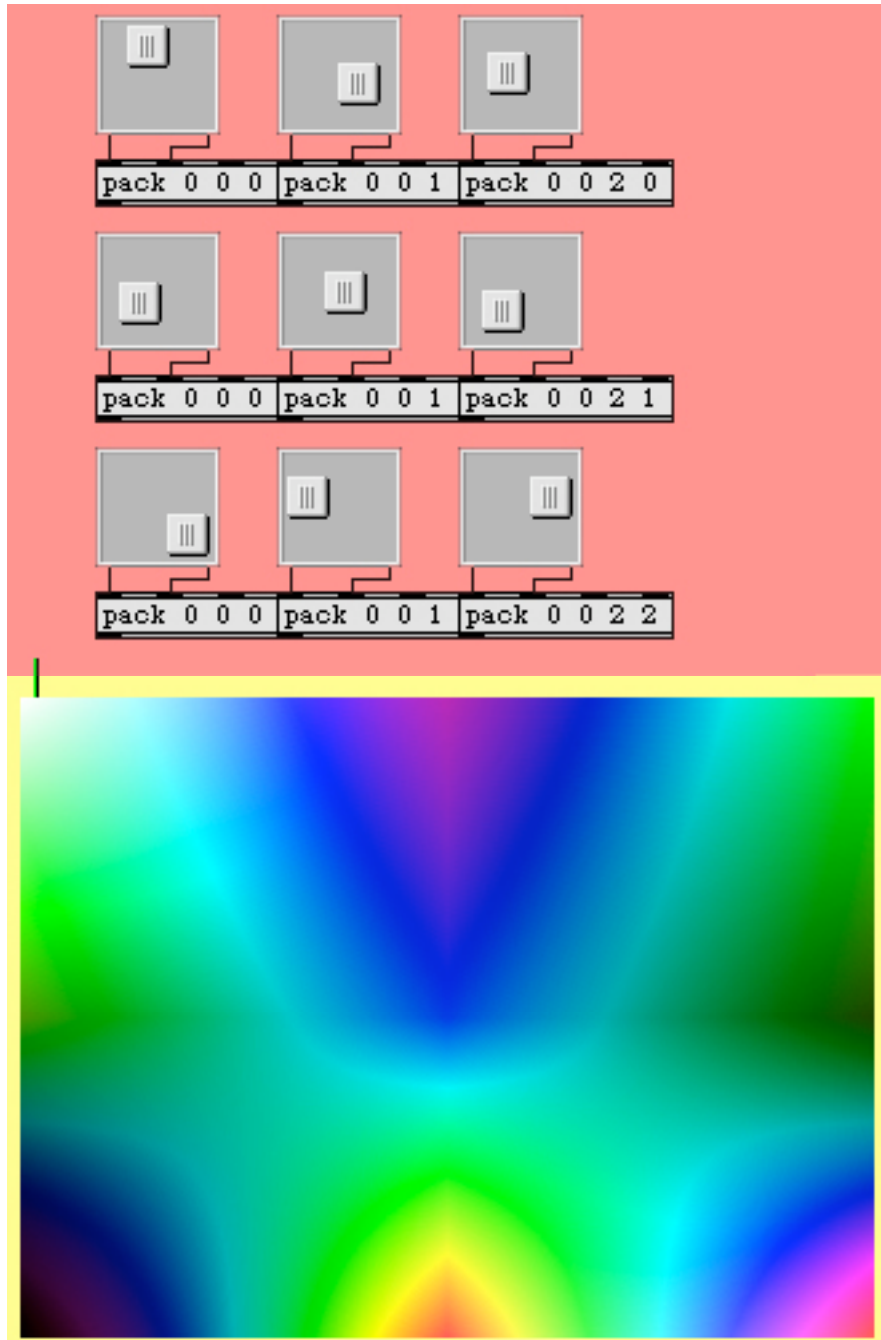
- Click the first box in the preset object in the area of the patch labeled *Generate "stretch" matrix*. You'll see the **pictslider** objects snap to various positions and a gradient progressing from magenta to green appear in the **jit.pwindow** at the bottom of that section.

We now see a normal looking image appear in the **jit.pwindow** at the bottom of our patch.

The Wide World of Repos

- Try manipulating the **pictslider** objects that set values in the stretch matrix. Notice that the effect is to stretch and twist our image.

By manipulating the **pictslider** objects one at a time, we can see that they distort different areas of the screen in a 3x3 grid. Clicking the first box in the **preset** object will return the image to normal.



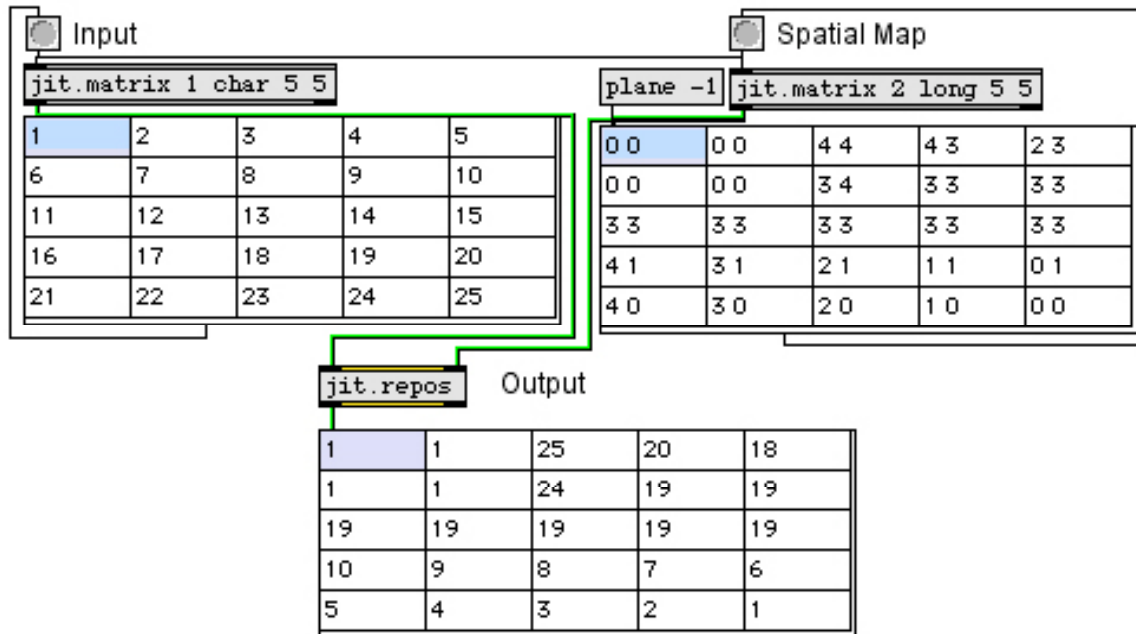
Hall of mirrors.

The **pictslider** objects in our patch set individual cells in a 2-plane, 3x3 Jitter matrix of type long. This matrix is then upsampled with interpolation to a 320x240 matrix named **stretch**. This matrix contains the instructions by which **jit.repos** maps the cells in our incoming

matrix. The interpolation here is necessary so that we can infer an entire 320x240 map from just nine specified coordinate points.

The spatial map provided to the **jit.repos** object in its right inlet contains a cell-by-cell description of which input cells are mapped to which output cells. The first plane (plane 0) in this matrix defines the x coordinates for the mapping; the second plane (plane 1) represents the y coordinates. For example, if the cell in the upper-left hand corner contains the values 50 75, then the cell at coordinates 50 75 from the matrix coming into the left inlet of the **jit.repos** object will be placed in the upper left-hand corner of the output matrix.

The following example will give us a better sense of what **jit.repos** is doing with its instructions:



A simple spatial map.

In this example, we have a simple single-plane 5x5 matrix filled with ascending numbers being remapped by a spatial map. Notice the correlation between the values in the spatial map and where the corresponding cells in the output matrix get their original values from in the input matrix. For example, the cell in the lower-left of the spatial map matrix reads 40. The cell at coordinate {4, 0} in the incoming matrix is set to value 5. As you can see, the cell in the lower-left of the output matrix contains that value as well.

Similarly, you can see that the cells comprising the middle row of the output matrix are all set to the same value (19). This is because the corresponding cells in the spatial map are all set to the same value (33). Cell {3, 3} in the input matrix contains the value 19; that value is then placed into that entire row in the output matrix.

In our tutorial patch, our spatial map needs to be able to contain values in a range that represents the full size (or dim) of the matrix to be processed. Because the image that we want to distort is 320x240, we need to use a matrix of type `long` for our spatial map. If it was of type `char`, we would be constrained to values in the range of 0 to 255, limiting what cells we can choose from in our spatial map. Note that in order to correctly view these `long` matrices in **jit.pwindow** objects we use the **jit.clip** object to constrain the values in the matrix before viewing to the same range as `char` matrices.

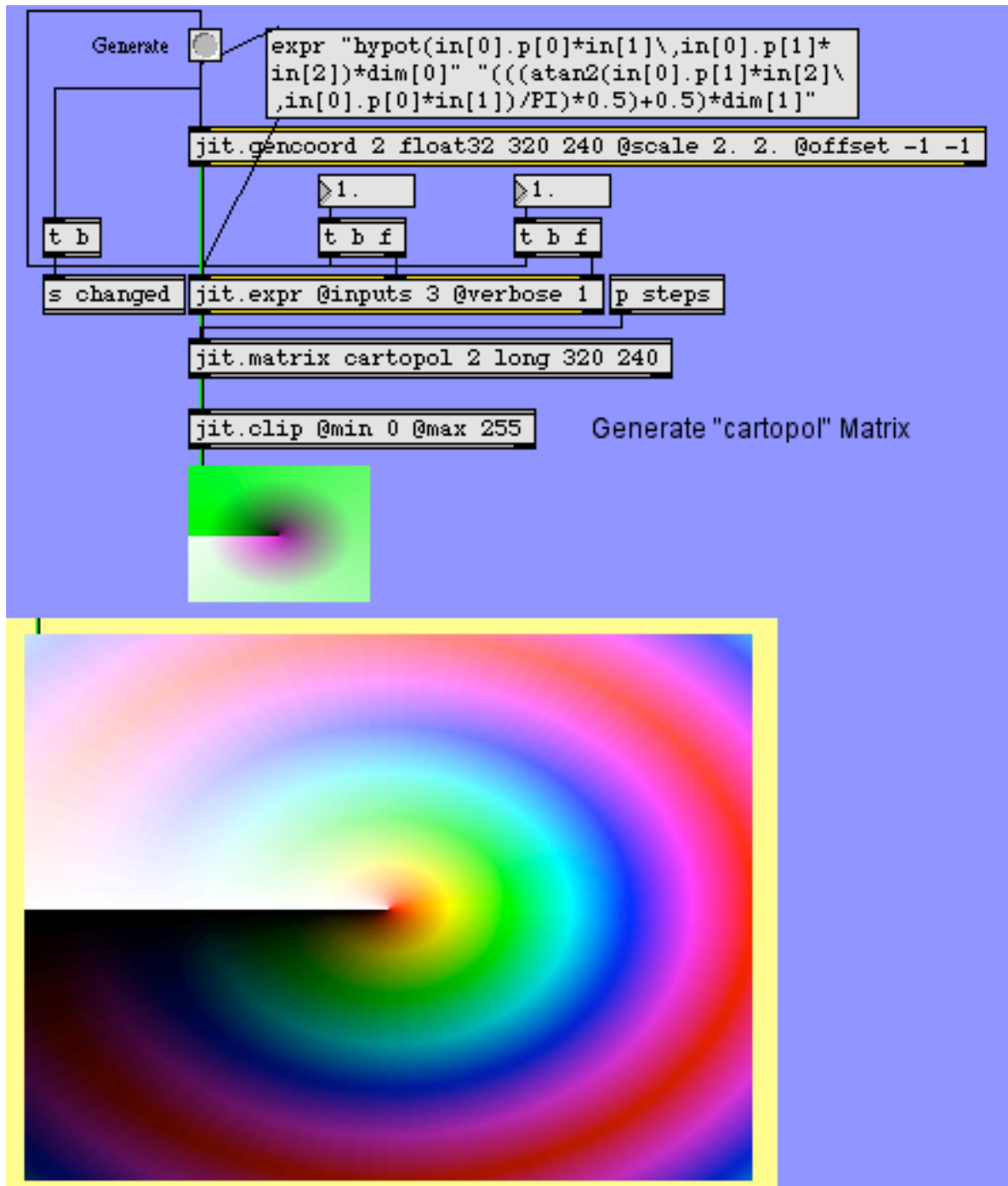
Technical Note: The default behavior of **jit.repos** is to interpret the spatial map as a matrix of absolute coordinates, i.e. a literal specification of which input cells map to which output cells. The `mode` attribute of **jit.repos**, when set to 1, places the object in relative mode, wherein the spatial map contains coordinates relative to their normal position. For example, if cell {15, 10} in the spatial map contained the values -3 7, then a **jit.repos** object working in relative mode would set the value of cell {15, 10} in the output matrix to the contents of cell {12, 17} in the input matrix (i.e. {15-3, 10+7}).

- Play around some more with the **pictslider** objects to get a better feel for how **jit.repos** is working. Notice that if you set an entire row (or column) of **pictslider** objects to approximately the same position you can “stretch” an area of the image to cover an entire horizontal or vertical band in the output image. If you set the **pictslider** objects in the left columns to values normally contained in the right columns (and vice versa), you can flip the image on its horizontal axis. By doing this only part-way, or by positioning the slider values somewhere in between where they sit on the “normal” map (as set by the **preset** object) you can achieve the types of spatial distortion commonplace in funhouse mirrors, where a kink in the glass bends the reflected light to produce a spatial exaggeration.

Spatial Expressions

Now that we’ve experimented with designing spatial maps by hand, as it were, we can look at some interesting ways to generate them algorithmically.

- In the lower-right section of the tutorial patch, click the **button** labeled *Generate*. Set the **number box** labeled *Crossfade* (attached to the **trigger** object in the main part of the patch) to 1.0.

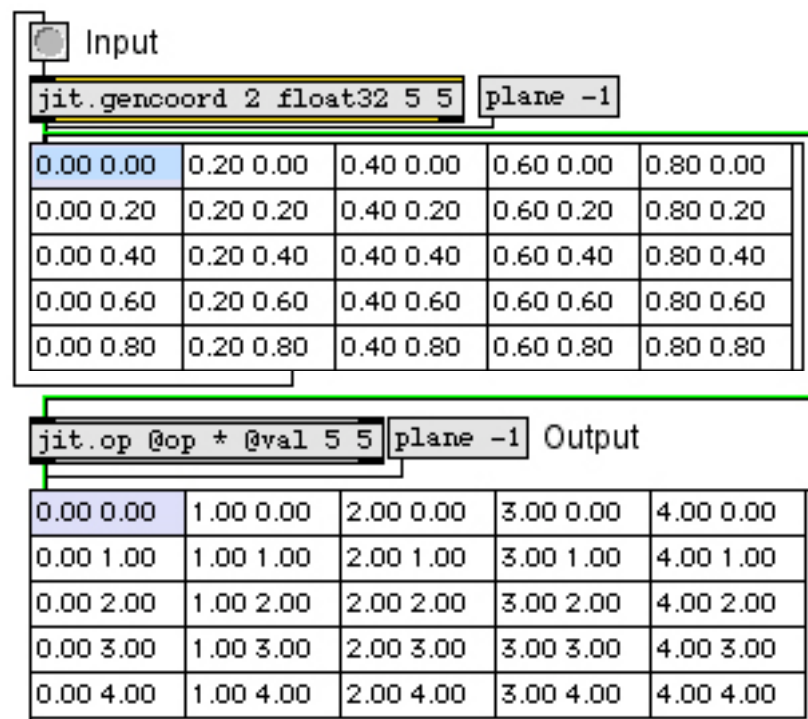


Lollipop space.

The cartopol spatial map has been set by a mathematical expression operated on the output of the **jit.gencoord** object by the **jit.expr** object. The results of this expression, performed on

float32 Jitter matrices, are then copied into a long matrix that serves as our spatial map. This spatial map causes **jit.repos** to spatially remap our incoming matrix from Cartesian to polar space, so that the leftmost edge of our incoming matrix becomes the center of the outgoing matrix, wrapping in a spiral pattern out to the final ring, which represents the rightmost edge of the input matrix. The boundmode attribute of **jit.repos** tells the object to wrap coordinates beyond that range back inwards, so that the final edge of our output image begins to fold in again towards the left side of the input image.

The **jit.gencoord** object outputs a simple Cartesian map of floating-point values; when you multiply the values in the matrix by the dim of the matrix, you can easily get coordinates which literally correspond to the locations of the cells, as shown in this illustration:



Normal Cartesian map.

The scale and offset attributes of **jit.gencoord** allow us to manipulate the range of values put out by the object, saving us the step of using multiple **jit.op** objects to get our Cartesian map into the range we want. For our equation to work, we want our starting map to be in the range of $\{-1, 1\}$ across both dimensions (hence we multiply the normal Cartesian space by 2 and offset it by -1).

The **jit.expr** object takes the incoming matrix provided by **jit.gencoord** and performs a mathematical expression on it, much as the Max **vexpr** object works on lists of numbers. The **expr** attribute of the **jit.expr** object defines the expression to use. The **in[i].p[j]** notation tells **jit.expr** that we want to operate on the values contained in plane *j* of the matrix

arriving at inlet i . The `dim[i]` keyword returns the size of the dimension i in the matrix. In our case, the notation `dim[0]` would resolve to 320; `dim[1]` would resolve to 240.

Note that there are actually two mathematical expressions defined in the **message box** setting the `expr` attribute. The first defines the expression to use for plane 0 (the x coordinates of our spatial map), the second sets the expression for plane 1 (the y coordinates).

Placed in a more legible notation, the **jit.expr** expression for plane 0:

```
hypot(in[0].p[0]*in[1],in[0].p[1]*in[2])*dim[0]
```

works out to something like this, if x is the value in plane 0 of the input, y is the value in plane 1 of the input, and a and b are the values in the **number box** objects attached to the second and third inlets of the **jit.expr** object:

$$\sqrt{ax^2+by^2} \times 320$$

Similarly, the expression for plane 1:

```
((atan2(in[0].p[1]*in[2],in[0].p[0]*in[1])/PI)*0.5)+0.5)*dim[1]
```

works out to something like this:

$$(((\text{atan}(by/ax))/2)+0.5) \times 240$$

Therefore, we can see that we are filling the x (plane 0) values of our spatial map with numbers corresponding to the length of the hypotenuse of our coordinates (i.e. their distance from the center of the map), while we set the y (plane 1) values to represent those values' angle (\emptyset) around the origin.

- In the lower-right section of the tutorial patch, change the **number box** objects attached to the **jit.expr** object. Note that they allow you to stretch and shrink the spatial map along the x and y axes.

The **jit.expr** object can take Max numerical values (int or float) in its inlets in lieu of matrices. Thus the `in[1]` and `in[2]` notation in the `expr` attribute refers to the numbers arriving at the middle and right inlets of the object.

Technical Note: The **jit.expr** object parses the mathematical expressions contained in the **expr** attribute as individual strings (one per plane). As a result, each individual expression should be contained in quotes and commas need to be escaped by backslash (\) characters. This is to prevent Max from atomizing the expression into a list separated by spaces or a sequence of messages separated by commas. We can gain some insight into how **jit.expr** parses its expressions by turning on the **verbose** attribute to the object (as we do in this tutorial). When an **expr** attribute is evaluated, the object prints the expression's evaluation hierarchy to the Max window so we can see exactly how the different parts of the expression are tokenized and interpreted.

- Open the **patcher** object labeled steps. Click through the different expressions contained in the **message** boxes, each of which shows a simpler stage of how we computed the cartopol map. The **norm[i]** and **snorm[i]** notation essentially replicates the function of **jit.gencoord** here, generating a map of Cartesian values (0 to 1) or signed Cartesian values (-1 to 1) across the specified dimension.

Steps to generate the "cartopol" Matrix

A normal cartesian spatial map

```
exprfill 0 "norm[0]*dim[0]", exprfill 1 "norm[1]*dim[1]"
```

A "signed" cartesian spatial map

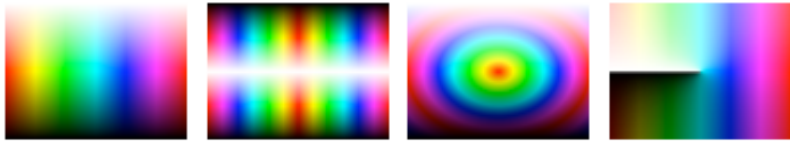
```
exprfill 0 "snorm[0]*dim[0]", exprfill 1 "snorm[1]*dim[1]"
```

The distance (x) coordinate of the polar map, with the y coordinate kept as a normal map

```
exprfill 0 "hypot(snorm[0]\,snorm[1])*dim[0]", exprfill 1 "norm[1]*dim[1]"
```

The theta (y) coordinate of the polar map, with the x coordinate kept as a normal map

```
exprfill 0 "norm[0]*dim[0]", exprfill 1 "(((atan2(snorm[1]\,snorm[0])/PI)+0.5)+0.5)*dim[1]"
```



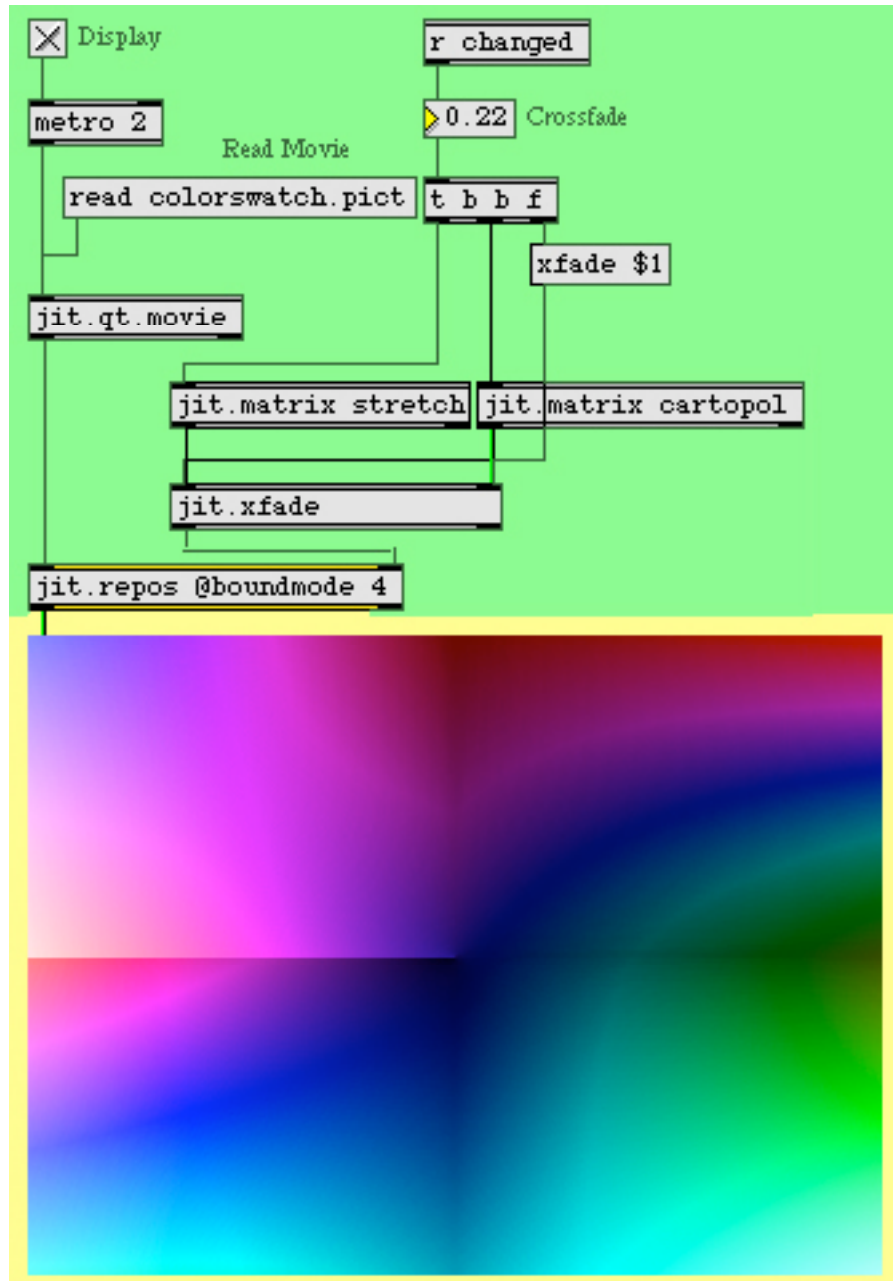
A B C D

Different stages of the Cartesian-to-polar mapping: (a) normal map, (b) normal map scaled and offset to a signed {-1, 1} range, (c) hypotenuse only, (d) theta only

An important thing to notice with this subpatch is that the **message box** objects are directly connected to the **jit.matrix** object in the main patch. The **exprfill** message to **jit.matrix** allows us to fill a Jitter matrix with values from a mathematical expression without having to resort to using **jit.expr** as a separate object. The syntax for the **exprfill** method is slightly different from the **expr** attribute to **jit.expr**: the first argument to **exprfill** is the plane to fill with values, so two **exprfill** messages (separated by commas) are necessary to generate the spatial map) using this method.

Still Just Matrices

- Slowly change the **number box** attached to the **trigger** object on the left of the patch from 1 to 0 and back again.



A partial cross-fade of the two spatial maps used in this tutorial.

The number box controls the xfade attribute of the **jit.xfade** object directly attached to the **jit.repos** object at the heart of our processing chain. As a result, we can smoothly transition

from our manually defined spatial map (the stretch matrix) to our algorithmically defined one (the cartopol matrix). The ability to use Jitter processing objects to composite different spatial maps adds another layer of potential flexibility. Because the two matrices containing our maps have the same typology (type, dim, and plane count) they can be easily combined to make an infinite variety of composites.

Summary

The **jit.repos** object processes an input matrix based on a spatial map encoded as a matrix sent to its second inlet. The default behavior is for **jit.repos** to process matrices spatially using absolute coordinates, such that the cell values in the second matrix correspond to coordinates in the input matrix whose cell values are copied into the corresponding position in the output matrix.

Spatial maps can be created by setting cells explicitly in a matrix and, if desired, upsampling with interpolation from a smaller initial matrix. Maps can also be created by generating matrices through algorithmic means, such as processing a “normal” Cartesian map generated by a **jit.gencoord** object through a mathematical expression evaluated by a **jit.expr** object. The **jit.expr** object takes strings of mathematical expressions, one per plane, and parses them using keywords to represent incoming cell values as well as other useful information, such as the size of the incoming matrix. If you want to fill a Jitter matrix with the results of a mathematical expression directly, you can use the `exprfill` message to a **jit.matrix** object to accomplish the same results without having to use a separate **jit.expr** object.

The **jit.repos** object can be used to generate a wide variety of spatial maps for different uses in image and data processing. The ‘jitter-examples/video/spatial’ folder of the Max ‘examples’ folder contains a number of patches that illustrate the possibilities of **jit.repos**.

Tutorial 40: Drawing in OpenGL using jit.gl.sketch

The OpenGL capabilities of Jitter provide us with a variety of tools for creating animated scenes in three dimensions. The Jitter OpenGL objects we've looked at thus far each perform a fairly straightforward task, be it creating simple geometries (**jit.gl.gridshape** and **jit.gl.plato**), importing *OBJ* models (**jit.gl.model**), or manipulating the position of objects and scenes (**jit.gl.handle**). More complex scenes can be built by using several of these objects at once. After a certain point, however, it may make sense to work with an object that understands OpenGL commands directly and can be used to perform a large number of drawing commands at once. This object is **jit.gl.sketch**, and it is the subject of this Tutorial.

The **jit.gl.sketch** object interfaces with the OpenGL system in Jitter just as other OpenGL objects; as a result, it may help to review *Tutorial 30: Drawing 3D Text* and *Tutorial 31: Rendering Destinations* before we begin. In addition, the messages and attributes understood by **jit.gl.sketch** bear a strong resemblance to the methods and properties of the JavaScript sketch object used in the Max **jsui** object. *Tutorial 51: Designing User Interfaces in JavaScript* in the *Max Tutorials and Topics* manual will give you some additional information on the capabilities of OpenGL vector graphics that is common to both of these systems.

The majority of the messages used by **jit.gl.sketch** are slightly modified versions of standard functions within the OpenGL API. The entire API is outside the scope of this Tutorial; however, the standard reference for these functions (the OpenGL “Redbook”) is available online at:

http://www.opengl.org/documentation/red_book_1.0/

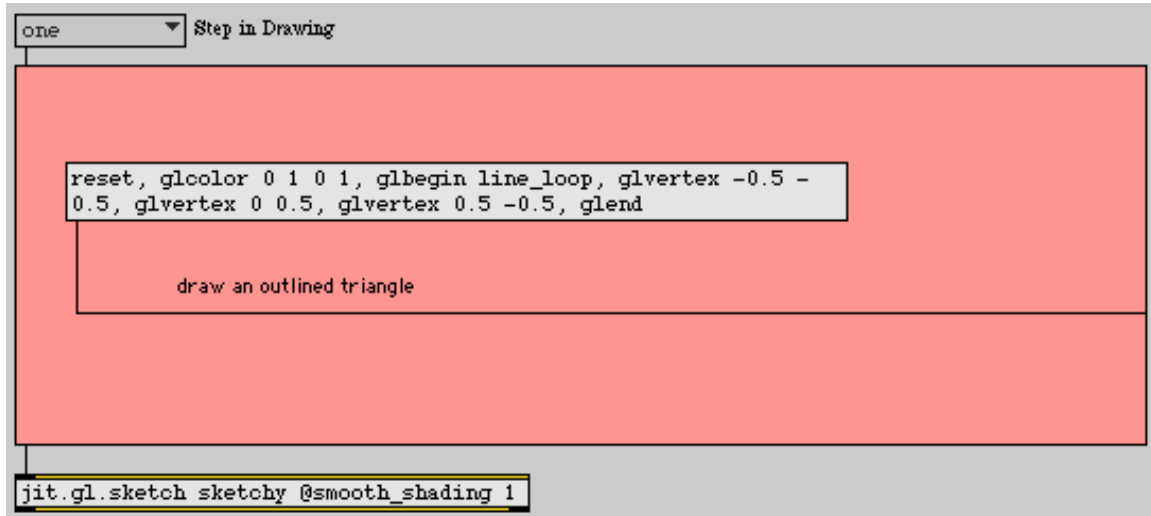
Converting between OpenGL code as given in the “Redbook” and elsewhere (in the C programming language) and messages to **jit.gl.sketch** is reasonably straightforward if the following conventions are kept in mind:

- All OpenGL commands are lowercase in **jit.gl.sketch**. Thus the OpenGL function *glColor()* becomes the Max message *glcolor* sent to **jit.gl.sketch**.
- OpenGL symbolic constants, in addition to being lowercase, lose their “GL_” prefix, so that *GL_CLIP_PLANE1* (in OpenGL) becomes *clip_plane1* when used with **jit.gl.sketch**, for example.

In addition to the basic OpenGL API, **jit.gl.sketch** also understands a number of high-level drawing commands to instruct the object to render basic shapes and perform vector graphics-style drawing operations.

- Open the tutorial patch *40jSketch.pat* in the Jitter Tutorials folder.

The Tutorial patch consists of a series of Jitter objects used for rendering in OpenGL: a **jit.gl.render** object, a **jit.pwindow** object, and a **jit.gl.sketch** object. The **jit.gl.sketch** object is sent messages from a **bpatcher** object containing **message box** objects containing lists of commands. Different views of the **bpatcher** can be obtained by selecting different “offsets” from the **ubumenu** object connected to it. In this Tutorial we’ll step through the different sections of the **bpatcher** one-by-one and look at the messages contained in each view.



*Our **jit.gl.sketch** object receiving messages from within a **bpatcher**.*

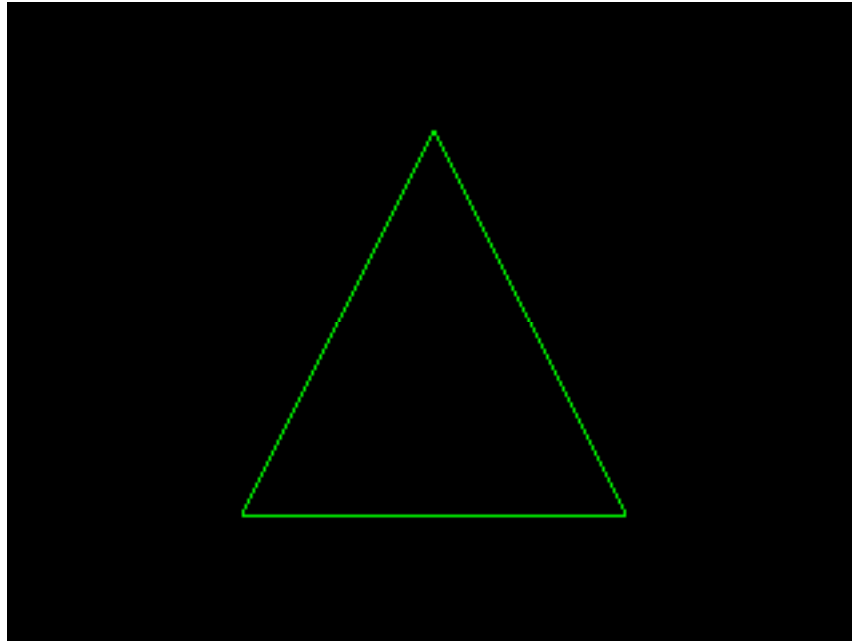
- Click on the **toggle box** labeled *Display* attached to the **qmetro** object at the top-left of the patcher.

By starting the **qmetro** object, we’ve begun rendering our OpenGL scene. Our **jit.gl.render** object shares it’s name (*sketchy*) with the **jit.pwindow** object and the **jit.gl.sketch** object in the patch. Because of this, the **jit.pwindow** will display anything drawn by the **jit.gl.render** object, which in turn will draw whatever the **jit.gl.sketch** object instructs it to draw.

The command list

- Set the **ubumenu** object attached to the **bpatcher** to read “one”. The **bpatcher** object will show a **message box** containing a series of messages separated by commas. Click on the **message box** to send its contents to our **jit.gl.sketch** object.

A green triangle should appear in the **jit.pwindow** object.



*A green triangle drawn with a sequence of messages to **jit.gl.sketch**.*

Technical Detail: The **jit.gl.sketch** object works by maintaining a *command list* of OpenGL commands that are then executed every time the scene is drawn (in our patch, when the **jit.gl.render** object receives a bang). Most of the messages we send to **jit.gl.sketch** become part of this command list. This command list can be stored in the internal memory of the object or, if the `displaylist` attribute of **jit.gl.sketch** is set to 1, it can be stored on the GPU of our computer. When working with large numbers of commands, turning on the `displaylist` attribute will speed up rendering time; however, new commands added onto the list may take longer to be added if they are sent very quickly, as they need to be loaded onto the GPU.

Our **message box** sends the following messages in sequence to **jit.gl.sketch**:

```
reset,  
glcolor 0 1 0 1,  
glbegin line_loop,  
glvertex -0.5 -0.5,  
glvertex 0 0.5,  
glvertex 0.5 -0.5,  
glend
```

The reset message to **jit.gl.sketch** simply tells the object to clear its command list and reinitialize itself. What follows are a sequence of OpenGL commands. The `glcolor`

command tells **jit.gl.sketch** to change the color it uses to draw. As with all OpenGL objects in Jitter, these colors are floating-point values in RGBA (red, green, blue, alpha) order; thus, `glcolor 0 1 0 1` tells **jit.gl.sketch** to draw in a fully opaque (alpha=1) green.

The `glbegin` message tells **jit.gl.sketch** to interpret what follows as instructions defining an *object*. The argument to `glbegin` specifies the *drawing primitive* to use when rendering the shape. The primitive we've chosen, `line_loop`, will take a set of points (called *vertices*) and connect them with lines, completing the shape by connecting the last vertex to the first vertex again. Thus in a shape with four points A, B, C, and D, our object will contain lines from A to B, B to C, C to D, and D to A.

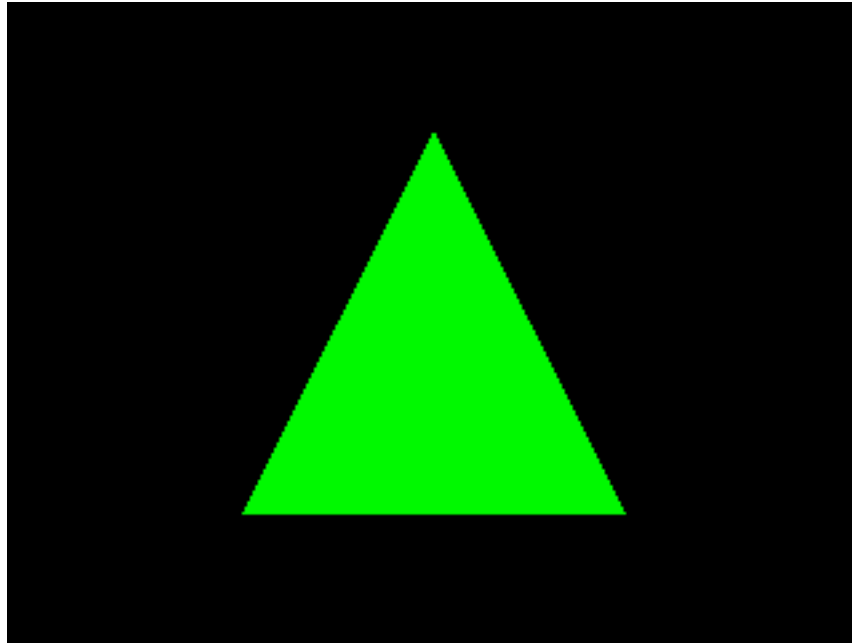
The `glvertex` messages contain the coordinates of the points in our shape. If only two coordinates are given, they are interpreted as *x* and *y* values. If three coordinates are given, the last value is interpreted as a *z* coordinate. The `glend` message tells **jit.gl.sketch** that we've finished defining the shape. We could then move on to another shape (or execute other commands) if we so desired.

Step one of our patch therefore resets the object, sets the color to green, and instructs it to draw an outlined shape with three vertices connected to one another. These messages (with the exception of the reset message) form the command list for **jit.gl.sketch** for the image we see in the **jit.pwindow**.

More about drawing primitives

- Select the **ubumenu** object and set it to “two”. Click on the **message box** that appears in the **bpatcher**.

Our outlined triangle should disappear and a filled green triangle should take its place.



A filled green triangle.

The list of commands in our **message box** is very similar to the first one:

```
reset,  
glcolor 0 1 0 1,  
glbegin triangles,  
glvertex -0.5 -0.5,  
glvertex 0 0.5,  
glvertex 0.5 -0.5,  
glend
```

Once again, we reset the object and then define a color (green) and an object with three vertices to be drawn. The only difference is the drawing primitive specified as an argument to the `glbegin` message. In this example, we're using `triangles` as our drawing primitive. This tells **jit.gl.sketch** to interpret triplets of vertices as *triangles*. Rather than outlining the shape as we did before, we've now instructed **jit.gl.sketch** to generate a *polygon*. As a result, the interior of our vertices is filled with the specified `glcolor` (green).

There are ten (10) different drawing primitives recognized by OpenGL and, as a result, by objects such as **jit.gl.render** and **jit.gl.sketch**. They are *points*, *lines*, *line_strip*, *line_loop*, *triangles*, *triangle_strip* (abbreviated *tri_strip*), *triangle_fan* (abbreviated *tri_fan*), *quads*, *quad_strip*, and *polygon*. These primitives each specify the algorithm by which a series of vertices will be connected. *Tutorial 37: Geometry Under the Hood* and Table 2-2 and Figure 2-7 of the OpenGL “Redbook” give descriptions and examples of each and describe their common uses.

- Set the **ubumenu** object in the Tutorial patch to “three” and click the **message box** that appears in the **bpatcher**. A triangle with a rainbow surface will appear in place of the solid green triangle.



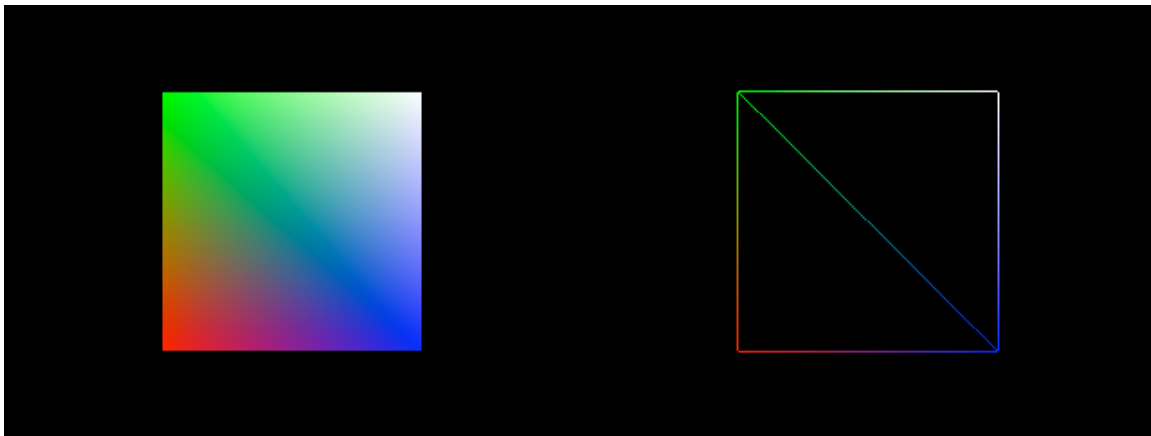
A shape with different colored vertices.

In this command list, we’ve embedded glColor commands for each vertex of our triangle:

```
reset,  
glbegin triangles,  
glcolor 1 0 0 1,  
glvertex -0.5 -0.5,  
glcolor 0 1 0 1,  
glvertex 0 0.5,  
glcolor 0 0 1 1,  
glvertex 0.5 -0.5,  
glend
```

As a result, our **jit.gl.sketch** object sets our three vertices to red, green, and blue, respectively, and fills in the interior surface with a color gradient that fades smoothly between the three colors. The `smooth_shading` attribute of **jit.gl.sketch** allows this behavior when set to 1 (as is the case with the **jit.gl.sketch** object in our Tutorial patch).

- Set the **ubumenu** object in the Tutorial patch to “four”. Set the **ubumenu** *inside* the **bpatcher** to “fill”. Our **jit.pwindow** now contains a square with a rainbow-colored surface. Set the **ubumenu** to “outline”. Now we only see rainbow-colored lines outlining the shape, as well as a diagonal across it.



A square made up of connected triangles, filled and unfilled.

This section of the **bpatcher** allows us to see how a drawing primitive works to make a more complex shape. We begin our command list with the `glpolygonmode` command, which tells **jit.gl.sketch** whether to fill in complete polygons or leave them as outlines, exposing the skeleton that defines how the vertices are connected. The first argument to `glpolygonmode` defines whether we’re talking about the front of the shape, the back of the shape, or both (`front_and_back`, as we’re using here). The second argument defines whether that side of the shape will be filled (`fill`) or left outlined (`line`). The `poly_mode` attribute of the Jitter OpenGL objects accomplishes the same thing.

After we’ve reset and decided how we want our shape drawn (filled or outlined), we supply a shape description:

```
glbegin tri_strip,  
glcolor 1 0 0 1,  
glvertex -0.5 -0.5,  
glcolor 0 1 0 1,  
glvertex -0.5 0.5,  
glcolor 0 0 1 1,  
glvertex 0.5 -0.5,  
glcolor 1 1 1 1,  
glvertex 0.5 0.5,  
glend
```

As with the triangle in our previous example, we provide a `glcolor` for each `glvertex` (in this example red, green, blue, and white for the four points of the square). The `tri_strip` drawing primitive used as an argument to `glbegin` tells **jit.gl.sketch** to draw the shape as a series of triangles where the last two vertices of a triangle are recycled into the next triangle. Thus a polygon with six vertices A, B, C, D, E, and F would be drawn as *four* triangles using the `tri_strip` primitive: ABC, CBD, CDE, and EDF (the ordering ensures that the triangles are all drawn with the same orientation). In our code, we get a shape with two triangles that when connected and filled appear to us as a square.

Rotation, translation, and scaling

- Set the **ubumenu** in the Tutorial patch to “five”. Click and drag the **number box** in the **bpatcher** slowly upwards from 0 to 360. The filled rainbow square from the previous example will reappear and rotate in a counter-clockwise direction.



An rotation matrix applied to an object.

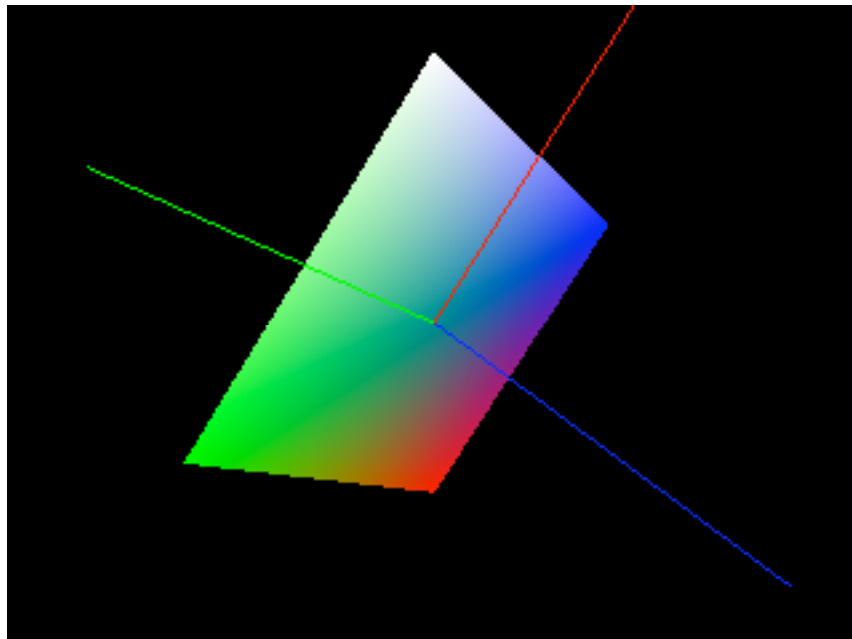
In addition to defining shapes containing vertices and colors, we can instruct **jit.gl.sketch** to manipulate these shapes as objects that can be animated and scaled. Our command list sets this up for us by supplying some additional commands:

```
reset,  
glmatrixmode modelview,  
glpushmatrix,  
glrotate $1 0 0 1,
```

After resetting everything, we send the `glmatrixmode` message, which tells **jit.gl.sketch** we are going to be performing perspective operations to define something called the modelview matrix. This allows us to perform rotation, translation, and scaling on the object. The `glpushmatrix` message copies our current modelview matrix onto a stack, allowing us to return to it later. This allows us to create nested translations for drawing complex objects. For example, we could draw a series of polygons, each with rotations relative to the *previous* polygon; in this way we could generate sequences of shapes that have the same orientation with respect to each other but could then all be rotated with respect to another group of shapes. The use of a stack for this process greatly aids in modeling as we can use simple face-on coordinates for the vertices in our shapes and then tell the renderer to compute the rotations for us. The `glrotate` command specifies the *degrees* of rotation for our shape followed by three values that specify the *axis* of rotation as an *xyz* vector. Thus we're rotating our object around the *z* axis (which spins the shape along the *x-y* plane of the screen).

Following the rotation, we continue with our command list as in the previous example. At the end of the shape specification, we complete our command list with the `glpopmatrix` command, which returns our **jit.gl.sketch** object to the previous rotation. New commands added to the command list after that would then use the original rotation vector, not the one we've applied to our shape.

- Set the **ubumenu** to “six”, and manipulate the **number box** objects attached to the **pak** object inside the **bpatcher**. We can now rotate our square around all three axes. In addition, lines are rendered to provide us with unit vectors to guide us in our rotation.



A rotation matrix applied to multiple objects (a square and three vectors).

Our command list here sets up a new `modelview` matrix and performs the rotation three times in series:

```
reset,  
glmatrixmode modelview,  
glpushmatrix,  
glrotate $1 1 0 0,  
glrotate $2 0 1 0,  
glrotate $3 0 0 1,
```

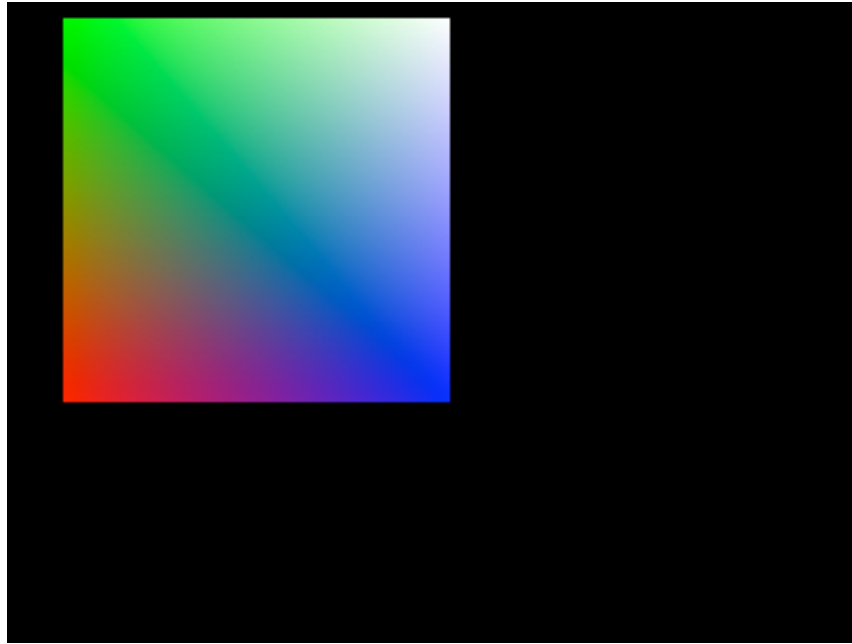
This allows us to rotate our shape along the x axis, y axis, and z axis in sequence according to the values of the **number box** objects in our patch. We then create *four* objects using this rotation before we send a `glpopmatrix` to return to our original orientation. After our square is rendered, we create three two-vertex lines in different colors oriented along the same axes as the plane:

```
glcolor 1 0 0 1, glBegin lines, glVertex 0 0 0, glVertex 1 0 0,  
glend,  
glcolor 0 1 0 1, glBegin lines, glVertex 0 0 0, glVertex 0 1 0,  
glend,  
glcolor 0 0 1 1, glBegin lines, glVertex 0 0 0, glVertex 0 0 1,  
glend,  
glpopmatrix
```

The drawing primitive `lines` simply connects all the vertices in the shape (unlike `line_loop`, it does *not* connect the last vertex back to the first). Notice that because these objects (and our square) are all within the same `modelview` matrix, they all share the same rotation.

- Set the **ubumenu** to “seven” and manipulate the **number box** objects in the **bpatcher**.

The colored square can be moved along the x and y axes of the space.

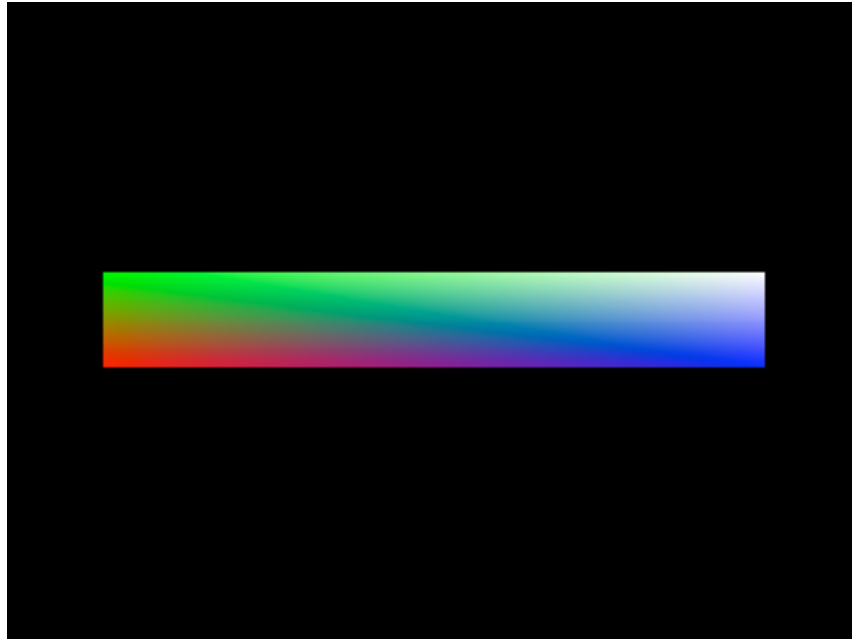


A translation matrix applied to our shape.

In addition to image rotation, the `modelview` matrix can apply translation of an image. This allows us to specify vertices for our object centered around $(0, 0)$ and then move the object anywhere we want in the space. The `gltranslate` message in our command list performs this function, taking as arguments the x and y offsets that will be applied to every vertex in the shape(s) affected by the transformation.

- Set the **ubumenu** to “eight” and manipulate the **number box** objects in the **bpatcher**.

The colored square can be stretched along both axes.



Our object transformed by scaling.

Just as we can rotate and translate a shape or shapes, we can also control their scaling with the `glScale` message. The vertices in our shape are *multiplied* by the arguments to the `glScale` message (specified as x , y , and optional z scaling factors).

The *order* in which you provide `glRotate`, `glTranslate`, and `glScale` commands to **jit.gl.sketch** is important, as they are processed *sequentially* and *cumulatively* to create the final transformation of the affected objects. For example, say you have a vertex at coordinates (0.5, 1). If you translate that vertex by (0.2, 0.2) and then scale it by (1.5, 1.5) the vertex will end up at (1.6, 1.7). If you reverse the order of those operations (scale *then* translate), your vertex will end up at (0.95, 1.7).

Summary

The **jit.gl.sketch** object gives us access to the powerful OpenGL API, allowing us to define and execute an OpenGL command list within a Max patch. Lists of vertices, specified by `glVertex` messages and bracketed by `glBegin` and `glEnd` messages, can be used to specify the shapes of objects. The `glBegin` command takes a *drawing primitive* as its argument that defines the algorithm by which the vertices are connected and filled. You can use the `glPolygonMode` command to expose the outlined structure of a shape, and you can color entire shapes or vertices using `glColor` messages. You can perform rotation, translation, and scaling of a defined shape by sending the `glMatrixView` command with the argument

modelview. Using `glrotate`, `gltranslate`, and `glscale` commands, you can then specify a viewing transformation. The `glpushmatrix` and `glpopmatrix` commands allow you to define a *stack* of these transformations so that you can change the rotation, position, and size of shapes either independently or in nested groups within a sequence of OpenGL commands.

Tutorial 41: Shaders

One of the primary purposes of graphics cards is to render 3D objects to a 2D frame buffer to display. How the object is rendered is determined by what is called the "shading model", which is typically fed information such as the object's color and position, lighting color and positions, texture coordinates, and material characteristics like how shiny or matte the object should appear. The program that executes in hardware or software to perform this calculation is called the *shader*. Traditionally graphics cards have used a fixed shading pipeline for applying a shading model to an object, but in recent years, graphics cards have acquired programmable pipelines so that custom shaders can be executed in place of the fixed pipeline. For a summary off many of the ways to control the fixed OpenGL pipeline, see *Tutorial 35: Lighting and Fog*.

Hardware Requirement: To fully experience this Tutorial, you will need a graphics card that supports programmable shaders, e.g. ATI Radeon 9200, NVIDIA GeForce 5000 series or later graphics cards. It is also recommended that you update your OpenGL driver with the latest available for your graphics card. On Macintosh, this is provided with the latest OS update. On PC, you can acquire the latest driver from either your graphics card manufacturer or your computer manufacturer.

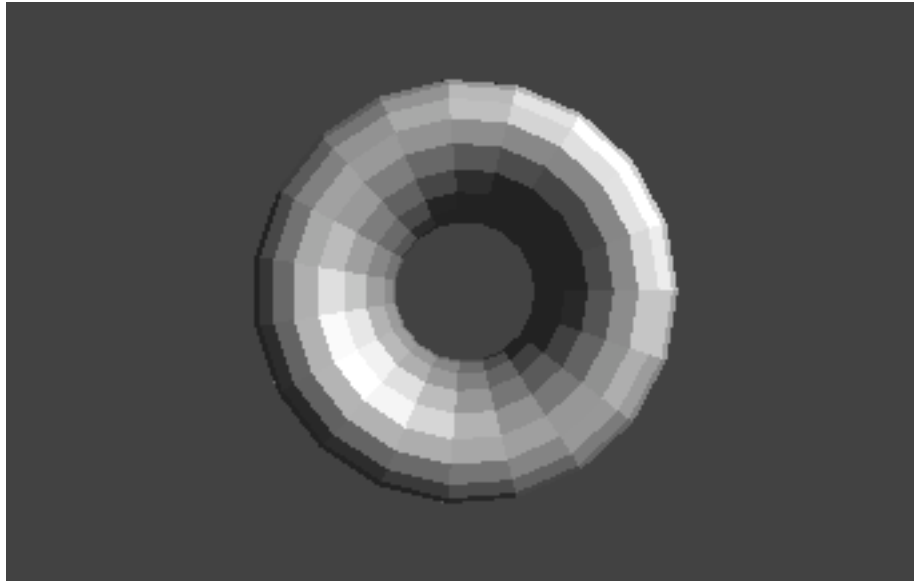
Flat Shading

One of the simplest shading models which takes lighting into account is called *Flat Shading* or *Facet Shading*, where each polygon has a single color across the entire polygon based on surface normal and lighting properties. As we saw demonstrated in *Tutorial 35*, this can be accomplished in Jitter by setting the `lighting_enable` attribute of a Jitter OpenGL object (e.g. `jit.gl.gridshape`) to 1.

Getting Started

- Open the Tutorial patch `41jShaders.pat` in the Jitter Tutorial folder. Click on the **toggle box** labeled *Start Rendering*.
- Click the **toggle box** object above the **message box** object reading `lighting_enable $1` to turn on lighting for the `jit.gl.gridshape` object drawing the torus. The `smooth_shading` attribute is 0 by default.

Now you should see a change in the lighting of the torus. Instead of the dull gray appearance it started with, you will see a shiny gray appearance like this:

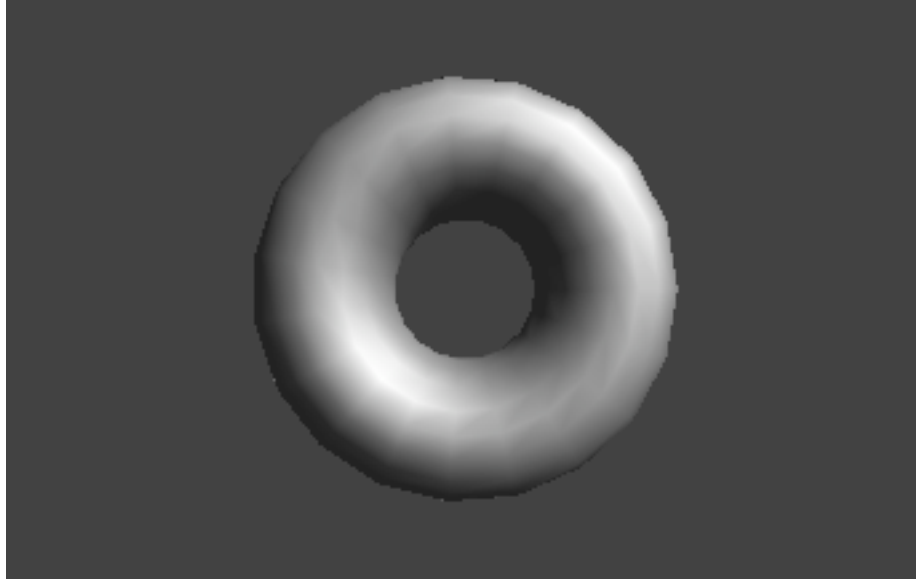


A rendered torus showing Flat shading.

Smooth Shading

While this might be a desirable look, the color of most objects in the real world changes smoothly across the surface. "Gouraud Shading" (1971) was one of the first shading models to efficiently address this problem by calculating a per vertex color based on surface normal and lighting properties. It then linearly interpolates color across the polygon for a smooth shading look. While the artifacts of this simple approach might look odd when using a small number of polygons to represent a surface, when using a large number of polygons, the visible artifacts of this approach are minimal. Due to the computational efficiency of this shading model, Gouraud Shading has been quite popular and remains the primary shading model used today by computer graphics cards in their standard fixed shading pipeline. In Jitter, this can be accomplished by setting the `smooth_shading` attribute to 1 (on). Let's do this in our patch, increasing and decreasing the polygon count by changing the `dim` attribute of the **`jit.gl.gridshape`** object.

- Click the **toggle box** attached to the **message box** reading `smooth_shading $1`. Try increasing and decreasing the polygon count by changing the **number box** attached to the message box reading `dim $1 $1` in the lower right of the patch.



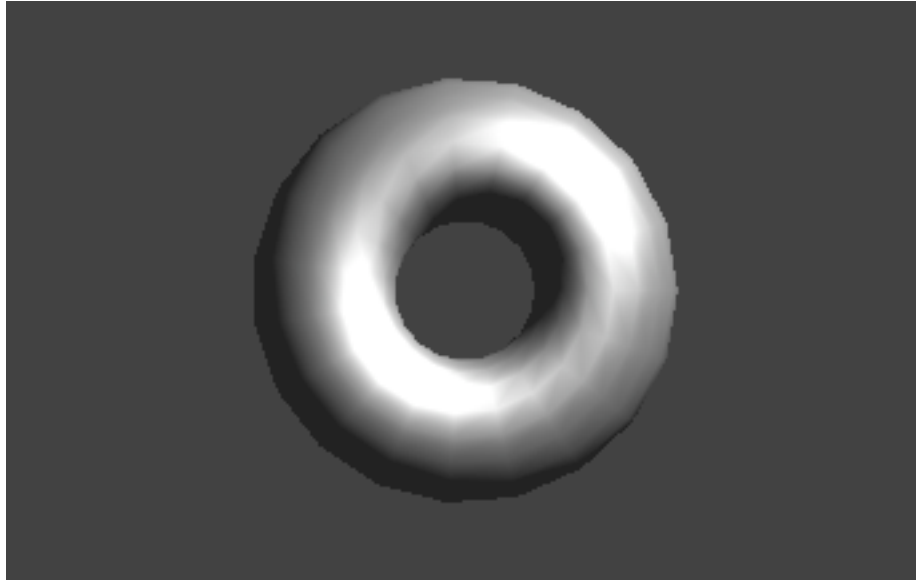
A torus rendered with smooth shading.

Per-Pixel Lighting

As mentioned, smooth shading with a low number of polygons has visible artifacts that result from just performing the lighting calculation on a per vertex basis, interpolating the color across the polygon. Phong Shading (1975) smooths not only vertex color across a polygon, but also smooths the surface normals for each vertex across the polygon, calculating a per fragment (or pixel) color based on the smoothed normal and lighting parameters. The calculation of lighting values on a per pixel basis is called "Per Pixel lighting". This is computationally more expensive than Gouraud shading but yields better results with fewer polygons. Per pixel shading isn't in the OpenGL fixed pipeline; however we can load a custom *shader* into the programmable pipeline to apply this shading model to our object.

- Load the per-pixel lighting shader by clicking the **message box** that says `read mat.dirperpixel.jxs` connected to the **jit.gl.shader** object.

- Apply the shader to our object by clicking the **message box** shader shademe connected to the **jit.gl.gridshape** object. This sets object's shader attribute to reference the **jit.gl.shader** object by its name (shademe).



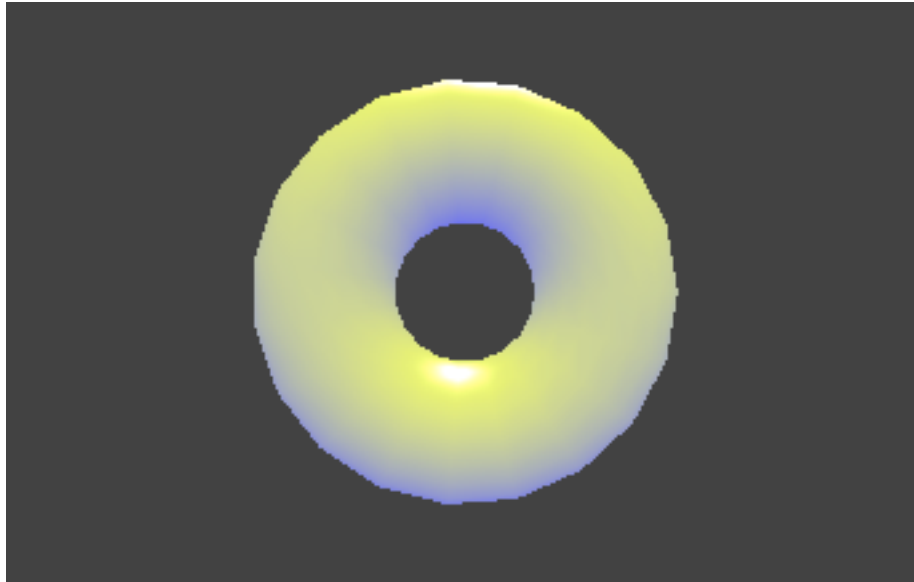
Per-pixel shading applied to the torus.

Programmable Shaders

In 1984, Robert Cook proposed the concept of a shade tree, where one could build arbitrary shading models out of some fundamental primitives using a "shading language". This shading language made it so that a rendering pipeline could be extended to support an infinite number of shaders rather than a handful of predefined ones. Cook's shading language was extended by Ken Perlin to contain control structures and became the model used by Pixar's popular *RenderMan* shading language. More recently the GPU-focused *Cg* and *GLSL* shading languages were established based on similar principles. The custom shaders used in this tutorial, including the per-pixel lighting calculation, were written in GLSL.

A brief introduction to how to write your own shaders will be covered in a subsequent Tutorial. For now, let's continue to use pre-existing shaders and show how we can dynamically change shader parameters. Gooch shading is a non-photorealistic shading model developed primarily for technical illustration. It outlines the object and uses warm and cool colors to give the sense of depth desired for technical illustrations. We have a custom shader that implements a simplified version of Gooch Shading which ignores the application of outlines.

- Load the simplified Gooch shader by clicking the **message box** read mat.gooch.jxs connected to the **jit.gl.shader** object.



The simplified Gooch shading model applied to our torus.

You should notice that the warm tone is a yellow color and the cool tone is a blue color. These values have been defined as defaults in our shader file, but are exposed as parameters that can be overridden by messages to the **jit.gl.shader** object.

- Using the **number boxes** in the patch, send the messages param warmcolor <red> <blue> <green> <alpha> and param coolcolor <red> <green> <blue> <alpha> to the **jit.gl.shader** object to change the tones used for the warm and cool colors.

We can determine parameters available to the shader by sending the **jit.gl.shader** object the message dump params to print the parameters in the Max window, or by sending the message getparamlist, which will output a parameter list out the object's rightmost (dump) outlet. An individual parameter's current value can be queried with the message getparamval <parameter-name>. A parameter's default value can be queried with the message getparamdefault <parameter-name>, and the parameter's type can be queried with getparamtype <parameter-name>.

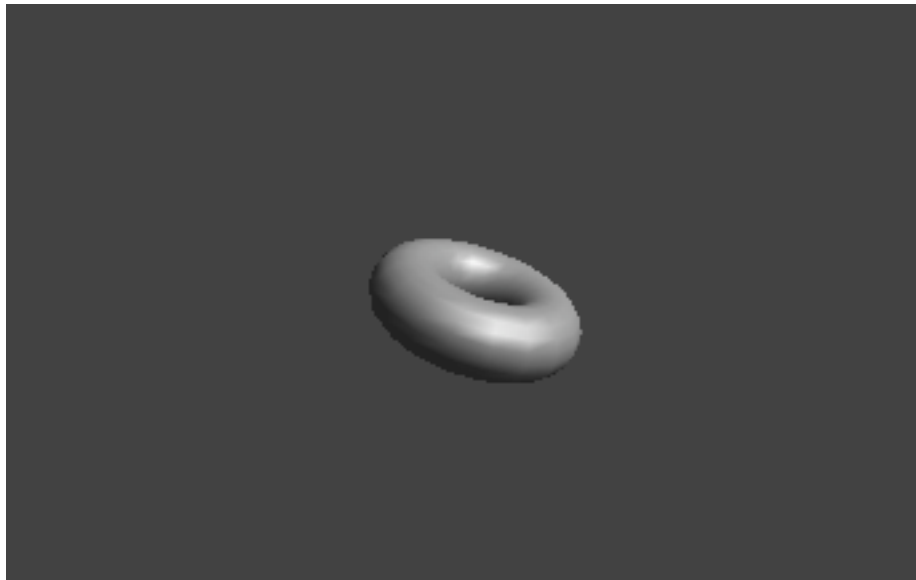
Vertex Programs

Shaders can be used not only to determine the surface color of objects, but also the position and attributes of our object's vertices, and as we'll see in our next Tutorials. For now, let's look at vertex processing. In the previous shaders we just discussed how the different shaders would render to pixels. In fact, for each of these examples we were

running two programs: one to process the vertices (the vertex program) and one to render the pixels (the fragment program). The vertex program is necessary to transform the object in 3D space (rotation, translation, scaling), as well as calculate per-vertex lighting, color, and other attributes. Since we see the object move and rotate as we make use of the **jit.gl.handle** object in our patch, obviously some vertex program must be running. Logically, the vertex program runs on the programmable vertex processor and the fragment program runs on the programmable fragment processor. This model fits with the fixed function pipeline that also separates vertex and fragment processing tasks.

The custom vertex program in the previous examples, however, didn't visibly perform any operation that is noticeably different than the fixed pipeline vertex program. So let's load a vertex shader that has a more dramatic effect. The *vd.gravity.js* shader can push and pull geometry based on the vertex distance from a point in 3D space.

- Load the simplified gravity vertex displacement shader by clicking the **message box** read *vd.gravity.js* connected to the **jit.gl.shader** object.
- Control the position and amount of the gravity vertex displacement shader by changing the **number boxes** connected to the **pak** object and **message box** outputting the messages *param gravpos <x> <y> <z>* and *param amount <n>*, respectively.



Vertex Distortion.

Summary

In this tutorial we discussed the fixed and programmable pipelines available on the graphics card and demonstrated how we can use the programmable pipeline by loading custom shaders with the **jit.gl.shader** object. Shaders can then be applied to 3D objects

to obtain different effects. We can also set and query shader parameters through messages to the **jit.gl.shader** object.

Tutorial 42: Slab: Data Processing on the GPU

We saw in the previous tutorial how custom shaders can be executed on the Graphics Processing Unit (GPU) to apply additional shading models to 3D objects. While the vertex processor and fragment processor are inherently designed to render 3D geometry, with a little creativity we can get these powerful execution units to operate on arbitrary matrix data sets to do things like image processing and other tasks. You might ask, "If we can already process arbitrary matrix data sets on the CPU with Jitter Matrix Operators (MOPs), why would we do something silly like use the graphics card to do this job?" The answer is speed.

Hardware Requirement: To fully experience this tutorial, you will need a graphics card which supports programmable shaders—e.g. ATI Radeon 9200, NVIDIA GeForce 5000 series or later graphics cards. It is also recommended that you update your OpenGL driver with the latest available for your graphics card. On Macintosh, this is provided with the latest OS update. On PC, this can be acquired from either your graphics card manufacturer, or computer manufacturer.

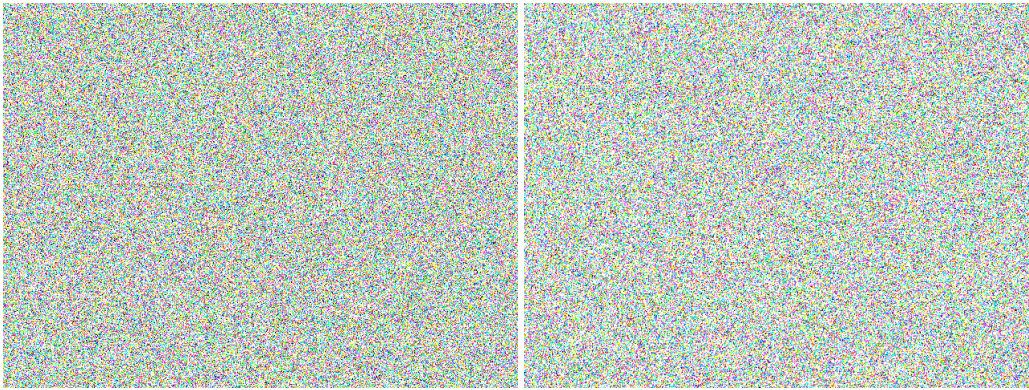
Recent Trends in Performance

The performance in CPUs over the past half century has more or less followed *Moore's Law*, which predicts the doubling of CPU performance every 18 months. However, because the GPU's architecture does not need to be as flexible as the CPU and is inherently *parallelizable* (i.e. multiple pixels can be calculated independently of one another), GPUs have been streamlined to the point where performance is advancing at a much faster rate, doubling as often as every 6 months. This trend has been referred to as *Moore's Law Cubed*. At the time of writing, high-end consumer graphics cards have up to 8 vertex pipelines and 24 fragment pipelines which can each operate in parallel, enabling dozens of image processing effects at DV resolution at full frame rate; we can even begin to process HD resolution images with full frame rate. Given recent history, it would seem that GPUs will continue to increase in performance at faster rates than the CPU.

Getting Started

- Open the Tutorial patch *42jSlab-comparisonCPU.pat* in the Jitter Tutorial folder. Click on the **toggle box** connected to the **qmetro** object. Note the performance of the patch as displayed by the **jit.fpsgui** object at the bottom.
- Close the patch and open the Tutorial patch *42jSlab-comparisonGPU.pat* in the Jitter Tutorial folder. Click on the **toggle box** connected to the **qmetro** object. Note the performance in the **jit.fpsgui** and compare it to what you got with the CPU version.

The patches don't do anything particularly exciting; they are simply a cascaded set of additions and multiplies operating on a 640x480 matrix of noise (random values of type char). One patch performs these calculations on the CPU, the other on the GPU. In both examples the noise is being generated on the CPU (this doesn't come without cost). The visible results of these two patches should be similar; however, as you probably will notice if you have a recent graphics card, the performance is much faster when running on the graphics card (GPU). Note that we are just performing some simple math operations on a dataset, and this same technique could be used to process arbitrary matrix datasets on the graphics card.



CPU (left) and GPU (right) processed noise.

What about the shading models?

Unlike the last tutorial, we are not rendering anything that appears to be 3D geometry based on lighting or material properties. As a result, this doesn't really seem to be the same thing as the shaders we've already covered, does it? Actually, we are still using the same vertex processor and fragment processor, but with extremely simple geometry where the pixels of the texture coordinates applied to our geometry maps to the pixel coordinates of our output buffer. Instead of lighting and material calculations, we can perform arbitrary calculations per pixel in the fragment processor. This way we can use shader programs in a similar fashion to Jitter objects which process matrices on the CPU (Jitter MOPs).

- Open the Tutorial patch *42jSlab-compositeDV.pat* in the Jitter Tutorial folder. Click on the **toggle** connected to the leftmost **qmetro** object.
- Click the **message boxes** containing *dvducks.mov* and *dvkite.mov* to load two DV movies, and turn on the corresponding **metro** objects to enable playback.

- Load a desired compositing operator from the **ubumenu** object connected to the topmost instance of **jit.gl.slab**.



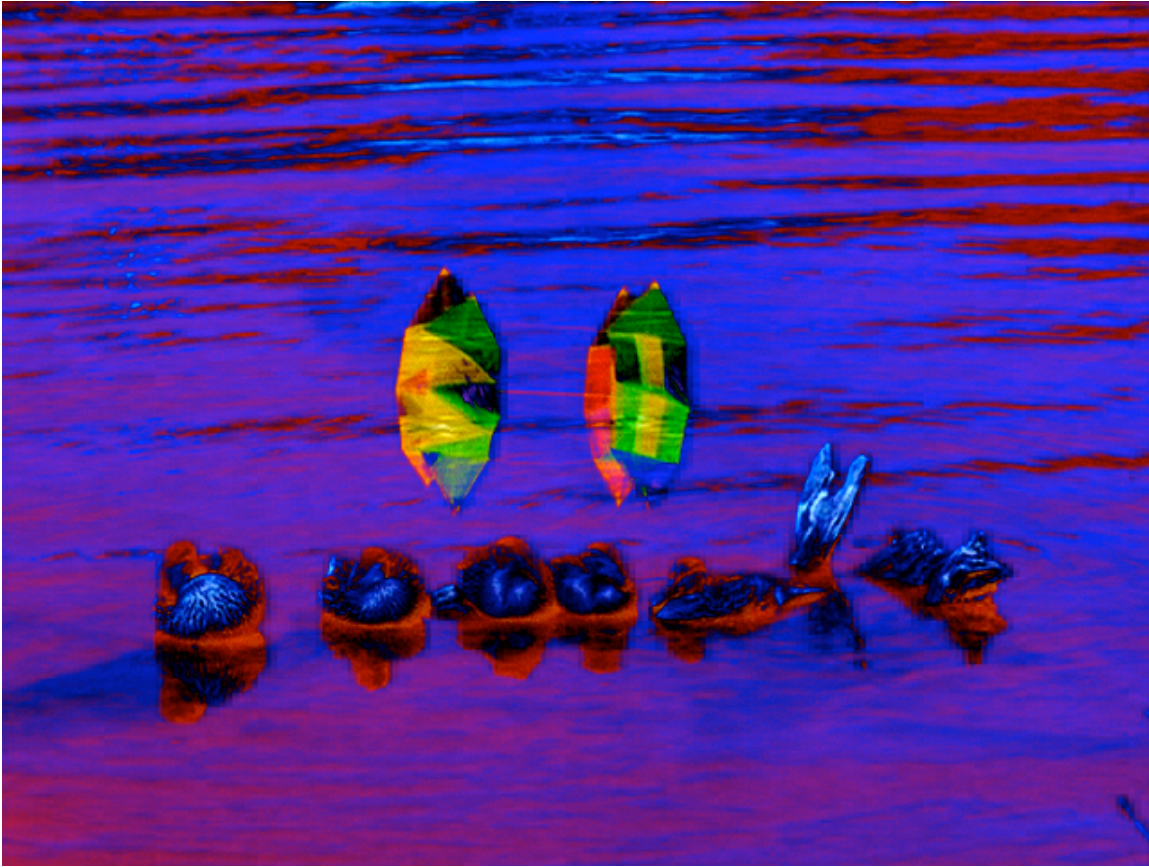
UYVY DV footage composited on GPU using "difference" op.

Provided that our hardware can keep up, we are now mixing two DV sources in real time on the GPU. You will notice that the **jit.qt.movie** objects and the topmost **jit.gl.slab** object each have their `colormode` attribute set to `uyvy`. As covered in the *Tutorial 49: Colorspaces*, this instructs the **jit.qt.movie** objects to render the DV footage to chroma-reduced YUV 4:2:2 data, and the **jit.gl.slab** object to interpret incoming matrices as such. We are able to achieve more efficient decompression of the DV footage using `uyvy` data because DV is natively a chroma-reduced YUV format. Since `uyvy` data takes up one half the memory of ARGB data, we can achieve more efficient memory transfer to the graphics card.

Let's add some further processing to this chain.

- Click the **message boxes** containing `read cf.emboss.jxs` and `read cc.scalebias.jxs` connected to the lower two instances of **jit.gl.slab**.

- Adjust these two effects by playing with the **number boxes** to the right to change the parameters of the two effects.



Additional processing on GPU.

How Does It Work?

The **jit.gl.slab** object manages this magic, but how does it work? The **jit.gl.slab** object receives either `jit_matrix` or `jit_gl_texture` messages as input, uses them as input textures to render this simple geometry with a shader applied, capturing the results in another texture which it sends down stream via the `jit_gl_texture <texturename>` message. The `jit_gl_texture` message works similarly to the `jit_matrix` message, but rather than representing a matrix residing in main system memory, it represents a texture image residing in memory on the graphics hardware.

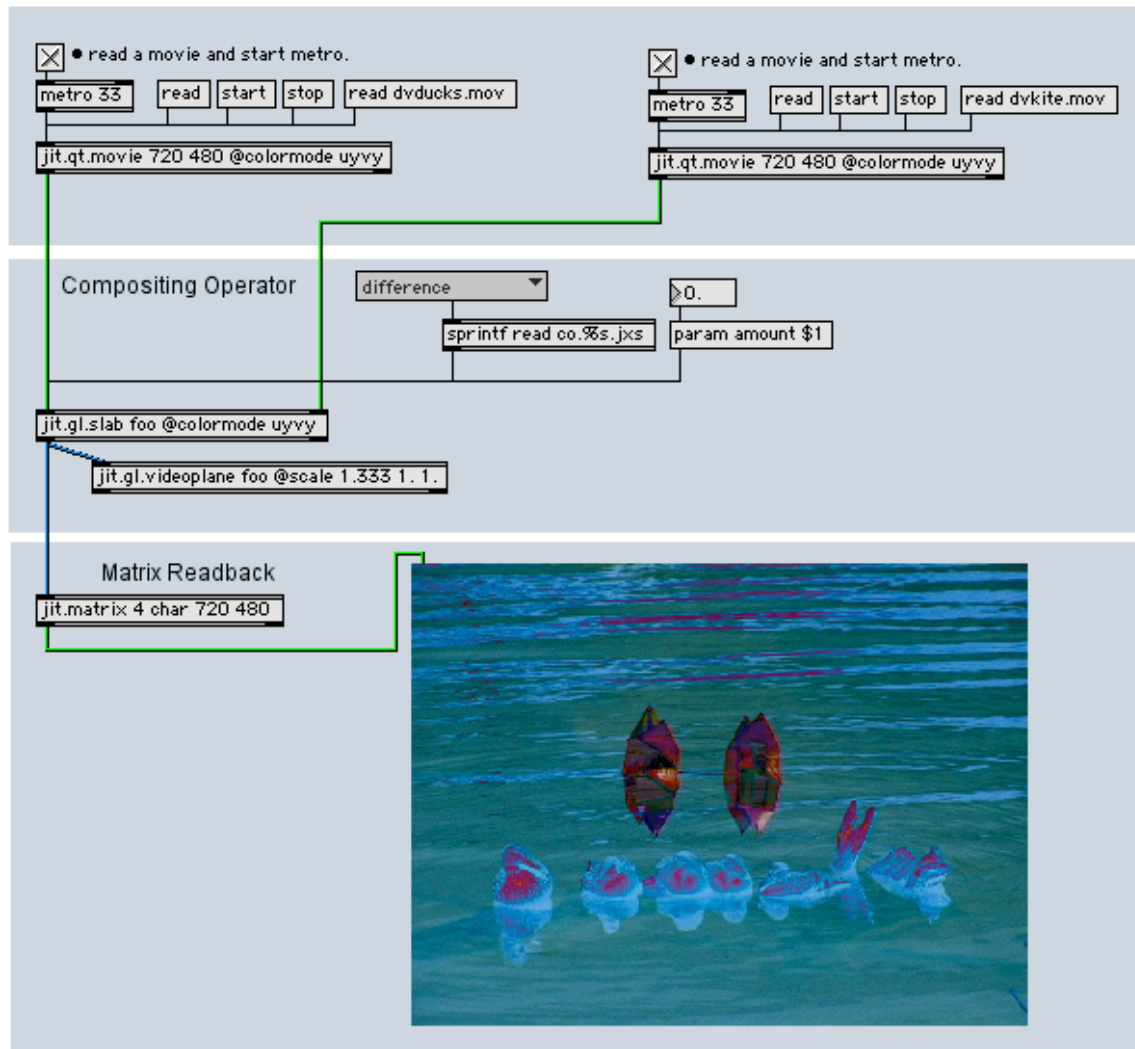
The final display of our composited video is accomplished using a **jit.gl.videoplane** object that can accept either a `jit_matrix` or `jit_gl_texture` message, using the received input as a texture for planar geometry. This could optionally be connected to some other object like **jit.gl.gridshape** for texturing onto a sphere, for example.

Moving from VRAM to RAM

The instances of **jit.gl.texture** that are being passed between **jit.gl.slab** objects by name refer to resources that exist on the graphics card. This is fine for when the final application of the texture is onto 3D geometry such as **jit.gl.videoplane** or **jit.gl.gridshape**, but what if we want to make use of this image in some CPU based processing chain, or save it to disk as an image or movie file? We need some way to transfer this back to system memory. The **jit.matrix** object accepts the `jit_gl_texture` message and can perform what is called *texture readback*, which transfers texture data from the graphics card (VRAM) to main system memory (RAM).

- Open the Tutorial patch *42jSlab-readback.pat* in the Jitter Tutorial folder. Click on the **toggle boxes** connected to the leftmost **qmetro** object. As in the last patch we looked at, read in the movies by clicking on the **message boxes** and start the **metro** object on the right side of the patch.

DV Source Movies



Matrix readback from GPU.

Here we see that the image is being processed on the GPU with **jit.gl.slabs** object and then copied back to RAM by sending the `jit_gl_texture <texturename>` message to the **jit.matrix** object. This process is typically not as fast as sending data to the graphics card, and does not support reading back in a chroma-reduced UYVY format. However, if the GPU is performing a fair amount of processing, even with the transfer from the CPU to the GPU and back, this technique can be faster than performing the equivalent processing

operation on the CPU. It is worth noting that readback performance is being improved in recent generation GPUs; in particular, PCI-e based graphics cards typically offer better readback performance than AGP based graphics cards.

Summary

In this tutorial we discussed how to make use of **jit.gl.slabs** object to use the GPU for general-purpose data processing. While the focus was on processing images, the same techniques could be applied to arbitrary matrix datasets. Performance tips by using chroma reduced uyvy data were also covered, as was how to read back an image from the GPU to the CPU.

Tutorial 43: A Slab of Your Very Own

While there are many shaders that provided with the Jitter distribution, many of which can be composed to form more complicated operations, one of the most exciting aspects of Jitter's shader support is that you can write your own. Writing shaders is fundamentally not a difficult process; however, it does involve text-based programming, so some understanding of a programming language like C, Java, or Javascript is recommended before you take a crack at writing a shader.

Since there are a few more things to consider once you take lighting and complex geometry into account, we are going to focus on the simple data processing of the **jit.gl.slab** object. We will use the GLSL shader language for this Tutorial, but note that Jitter also supports programs written in Cg, as well as the ARB and NV assembly languages. Unfortunately it is out of the scope of the Jitter documentation to fully describe any of these languages. This Tutorial is meant to give you a very simple introduction to GLSL and demonstrate how to integrate shader code into a format Jitter understands. We will show you the basics of what you need to know and refer you to more authoritative sources to find out more.

Hardware Requirement: To fully experience this tutorial, you will need a graphics card that supports programmable shaders--e.g. ATI Radeon 9200, NVIDIA GeForce 5000 series or later graphics cards. It is also recommended that you update your OpenGL driver with the latest available for your graphics card. On Macintosh, this is provided with the latest OS update. On PC, this can be acquired from either your graphics card manufacturer, or computer manufacturer.

Mixing Multiple Sources

Let's start with a 4-source video mixer. This could already be accomplished with a handful of instances of either the provided math or compositing shaders, but it will be more efficient to do this all at once in a single shader. A mathematical formula for this 4 source mix might look something like the following, where *a*, *b*, *c*, and *d* are constants for how much of each input we want to accumulate.

output = a*input0 + b*input1 + c*input2 + d*input3

In GLSL, this might look something like the following:

```
uniform vec4 a;
uniform vec4 b;
uniform vec4 c;
uniform vec4 d;

void main (void)
{
    vec4 input0;
    vec4 input1;
    vec4 input2;
    vec4 input3;
    vec4 output;

    output = a*input0 + b*input1 + c*input2 + d*input3;
}
```

We’ve defined a few global variables: *a*, *b*, *c*, and *d*, as well as a function called `main()` that contains local variables *input0*, *input1*, *input2*, *input3*, and *output*. The `uniform` keyword signifies that the value will be uniform (constant) for the extent of the fragment program’s execution—i.e. all fragments will have the same values for *a*, *b*, *c*, and *d*. The `vec4` type is a vector with four values, which typically refer to a point (x,y,z,w) in homogenous coordinates or a color (r,g,b,a). When we multiply two `vec4` elements in GLSL as above, it is a pointwise multiplication, resulting in another `vec4` where each element of the multiplication is the product of the same element from the two operands—e.g. with $m = p * q$: $m.r = p.r * q.r$; $m.g = p.g * q.g$; $m.b = p.b * q.b$; $m.a = p.a * q.a$.

This program is a fragment program, and our main function will be executed once for each fragment (or pixel) in the image. It has no knowledge about adjacent output pixels or when it is being run. This is what permits it to run several instances in parallel for multiple fragments in order to obtain such high performance. CPUs don’t have the same kind of restrictions, but then the computation is not inherently parallelizable.

For detailed information about all things GLSL related (keywords, built in operators, built-in variables, syntax, etc.), we recommend reading the *OpenGL Shading Language Reference* (aka “The Orange Book”) and the *OpenGL Shading Language Specification*. The specification is available online at opengl.org. There are also several tutorials online for GLSL such as the one hosted at lighthouse3d.com.

Fragment Program Input and Output

The above code is valid GLSL and looks similar to our original formula. However, it won’t work yet. Right now we’ve declared variables for input and output, but these values

aren't actually making use of any input and not assigning any values to the output in the OpenGL pipeline. To get input, we need to sample values from textures, and to write to our output, we need to write to the built-in `gl_FragColor` variable. With this in mind, we make the following modifications to our code:

```
uniform vec4 a;
uniform vec4 b;
uniform vec4 c;
uniform vec4 d;

// define our varying texture coordinates
varying vec2 texcoord0;
varying vec2 texcoord1;
varying vec2 texcoord2;
varying vec2 texcoord3;

// define our rectangular texture samplers
uniform sampler2DRect tex0;
uniform sampler2DRect tex1;
uniform sampler2DRect tex2;
uniform sampler2DRect tex3;

void main (void)
{
    // sample our textures
    vec4 input0 = texture2DRect(tex0, texcoord0);
    vec4 input1 = texture2DRect(tex1, texcoord1);
    vec4 input2 = texture2DRect(tex2, texcoord2);
    vec4 input3 = texture2DRect(tex3, texcoord3);
    vec4 output;

    // perform our calculation
    output = a*input0 + b*input1 + c*input2 + d*input3;

    // write our data to the fragment color
    gl_FragColor = output;
}
```

The `varying` keyword means that the parameter will be changing for each fragment. For example, `texcoord0` will reference the interpolated (x,y) pixel coordinate to extract from our leftmost input texture. We are able to sample this texture (i.e. get the color value associated with the texture coordinate) by calling the `texture2DRect()` function with the corresponding sampler ID and texture coordinate vector. The `sampler2DRect` data type and `texture2DRect()` function indicate that we are using *rectangular* textures. Ordinary textures in OpenGL have dimensions restricted to powers of two for each dimension (e.g. 256x256) and the values are indexed with “normalized” coordinates (fractional values from 0.-1.). Rectangular textures on the other hand permit arbitrary dimensions (e.g. 720x480), and the coordinates are in terms of pixel position (e.g. 0.-720., and 0.-480.). In Jitter we use rectangular textures as our default texture type for increased

performance when working with datasets that are not powers of two, as is often the case when working with video. Ordinary textures can still be used via the `sampler2D` data type and the `texture2D()` function, but they require that the input textures have been created with the `rectangle` attribute set to 0.

The Vertex Program

The above code is now our complete GLSL fragment program, but it still won't do anything yet, because it requires a vertex program pass on our texture coordinates. As the name suggests, the vertex program runs once for each vertex in the geometry. It is where any modification to the geometry is calculated, including the attachment of texture coordinates to that geometry. Values generated by the vertex program can be automatically interpolated across the surface of the geometry as we typically want to do with texture coordinates. For our vertex program, we will want to transform our vertex by the current model-view projection matrix, transform our texture coordinates by the current texture transform matrix (this is how we scale and possibly flip our rectangular texture coordinates), and then pass on our texture coordinates as varying values to our fragment program. Such a vertex program might look like this:

```
// define our varying texture coordinates
varying vec2 texcoord0;
varying vec2 texcoord1;
varying vec2 texcoord2;
varying vec2 texcoord3;

void main( void )
{
    // the output vertex position to the input vertex position
    // transformed by the current ModelViewProjection matrix
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    // assign our varying texture coordinates to the
    // input texture coordinate values transformed
    // by the appropriate texture matrix. This is
    // necessary for rectangular and flipped textures
    texcoord0 = vec2(gl_TextureMatrix[0] * gl_MultiTexCoord0);
    texcoord1 = vec2(gl_TextureMatrix[1] * gl_MultiTexCoord1);
    texcoord2 = vec2(gl_TextureMatrix[2] * gl_MultiTexCoord2);
    texcoord3 = vec2(gl_TextureMatrix[3] * gl_MultiTexCoord3);
}
```

Wrapping in a Jitter XML Shader (JXS)

These programs together will perform all the hard work to process all of our pixel data, but we're still missing one final component. In order for Jitter to load these programs and expose parameters to the user, we need to package these programs in a *Jitter XML Shader*

file (JXS). In this file we will specify user settable variables *a*, *b*, *c*, and *d* with default values, bind our multiple texture units so that the program can access them properly, define our programs, and bind our user variables to our program variables:

```
<jittershader name="fourwaymix">
  <param name="a" type="vec4" default="0.25 0.25 0.25 0.25" />
  <param name="b" type="vec4" default="0.25 0.25 0.25 0.25" />
  <param name="c" type="vec4" default="0.25 0.25 0.25 0.25" />
  <param name="d" type="vec4" default="0.25 0.25 0.25 0.25" />
  <param name="tex0" type="int" default="0" />
  <param name="tex1" type="int" default="1" />
  <param name="tex2" type="int" default="2" />
  <param name="tex3" type="int" default="3" />
  <language name="glsl" version="1.0">
    <bind param="a" program="fp" />
    <bind param="b" program="fp" />
    <bind param="c" program="fp" />
    <bind param="d" program="fp" />
    <bind param="tex0" program="fp" />
    <bind param="tex1" program="fp" />
    <bind param="tex2" program="fp" />
    <bind param="tex3" program="fp" />
    <program name="vp" type="vertex"
source="43j-fourwaymix.vp.glsl" />
    <program name="fp" type="fragment"
source="43j-fourwaymix.fp.glsl" />
  </language>
</jittershader>
```

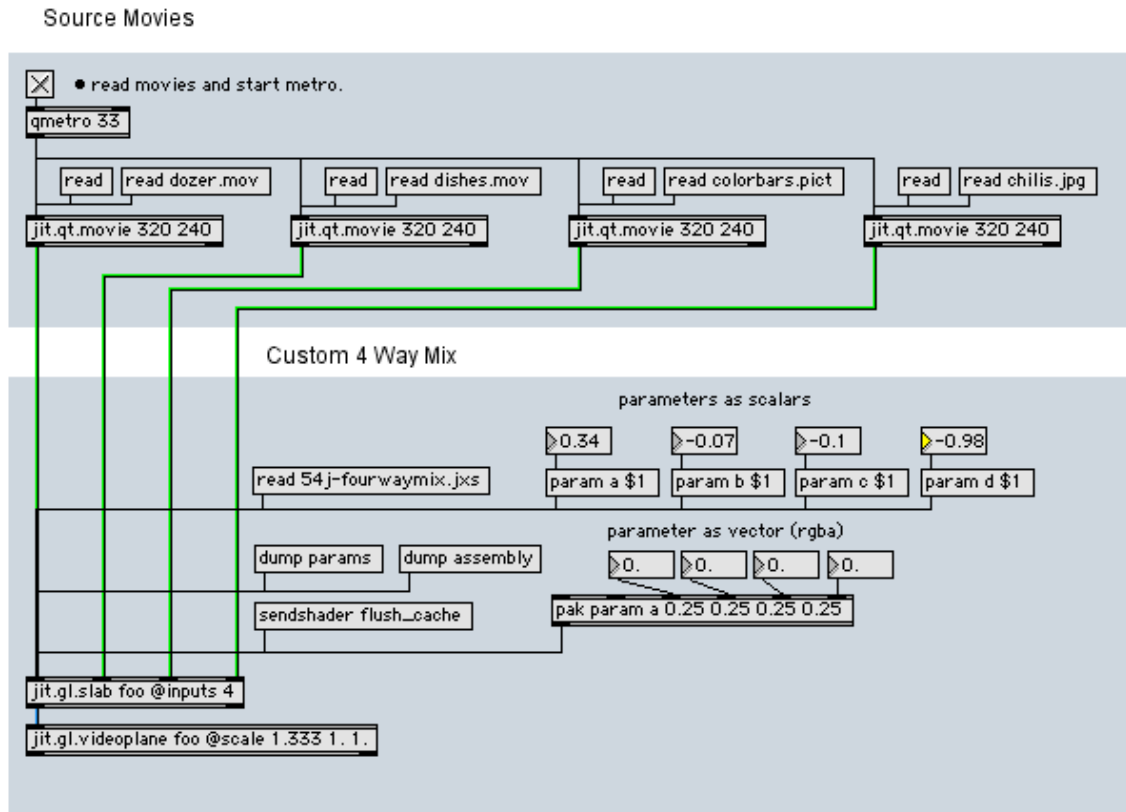
The `jittershader` tag defines our shader with an optional name. The `param` tag defines a parameter to be exposed to the Max environment with a name, type, and optional default value. The `language` tag defines a block of programs in the specified language and version. The `bind` tag binds user-exposed parameters to variables in specific programs. Finally, the `program` tag defines our programs with a name, type, and source file. The source may optionally be embedded in the XML file itself by including it inside either an XML comment or CDATA block; this is the case with many of the shader files provided in the Jitter distribution. For a complete reference of XML tags available, please see *Appendix D: The JXS File Format*.

Ready for Action

Now that all the pieces are in place, let's see our program do its job.

- Open the Tutorial patch *43jYourVeryOwnSlab.pat* in the Jitter Tutorial folder. Click on the **toggle box** connected to the **qmetro** object.
- Read in a movie for each instance of **jit.qt.movie**, using either those provided in the **message box** objects or media of your own choosing.

- Load the shader by sending the message read 43j-fourwaymix.jxs to the **jit.gl.slab** object.



The patch that puts it all together.

You will notice that the **jit.gl.slab** object has its inputs attribute set to 4. This is necessary to have more than the default 2 inputs, as we want for our 4-way mix shader. We can also find out a bit about our program by sending the message dump params or dump assembly to see what our code was compiled down to. Lastly, it is worth noting that for optimal performance when loading and compiling shaders, Jitter keeps a cache of all compiled shaders; it will only recompile the shader if it detects that the source files have been modified on disk. To flush the cache, you can send **jit.gl.slab** the message flush_cache.

Summary

In this tutorial we demonstrated how to build a shader from scratch in GLSL that mixes 4 input sources on the GPU via a **jit.gl.slab** object. This was accomplished by writing a fragment program, a vertex program, and wrapping them in a Jitter XML Shader file so that our shader could be loaded into Jitter with parameters exposed to the user in Max. We also demonstrated the use of more than 2 inputs for the **jit.gl.slab** object by using the inputs attribute.

Tutorial 44: Flash Interactivity

Macromedia's Flash file format has been a highly successful and widely used means for creating and distributing interactive content—particularly over the Internet. Since the introduction of Jitter, users have been able to take advantage of QuickTime's ability to read Flash media within the **jit.qt.movie** object, although the amount of control available was limited to basic temporal and spatial manipulation. In this chapter, we'll explore some of the possibilities available to users of Macromedia's Flash environment.

A Little Background

Flash was introduced in 1996 as a vector-based animation development tool, with output files that were optimized for internet-based distribution (that means small!). Over the years, the basic toolset has expanded mightily, gradually encompassing functionality familiar to users of Macromedia's other popular animation package, Director. This expansion included object-based interactivity and—in the last several versions—a fully featured scripting language called ActionScript, which is based on JavaScript. The Flash file format has remained extremely compact and has generated such broad interest that other software manufacturers have provided support. This is particularly due to the fact that the format specification itself is open.

Jitter's support for Flash falls into two basic categories: on-screen interactivity and script interactivity. On-screen interactivity refers to things like button-clicks, mouse movements and keyboard events—the kinds of interactions that we're familiar with from using Flash in web browsers. Script interactivity is the ability to learn and change the properties of ActionScript objects and variables. Working with ActionScript properties from Max provides a powerful means for developers of Flash and Max content to link the two environments.

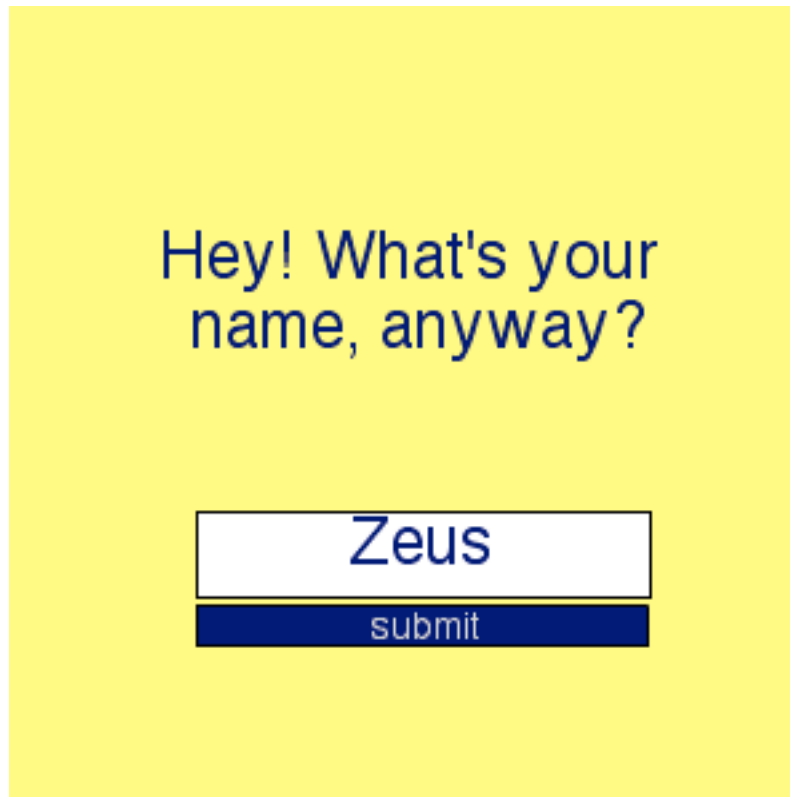
Clicks and Pops

Using the on-screen Flash interactivity features of Jitter doesn't require any knowledge of Flash or the underlying programming inside of a Flash document.

- Open the tutorial patch *44jFlashInteractivity.pat* in the Jitter Tutorials folder.

In fact, there should be no unfamiliar objects in this patch—all of Jitter's Flash functionality is bundled inside the **jit.qt.movie** object. With the help of the additional objects in the patch, we are able to route mouse and keyboard data to a Flash movie.

- Click the **message box** labeled read 44jBasic.swf. Click the **toggle box** labeled *Display* connected to the **metro** object at the top of the patch. You should hear some soothing music and see an introductory screen in the **jit.pwindow** object.



The pinnacle of interface design: the introductory screen of 44jBasic.swf.

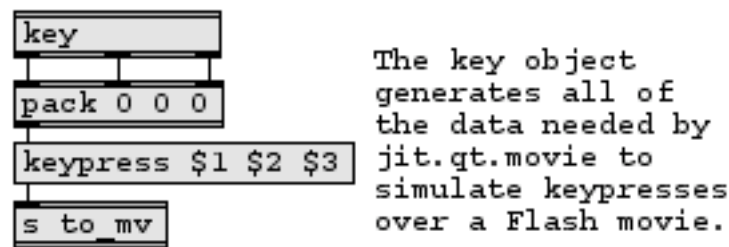
- Take a few moments to become familiar with the Flash movie by clicking on the interface in the **jit.pwindow** object. There are opportunities to select and enter text, to rollover and click on buttons and to hear some excellent sounds.

Pretty great Flash movie, right? So, let's understand how we're interacting with it. Since the movie isn't being played in a special window that knows the type of media being displayed (like with Apple's QuickTime Player, which can automatically pass events from the user interface to the movie), we have to manually pass events to the **jit.qt.movie** object, which passes them along to the movie. Luckily, Max and Jitter already contain objects that can sense things like mouse position and keyboard events, so most of our work has already been done for us.

We're using the mouse and mouseidle notification messages of the **jit.pwindow** object to find out where the mouse is relative to the object's area. By default, the **jit.pwindow** object outputs a mouse message every time the mouse is clicked over the object. We've also told the object to report idle mouse movements by sending it the `idlemouse 1` message (when we save the patch, this `idlemouse` preference is saved). This causes the `mouseidle` message to be output whenever the mouse enters the screen region of the **jit.pwindow** object. The **jit.window** object outputs similar information, or we could also use Max's **mousestate** object if we wanted to.

The **jit.qt.movie** object understands the message mouse in exactly the same format output by the **jit.pwindow** object to simulate mouse movement or clicks over a Flash movie. Since **jit.qt.movie** doesn't understand the mouseidle message, we have to change all of the messages beginning with the word mouseidle to begin with the word mouse, which we're doing with the **route** and **prepend** objects. The newly formatted message is then sent to the **jit.qt.movie** object and will trigger the appropriate interactivity in the Flash movie.

Similarly, to handle keyboard events, we can use Max's **key** object to acquire the necessary information, which we pass along to the **jit.qt.movie** object as a keypress message (the three arguments for the keypress message match exactly with the three numbers output from the **key** object).



*From the **key** object to a keypress message and beyond.*

That's it! On-screen Flash interaction in Jitter is really very simple to set up, and offers access to a large and exciting selection of interactive media.

Hips to Scripts

If you're a Flash developer (or interested in becoming one), you'll be pleased to know that we can interact directly with ActionScript inside of Flash movies. We can change variables and otherwise influence object behavior by sending Max messages to the **jit.qt.movie** object.

- Click the **message box** labeled read 44jPhysics.swf. If the **toggle box** labeled *Display* is turned off, turn it back on in order to see the output of the Flash movie.

This movie is a simple physical simulator. By clicking and "throwing" the grey ball, we can see that the ball behaves similarly to real thrown objects in the world. It travels through the air and, depending on how hard it is thrown, bounces off of walls, losing a little bit of energy as it does, until it comes to rest on the ground.

This behavior has been coded into the Flash movie using Macromedia's scripting language, ActionScript.

```
onClipEvent(enterFrame) {
    // less is unnecessary
    if (_root.airfriction < 0.5)
        _root.airfriction = 0.5;

    if (!dragging) {
        vel.y += _root.gravity;

        pos.x += vel.x;
        pos.y += vel.y;

        // Has the ball left the bottom of the stage?...
        if( pos.y + radius > movie.height ) {
            pos.y = movie.height - radius;
            vel.y *= -_root.loss;
            vel.x *= _root.friction;
        }
    }
}
```

An excerpt from the ActionScript code used in our physics simulator.

If you're familiar with Max's JavaScript implementation (or JavaScript in general), ActionScript should look familiar. In fact, ActionScript *is* JavaScript, albeit with some extra Flash-specific Objects and functions (just like Max's implementation is standard JavaScript, which contains specific Objects and functions that are only relevant to Max).

There are now two versions of ActionScript, called ActionScript 1.0 and ActionScript 2.0. At this time, QuickTime (and Jitter) only supports ActionScript 1.0 features in Flash files saved with Flash Player 5 compatibility. This means that some of Flash's newer features (Flash MX and later) are not currently available for Jitter users.

Our Flash document (the *44jPhysics.fla* file used to generate our Flash movie) defines some global ActionScript variables. Taking a look at the Actions for our Frame, we see:

```
// factor to multiply our energy by, when we hit a wall
var loss = 0.6;
// factor to divide our velocity by, as we travel through the air
var airfriction = 2.;
```

We can send the message `flash_var` to the **jitter.qt.movie** object to modify the values of these variables.

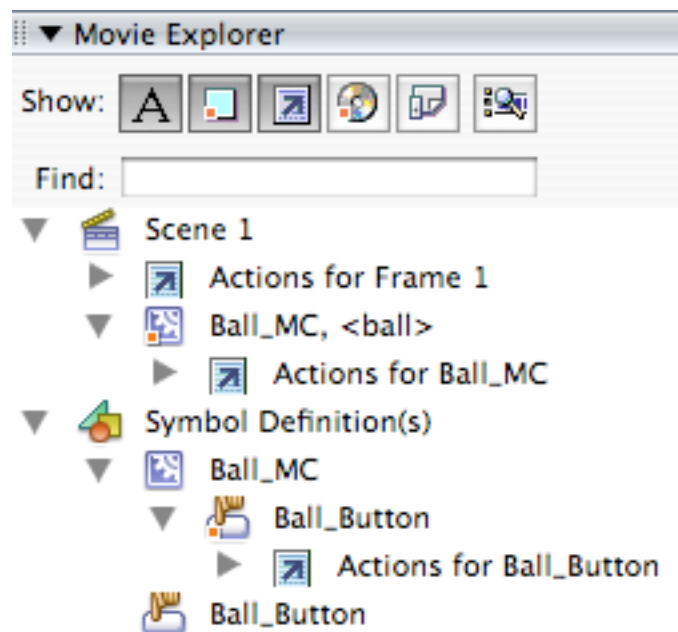
- Change the values in the **number boxes** labeled *Loss Factor* and *Air Friction Divisor* and note how the ball's behavior changes.

We can also query the value of a variable by sending the message `getflash_var` to the **jit.qt.movie** object.

- Click the message boxes labeled `getflash_var airfriction` and `getflash_var loss` and verify that your changes were updated inside of the Flash movie. The output should be posted in the Max Window.

Although the output appears to be numeric, all output from the `getflash_var` message is symbolic. If you are working with numeric data, rather than strings, you will need to use the **fromsymbol** object to convert the symbols to numbers. Input can be any type.

So far, we've seen how to set and get global ActionScript variables. But what about nested variables? Examining the *44jPhysics.fla* document within the Macromedia Flash application (sold separately), we can see the structure of our movie.



The structure of our Flash movie, and its symbol definitions, as seen in Flash

Our movie is really quite simple—1 scene and 1 frame. The frame contains 2 items: some actions (ActionScript code) and a Movie Clip instance of the symbol *Ball_MC*, whose instance name is *ball*. Looking at *Ball_MC*'s symbol definition, we can see that *Ball_MC* contains a Button called *Ball_Button*, which itself contains some actions.

If we want to know the x coordinate of our Movie Clip instance *ball*, we can use Flash's path syntax: */path/to.variable*. In our case, this is */ball._x*, since *ball* is at the root level of the movie, and *_x* is the built-in variable describing the x position of an object on the Flash Stage.

- Click on the **message box** labeled `getflash_var /ball._x`, `getflash_var /ball._y` and watch the values in the floating point **number boxes** change to reflect the position of the ball. Note the use of the **fromsymbol** objects to convert the symbolic Flash output to floating point numbers.
- Turn on the **toggle box** labeled *Enable Sound*. We're analyzing the x and y positions of the ball to detect when the ball has hit a wall, and playing a tone in response.

Interestingly, we can't *set* the coordinates in this direct fashion. We need to have a special variable representing the built-in variable, which gets checked every frame. See the `more_setting patcher` and the source code in the *44jPhysics.fla* document for more information on how this is done.

At the time of this writing, there is an additional limitation to getting and setting Flash variables. Variables must be simple—arrays don't work properly. As you design your ActionScript, this should be kept in mind to avoid a lot of editing later in the process!

Summary

In this tutorial, we learned how to take advantage of Flash interactivity using the **jit.qt.movie** object's mouse and keypress messages to simulate mouse-movements and keyboard activity. We also examined the `flash_var` and `getflash_var` messages, which can be used to set and get ActionScript variables and influence the running of scripts within Flash movies.

Tutorial 45: Introduction to using Jitter within JavaScript

The Max **js** object, introduced in Max 4.5, allows us to use procedural code written in the JavaScript language within Max. In addition to implementing the core JavaScript 1.5 language, the Max **js** object, contains a number of additional objects and methods specific to Max, e.g. the **Patcher** object (designed for interfacing with the Max patcher) or the `post ()` method (for printing messages into the Max window). There are a number of extensions to the **js** object that allow us to perform Jitter functions directly from the JavaScript language when working with Jitter. For example, the Jitter extensions to **js** allow us to:

- Instantiate Jitter objects directly within JavaScript and create function chains of Jitter processes within procedural code.
- Create Jitter matrices with JavaScript and access and set the values and parameters of Jitter matrices from within JavaScript functions.
- Use Jitter library operations (e.g. **op** and **functor** objects) to do fast matrix operations on Jitter matrices within JavaScript to create low-level Jitter processing systems.
- Receive callbacks from Jitter objects by listening to them and calling functions based on the results (e.g. triggering a new function whenever a movie loops).

Before beginning this tutorial, you should review the basics of using JavaScript in Max by looking at several tutorials found in the **Max Tutorials and Topics** manual: *Tutorial 48: Basic Javascript* and *Tutorial 49: Scripting and Custom Methods in JavaScript*. These tutorials cover the basics of instantiating and controlling a function chain of Jitter objects within **js** JavaScript code.

- Open the tutorial patch *45jJavaScriptIntro.pat* in the Jitter Tutorials folder.

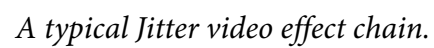
The tutorial patch shows us two columns of Max objects side-by-side. The right column contains a patch that processes a movie through an effect and displays it. All of this is done using Jitter objects connected within the Max patch between the **qmetro** object and the **jit.pwindow** object. The left column shows the **qmetro** and **jit.pwindow** objects, but contains only a **js** object loading the JavaScript file *45jWakefilter.js* in between. As we will learn, the two sides of the patch do pretty much the exact same thing. First, we'll look at the patch on the right side to see what's happening.

Waking Up

- On the right side of the patch, click the **toggle box** labeled *Display Processing using Patcher*. Click the **message box** that reads *read countdown.mov*, also on the right side.

We use **qmetro** objects instead of **metro** objects in our patch because of the potential for scheduler backlog when working with JavaScript. The normal behavior of the Max **js** object is for it to create a queue of pending events while it executes the current one; as a result, a fast **metro** object will quickly accumulate a large backlog of bang messages for the **js** object to deal with. The **qmetro** object sends bang messages to the back of the low priority queue where they can be *usurped* by subsequent messages. See *Tutorial 16: Using Named Jitter Matrices* for a more in-depth discussion on this topic.

- The movie should appear in the **jit.pwindow** with a gradually changing colored effect.



This side of the patch plays back a QuickTime movie (using a **jit.qt.movie** object) into an edge detection object (**jit.robcross**), which is then multiplied by the output of a named matrix called **bob** (using **jit.op**). The matrix output by **jit.op** is then processed by a **jit.wake** object, which applies a temporal feedback and spatial convolution to the matrix, the parameters of which can be controlled independently for each plane. The output of the **jit.wake** object is then brightened slightly (with a **jit.brcosa** object) and then stored back into our named matrix (**bob**). The output of our effects chain as seen in the **jit.pwindow** is the output of the **jit.wake** object.

The technique of using named Jitter matrices for feedback is covered in *Tutorial 17: Feedback Using Named Matrices*. The **jit.robcross** object applies the *Robert's Cross* edge detection algorithm (a similar object that allows us to use two other algorithms is called **jit.sobel**). The **jit.wake** object contains an internal feedback matrix that is used in conjunction with image convolution to create a variety of motion and spatial blur effects (similar effects could be constructed using objects such as **jit.slide** and **jit.convolve**).

- Open the **patcher** object named **random_bleed**.

The key to the variation of our processing algorithm is this subpatch, containing twelve **random** objects that are controlling different parameters of the **jit.wake** object in the main processing chain. The output of these **random** objects is scaled for us (with **scale** objects) to convert our integer random numbers (0 to 999) into floating-point values in the range 0 to 0.6. These values are then smoothed with the two ***** objects and the **+** object, implementing a simple one-pole filter:

$$y_n = 0.01x_n + 0.99y_{n-1}$$

These smoothed values then set the attributes of **jit.wake** that control how much bleed occurs in different directions (up, down, left, right) in different planes (specified as the color channels red, green, and blue). You'll notice that the smoothing algorithm is such that the values in all of the **number box** objects showing the smoothed output tend to hover around 0.3 (or half of 0.6). Our Jitter algorithm exhibits a slowly varying (random) color shift because of the minute differences between these sets of attributes.

- Back in the main tutorial patcher, try changing movies by clicking the **message box** objects reading **read wheel.mov** and **read dozer.mov**. Compare the effects on these two movies with the effect on the "countdown" movie. Note that when we read in new movies, we initialize the **bob** matrix to contain all values of 255 (effectively clearing it to white).

The Javascript Route

- Shut off the **qmetro** object on the right side of the patch by clicking the **toggle box** above it. Activate the **qmetro** object on the left side of the patch by clicking the **toggle box** attached to it.

Click the **message box** that reads read countdown.mov on the left side of the patch.



Look familiar?

The video on the left side of the patch looks strikingly familiar to that displayed on the right side. This is because the **js** object on the left side of the patch contains all the objects and instructions necessary to read in our movie and perform the matrix processing for our effect.

- Double-click the **js** object in our Tutorial patch. A text editor will appear, containing the source code for the **js** object in the patch. The code is saved as a file called '45jWakefilter.js' in the same folder as the Tutorial patch.

Our JavaScript code contains the familiar comment block at the top, describing the file, followed by a block of global code (executed when the **js** object is instantiated) followed by a number of functions we've defined, most of which respond to various messages sent into the **js** object from the Max patcher.

Creating Matrices

- Look at the code for the global block (i.e. the code before we arrive at the bang () function).

Our code begins with the familiar statement of how many inlets and outlets we'd like in our **js** object:

```
// inlets and outlets
inlets = 1;
outlets = 1;
```

Following this, we have a number of statements we may never have seen before:

```
// Jitter matrices to work with (declared globally)
var mymatrix = new JitterMatrix(4, "char", 320, 240);
var mywakematrix = new JitterMatrix(4, "char", 320, 240);
var myfbmatrix = new JitterMatrix(4, "char", 320, 240);

// initialize feedback matrix to all maximum values
myfbmatrix.setall(255, 255, 255, 255);
```

This block of code defines that we will be working with a number of Jitter matrices within our **js** object. The variables *mymatrix*, *mywakematrix*, and *myfbmatrix* are defined to be instances of the JitterMatrix object, much as we would declare an Array, Task, or instance of the **jsui** sketch object. The arguments to our new JitterMatrix objects are exactly the same as would be used as arguments to a **jit.matrix** object, i.e. an optional name, a planecount, a type, and a list of values for the dim.

It's important not to confuse the *name* attribute of a Jitter matrix with the *variable* name that represents it inside the JavaScript code. For example, we've created a JitterMatrix object in our code assigned to the variable *mymatrix*. Sending the message `jit_matrix mymatrix` to a **jit.pwindow** in our Max patch would not, however, display that matrix. Our *mymatrix* object has a *name* property that is generated automatically if not provided using the same convention used in other Jitter objects (e.g. *uxxxxxxxxxx*). The distinction is similar to that employed by the JavaScript Global object used to share data with Max patches.

All three of our JitterMatrix objects are created with the same typology. On the fourth line of this part of our code, we take the JitterMatrix *myfbmatrix* and set all its values to 255. The `setall()` method of the JitterMatrix object does this for us, much as the `setall` message to a **jit.matrix** object would. In fact, all of the messages and attributes used by the **jit.matrix** object are exposed as methods and properties of the JitterMatrix object within JavaScript. A few examples:

```
// set all the values in our matrix to 0:
mymatrix.clear();
// set the variable foo to the value of cell (40,40):
var foo = mymatrix.getcell(40,40);
// set cell (30,20) to the values (255,255,0,0):
mymatrix.setcell12d(30,20,255,255,0,0);
```

The `setcell12d()` method allows us to set a value of a single cell in a matrix using an array of values where the first two arguments are assumed to be the position in the matrix. The cell is then set to the values contained in subsequent arguments. There are also utility functions for one- and three-dimensional matrices (`setcell1d()` and `setcell13d()`, respectively). For a general purpose solution, we can use the plain `setcell()` function just as we would in a Max message, e.g. `mymatrix.setcell(20, 30, "val", 0, 0, 255, 255)`.

Creating Objects

- Continue perusing the global block. Now that we've created some matrices to work with, we have to create some objects to manipulate them with.

```
// Jitter objects to use (also declared globally)
var myqtmovie = new JitterObject("jit.qt.movie", 320, 240);
var myrobcross = new JitterObject("jit.robcross");
var mywake = new JitterObject("jit.wake");
var mybrcosa = new JitterObject("jit.brcosa");
```

These four lines create instances of the JitterObject objects. We need four of them (*myqtmovie*, *myrobcross*, *mywake*, and *mybrcosa*) corresponding to the four equivalent objects on the right side of our Max patch (**jit.qt.movie**, **jit.robcross**, **jit.wake**, and **jit.brcosa**). These JitterObject objects behave just as the equivalent Jitter objects would in a Max patcher, as we'll see a bit later on. The first argument when we instantiate a JitterObject is the class of Jitter object we'd like it to load (e.g. "jit.qt.movie" will give us a **jit.qt.movie** object loaded into JavaScript). Further arguments to the object can be passed just as they would in Max patchers, so that we can tell our new **jit.qt.movie** JitterObject to have a dim of 320x240 by supplying those values as arguments.

Just as we would initialize attributes by typing them into the object box following the object's name (e.g. **jit.brcosa @saturation 1.1**), we can use our global JavaScript code to initialize attributes of our the JitterObject objects we've created:

```
myrobcross.thresh = 0.14; // set edge detection threshold
mywake.rfb = 0.455; // set wake feedback for red channel
mywake.gfb = 0.455; // set wake feedback for green channel
mywake.bfb = 0.455; // set wake feedback for blue channel
mybrcosa.brightness = 1.5; // set brightness for feedback stage
```

Note that the properties of a JitterObject correspond directly to the attributes used by the Jitter object loaded into it, e.g. a JitterObject loading a **jit.brcosa** object will have properties for *brightness*, *contrast*, and *saturation*. In our code above, we initialize the *thresh* property of the JitterObject *myrobcross* to 0.14, mirroring the **jit.robcross** object on the right side of our patch. In the same way, we initialize attributes for our *mywake* and *mybrcosa* objects as well.

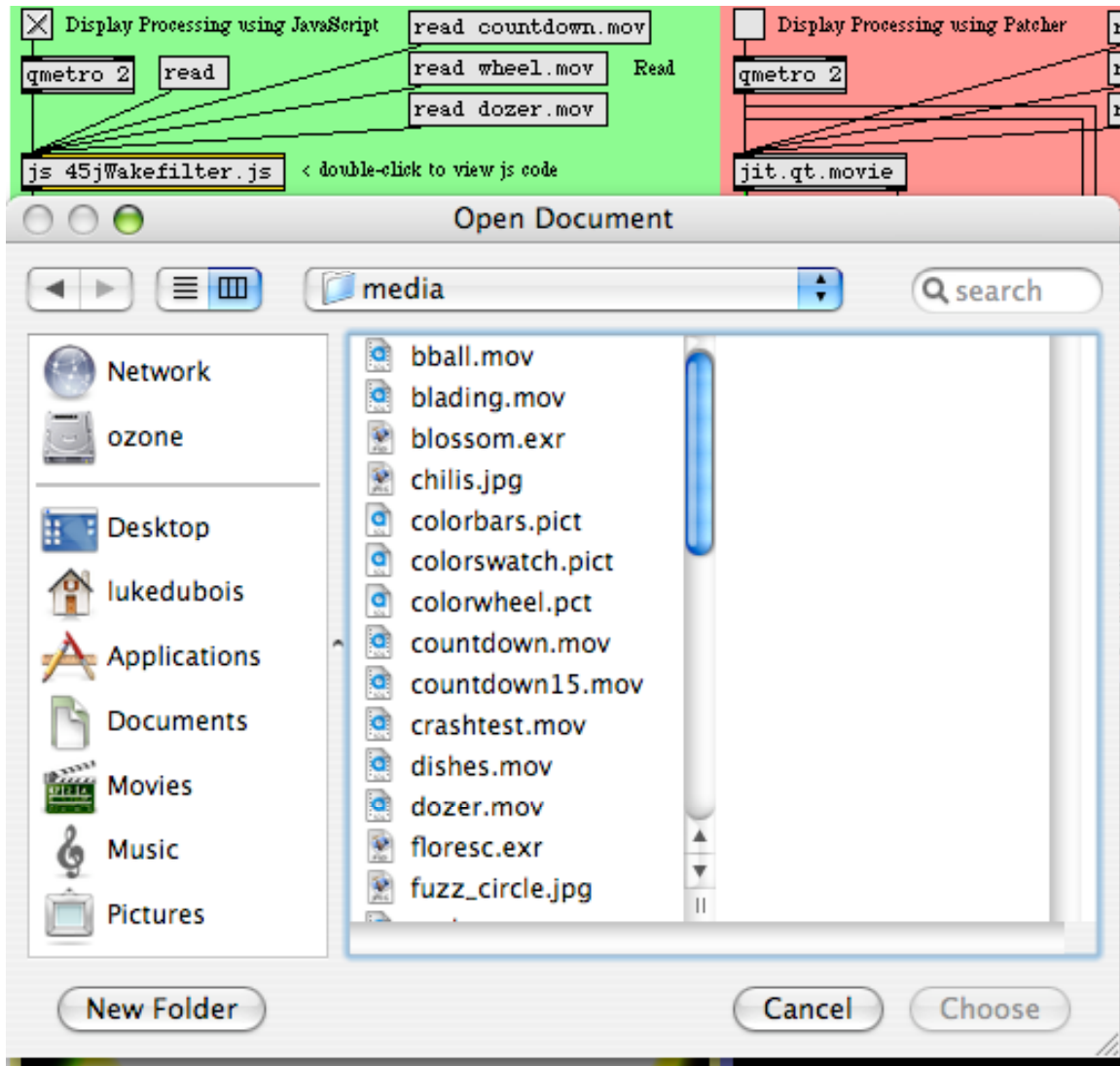
JavaScript Functions calling Jitter Object Methods

- Look at the code for the `read()` function. This function is called when our **js** object receives the read message.

```
function read(filename) // read a movie
{
    if(arguments.length==0) {
        // no movie specified, so open a dialog
        myqtmovie.read();
    }
    else { // read the movie specified
        myqtmovie.read(filename);
    }
    // initialize feedback matrix to all maximum values
    myfbmatrix.setall(255, 255, 255, 255);
}
```

Our `read()` function parses the arguments to the read message sent to our **js** object. If no arguments appear, it will call the `read()` method of our *myqtmovie* object with no arguments. If an argument is specified, our *myqtmovie* object will be told to read that argument as a filename.

- Click the **message box** that labeled read on the left side of the patch. Notice that a dialog box pops up, just as if you had sent a read message into a **jit.qt.movie** object in a Max patcher. Cancel the dialog or load in a new movie to see what our algorithm does to it.



JitterObjects in JavaScript behave just like those in Max patchers

If we wanted to, we could have looked at the Array returned by the `read()` method to ensure that it didn't fail. For right now, however, we'll trust that the arguments to the read message sent to our **js** object are legitimate filenames of QuickTime movies in the search path.

After we read in our movie (or instruct our *myqtmovie* object to open a **jit.qt.movie** “Open Document” dialog), we once again initialize our JitterMatrix *myfbmatrix* to values of all 255.

The Perform Routine

Just as a typical Jitter processing chain might run from **jit.qt.movie** to output through a series of Jitter objects in response to a **qmetro**, our JavaScript Jitter algorithm performs one loop of its processing algorithm (outputting a single matrix) in response to a bang from an outside source.

- Look at the `bang()` function in our JavaScript code. Notice that, just as in our Max patcher, each JitterObject gets called in sequence, processing matrices in turn.

```
function bang()  
// perform one iteration of the playback / processing loop  
{  
  // setup  
  
  // calculate bleed coefficients for new matrix:  
  calccoeffs();  
  
  // process  
  
  // get new matrix from movie ([jit.qt.movie]):  
  myqtmovie.matrixcalc(mymatrix, mymatrix);  
  
  // perform edge detection ([jit.robcross]):  
  myrobcross.matrixcalc(mymatrix, mymatrix);  
  
  // multiply with previous (brightened) output  
  mymatrix.op("*", myfbmatrix);  
  
  // process wake effect (can't process in place) ([jit.wake]):  
  mywake.matrixcalc(mymatrix, mywakematrix);  
  
  // brighten and copy into feedback matrix ([jit.brcosa]):  
  mybrcosa.matrixcalc(mywakematrix, myfbmatrix);  
  
  // output processed matrix into Max  
  outlet(0, "jit_matrix", mywakematrix.name);  
}
```

The `calccoeffs()` function called first in the `bang()` function sets up the properties of our *mywake* object (more on this below). Following this is the processing chain of Jitter objects that take a new matrix from our *myqtmovie* object and transform it. The `matrixcalc()` method of a JitterObject is the equivalent to sending a Jitter object in Max a bang (in the case of Jitter objects which *generate* matrices) or a `jit_matrix` message (in

Jitter objects which *process* or *display* matrices). The arguments to the `matrixcalc()` method are the input matrix followed by the output matrix. Our *myqtmovie* object has a redundant argument for its input matrix that is ignored; we simply provide the name of a valid JitterMatrix. If we were working with a Jitter object that needs more than one input or output (e.g. **jit.xfade**), we would supply our `matrixcalc()` method with Arrays of matrices set inside brackets ([,]).

The `op()` method of a JitterMatrix object is the equivalent of running the matrix through a **jit.op** object, with arguments corresponding to the `op` attribute and the scalar (`val`) or matrix to act as the second operand. In a narrative form, therefore, the following things are happening in the “process” section of our `bang()` function:

- Our *myqtmovie* object generates a new matrix from the current frame of the loaded video file, storing it into the JitterMatrix *mymatrix*.
- Our *myrobcross* object takes the *mymatrix* object and performs an edge detection on it, storing the results back into the *same* matrix (more about this below).
- We then multiply our *mymatrix* JitterMatrix with the contents of *myfbmatrix* using the `op()` method to *mymatrix*. This multiplication is done “in place” as in the previous step.
- We then process the *mymatrix* JitterMatrix through our *mywake* object, storing the output in a third JitterMatrix, called *mywakematrix*.
- Finally, we brighten the JitterMatrix *mywakematrix*, storing the output in *myfbmatrix* to be used on the next iteration of the `bang()` function. In our JavaScript code, therefore, the matrix *myfbmatrix* is being used exactly as the named matrix *bob* was used in our Max patch.

Technical Note: Depending on the class of Jitter object loaded, a JitterObject may be able to use the same matrix for both its input and output in its `matrixcalc()` method. This use of “in place” processing allows you to conserve processing time and memory copying data into new intermediary matrices. Whether this works depends entirely on the inner workings of the Jitter object in question; for example, a **jit.brcosa** object will behave correctly, whereas a **jit.wake** object (because it depends on its previous output matrices for performing feedback) will not. By a similar token, the `op()` method to a JitterMatrix object will do its processing “in place” as well.

Our processed matrix (the output of the *mywake* object stored in the *mywakematrix* matrix) is then sent out to the patcher by using an `outlet()` function:

```
outlet(0, "jit_matrix", mywakematrix.name);
```

We use the name property of our JitterMatrix in this call to send the matrix's name (uxxxxxxxxxx) to the receiving object in the Max patch.

Other Functions

- Take a look at the `calccoeffs()` function in our JavaScript code. This function is called internally by the `bang()` function every time it runs.

```
function calccoeffs() // computes the 12 bleed coefficients for
the convolution state of the [jit.wake] object
```

```
{
  // red channel
  mywake.rupbleed*=0.99;
  mywake.rupbleed+=Math.random()*0.006;
  mywake.rdownbleed*=0.99;
  mywake.rdownbleed+=Math.random()*0.006;
  mywake.rleftbleed*=0.99;
  mywake.rleftbleed+=Math.random()*0.006;
  mywake.rrightbleed*=0.99;
  mywake.rrightbleed+=Math.random()*0.006;
  // green channel
  mywake.gupbleed*=0.99;
  mywake.gupbleed+=Math.random()*0.006;
  mywake.gdownbleed*=0.99;
  mywake.gdownbleed+=Math.random()*0.006;
  mywake.gleftbleed*=0.99;
  mywake.gleftbleed+=Math.random()*0.006;
  mywake.grightbleed*=0.99;
  mywake.grightbleed+=Math.random()*0.006;

  // blue channel
  mywake.bupbleed*=0.99;
  mywake.bupbleed+=Math.random()*0.006;
  mywake.bdownbleed*=0.99;
  mywake.bdownbleed+=Math.random()*0.006;
  mywake.bleftbleed*=0.99;
  mywake.bleftbleed+=Math.random()*0.006;
  mywake.brightbleed*=0.99;
  mywake.brightbleed+=Math.random()*0.006;
}
calccoeffs.local = 1; // can't call from the patcher
```

We see that the `calccoeffs()` function literally duplicates the functionality of the `random_bleed` **patcher** on the right side of our patch. It sets a variety of properties of the *mywake* JitterObject, corresponding to the various attributes of the **jit.wake** object it contains. Notice that we can use these properties as ordinary variables, getting their values as well as setting them. This allows us to change their values using in place operators, e.g.:

```
mywake.rupbleed*=0.99;
mywake.rupbleed+=Math.random()*0.006;
```

This code (replicated twelve times for different properties of the *mywake* object) uses the *current* value of the `rupbleed` property of *mywake* as a starting point, multiplies it by 0.99, and adds a small random value (between 0 and 0.006) to it.

Summary

You can use JavaScript code within Max to define procedural systems using Jitter matrices and objects. The JitterMatrix object within **js** allows you to create, set, and query attributes of Jitter matrices from within JavaScript—the `setall()` method of JitterMatrix, sets all of its cells to a certain value, for example. You can also apply mathematical operations to a JitterMatrix “in place” using the `op()` method, which contains the complete set of mathematical operators used in the **jit.op** object. Jitter objects can be loaded as classes into the JitterObject object. Upon instantiation, a JitterObject acquires properties and methods equivalent to the Jitter object’s messages and attributes. The `matrixcalc()` method of a JitterObject performs the equivalent of sending the Jitter object a `bang` or a `jit_matrix` message, whichever is relevant for that class of object. This allows you to port complex function graphs of Jitter processes into JavaScript.

In the next two Tutorials, we’ll look at other ways to use JavaScript to expand the possibilities when working with Jitter.

Code Listing

```
// 45jWakefilter.js
//
// a video playback processing chain demonstrating the use of
// Jitter objects and matrices within [js].
//
// rld, 6.05
//
// inlets and outlets
inlets = 1;
outlets = 1;
```



```

// Jitter matrices to work with (declared globally)
var mymatrix = new JitterMatrix(4, "char", 320, 240);
var mywakematrix = new JitterMatrix(4, "char", 320, 240);
var myfbmatrix = new JitterMatrix(4, "char", 320, 240);

// initialize feedback matrix to all maximum values
myfbmatrix.setall(255, 255, 255, 255);

// Jitter objects to use (also declared globally)
var myqtmovie = new JitterObject("jit.qt.movie", 320, 240);
var myrobcross = new JitterObject("jit.robcross");
var mywake = new JitterObject("jit.wake");
var mybrcosa = new JitterObject("jit.brcosa");

// set some initial attributes for our JitterObjects
myrobcross.thresh = 0.14; // set edge detection threshold
mywake.rfb = 0.455; // set wake feedback for red channel
mywake.gfb = 0.455; // set wake feedback for green channel
mywake.bfb = 0.455; // set wake feedback for blue channel
mybrcosa.brightness = 1.5; // set brightness for feedback stage

function read(filename) // read a movie
{
    if(arguments.length==0) {
        // no movie specified, so open a dialog
        myqtmovie.read();
    }
    else { // read the movie specified
        myqtmovie.read(filename);
    }
    // initialize feedback matrix to all maximum values
    myfbmatrix.setall(255, 255, 255, 255);
}

function bang()
// perform one iteration of the playback / processing loop
{
    // setup

    // calculate bleed coefficients for new matrix:
    calccoeffs();

    // process

    // get new matrix from movie ([jit.qt.movie]):
    myqtmovie.matrixcalc(mymatrix, mymatrix);

    // perform edge detection ([jit.robcross]):
    myrobcross.matrixcalc(mymatrix, mymatrix);

    // multiply with previous (brightened) output
    mymatrix.op("*", myfbmatrix);
}

```

```

    // process wake effect (can't process in place) ([jit.wake]):
    mywake.matrixcalc(mymatrix, mywakematrix);

    // brighten and copy into feedback matrix ([jit.brcosa]):
    mybrcosa.matrixcalc(mywakematrix, myfbmatrix);

    // output processed matrix into Max
    outlet(0, "jit_matrix", mywakematrix.name);
}

function calccoeffs() // computes the 12 bleed coefficients for
the convolution state of the [jit.wake] object
{
    // red channel
    mywake.rupbleed*=0.99;
    mywake.rupbleed+=Math.random()*0.006;
    mywake.rdownbleed*=0.99;
    mywake.rdownbleed+=Math.random()*0.006;
    mywake.rleftbleed*=0.99;
    mywake.rleftbleed+=Math.random()*0.006;
    mywake.rrightbleed*=0.99;
    mywake.rrightbleed+=Math.random()*0.006;
    // green channel
    mywake.gupbleed*=0.99;
    mywake.gupbleed+=Math.random()*0.006;
    mywake.gdownbleed*=0.99;
    mywake.gdownbleed+=Math.random()*0.006;
    mywake.gleftbleed*=0.99;
    mywake.gleftbleed+=Math.random()*0.006;
    mywake.grightbleed*=0.99;
    mywake.grightbleed+=Math.random()*0.006;
    // blue channel
    mywake.bupbleed*=0.99;
    mywake.bupbleed+=Math.random()*0.006;
    mywake.bdownbleed*=0.99;
    mywake.bdownbleed+=Math.random()*0.006;
    mywake.bleftbleed*=0.99;
    mywake.bleftbleed+=Math.random()*0.006;
    mywake.brightbleed*=0.99;
    mywake.brightbleed+=Math.random()*0.006;
}
calccoeffs.local = 1; // can't call from the patcher

```

Tutorial 46: Manipulating Matrix Data using JavaScript

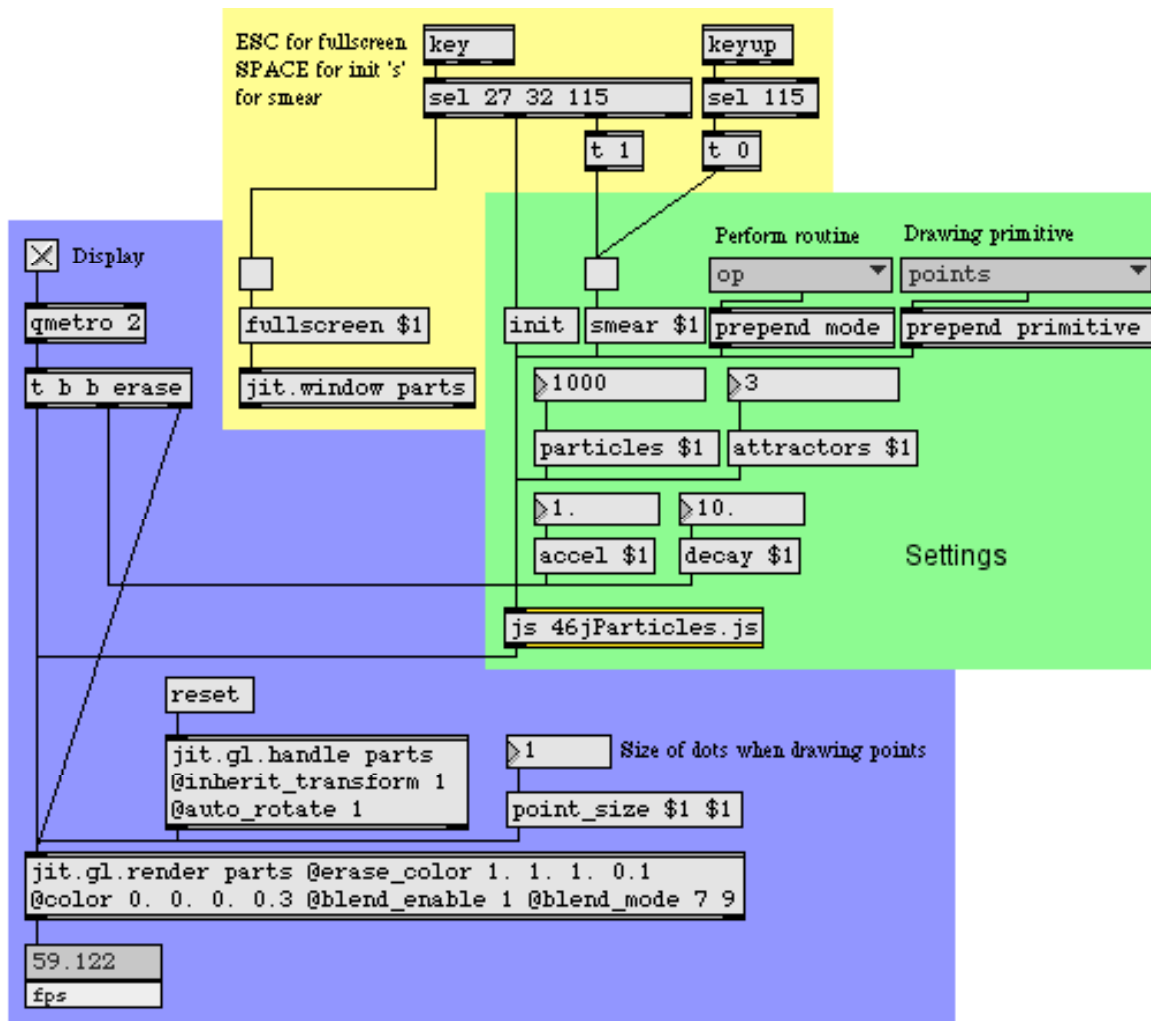
As we saw in the last tutorial, we can use the Max **js** object to design a pipeline of Jitter objects within procedural JavaScript code. The `JitterObject` and `JitterMatrix` objects within JavaScript allow us to create new Jitter objects and matrices and work with them more or less as we would within a Max patcher. In many situations, we need to manipulate data stored in a Jitter matrix in a manner that would be awkward or difficult to do using a Max patcher. This tutorial looks at a variety of solutions for how to manipulate matrix data within **js** using methods and properties of the `JitterMatrix` object, as well as using the **jit.expr** object within JavaScript.

This tutorial assumes you've read through the previous *Tutorial 45: Introduction to using Jitter within JavaScript*. In addition, this tutorial works with Jitter OpenGL objects as well as **jit.expr**, so you may want to review *Tutorial 30: Drawing 3D Text*, *Tutorial 31: Rendering Destinations*, and *Tutorial 39: Spatial Mapping* before we begin.

- Open the tutorial patch *46jJavaScriptOperators.pat* in the Jitter Tutorials folder.

This tutorial patch uses a **js** object loading a file called *46jParticles.js*. The JavaScript code generates a Jitter matrix in response to a bang that we then send to the **jit.gl.render** object in the patcher.

The file contains a number of functions to respond to various settings we can send as messages from our patch.

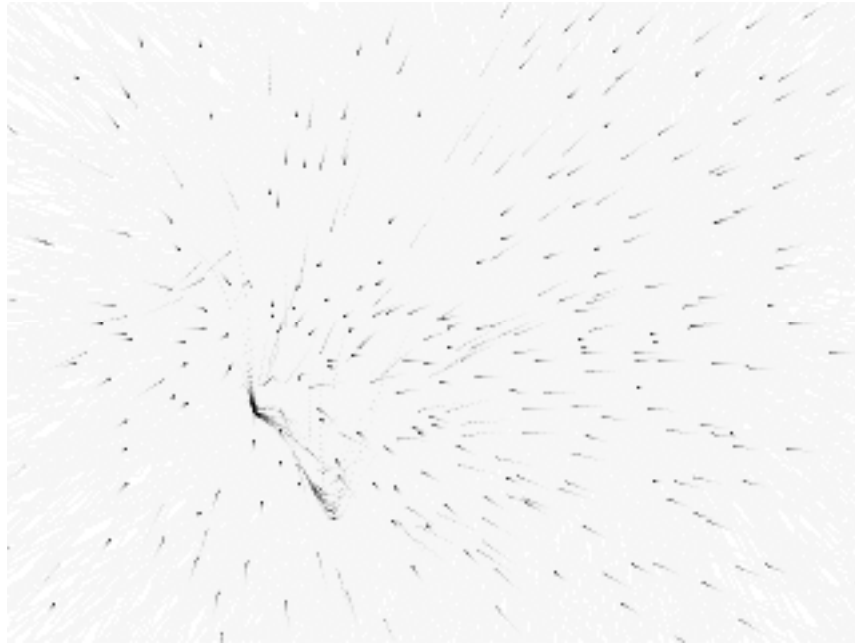


Our patcher containing the JavaScript file.

- Click the **toggle box** above the **qmetro** object on the left hand side of the patch. Observe the results in the **jit.window** named parts.

Our JavaScript code generates a set of points that represent a simple *particle system*. A particle system is essentially an algorithm that operates on a (often very large) number of spatial points called *particles*. These particles have rules that determine how they move over time in relation to each other or in relation to other actors within the space. At their basic level, particles contain only their spatial coordinates. Particle systems may also contain other information and can be used to simulate a wide variety of natural processes such as running water or smoke.

Particle systems are widely used in computer simulations of our environment, and as such are a staple technology in many computer-generated imagery (CGI) applications.



A particle system generated as a Jitter matrix.

The Wide World of Particles

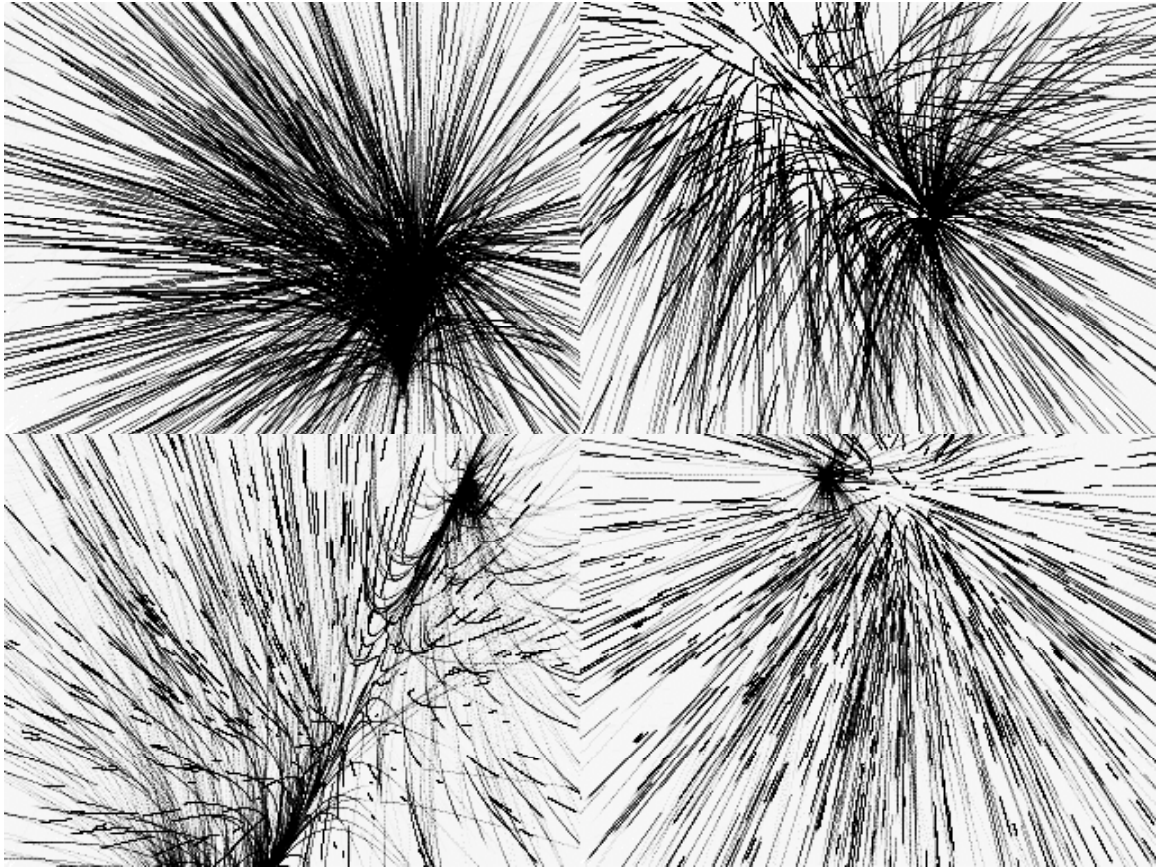
The particle system used in our JavaScript simulation works by generating two sets of random points representing the positions and velocities of the particles and a number of spatial positions in the 3D space that contain gravity. These gravity points, called *attractors*, act upon the particles in each frame to gradually pull particles towards them. As we can see in the **jit.window**, the points in our particle system gradually collapse towards one or more points of singularity. Alternately, the attractor points may be such that the particles oscillate between them, caught in a conflicting gravity field that gradually stabilizes.

- With the patcher as the frontmost window, press the *spacebar* on your computer keyboard or click on the **message box** labeled `init` attached to the **js** object. Try this a few times to observe the different behaviors of our particle system.

The `init` message to our **js** object reboots the particle system. It randomly scatters the particles and generates new attractor points.

- Press and hold the 's' key on our computer keyboard, or click the **toggle box** attached to the **message box** labeled `smear $1`. The particles will now leave trails behind them as they move. Release the 's' key (or uncheck the **toggle box**) to return to a normal visualization.

By observing our particle system over time we can view the trajectories of our points as they collapse towards the attractors.



A variety of behaviors from a simple set of rules.

- Click in the **jit.window**. A set of colored red, green, and blue axes will appear around the world. Rotate the world with your mouse. Try zooming out (by holding down the ALT/Option key and dragging in the window). Restart the particle system from different vantage points.

A **jit.gl.handle** object controls our **jit.gl.render** object, allowing us to see that our particle system inhabits a three-dimensional world. By rotating the space around to different perspectives we can see how the attractors pull in particles from all around them.

Under the Hood

Now that we've seen a good part of the functionality of the patch (we'll return to it later), let's look at the JavaScript code to see how the algorithm is constructed.

- Double-click the **js** object in our Tutorial patch. A text editor will appear, containing the source code for the **js** object in the patch. The code is saved as a file called '46jParticles.js' in the same folder as the Tutorial patch.

Our JavaScript code works by manipulating our particles and attractors as Jitter matrices. The particle system is updated by a generation every time our **js** object receives a bang. This is accomplished by performing a series of operations on the data in the Jitter matrices. We can take advantage of the Jitter architecture to perform mathematical operations on *entire* matrices at once since we've encoded our system as matrices. This provides a significant advantage in speed, clarity, and efficiency over working with our data as individual values that must be adjusted one at a time, in many cases (as we would were we to encode our particles as an Array, for example).

Our JavaScript code actually contains three ways of updating our particle system from generation to generation, each of which uses different techniques for manipulating the matrix data representing the particles. We can process the particle system as a single entity using a series of `op ()` methods to the JitterMatrix object, by using a `jit.expr` object, or by iterating through the particle system point-by-point (or cell-by-cell). We'll look at each in turn once we investigate the code common to them all.

- Look at the global block of code that begins the JavaScript file.

After the initial comment block and inlet/outlet declarations, we can see a number of variables declared and initialized, including a few JitterObject and JitterMatrix objects. If we examine this code in detail, we can see the outline of how we'll perform our particle system.

```
var PARTICLE_COUNT = 1000; // initial number of particle vertices
var ATTRACTOR_COUNT = 3; // initial number of points of gravity
```

These two global variables (`PARTICLE_COUNT` and `ATTRACTOR_COUNT`) are used to decide how many particles and how many points of attraction we'd like to work with in our simulation. These will determine the dim of the Jitter matrices containing the particle and attractor information.

```
// create a [jit.noise] object for particle and velocity
generation
var noisegen = new JitterObject("jit.noise");
noisegen.dim = PARTICLE_COUNT;
noisegen.planecount = 3;
noisegen.type = "float32";

// create a [jit.noise] object for attractor generation
var attgen = new JitterObject("jit.noise");
attgen.dim = ATTRACTOR_COUNT;
attgen.planecount = 3;
attgen.type = "float32";
```

Our particle systems are generated randomly by the `init()` function, which we will investigate presently. The **jit.noise** objects created here as `JitterObject` objects will perform that function for us, by generating one-dimensional matrices of `float32` values of a size (`dim`) corresponding to the number of particles and attractors specified for our system. The matrices generated by our **jit.noise** objects have a `planeCount` of 3, corresponding to x, y, and z spatial data.

```
// create two [jit.expr] objects for the bang_expr() function

// first expression: sum all the planes in the input matrix
var myexpr = new JitterObject("jit.expr");
myexpr.expr = "in[0].p[0]+in[0].p[1]+in[0].p[2]";
// second expression: evaluate a+((b-c)*d/e)
var myexpr2 = new JitterObject("jit.expr");
myexpr2.expr = "in[0]+((in[1]-in[2])*in[3]/in[4])";
```

One of the ways we will update our particle system is to use two **jit.expr** objects within our JavaScript code. This part of the code creates the `JitterObject` objects and defines the mathematical expression to be used by them (the `expr` attribute). We'll step through this when we investigate the code that uses them later on.

```
// create the Jitter matrices we need to store our data

// matrix of x,y,z particle vertices
var particlemat = new JitterMatrix(3, "float32", PARTICLE_COUNT);
// matrix of x,y,z particle velocities
var velomat = new JitterMatrix(3, "float32", PARTICLE_COUNT);
// matrix of x,y,z points of attraction (gravity centers)
var attmat = new JitterMatrix(3, "float32", ATTRACTOR_COUNT);
// matrix for aggregate distances
var distmat = new JitterMatrix(3, "float32", PARTICLE_COUNT);
// temporary matrix for the bang_op() function
var tempmat = new JitterMatrix(3, "float32", PARTICLE_COUNT);
// temporary summing matrix for the bang_op() function
var summat = new JitterMatrix(1, "float32", PARTICLE_COUNT);
// another temporary summing matrix for the bang_op() function
var summat2 = new JitterMatrix(1, "float32", PARTICLE_COUNT);
// a scalar matrix to store the current gravity point
var scalarmat = new JitterMatrix(3, "float32", PARTICLE_COUNT);
// a scalar matrix to store acceleration (expr_op() function
only)
var amat = new JitterMatrix(1, "float32", PARTICLE_COUNT);
```

Our algorithm calls for a number of `JitterMatrix` objects to store information about our particle system and to be used as intermediate storage during the processing of each generation of the system. The first three matrices (bound to the variables *particlemat*, *velomat*, and *attmat*) store the x,y,z positions of our particles, the x,y,z velocities of our particles, and the x,y,z positions of our attractors, respectively. The other six matrices are used in computing each generation of the system.


```
var a = 0.001; // acceleration factor
var d = 0.01; // decay factor
```

These two variables control two important aspects of the behavior of our particle system: the variable *a* controls how rapidly the particles accelerate toward an attractor, while the variable *d* controls how much the particles' current velocity decays with each generation. This second variable influences how easy it is for a particle to change direction and be drawn to other attractors.

```
var perform_mode="op"; // default perform function
var draw_primitive = "points"; // default drawing primitive
```

These final variables define which of the three techniques we'll use to process our particle system (the *perform_mode*) and the way in which our **jit.gl.render** object visualizes the particle matrix once our JavaScript code sends it out to the Max patcher (the *draw_primitive*).

The Initialization Phase

The first step in creating a particle system is to generate an *initial state* for the particles and any factors that will act upon them (in our case, the attractor points). For example, were we to attempt to simulate a waterfall, we would start all our particles at the top of the space, with a fixed gravity field at the bottom of the space acting upon the particles with each generation. Our system is slightly less ambitious in terms of real-world accuracy—the particles and attractors will simply be in random positions throughout the 3D scene.

- Look at the code for the `loadbang()` and `init()` functions in the JavaScript code.

```
function loadbang() // execute this code when our Max patch opens
{
    init(); // initialize our matrices
    post("particles initialized.\n");
}

function init()
// initialization routine... call at load, as well as
// when we change the number of particles or attractors
{
    // generate a matrix of random particles spread between -1 and
    1
    noisegen.matrixcalc(particlemat, particlemat);
    particlemat.op("*", 2.0);
    particlemat.op("-", 1.0);
    // generate a matrix of random velocities spread between -1
    and 1
    noisegen.matrixcalc(velomat, velomat);
```

```

    velomat.op("*", 2.0);
    velomat.op("-", 1.0);
    // generate a matrix of random attractors spread between -1
and 1
    attgen.matrixcalc(attmat, attmat);
    attmat.op("*", 2.0);
    attmat.op("-", 1.0);
}

```

The `loadbang()` function in a **js** object runs whenever the Max patcher containing the **js** file is loaded. This happens *after* the object is instantiated with the rest of the patch, and is triggered at the same time as messages triggered by **loadbang** and **loadmess** objects in a Max patch would be. Our `loadbang()` function simply calls the `init()` function and then prints a friendly message to the Max window telling us that all is well.

The `loadbang()` function of JavaScript code only executes when the patcher containing the **js** object is opened. This function does *not* execute when you change and recompile the JavaScript code.

Our `init()` function runs when we open our patch as well as whenever we call it from our Max patcher (through the **message box** triggered by the *spacebar*). The `init()` function also gets called whenever we change the number of attractors and particles in our simulation. The `matrixcalc()` method of **jit.noise** fills the output matrix (the second argument to the method) with random values between 0 and 1. This is the same as sending a bang to a **jit.noise** object in a patcher. In our `init()` function we fill three matrices with random values in 3 planes. These matrices represent the initial position of our particles (*particlemat*), the initial velocity of our particles (*velomat*) and the position of our attractors (*attmat*). Using the `op()` method to our JitterMatrix objects, we then scale these random values to be between -1 and 1. We do this by multiplying the matrix values by 2 and then subtracting 1.

Now that we have our initial state set up for our particle system, we need to look at how we process the particles with each generation. This is accomplished through one of three different methods in our JavaScript code determined by the *perform_mode* variable.

- In the Tutorial patcher, restart the particle system and switch the **ubumenu** object labeled *Perform routine* from “op” to “expr”. The particle system should continue to behave exactly as before. Switch the **ubumenu** again to “iter”. The particle system will still run, but very slowly (note the frame rate in the **fpsgui** object attached to the **jit.gl.render** object in the lower left part of the patch). Switch the **ubumenu** back to “op”.

The **ubumenu** changes the value of the *perform_mode* variable via the `mode()` function in our JavaScript code. We will look at later in this Tutorial, but it’s important to note that one of the methods used (“iter”) runs much more slowly than the other two. This is

largely due to the technique used to update the particle system. We'll look at why this is when we investigate the function that performs that task.

- Look at the `bang()` function in the JavaScript code.

```
function bang() // perform one iteration of our particle system
{
  switch(perform_mode) { // choose from the following...
    case "op": // use Jitter matrix operators
      bang_op();
      break;
    case "expr": // use [jit.expr] for the bulk of the
algorithm
      bang_expr();
      break;
    case "iter": // iterate cell-by-cell through the matrices
      bang_iter();
      break;
    default: // use bang_op() as our default
      bang_op();
      break;
  }
  // output our new matrix of particle vertices
  // with the current drawing primitive
  outlet(0, "jit_matrix", particlemat.name, draw_primitive);
}
```

Our `bang()` function uses a JavaScript `switch()` statement to decide what function to call from within it to do the actual processing of our particle system. Depending on the *perform_mode* we choose in the Max patcher, we select from one of three different functions (`bang_op()`, `bang_expr()`, or `bang_iter()`). Assuming all goes well, we then output the message `jit_matrix`, followed by the name of our *particlemat* matrix (which contains the current coordinates of the simulation's particles), followed by the name of our OpenGL *draw_primitive*, back into Max.

In the grand tradition of *Choose Your Own Adventure* and *Let's Make a Deal*, we'll now investigate the three different perform routines represented by the different functions mentioned above.

Door #1: The `op()` route

- Look at the JavaScript source for the `bang_op()` function.

Our `bang_op()` function updates our particle system by using, whenever possible, the `op()` method to the JitterMatrix object to mathematically alter the contents of matrices *all at once*. Whenever possible, we do this processing *in place* to limit the number of separate Jitter matrices we need to get through the algorithm. We perform the bulk of the

processing multiple times within a `for ()` loop, once for each attractor in our particle system. Once this loop completes, we get an updated version of the velocity matrix (*velomat*), which we then add to the particle matrix (*particlemat*) to define the new positions of the particles.

In a nutshell, we do the following:

```
function bang_op() // create our particle matrix using Matrix
operators
{
  for(var i = 0; i < ATTRACTOR_COUNT; i++)
    // do one iteration per gravity point
    {
```

We perform the code until the closing brace `{}` once for every attractor, setting the attractor we're currently working on to the variable *i*.

```
    // create a scalar matrix out of the current attractor:
    scalarmat.setall(attmat.getcell(i));
```

The `getcell ()` method of a `JitterMatrix` object returns the values of the numbers in the cell specified as its argument. The `setall ()` method sets *all* the cells of a matrix to a value (or array of values). These methods work the same as the corresponding messages to the **jit.matrix** object in a Max patcher. This line tells our **js** object to copy the current attractors coordinates out of the attractor matrix (*attmat*) and set every single cell in the `JitterMatrix` *scalarmat* to those values. The *scalarmat* matrix has the same dim as the *particlemat* matrix (equal to the number of particles in our system). This allows us to use it as a scalar operand in our `op ()` methods.

```
    // subtract our particle positions from the current
    attractor
    // and store in a temporary matrix (x,y,z):
    tempmat.op("-", scalarmat, particlemat);
```

This code subtracts our particle positions (*particlemat*) from the position of the attractor we're currently working with (*scalarmat*). The result is then stored in a temporary matrix with the same dim as two used in the `op ()` function. This matrix represents the distances from each particle to the current attractor.

```
    // square to create a cartesian distance matrix (x*x, y*y,
    z*z):
    distmat.op("*", tempmat, tempmat);
```

This code multiplies the *tempmat* matrix *by itself*, as a simple way of squaring it. The result is then stored in the *distmat* matrix.

```

// sum the planes of the distance matrix (x*x+y*y+z*z)
summat.planemap = 0;
summat.frommatrix(distmat);
summat2.planemap = 1;
summat2.frommatrix(distmat);
summat.op("+", summat, summat2);
summat2.planemap = 2;
summat2.frommatrix(distmat);
summat.op("+", summat, summat2);

```

In this block of code, we take the separate *planes* of the *distmat* matrix and add them together into a single-plane matrix called *summat*. In order to do this, we use the *planemap* property of *JitterMatrix* to specify which plane of the source matrix to use when copying from using the *frommatrix()* method. To sum everything we need a second temporary matrix (*summat2*) to help with the operation. First we copy plane 0 of *distmat* (the squared *x* distances) into *summat*. We then copy plane 1 of *distmat* (the squared *y* distances) into *summat2*. We then add *summat* and *summat2*, storing the results back in *summat* (using an *op()* method). We then copy plane 2 of *distmat* (the squared *z* distances) into *summat2*. We then add *summat* and *summat2* again, keeping in mind that *summat* at this point *already* contains the sum of the first two planes of *distmat*. The result of this second sum is stored back into *summat*, which now contains the sum of all three planes of *distmat*.

```

// scale our distances by the acceleration value:
tempmat.op("*", a);
// divide our distances by the sum of the distances
// to derive gravity for this frame:
tempmat.op("/", summat);
// add to the current velocity bearings to get the
// amount of motion for this frame:
velomat.op("+", tempmat);
}

```

This is the last block of code in our per-attractor loop. We multiply the *tempmat* matrix (which contains the distances of our particles from the current attractor) by the value stored in the variable *a*, representing acceleration. We then divide that result by the *summat* matrix (the sum of the squared distances), and add those results to the current velocities of each particle as stored in the *velomat* matrix. The result of the addition is stored in *velomat*.

This *entire* process is repeated again for each attractor. As a result, the *velomat* matrix is added to each time based on how far our particles are from each attractor. By the time the loop finishes (when *i* reaches the last attractor index), *velomat* contains a matrix of velocities corresponding to the aggregate pull of all our attractors on all our particles.

```

        // offset our current positions by the amount of motion:
        particlemat.op("+", velomat);
        // reduce our velocities by the decay factor for the next
frame:
        velomat.op("*", d);
}

```

Finally, we add these velocities to our matrix of particles (*particlemat* + *velomat*). Our particle matrix is now updated to a new set of particle positions. We then decay the velocity matrix by the amount stored in the variable *d*, so that the simulation retains a remnant of this generation's velocity for the next generation of the particle system.

The use of a cascading series of `op()` methods to perform our algorithm on entire matrices gives us a big advantage in terms for speed, as Jitter can perform a simple mathematical operation on a large set of data very quickly. However, there are a few points (particularly in the generation of the summing matrix *summat*) where the code may have seemed more awkward than necessary. We can use **jit.expr** to define a more complex mathematical expression to perform much of this work in a single operation.

Door #2: The `expr()` route

- Back in the global block of our JavaScript file, revisit the code that instantiates the **jit.expr** objects.

```

// create two [jit.expr] objects for the bang_expr() function

// first expression: sum all the planes in the input matrix
var myexpr = new JitterObject("jit.expr");
myexpr.expr = "in[0].p[0]+in[0].p[1]+in[0].p[2]";
// second expression: evaluate a+((b-c)*d/e)
var myexpr2 = new JitterObject("jit.expr");
myexpr2.expr = "in[0]+((in[1]-in[2])*in[3]/in[4])";

```

At the beginning of our JavaScript code we created two `JitterObject` objects (*myexpr* and *myexpr2*) that instantiated **jit.expr** objects. The expression for the first object takes a single matrix (*in[0]*) and sums its planes (the *.p[n]* notation refers to the data stored in plane *n* of that matrix). The second expression takes five matrices (*in[0]* – *in[4]*) and adds the first matrix (*A*) to the result of the second subtracted from the third (*B-C*) multiplied by a fourth (*D*) and divided by a fifth (*E*). Our *myexpr2* `JitterObject` therefore evaluates the expression:

$$A + ((B - C) * D / E)$$

- Look at the code for the `bang_expr()` function. Compare it to what we've used in the `bang_op()` function.

The basic outline of our `bang_expr()` function is equivalent to the `bang_op()` function, i.e. we iterate through a loop based on the number of attractors in our simulation, eventually ending up with an aggregate velocity matrix (*velomat*) that we then use to offset our particle matrix (*particlemat*). The key difference lies in where we insert the calls to **jit.expr**:

```
function bang_expr() // create our particle matrix using
[jit.expr]
{
    // create a scalar matrix out of our acceleration value:
    amat.setall(a);
```

The above line fills every cell in the *amat* matrix with the value of the variable *a* (the acceleration factor). This allows us to use it as an operand in one of the **jit.expr** expressions later on.

```
    for(var i = 0; i < ATTRACTOR_COUNT; i++)
    // do one iteration per gravity point
    {
        // create a scalar matrix out of the current attractor:
        scalarmat.setall(attmat.getcell(i));
        // subtract our particle positions from the current
        attractor
        // and store in a temporary matrix (x,y,z):
        tempmat.op("-", scalarmat, particlemat);
        // square to create a cartesian distance matrix (x*x, y*y,
        z*z):
        distmat.op("*", tempmat, tempmat);
```

This is all the same as in `bang_op()`. We derive a squared distance matrix based on the difference between the current attractor and our particle positions.

```
    // sum the planes of the distance matrix (x*x+y*y+z*z) :
    // "in[0].p[0]+in[0].p[1]+in[0].p[2]" :
    myexpr.matrixcalc(distmat, summat);
```

Instead of summing the *distmat* matrix plane-by-plane using `op()` and `frommatrix()` methods, we simply evaluate our first mathematical expression using *distmat* as a 3-plane input matrix and *summat* as a 1-plane output matrix.

```
    // derive amount of motion for this frame :
    // "in[0]+((in[1]-in[2])*in[3]/in[4])" :

    myexpr2.matrixcalc([velomat,scalarmat,particlemat,amat,summat]
    ,
    velomat);
}
```

Similarly, at the end of our attractor loop we can derive the velocity matrix *velomat* in one compound expression based on the previous velocity matrix (*velomat*), the scalar matrix containing the current attractor point (*scalarmat*), the current particle positions (*particlemat*), the scalar matrix containing the acceleration (*amat*), and the matrix containing the distance sums (*summat*). This is much simpler (and a cleaner read) than using a whole sequence of `op()` functions working with intermediary matrices. Note that we use brackets (`[` and `]`) to establish an array of input matrices in the `matrixcalc()` method to the *myexpr2* object.

```

        // offset our current positions by the amount of motion:
        particlemat.op("+", velomat);
        // reduce our velocities by the decay factor for the next
frame:
        velomat.op("*", d);
    }

```

This is the same as in `bang_op()`. We generate the new particle positions and decay the new velocities for use as initial velocities in the next generation of the system.

Door #3: Cell-by-cell

- Look at the code for the `bang_iter()` function.

The `bang_iter()` function works in a different way from the other two perform routines we're using in our JavaScript code. Rather than working on the matrices as single entities, we work on everything on a cell-by-cell basis, iterating through not only the matrix of attractor positions (*attmat*), but also through the matrices of particles and velocities. We do this through a pair of nested `for()` loops, temporarily storing each cell value in different Array objects. We use the `getCell()` and `setCellId()` methods to the *JitterMatrix* object to retrieve and store values from these Arrays.

```

function bang_iter() // create our particle matrix cell-by-cell
{
    var p_array = new Array(3); // array for a single particle
    var v_array = new Array(3); // array for a single velocity
    var a_array = new Array(3); // array for a single attractor

    for(var j = 0; j < PARTICLE_COUNT; j++)
    // do one iteration per particle
    {
        // fill an array with the current particle:
        p_array = particlemat.getCell(j);
        // fill an array with the current particle's velocity:
        v_array = velomat.getCell(j);
    }
}

```



```

for(var i = 0; i < ATTRACTOR_COUNT; i++)
// do one iteration per gravity point
{
    // fill an array with the current attractor:
    a_array = attmat.getcell(i);

    // find the distance from this particle to the
    // current attractor:
    var distsum = (a_array[0]-p_array[0])*(a_array[0]-
p_array[0]);
    distsum+= (a_array[1]-p_array[1])*(a_array[1]-
p_array[1]);
    distsum+= (a_array[2]-p_array[2])*(a_array[2]-
p_array[2]);

    // derive the amount of motion for this frame:
    v_array[0]+= (a_array[0]-p_array[0])*a/distsum; // x
    v_array[1]+= (a_array[1]-p_array[1])*a/distsum; // y
    v_array[2]+= (a_array[2]-p_array[2])*a/distsum; // z
}

// offset our current positions by the amount of motion
p_array[0]+=v_array[0]; // x
p_array[1]+=v_array[1]; // y
p_array[2]+=v_array[2]; // z

// reduce our velocities by the decay factor for the next
frame:
v_array[0]*=d; // x
v_array[1]*=d; // y
v_array[2]*=d; // z

// set the position for this particle in the Jitter matrix:
particlemat.setcell1d(j, p_array[0],p_array[1],p_array[2]);
// set the velocity for this particle in the Jitter matrix:
velomat.setcell1d(j, v_array[0],v_array[1],v_array[2]);
}
}

```

Note that by updating our particle system bit-by-bit (and using intermediary Array objects to store data for each cell) we're essentially replicating the same operation, as many times as there are particles in our system! While this may not be noticeably inefficient with a small number of particles, once you begin to work with thousands of points it will become noticeably slower.

Other functions

- Back in the Max patcher, change the **number box** objects attached to the **message boxes** labeled particles \$1, attractors \$1, accel \$1, and decay \$1. Try setting the number of particles to very large and very small numbers. Try to work out how the accel and decay attributes

alter the responsiveness of the system. Look at the code for these functions in the JavaScript file.

The bulk of these functions simply change variables, sometimes scaling them first (e.g. `accel()` and `decay()` simply change the values of *a* and *d*, respectively). Similarly, the `mode()` function changes the value of the *perform_mode* variable to a string that we use to decide the perform routine:

```
function mode(v) // change perform mode
{
    perform_mode = v;
}
```

The `particles()` and `attractors()` functions, however, need to not only change the value of a variable (*PARTICLE_COUNT* and *ATTRACTOR_COUNT*, respectively), but they need to change the `dim` of the matrices that depend on those values as well as reboot the particle simulation (by calling the `init()` function).

```
function particles(v) // change the number of particles we're
working with
{
    PARTICLE_COUNT = v;

    // resize matrices
    noisegen.dim = PARTICLE_COUNT;
    particlemat.dim = PARTICLE_COUNT;
    velomat.dim = PARTICLE_COUNT;
    distmat.dim = PARTICLE_COUNT;
    attmat.dim = PARTICLE_COUNT;
    tempmat.dim = PARTICLE_COUNT;
    summat.dim = PARTICLE_COUNT;
    summat2.dim = PARTICLE_COUNT;
    scalarmat.dim = PARTICLE_COUNT;
    amat.dim = PARTICLE_COUNT;

    init(); // re-initialize particle system
}

function attractors(v)
// change the number of gravity points we're working with
{
    ATTRACTOR_COUNT = v;

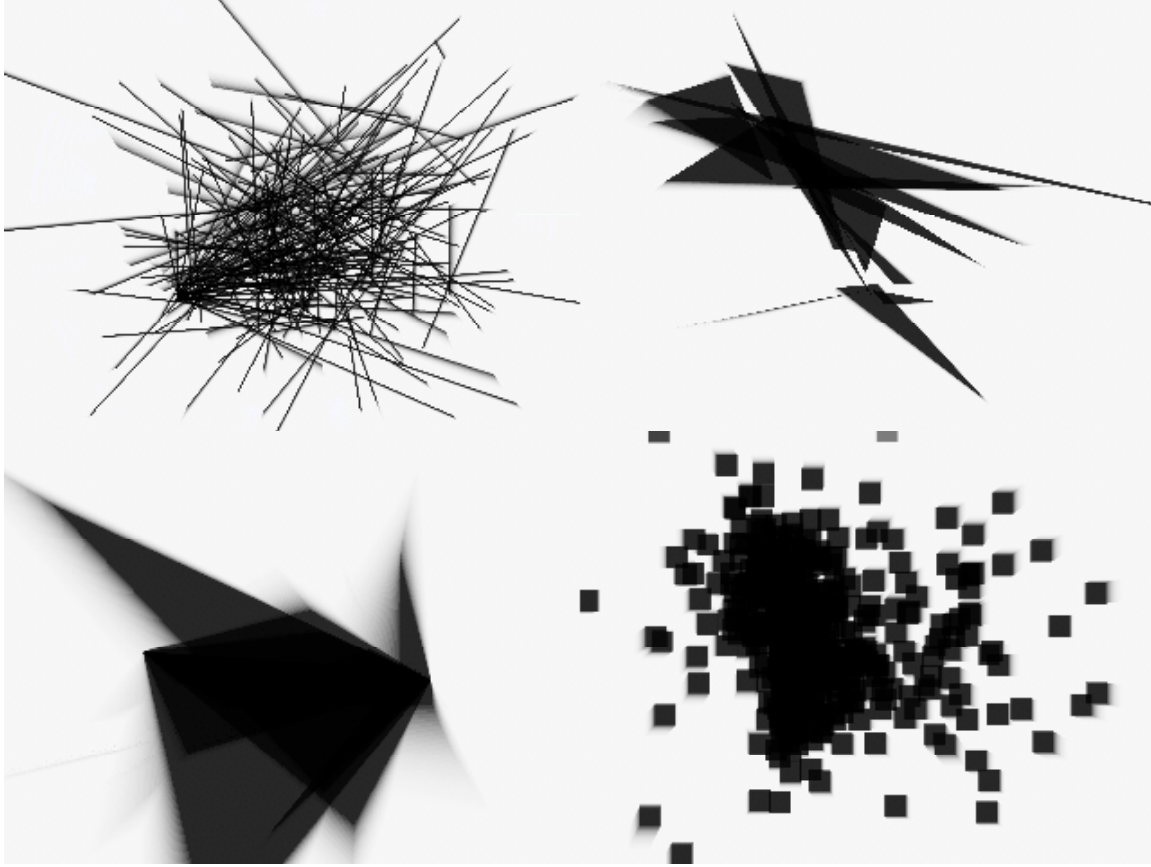
    // resize attractor matrix
    attgen.dim = ATTRACTOR_COUNT;

    init(); // re-initialize particle system
}
```

- In the Max patcher, change the **ubumenu** object labeled *Drawing primitive*. Try different settings and notice how it changes the way the particle system is drawn. The `primitive()` function in our JavaScript code changes the value of the variable *draw_primitive*.

Our particle system is visualized by sending the matrix of particle positions (referred to as *particlemat* in our JavaScript code) to the **jit.gl.render** object. The matrix contains 3 planes of *float32* data, which **jit.gl.render** interprets as x,y,z vertices in a geometry. The *drawing primitive*, which is a symbol appended to the `jit_matrix uxxxxxxxxx` message sent from the **js** object to the **jit.gl.render** object, defines the way in which our OpenGL drawing context visualizes the data.

For more information on the specifics of these drawing primitives and the OpenGL matrix format, consult *Appendix B: The OpenGL Matrix Format* or the OpenGL “Redbook.”



Different ways of visualizing our particles using different drawing primitives.

- Look at the JavaScript code for the `smear()` function. This is the function that allows us to leave trails when we hold down the ‘s’ key on our keyboard (which triggers the **toggle box** attached to the `smear $1 message box`).

```
function smear(v) // turn on drawing smear by zeroing the alpha
on the renderer's erase color
{
  if(v) {
    // smear on (alpha=0):
    outlet(0, "erase_color", 1., 1., 1., 0.);
  }
  else {
    // smear off (alpha=0.1):
    outlet(0, "erase_color", 1., 1., 1., 0.1);
  }
}
```

Sending a smear message to our **js** object causes our JavaScript code to send an `erase_color` message to our **jit.gl.render** object. If the argument to `smear` is `1`, we lower the *alpha* of the `erase_color` to `0`. This has the result of the drawing context doing nothing in response to the `erase` message triggered by the **qmetro** in our patch. A smear value of `1` sets the **jit.gl.render** object's `erase_color` attribute back to an alpha of `0.1`, which makes the renderer erase 10% of the image in response to the `erase` message. This causes a small amount of trailing which aids in the visualization of the particle movement.

- Play around with the patch some more, looking at the different ways we can generate and visualize our particles. A wide variety of interesting systems can be created simply by passing unadorned `x,y,z` values to the **jit.gl.render** object as 3-plane matrices.

Summary

JavaScript can be a powerful language to use when designing algorithms that manipulate matrix data in Jitter. The ability to perform mathematical operations directly on matrices using a variety of techniques (`op()` methods, **jit.expr** objects, and cell-by-cell iteration) within procedural code lets you take advantage of Jitter as a tool to process large sets of data at once.

In the next tutorial, we'll look at ways to trigger callback functions in JavaScript based on the actions of `JitterObject` objects themselves.

Code Listing

```
// 46jParticles.js
//
// a 3-D particle generator with simple gravity simulation
// demonstrating different techniques for mathematical
// matrix manipulation using Jitter objects in [js].
//
// rld, 7.05
//

inlets = 1;
outlets = 1;

var PARTICLE_COUNT = 1000; // initial number of particle vertices
var ATTRACTOR_COUNT = 3; // initial number of points of gravity

// create a [jit.noise] object for particle and velocity
// generation
var noisegen = new JitterObject("jit.noise");
noisegen.dim = PARTICLE_COUNT;
noisegen.planecount = 3;
noisegen.type = "float32";
```

```

// create a [jit.noise] object for attractor generation
var attgen = new JitterObject("jit.noise");
attgen.dim = ATTRACTOR_COUNT;
attgen.planecount = 3;
attgen.type = "float32";

// create two [jit.expr] objects for the bang_expr() function

// first expression: sum all the planes in the input matrix
var myexpr = new JitterObject("jit.expr");
myexpr.expr = "in[0].p[0]+in[0].p[1]+in[0].p[2]";
// second expression: evaluate a+((b-c)*d/e)
var myexpr2 = new JitterObject("jit.expr");
myexpr2.expr = "in[0]+((in[1]-in[2])*in[3]/in[4])";

// create the Jitter matrices we need to store our data

// matrix of x,y,z particle vertices
var particlemat = new JitterMatrix(3, "float32", PARTICLE_COUNT);
// matrix of x,y,z particle velocities
var velomat = new JitterMatrix(3, "float32", PARTICLE_COUNT);
// matrix of x,y,z points of attraction (gravity centers)
var attmat = new JitterMatrix(3, "float32", ATTRACTOR_COUNT);
// matrix for aggregate distances
var distmat = new JitterMatrix(3, "float32", PARTICLE_COUNT);
// temporary matrix for the bang_op() function
var tempmat = new JitterMatrix(3, "float32", PARTICLE_COUNT);
// temporary summing matrix for the bang_op() function
var summat = new JitterMatrix(1, "float32", PARTICLE_COUNT);
// another temporary summing matrix for the bang_op() function
var summat2 = new JitterMatrix(1, "float32", PARTICLE_COUNT);
// a scalar matrix to store the current gravity point
var scalarmat = new JitterMatrix(3, "float32", PARTICLE_COUNT);
// a scalar matrix to store acceleration (expr_op() function
only)
var amat = new JitterMatrix(1, "float32", PARTICLE_COUNT);

var a = 0.001; // acceleration factor
var d = 0.01; // decay factor

var perform_mode="op"; // default perform function
var draw_primitive = "points"; // default drawing primitive

function loadbang() // execute this code when our Max patch opens
{
    init(); // initialize our matrices
    post("particles initialized.\n");
}

function init()
// initialization routine... call at load, as well as
// when we change the number of particles or attractors
{

```

```

    // generate a matrix of random particles spread between -1 and
1    noisegen.matrixcalc(particlemat, particlemat);
    particlemat.op("*", 2.0);
    particlemat.op("-", 1.0);
    // generate a matrix of random velocities spread between -1
and 1    noisegen.matrixcalc(velomat, velomat);
    velomat.op("*", 2.0);
    velomat.op("-", 1.0);
    // generate a matrix of random attractors spread between -1
and 1    attngen.matrixcalc(attmat, attmat);
    attmat.op("*", 2.0);
    attmat.op("-", 1.0);
}

function bang() // perform one iteration of our particle system
{
    switch(perform_mode) { // choose from the following...
        case "op": // use Jitter matrix operators
            bang_op();
            break;
        case "expr": // use [jit.expr] for the bulk of the
algorithm    bang_expr();
            break;
        case "iter": // iterate cell-by-cell through the matrices
            bang_iter();
            break;
        default: // use bang_op() as our default
            bang_op();
            break;
    }

    // output our new matrix of particle vertices
    // with the current drawing primitive
    outlet(0, "jit_matrix", particlemat.name, draw_primitive);
}

```

```

function bang_op() // create our particle matrix using Matrix
operators
{
    for(var i = 0; i < ATTRACTOR_COUNT; i++)
        // do one iteration per gravity point
        {
            // create a scalar matrix out of the current attractor:
            scalarmat.setall(attmat.getcell(i));

            // subtract our particle positions from the current
            attractor
            // and store in a temporary matrix (x,y,z):
            tempmat.op("-", scalarmat, particlemat);

            // square to create a cartesian distance matrix (x*x, y*y,
            z*z):
            distmat.op("*", tempmat, tempmat);

            // sum the planes of the distance matrix (x*x+y*y+z*z)
            summat.planemap = 0;
            summat.frommatrix(distmat);
            summat2.planemap = 1;
            summat.frommatrix(distmat);
            summat.op("+", summat, summat2);
            summat2.planemap = 2;
            summat2.frommatrix(distmat);
            summat.op("+", summat, summat2);

            // scale our distances by the acceleration value:
            tempmat.op("*", a);
            // divide our distances by the sum of the distances
            // to derive gravity for this frame:
            tempmat.op("/", summat);
            // add to the current velocity bearings to get the
            // amount of motion for this frame:
            velomat.op("+", tempmat);
        }

        // offset our current positions by the amount of motion:
        particlemat.op("+", velomat);
        // reduce our velocities by the decay factor for the next
        frame:
        velomat.op("*", d);
    }
}

```



```

function bang_expr() // create our particle matrix using
[jit.expr]
{
    // create a scalar matrix out of our acceleration value:
    amat.setall(a);

    for(var i = 0; i < ATTRACTOR_COUNT; i++)
    // do one iteration per gravity point
    {
        // create a scalar matrix out of the current attractor:
        scalarmat.setall(attmat.getcell(i));
        // subtract our particle positions from the current
        attractor
        // and store in a temporary matrix (x,y,z):
        tempmat.op("-", scalarmat, particlemat);
        // square to create a cartesian distance matrix (x*x, y*y,
        z*z):
        distmat.op("*", tempmat, tempmat);

        // sum the planes of the distance matrix (x*x+y*y+z*z) :
        // "in[0].p[0]+in[0].p[1]+in[0].p[2]" :
        myexpr.matrixcalc(distmat, summat);

        // derive amount of motion for this frame :
        // "in[0]+((in[1]-in[2])*in[3]/in[4])" :

        myexpr2.matrixcalc([velomat,scalarmat,particlemat,amat,summat]
        ,
        velomat);
    }

    // offset our current positions by the amount of motion:
    particlemat.op("+", velomat);
    // reduce our velocities by the decay factor for the next
    frame:
    velomat.op("*", d);
}

```

```

function bang_iter() // create our particle matrix cell-by-cell
{
    var p_array = new Array(3); // array for a single particle
    var v_array = new Array(3); // array for a single velocity
    var a_array = new Array(3); // array for a single attractor

    for(var j = 0; j < PARTICLE_COUNT; j++)
    // do one iteration per particle
    {
        // fill an array with the current particle:
        p_array = particlemat.getcell(j);
        // fill an array with the current particle's velocity:
        v_array = velomat.getcell(j);

        for(var i = 0; i < ATTRACTOR_COUNT; i++)
        // do one iteration per gravity point
        {
            // fill an array with the current attractor:
            a_array = attmat.getcell(i);

            // find the distance from this particle to the
            // current attractor:
            var distsum = (a_array[0]-p_array[0])*(a_array[0]-
p_array[0]);
            distsum+= (a_array[1]-p_array[1])*(a_array[1]-
p_array[1]);
            distsum+= (a_array[2]-p_array[2])*(a_array[2]-
p_array[2]);

            // derive the amount of motion for this frame:
            v_array[0]+= (a_array[0]-p_array[0])*a/distsum; // x
            v_array[1]+= (a_array[1]-p_array[1])*a/distsum; // y
            v_array[2]+= (a_array[2]-p_array[2])*a/distsum; // z
        }

        // offset our current positions by the amount of motion
        p_array[0]+=v_array[0]; // x
        p_array[1]+=v_array[1]; // y
        p_array[2]+=v_array[2]; // z

        // reduce our velocities by the decay factor for the next
frame:
        v_array[0]*=d; // x
        v_array[1]*=d; // y
        v_array[2]*=d; // z

        // set the position for this particle in the Jitter matrix:
        particlemat.setcellld(j, p_array[0],p_array[1],p_array[2]);
        // set the velocity for this particle in the Jitter matrix:
        velomat.setcellld(j, v_array[0],v_array[1],v_array[2]);
    }
}

```

```

function particles(v) // change the number of particles we're
working with
{
    PARTICLE_COUNT = v;

    // resize matrices
    noisegen.dim = PARTICLE_COUNT;
    particlemat.dim = PARTICLE_COUNT;
    velomat.dim = PARTICLE_COUNT;
    distmat.dim = PARTICLE_COUNT;
    attmat.dim = PARTICLE_COUNT;
    tempmat.dim = PARTICLE_COUNT;
    summat.dim = PARTICLE_COUNT;
    summat2.dim = PARTICLE_COUNT;
    scalarmat.dim = PARTICLE_COUNT;
    amat.dim = PARTICLE_COUNT;

    init(); // re-initialize particle system
}

function attractors(v)
// change the number of gravity points we're working with
{
    ATTRACTOR_COUNT = v;

    // resize attractor matrix
    attgen.dim = ATTRACTOR_COUNT;

    init(); // re-initialize particle system
}

function accel(v) // set acceleration
{
    a = v*0.001;
}

function decay(v) // set decay
{
    d = v*0.001;
}

function mode(v) // change perform mode
{
    perform_mode = v;
}

function primitive(v) // change OpenGL drawing primitive
{
    draw_primitive = v;
}

```

```

function smear(v) // turn on drawing smear by zeroing the alpha
on the renderer's erase color
{
    if(v) {
        // smear on (alpha=0):
        outlet(0, "erase_color", 1., 1., 1., 0.);
    }
    else {
        // smear off (alpha=0.1):
        outlet(0, "erase_color", 1., 1., 1., 0.1);
    }
}

function bang() // perform one iteration of our particle system
{
    switch(perform_mode) { // choose from the following...
        case "op": // use Jitter matrix operators
            bang_op();
            break;
        case "expr": // use [jit.expr] for the bulk of the
algorithm
            bang_expr();
            break;
        case "iter": // iterate cell-by-cell through the matrices
            bang_iter();
            break;
        default: // use bang_op() as our default
            bang_op();
            break;
    }

    // output our new matrix of particle vertices with the current
drawing primitive
    outlet(0, "jit_matrix", particlemat.name, draw_primitive);
}

function bang_op() // create our particle matrix using Matrix
operators
{
    for(var i = 0; i < ATTRACTOR_COUNT; i++) // do one iteration
per gravity point
    {
        scalarmat.setall(attmat.getcell(i)); // create a scalar
matrix out of the current attractor

        tempmat.op("-", scalarmat, particlemat); // subtract our
particle positions from the current attractor and store in a
temporary matrix (x,y,z)

        distmat.op("*", tempmat, tempmat); // square to create our
cartesian distance matrix (x*x, y*y, z*z)
    }
}

```

```

    // sum the planes of the distance matrix (x*x+y*y+z*z)
    summat.planemap = 0;
    summat.frommatrix(distmat);
    summat2.planemap = 1;
    summat.frommatrix(distmat);
    summat.op("+", summat, summat2);
    summat2.planemap = 2;
    summat2.frommatrix(distmat);
    summat.op("+", summat, summat2);

    tempmat.op("*", a); // scale our distances by the
acceleration value
    tempmat.op("/", summat); // divide our distances by the sum
of the distances to derive gravity for this frame
    velomat.op("+", tempmat); // add to the current velocity
bearings to get the amount of motion for this frame
}

    particlemat.op("+", velomat); // offset our current
positions by the amount of motion
    velomat.op("*", d); // reduce our velocities by the decay
factor for the next frame
}

function bang_expr() // create our particle matrix using
[jit.expr]
{
    amat.setall(a); // create a scalar matrix out of our
acceleration value

    for(var i = 0; i < ATTRACTOR_COUNT; i++) // do one iteration
per gravity point
    {
        scalarmat.setall(attmat.getcell(i)); // create a scalar
matrix out of the current attractor

        tempmat.op("-", scalarmat, particlemat); // subtract our
particle positions from the current attractor and store in a
temporary matrix (x,y,z)

        distmat.op("*", tempmat, tempmat); // square to create our
cartesian distance matrix (x*x, y*y, z*z)

        myexpr.matrixcalc(distmat, summat); // sum the planes of
the distance matrix (x*x+y*y+z*z) :
"in[0].p[0]+in[0].p[1]+in[0].p[2]"

        myexpr2.matrixcalc([velomat,scalarmat,particlemat,amat,summat]
, velomat); // derive amount of motion for this frame :
"in[0]+((in[1]-in[2])*in[3]/in[4])"
    }
}

```

```

        particlemat.op("+", velomat); // offset our current
positions by the amount of motion
        velomat.op("*", d); // reduce our velocities by the decay
factor for the next frame
    }

function bang_iter() // create our particle matrix cell-by-cell
{
    var p_array = new Array(3); // create an array for a single
particle (x,y,z)
    var v_array = new Array(3); // create an array for a single
velocity (x,y,z)
    var a_array = new Array(3); // create an array for a single
attractor (x,y,z)

    for(var j = 0; j < PARTICLE_COUNT; j++) // do one iteration
per particle
    {
        p_array = particlemat.getcell(j); // fill an array with the
current particle
        v_array = velomat.getcell(j); // fill an array with the
current particle's velocity

        for(var i = 0; i < ATTRACTOR_COUNT; i++) // do one
iteration per gravity point
        {
            a_array = attmat.getcell(i); // fill an array with the
current attractor

            // find the distance from this particle to the current
attractor
            var distsum = (a_array[0]-p_array[0])*(a_array[0]-
p_array[0]);
            distsum+= (a_array[1]-p_array[1])*(a_array[1]-
p_array[1]);
            distsum+= (a_array[2]-p_array[2])*(a_array[2]-
p_array[2]);

            v_array[0]+= (a_array[0]-p_array[0])*a/distsum; // derive
the amount of motion for this frame (x)
            v_array[1]+= (a_array[1]-p_array[1])*a/distsum; // derive
the amount of motion for this frame (y)
            v_array[2]+= (a_array[2]-p_array[2])*a/distsum; // derive
the amount of motion for this frame (z)
        }

        p_array[0]+=v_array[0]; // offset our current positions by
the amount of motion (x)
        p_array[1]+=v_array[1]; // offset our current positions by
the amount of motion (y)
        p_array[2]+=v_array[2]; // offset our current positions by
the amount of motion (z)
    }
}

```

```

        v_array[0]*=d; // reduce our velocities by the decay factor
for the next frame (x)
        v_array[1]*=d; // reduce our velocities by the decay factor
for the next frame (y)
        v_array[2]*=d; // reduce our velocities by the decay factor
for the next frame (z)

        particlemat.setcellld(j, p_array[0],p_array[1],p_array[2]);
// set the position for this particle in the Jitter matrix
        velomat.setcellld(j, v_array[0],v_array[1],v_array[2]); //
set the velocity for this particle in the Jitter matrix
    }
}

function particles(v) // change the number of particles we're
working with
{
    PARTICLE_COUNT = v;

    // resize matrices
    noisegen.dim = PARTICLE_COUNT;
    particlemat.dim = PARTICLE_COUNT;
    velomat.dim = PARTICLE_COUNT;
    distmat.dim = PARTICLE_COUNT;
    attmat.dim = PARTICLE_COUNT;
    tempmat.dim = PARTICLE_COUNT;
    summat.dim = PARTICLE_COUNT;
    summat2.dim = PARTICLE_COUNT;
    scalarmat.dim = PARTICLE_COUNT;
    amat.dim = PARTICLE_COUNT;

    init(); // re-initialize particle system
}

function attractors(v) // change the number of gravity points
we're working with
{
    ATTRACTOR_COUNT = v;

    // resize attractor matrix
    attgen.dim = ATTRACTOR_COUNT;

    init(); // re-initialize particle system
}

function accel(v) // set acceleration
{
    a = v*0.001;
}

```

```

function decay(v) // set decay
{
    d = v*0.001;
}

function mode(v) // change perform mode
{
    perform_mode = v;
}

function primitive(v) // change OpenGL drawing primitive
{
    draw_primitive = v;
}

function smear(v) // turn on drawing smear by zeroing the alpha
on the renderer's erase color
{
    if(v) {
        outlet(0, "erase_color", 1., 1., 1., 0.); // smear on
(alpha=0)
    }
    else {
        outlet(0, "erase_color", 1., 1., 1., 0.1); // smear off
(alpha=0.1)
    }
}

```


Tutorial 47: Using Jitter Object Callbacks in JavaScript

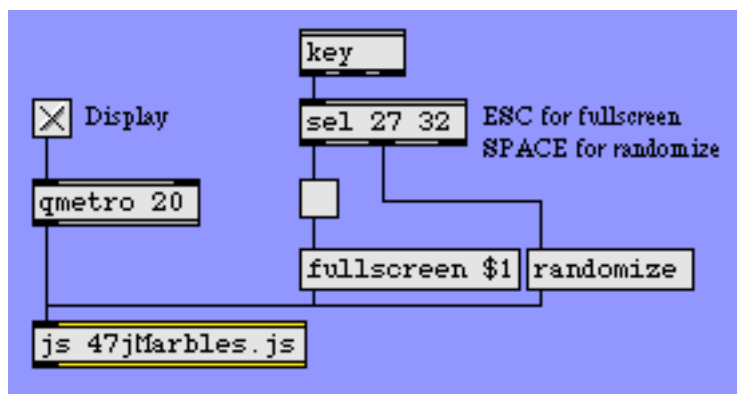
In the last two tutorials, we've looked at some of the possibilities and advantages of using Jitter objects and matrices within JavaScript code. In this tutorial we'll encapsulate an entire Jitter OpenGL patch inside of JavaScript, using many of the techniques we've already seen. Most importantly, however, we'll learn how to "listen" to the *output* of Jitter objects that have been encapsulated in JavaScript in order to design functions that respond interactively to these objects.

A number of Jitter objects (such as **jit.qt.movie**, **jit.pwindow**, and **jit.window**) output messages from their status (right-hand) outlets in response to processes initiated by the user in a Max patcher. For example, when you read a movie file into **jit.qt.movie**, the object outputs the message read followed by the movie's filename and a status number out its status outlet. Similarly, the **jit.pwindow** and **jit.window** objects can respond to mouse events in their windows by sending messages out their status outlet. Because Jitter objects instantiated within JavaScript have no inlets and outlets *per se*, we need to explicitly create an object (called a JitterListener) to catch those messages and call a function (called a "callback" function) in response to them.

This Tutorial assumes you've looked at the earlier JavaScript tutorials, as well as the first OpenGL tutorial *Tutorial 30: Drawing 3D Text*.

- Open the tutorial patch *47jJitterListener.pat* in the Jitter Tutorials folder.

The first thing we notice about this tutorial patch is that there is very little in it, in terms of Max objects. Virtually all the work done in the patcher is accomplished inside the **js** object in the patch.



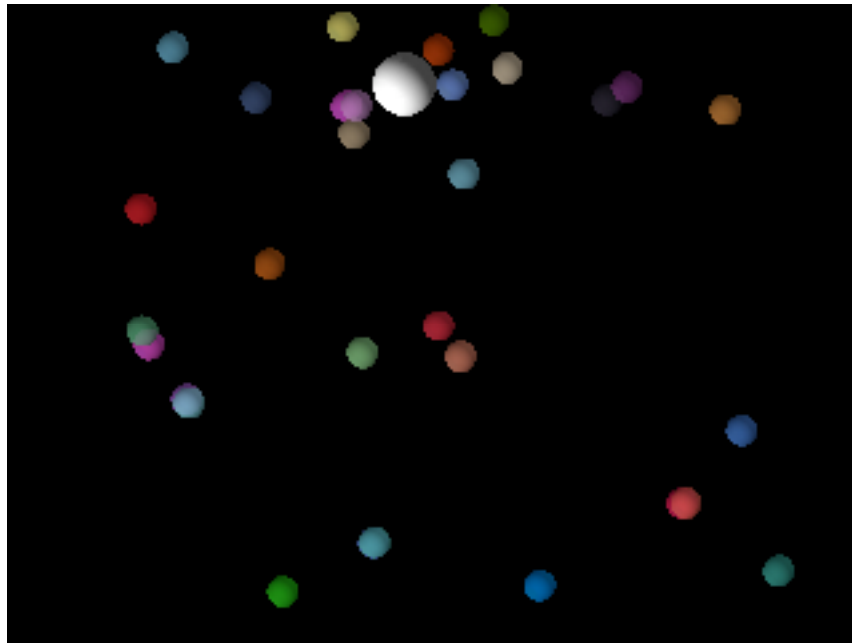
Our Tutorial patcher: not much to look at.

- Click on the **toggle box** labeled *Display*. The **qmetro** object will begin to send bang messages into the **js** object, and the **jit.window** object should fill with a number of shapes (one large sphere and a few dozen small spheres) on a black background.

Our **js** object contains a complete set of Jitter objects performing an OpenGL rendering (including a **jit.window** object).

- Move your mouse over the **jit.window**. The large sphere will “glue” to your mouse, following you along in the window. As the large sphere touches the smaller spheres, it will “push” them along as if they were solid objects colliding. Try to get a feel for moving the smaller spheres around with the larger one.

Our **js** file not only draws our OpenGL scene, but also handles interactive events from the mouse. This is done through a listening and callback mechanism (called a JitterListener) that we’ll learn about in the JavaScript code.



Pushing the spheres around.

Creating OpenGL objects in JavaScript

- Double-click the **js** object in our Tutorial patch. A text editor will appear that contains the source code for the **js** object in the patch. The code is saved as a file called ‘47jMarbles.js’ in the same folder as the Tutorial patch. Look at the global code block at the top of the file.

Our JavaScript code creates all the JitterObject objects we need to render our scene: a **jit.window** object to display the drawing context, a **jit.gl.render** object to perform the rendering, and a number of **jit.gl.gridshape** objects to draw and animate the spheres in the scene. These objects are instantiated in the global block of code at the beginning of the JavaScript file.

```
var OBJECT_COUNT = 32; // number of little spheres
```

This line declares a variable (*OBJECT_COUNT*) that determines the number of smaller spheres in our scene.

```
// create a [jit.window] object for our display
// (this is the object we'll "listen" to):
var mywindow = new JitterObject("jit.window","ListenWindow");
// turn off depth testing... we're using blending instead:
mywindow.depthbuffer = 0;
// turn ON idlemousing... we want to listen for it:
mywindow.idlemouse = 1;
```

Next, we create a **jit.window** object to display everything. We'll refer to it in our JavaScript code by the variable *mywindow*. Just as in a Max patch, the **jit.window** needs a name ("ListenWindow") that must match the OpenGL drawing context we're using. We've turned off the depthbuffer and turned on the idlemouse attribute, which allows the **jit.window** object to track mouse events in the window regardless of whether we've clicked our mouse down or not.

```
// create a [jit.gl.render] object for drawing into our window:
var myrender = new JitterObject("jit.gl.render","ListenWindow");
// use a 2-dimensional projection:
myrender.ortho = 2;
// set background to black with full erase opacity (no trails):
myrender.erase_color = [0,0,0,1];
```

Our **jit.gl.render** object (assigned to the variable *myrender*) defines a drawing context called "ListenWindow". The **jit.window** object (above) and the **jit.gl.gridshape** objects (below) share this name. We've decided to use an *orthographic* projection (by setting the ortho attribute to 2). This means that the z axis of our drawing context is ignored in terms of sizing objects based on their distance from the virtual camera. This essentially creates a 2-dimensional rendering of our scene. We've set the background to black (with a full erase between successive drawing) by setting our *erase_color* to 0001.

```
// create a [jit.gl.gridshape] object for use to control with the
mouse
var mywidget = new
JitterObject("jit.gl.gridshape","ListenWindow");
mywidget.shape = "sphere";
mywidget.lighting_enable = 1;
mywidget.smooth_shading = 1;
mywidget.scale = [0.1,0.1,0.1];
mywidget.color = [1,1,1,0.5] ;
mywidget.blend_enable = 1;
mywidget.position = [0,0]; // no z necessary in orthographic
projection
```

The first **jit.gl.gridshape** object we create (assigned to the variable *mywidget*) corresponds to the large sphere moved by our mouse in the **jit.window**. After creating it, we set all the relevant attributes we want the sphere to have, just as we would in a Max patcher. Note that in setting the position attribute we only use two arguments (*x* and *y*). In an orthographic projection the *z* axis is irrelevant to how the shapes are drawn.

```
// create an array of [jit.gl.gridshape] objects
// randomly arrayed across the window
var mysphere = new Array(OBJECT_COUNT);
```

Rather than naming our smaller spheres individually, we're treating them as an Array (called *mysphere*). Each element in this array will then be a separate **jit.gl.gridshape** object that we can access by array notation (e.g. *mysphere[5]* will be the *sixth* sphere, counting up from 0).

```
// initialize our little spheres with random colors and positions
(x,y)
for(var i=0;i<OBJECT_COUNT;i++) {
    mysphere[i] = new
    JitterObject("jit.gl.gridshape","ListenWindow");
    mysphere[i].shape = "sphere";
    mysphere[i].lighting_enable = 1;
    mysphere[i].smooth_shading = 1;
    mysphere[i].scale = [0.05,0.05,0.05];
    mysphere[i].color =
    [Math.random(),Math.random(),Math.random(),0.5] ;
    mysphere[i].position = [Math.random()*2.-1, Math.random()*2.-
    1];
    mysphere[i].blend_enable = 1;
}
```

This code actually creates the individual **jit.gl.gridshape** objects as individual JitterObjects that are members of the Array *mysphere*. We use a JavaScript `for ()` loop to set the initial attributes of all these objects in one block of code. We give them random initial color and position attributes, scattering them over the screen and coloring them all differently.

```
// create a JitterListener for our [jit.window] object
var mylistener = new JitterListener(mywindow.getregisteredname(),
thecallback);
```

Finally, we create a variable called *mylistener* that we assign to be a JitterListener object. JitterListener objects take two arguments: the *object* that they “listen” to, and the *function* that will be called when the object triggers an event. Our JitterListener object is set to listen to our **jit.window** object (*mywindow*). The `getregisteredname ()` property of a JitterObject object returns the name by which that object can be accessed by the JitterListener (in the case of **jit.window** objects, this will be the same as name of the drawing context). Whenever our **jit.window** object generates an event, a function called `thecallback ()` will be triggered in our JavaScript code. Now that we've instantiated a

JitterListener, we can (in most cases) leave it alone and simply deal with the mechanics of the callback function it triggers in response to an event from the object it listens to.

- Back in the Tutorial patcher, click the **message box** labeled randomize or hit the *spacebar* on your computer keyboard. The small spheres should scatter and change colors. Look at the code for the `randomize()` function in our **js** file.

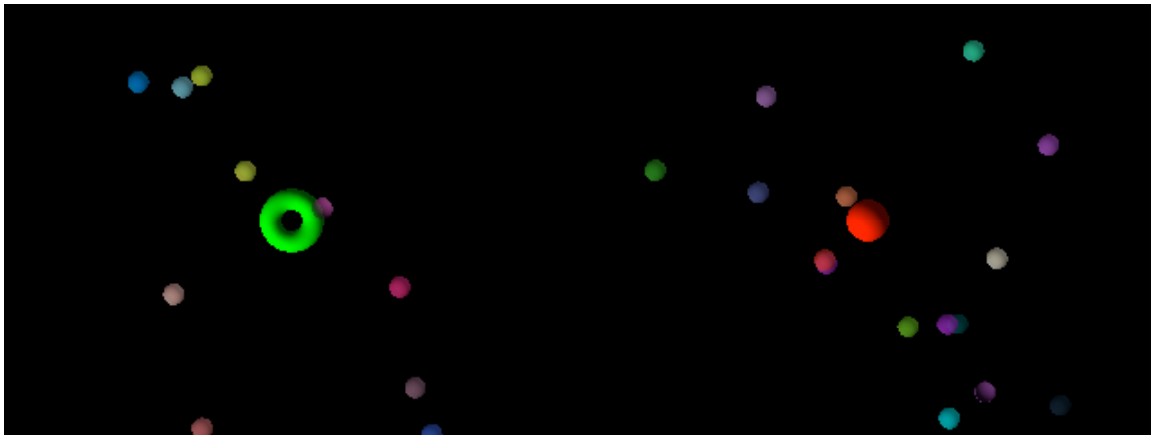
```
function randomize() // reorient the little spheres
{
  for(var i=0;i<32;i++) {
    mysphere[i].color =
[Math.random(),Math.random(),Math.random(),0.5];
    mysphere[i].position = [Math.random()*2.-1,
Math.random()*2.-1];
  }
}
```

Our `randomize()` function takes our little spheres and scatters them randomly, changing their colors. This allows us to basically restart our simulation whenever we like.

The callback function

The **jit.window** object responds to a variety of mouse events. When the `idlemouse` attribute is set to 1 (as it is in our code) the **jit.window** object outputs mouse information (the `mouseidle` message) when you drag the pointer over the window. It also tells us when the pointer has left the window area (the `mouseidleout` message). Regardless of the setting of the `idlemouse` attribute, the **jit.window** object also responds to mouse clicks inside the window (the `mouse` message).

- In the Tutorial patcher, move the pointer into the **jit.window** and click the mouse. The white sphere will turn into a green torus. Release the mouse and the torus will turn into a red sphere. Moving the pointer will turn the sphere white again.



How our sphere responds to mouse click events (mouse down and mouse up).

- Look at the code for the function called `thecallback()` in our JavaScript code. This code executes whenever our `JitterListener` object receives an event from our `jit.window` object.

```
function thecallback(event)
// callback function to handle events triggered by mousing
// in our [jit.window]
{
    var x,y,button; // some local variables
```

The *event* constructor stores the message sent by the `jit.window` object to the `JitterListener` object in our patch. We also declare a few local variables (*x*, *y*, and *button*) to store information about the pointer derived from the *event* message's arguments.

We then use JavaScript `if()` statements based on the `eventname` property of the *event* that triggered the callback function. The `eventname` property will contain the name of the message (`mouse`, `mouseidle`, etc.) that triggered the function.

```
    if (event.eventname=="mouse") {
        // we're entering, dragging within, or leaving a "mouse click"
event
```

This part of the code handles a mouse click event (a click down, dragging of the mouse, and the release of a click).

```
        // arguments are
(x,y,button,cmd,shift,capslock,option,ctrl)...
        // we only care about the first three
        x = event.args[0];
        y = event.args[1];
        button = event.args[2];
```

The first three arguments to the mouse event contain the position of the mouse (*x* and *y*) and a flag telling us whether the mouse button is down (1) or up (0). We store these settings into local variables (*x*, *y*, and *button*) for use later on.

```

        if (button) // we're clicked down
        {
            mywidget.color = [0,1,0,1]; // color our control object
green    mywidget.shape = "torus"; // change it to a donut shape
        }
        else // we've just unclicked
        {
            mywidget.color = [1,0,0,1]; // color our object red
            mywidget.shape = "sphere"; // change back to a sphere
        }
    }
}

```

If our button is down, we turn our **jit.gl.gridshape** object *mywidget* from a white sphere into a green torus. When we release the mouse, we turn it into a red sphere.

```

        else if (event.eventname=="mouseidle") {
            // we're mousing over the window with the mouse up
            x = event.args[0];
            y = event.args[1];
            mywidget.color = [1,1,1,1] ; // color our object white
        }
    }
}

```

This code executes whenever we drag our pointer across the **jit.window** object without clicking the pointer. All we do here is store the variables *x* and *y* based on the pointer position. We also turn our **jit.gl.gridshape** object (*mywidget*) white, in case the previous callback had turned it a different color.

```

        else if (event.eventname=="mouseidleout") {
            // we're no longer mousing over the window
            x = event.args[0];
            y = event.args[1];
            mywidget.color = [1,1,1,0.5] ; // make our object
translucent
        }
    }
}

```

When our pointer leaves the **jit.window**, the *eventname* property of our callback will be “mouseidleout”. We treat this the same as the “mouseidle” event. If we wanted to, we could treat it differently (e.g. make the white sphere disappear, or randomize the spheres).

```

        // move our control object to the drawing context's
        // equivalent of where our mouse event occurred:
        mywidget.position = myrender.screentoworld(x,y);
    }
}

```

Finally, we update the position of the **jit.gl.gridshape** object controlled by our mouse by updating the position attribute of the JitterObject object *mywidget*. Because our drawing context works in OpenGL world coordinates (where the default center of the space is (0, 0) and positions are expressed in decimal values), we need to convert our *x* and *y* values into numbers that make sense for our drawing context. The `screenToWorld()` method to a **jit.gl.render** object will convert those values from pixel coordinates on the display to world coordinates for the rendering.

```
}  
// don't allow this function to be called from Max  
thecallback.local = 1;
```

Because our `thecallback()` function is intended to be executed by our JitterListener object, we set its local property to 1 to prevent it from being executed directly by a Max message sent to the **js** object.

Drawing the scene

Now that we've seen how to respond to mouse events and update our *mywidget* object's position, color, and shape accordingly, we need to look at how we perform the actual rendering of our scene in response to a bang message from the **qmetro** object in our Max patcher. The `bang()` function in our JavaScript code handles the drawing as well as the animation of the little spheres in response to the new position of our large sphere.

- Look at the code for the `bang()` function.

The majority of our `bang()` function iterates through the positions of all the little spheres and compares them to the position of the large one. If the distance between them is small enough that the spheres would overlap when drawn, we move the little sphere away the minimum amount necessary to have the spheres touch. This type of processing is known as *collision detection* and is a ubiquitous technique in the field of computer animation.

```
function bang()  
// main drawing loop... figure out which little spheres to move  
// and drive the renderer  
{  
  // collision detection block. we need to iterate through  
  // the little spheres and check their distance from the  
control  
  // object. if we're touching we move the little sphere away  
  // along the correct angle of contact.  
  for(var i = 0; i<OBJECT_COUNT; i++) {
```


We enter a JavaScript `for ()` loop to check the position of our large sphere against every small sphere, one at a time.

```
// cartesian distance along the x and y axis
var distx = mywidget.position[0]-mysphere[i].position[0];
var disty = mywidget.position[1]-mysphere[i].position[1];

// polar distance between the two objects
var r = Math.sqrt(distx*distx+disty*disty);
// angle of little sphere around control object
var theta = Math.atan2(disty,distx);
```

By subtracting the x and y positions of the two spheres (large and small) from one another, we get their Cartesian distance. We can convert this to polar (absolute) distance by applying the Pythagorean theorem to derive the hypotenuse (stored in the variable r). We then derive the angle of the small sphere around the large one by taking the arctangent of the rise (x) over the run (y).

```
// check for collision...
if(r<0.15)
// control object is size 0.1, little spheres are 0.05,
// so less than 0.15 and it's a hit...
{
```

Because our large sphere has a scale attribute of 0.1 0.1 0.1 and our small spheres have scale attributes of 0.05 0.05 0.05, we can infer that the objects are overlapping if the distance between them is less than 0.15. This triggers a block of code that deals with the collision.

```
    // convert polar->cartesian to figure out x and y
displacement
    var movex = (0.15-r)*Math.cos(theta);
    var movey = (0.15-r)*Math.sin(theta);

    // offset the little sphere to the new position,
    // which should be just beyond touching at the
    // angle of contact we had before. the result
    // should look like we've "pushed" it along...
    mysphere[i].position = [mysphere[i].position[0]-movex,
mysphere[i].position[1]-movey];
    }
}
```

We use the polar distance and angle, combined with the minimum allowable distance between the objects (0.15), to derive how far we need to “nudge” the little sphere along the x and y axes to stop it from overlapping the large sphere. We figure out these values by converting the polar coordinates back into Cartesian values stored in the variables *movex* and *movey*. We then subtract these two variables’ values from the current position of the small sphere, pushing it out of the way.

```

    // rendering block...
    myrender.erase(); // erase the drawing context
    myrender.drawclients(); // draw the client objects
    myrender.swap(); // swap in the new drawing
}

```

This last block of code does the actual rendering. Just as you would when working with Jitter OpenGL objects in a Max patcher, you send an `erase` message to the **jit.gl.render** object (*myrender*). The `drawclients()` method to **jit.gl.render** collects all the relevant information from the OpenGL objects attached to our drawing context and draws them; any OpenGL objects with an automatic attribute set to 0 will have to be drawn manually here. The `swap()` method then replaces the old rendering with the new one in the **jit.window** object, showing us the updated scene. The `drawclients()` and `swap()` method are combined into one operation when you send a `bang` message to a **jit.gl.render** object in a Max patcher.

- Back in the Tutorial patcher, click the **toggle box** attached to the **message box** labeled `fullscreen $1` (or hit the *escape* key on your computer keyboard). The **jit.window** object will fill the screen. Notice that the coordinate conversion that allows you to move the large sphere around the screen with the mouse still works fine. You can take the **jit.window** object out of fullscreen mode by pressing the *escape* key on your keyboard. In the JavaScript code, look at the `fullscreen()` function:

```

function fullscreen(v)
// function to send the [jit.window] into fullscreen mode
{
    mywindow.fullscreen = v;
}

```

The `fullscreen()` attribute of a **jit.window** JitterObject object behaves just as the `fullscreen` message sent to a **jit.window** object in a Max patcher.

Summary

An entire OpenGL Jitter patch can be encapsulated in JavaScript by instantiating **jit.gl.render**, **jit.window**, and other OpenGL objects within procedural code written for the **js** object. These objects have all their messages and attributes exposed as corresponding methods and properties. You can use a JitterListener object to respond to events triggered by a Jitter object within JavaScript. The JitterListener then executes a callback function, passing the calling message to it as its argument. This allows you to write JavaScript functions to respond to mouse interactivity in a **jit.window** object, file reading in a **jit.qt.movie** object, and other situations where you would want to respond to an event triggered by a message sent by a Jitter object out its status outlet in a Max patcher.

Code Listing

```
// 47jMarbles.js
//
// Demonstrates how to set up a JitterListener object to listen
// to the output of a Jitter object instantiated within [js].
//
// rld, 7.05
//

var OBJECT_COUNT = 32; // number of little spheres

// create a [jit.window] object for our display
// (this is the object we'll "listen" to):
var mywindow = new JitterObject("jit.window","ListenWindow");
// turn off depth testing... we're using blending instead:
mywindow.depthbuffer = 0;
// turn ON idlemousing... we want to listen for it:
mywindow.idlemouse = 1;

// create a [jit.gl.render] object for drawing into our window:
var myrender = new JitterObject("jit.gl.render","ListenWindow");
// use a 2-dimensional projection:
myrender.ortho = 2;
// set background to black with full erase opacity (no trails):
myrender.erase_color = [0,0,0,1];

// create a [jit.gl.gridshape] object for use to control with the
mouse
var mywidget = new
JitterObject("jit.gl.gridshape","ListenWindow");
mywidget.shape = "sphere";
mywidget.lighting_enable = 1;
mywidget.smooth_shading = 1;
mywidget.scale = [0.1,0.1,0.1];
mywidget.color = [1,1,1,0.5] ;
mywidget.blend_enable = 1;
mywidget.position = [0,0]; // no z necessary in orthographic
projection

// create an array of [jit.gl.gridshape] objects randomly arrayed
across the window
var mysphere = new Array(OBJECT_COUNT);

// initialize our little spheres with random colors and positions
(x,y)
for(var i=0;i<OBJECT_COUNT;i++) {
    mysphere[i] = new
JitterObject("jit.gl.gridshape","ListenWindow");
    mysphere[i].shape = "sphere";
    mysphere[i].lighting_enable = 1;
    mysphere[i].smooth_shading = 1;
```

```

    mysphere[i].scale = [0.05,0.05,0.05];
    mysphere[i].color =
[Math.random(),Math.random(),Math.random(),0.5] ;
    mysphere[i].position = [Math.random()*2.-1, Math.random()*2.-
1];
    mysphere[i].blend_enable = 1;
}

// create a JitterListener for our [jit.window] object
var mylistener = new JitterListener(mywindow.getregisteredname(),
thecallback);

function randomize() // reorient the little spheres
{
    for(var i=0;i<32;i++) {
        mysphere[i].color =
[Math.random(),Math.random(),Math.random(),0.5] ;
        mysphere[i].position = [Math.random()*2.-1,
Math.random()*2.-1];
    }
}

function thecallback(event)
// callback function to handle events triggered by mousing
// in our [jit.window]
{
    var x,y,button; // some local variables

    if (event.eventname=="mouse") {
        // we're entering, dragging within, or leaving a "mouse click"
event

        // arguments are
(x,y,button,cmd,shift,capslock,option,ctrl)...
        // we only care about the first three
        x = event.args[0];
        y = event.args[1];
        button = event.args[2];
    }
}

```

```

        if (button) // we're clicked down
        {
            mywidget.color = [0,1,0,1]; // color our control object
green
            mywidget.shape = "torus"; // change it to a donut shape
        }
        else // we've just unclicked
        {
            mywidget.color = [1,0,0,1]; // color our object red
            mywidget.shape = "sphere"; // change back to a sphere
        }
    }
    else if (event.eventname=="mouseidle") {
        // we're mousing over the window with the mouse up
        x = event.args[0];
        y = event.args[1];
        mywidget.color = [1,1,1,1] ; // color our object white
    }
    else if (event.eventname=="mouseidleout") {
        // we're no longer mousing over the window
        x = event.args[0];
        y = event.args[1];
        mywidget.color = [1,1,1,0.5] ; // make our object
translucent
    }

    // move our control object to the drawing context's
    // equivalent of where our mouse event occurred:
    mywidget.position = myrender.screentoworld(x,y);
}
// don't allow this function to be called from Max
thecallback.local = 1;

function bang()
// main drawing loop... figure out which little spheres to move
// and drive the renderer
{
    // collision detection block. we need to iterate through
    // the little spheres and check their distance from the
control
    // object. if we're touching we move the little sphere away
    // along the correct angle of contact.
    for(var i = 0;i<OBJECT_COUNT;i++) {
        // cartesian distance along the x and y axis
        var distx = mywidget.position[0]-mysphere[i].position[0];
        var disty = mywidget.position[1]-mysphere[i].position[1];

        // polar distance between the two objects
        var r = Math.sqrt(distx*distx+disty*disty);
        // angle of little sphere around control object
        var theta = Math.atan2(disty,distx);

        // check for collision...

```

```

        if(r<0.15)
        // control object is size 0.1, little spheres are 0.05,
        // so less than 0.15 and it's a hit...
        {
            // convert polar->cartesian to figure out x and y
displacement
            var movex = (0.15-r)*Math.cos(theta);
            var movey = (0.15-r)*Math.sin(theta);

            // offset the little sphere to the new position,
            // which should be just beyond touching at the
            // angle of contact we had before. the result
            // should look like we've "pushed" it along...
            mysphere[i].position = [mysphere[i].position[0]-movex,
mysphere[i].position[1]-movey];
        }
    }

    // rendering block...
    myrender.erase(); // erase the drawing context
    myrender.drawclients(); // draw the client objects
    myrender.swap(); // swap in the new drawing
}

function fullscreen(v)
// function to send the [jit.window] into fullscreen mode
{
    mywindow.fullscreen = v;
}

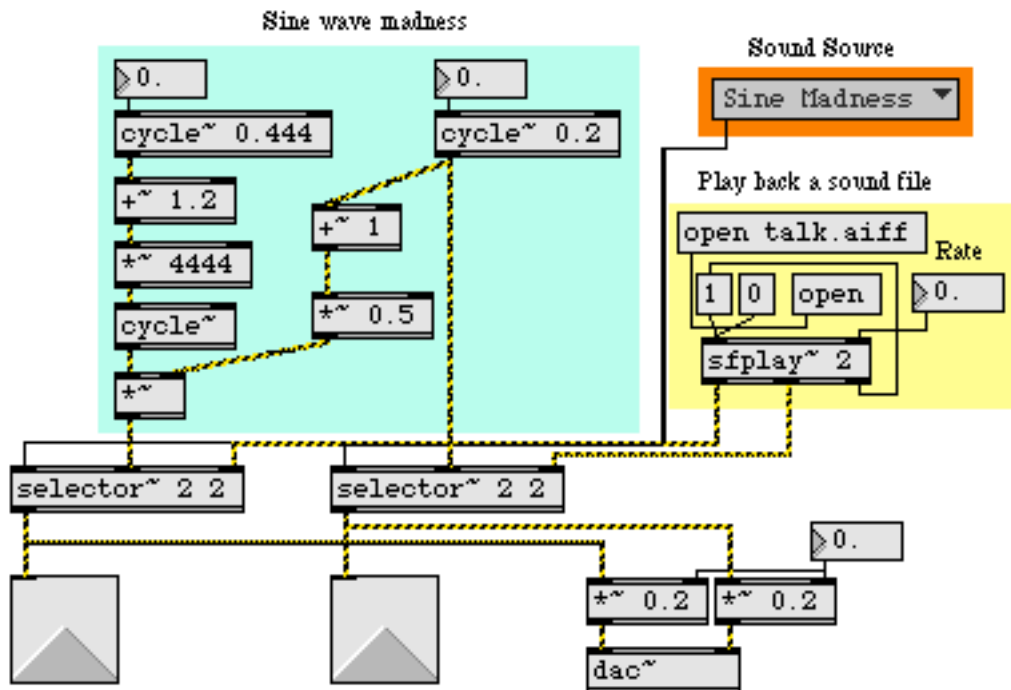
```

Tutorial 48: Frames of MSP signals

In *Tutorial 27: Using MSP Audio in a Jitter Matrix* we learned how to use the **jit.poke~** object to copy an MSP audio signal sample-by-sample into a Jitter matrix. This tutorial introduces **jit.catch~**, another object that moves data from the signal to the matrix domain. We'll see how to use the **jit.graph** objects to visualize audio and other one-dimensional data, and how **jit.catch~** can be used in a frame-based analysis context. We'll also meet the **jit.release~** object, which moves data from matrices to the signal domain, and see how we can use Jitter objects to process and synthesize sound.

- Open the Tutorial patch *48jSignals.pat* in the Jitter Tutorials folder.

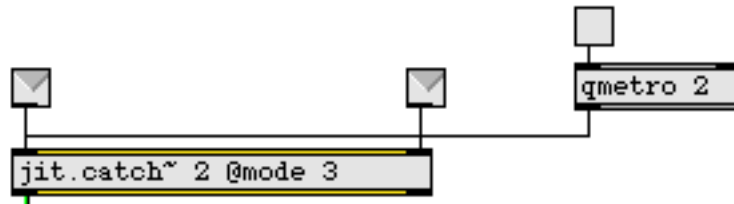
The patch is divided into several smaller subpatches. In the upper left the audio **patcher** contains a network labeled *Sine wave madness* where some **cycle~** objects modulate each other. Playback of an audio file is also possible through the **sfplay~** object in the upper-right hand corner of the subpatch.



Inside the Sine wave madness.

Basic Viz

- Click the **message box** labeled dsp start to begin processing audio vectors.
- Open the **patcher** named basic visualization. Click on the **toggle box** connected to the **qmetro** object.



Capturing and outputting the audio signals as Jitter matrices.

In the basic visualization **patcher** the two signals from the audio **patcher** are input into a **jit.catch~** object. The basic function of the **jit.catch~** object is to move signal data into the domain of Jitter matrices. The object supports the synchronous capture of multiple signals; the object's first argument sets the number of signal inputs; a separate inlet is created for each. The data from different signals is multiplexed into different planes of a float32 Jitter matrix. In the example in this subpatch, two signals are being captured. As a result, our output matrices will have two planes.

The **jit.catch~** object can operate in several different ways, according to the mode attribute. When the mode attribute is set to 0, the object simply outputs all the signal data that has been collected since the last bang was received. For example, if 1024 samples have been received since the last time the object received a bang, a one-dimensional 1024 cell float32 matrix would be output.

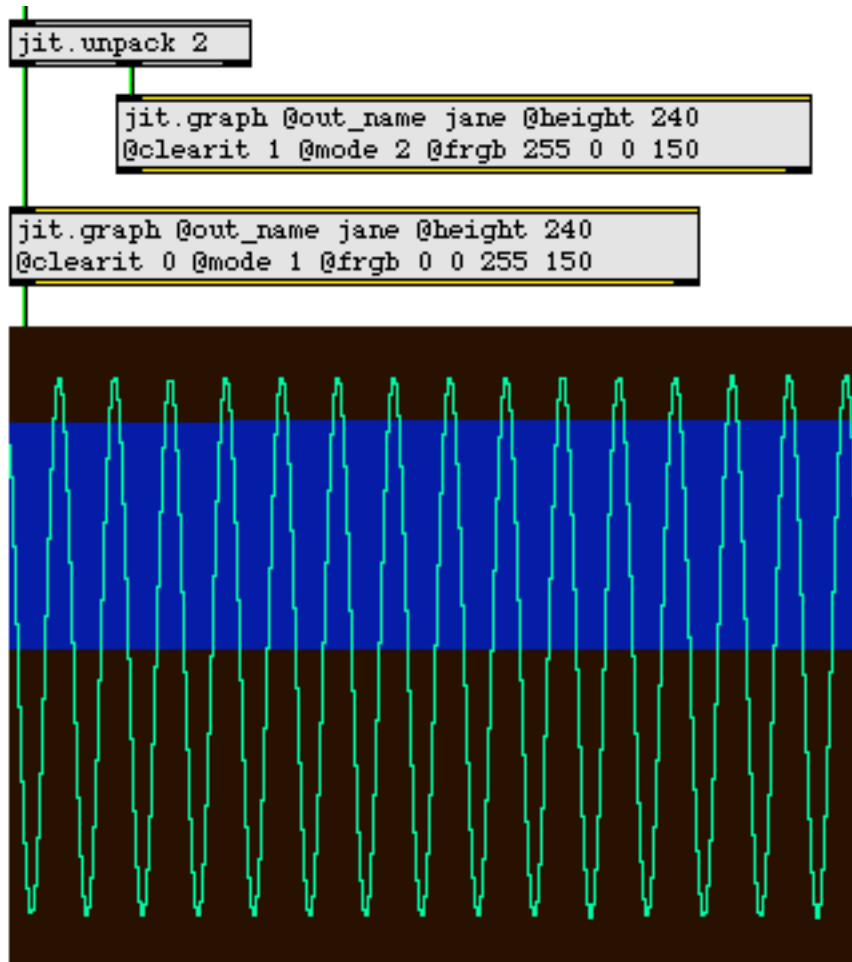
The object's mode 1 causes **jit.catch~** to output whatever fits in a multiple of a fixed frame size, the length of which can be set with the framesize attribute. The data is arranged in a two dimensional matrix with a width equal to the framesize. For instance, if the framesize were 100 and the same 1024 samples were internally cached waiting to be output, in mode 1 a bang would cause a 100x10 matrix to be output, and the 24 remaining samples would stay in the cache until the next bang. These 24 samples would be at the beginning of the next output frame as more signal data is received.

When working in mode 2, the object outputs the most recent data of framesize length. Using the example above (with a framesize of 100 and 1024 samples captured since our last output), in mode 2 our **jit.catch~** object would output the last 100 samples received.

In our basic visualization subpatch, our **jit.catch~** object is set to use mode 3, which causes it to act similarly to an oscilloscope in trigger mode. The object monitors the input data for

values that cross a threshold value set by the `trigthresh` attribute. This mode is most useful when looking at periodic waveforms, as we are in this example.

Note that mode 0 and mode 1 of the **jit.catch~** object attempt to output every signal value received exactly once, whereas mode 2 and mode 3 do not. Furthermore, the matrices output in mode 0 and mode 1 will be variable in size, whereas those output in mode 2 and mode 3 will always have the same dimensionality. For all modes, the **jit.catch~** object must maintain a pool of memory to hold the received signal values while they are waiting to be output. By default the object will cache 100 milliseconds of data, but this size can be increased by sending the object a `bufsize` message with an argument of the desired number of milliseconds. In mode 0 and mode 1, when more data is input than can be internally stored, the next bang will not output a matrix but instead will cause a buffer overrun message to be output from the dump (right-hand) outlet of the object.



Our basic visualization of the audio data.

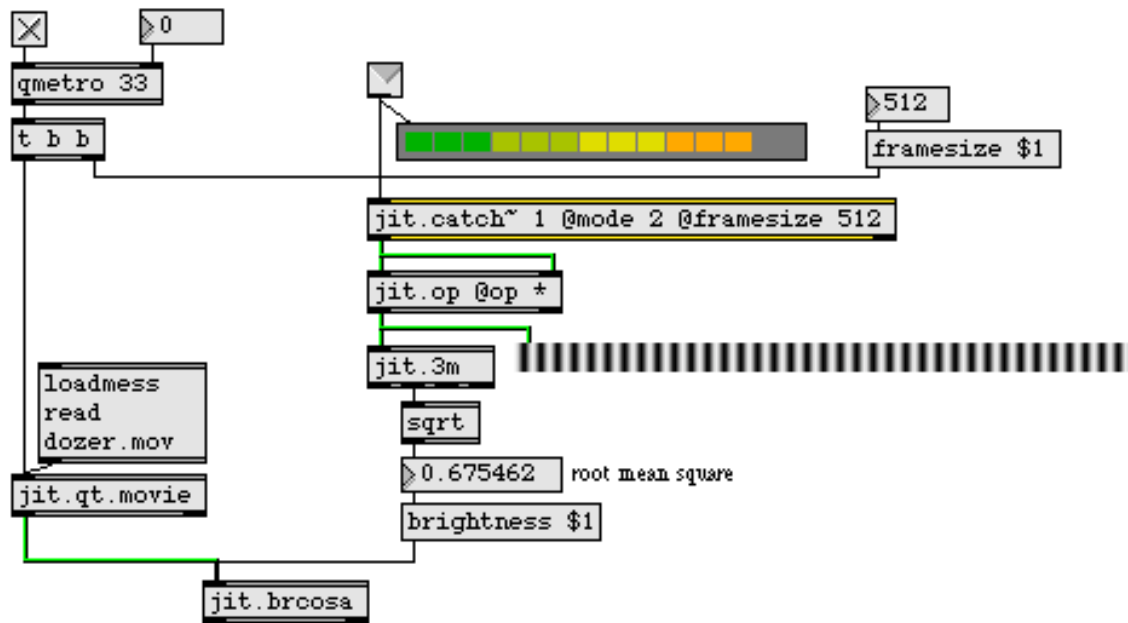
Since the two signals are multiplexed across different planes of the output matrices, we use the **jit.unpack** object to access the signals independently in the Jitter domain. In this example, each of the resulting single-plane matrices is sent to a **jit.graph** object. The **jit.graph** object takes one-dimensional data and expands it into a two-dimensional plot suitable for visualization. The low and high range of the plot can be set with the **rangehi** and **rangehi** attributes, respectively. By default the graph range is from -1.0 to 1.0.

These two **jit.graph** objects have been instantiated with a number of attribute arguments. First, the **out_name** attribute has been specified so that both objects render into a matrix named **jane**. Second, the right-hand **jit.graph** object (which will execute first) has its **clearit** attribute set to 1, whereas the left-hand **jit.graph** object has its **clearit** attribute set to 0. As you might expect, if the **clearit** attribute is set to 1 the matrix will be cleared before rendering the graph into it; otherwise the **jit.graph** object simply renders its graph on top of whatever is already in the matrix. To visualize two or more channels in a single matrix, then, we need to have the first **jit.graph** object clear the matrix (**clearit 1**); the rest of the **jit.graph** objects should have their **clearit** attribute set to 0.

The **jit.graph** object's **height** attribute specifies how many pixels high the rendered matrix should be. A **width** attribute is not necessary because the output matrix has the same width as the input matrix. The **frgb** attribute controls the color of the rendered line as four integer values representing the desired alpha, red, green, and blue values of the line. Finally, the **mode** attribute of **jit.graph** allows four different rendering systems: **mode 0** renders each cell as a point; **mode 1** connects the points into a line; **mode 2** shades the area between each point and the zero-axis; and the bipolar **mode 3** mirrors the shaded area about the zero-axis.

Frames

- Turn off the **qmetro** (by un-checking the **toggle box**) and close the basic visualization subpatch. Open the **patcher** named frame-based analysis and click the **toggle box** inside to activate the **qmetro** object.



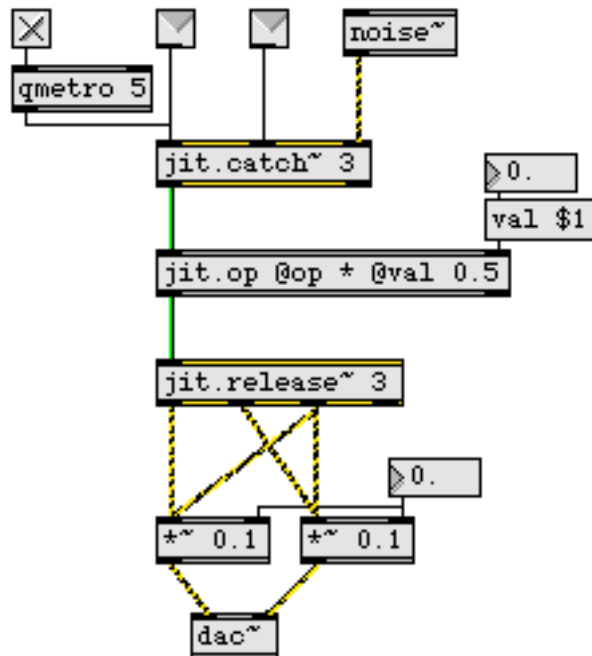
Performing a frame-based analysis of our audio.

The frame-based analysis subpatch shows an example of how one might use the **jit.catch~** object's ability to throw out all but the most recent data. The single-plane output of **jit.catch~** is being sent to a **jit.op** object which is multiplying it by itself, squaring each element of the input matrix. This is being sent to a **jit.3m** object, and the mean value of the matrix is then sent out the object's middle outlet to a **sqrt** object. The result is that we're calculating the *root mean square* (RMS) value of the signal – a standard way to measure signal strength. Using this value as the argument to the brightness attribute of a **jit.brcosa** object, our subpatch maps the amplitude of the audio signal to brightness of a video image.

Our frame-based analysis technique gives a good estimate of the average amplitude of the audio signal over the period of time immediately before the **jit.catch~** object received its most recent bang. The **peakamp~** object, which upon receiving a bang outputs the highest signal value reached since the last bang, can also be used to estimate the amplitude of an audio signal, but it has a couple of disadvantages compared to the **jit.catch~** technique presented here. First, the maximum value reached is only a single data point from the period of data. This maximum value provides no information about the other samples, and so may not be a good representative of that signal's strength over a period of time. On the other hand, it can be very expensive to consider every sample of data. The **qmetro** in

our frame-based analysis subpatch will output a bang roughly every 33 milliseconds. At a CD-quality sampling rate (44,100 samples per second), this represents about 1455 samples per bang. However, our **jit.catch~** technique examines only the final 512 samples, thereby striking a balance between accuracy and efficiency. The effect that this savings makes is amplified in situations where the analysis itself is more expensive, for example when performing an FFT analysis on the frame.

- Turn off the **qmetro** and close the frame-based analysis subpatch. Open the **patcher** named audio again and enter a 0 into the **number box** connected to the ***~** objects at the bottom of the patch; the generated signals will no longer being sent to your speakers. Close the audio subpatch and open the patcher named processing audio with jitter objects.



Using Jitter objects to process audio data, transforming the results back into signals.

The **jit.release~** object is the reverse of **jit.catch~**: multi-plane matrices are input into the object which then outputs multiple signals, one for each plane of the incoming matrix. The **jit.release~** object's latency attribute controls how much matrix data should be buffered before the data is output as signals. Given a Jitter network that is supplying a **jit.release~** object with data, the longer the latency the lower the probability that the buffer will underflow – that is, the *event-driven* Jitter network will not be able to supply enough data for the signal vectors that the **jit.release~** object must create at a constant (*signal-driven*) rate.

- Click on the **toggle box** into the subpatch to start the **qmetro** object. You should begin to hear our sine waves mixed with white noise (supplied by the **noise~** object). Change

the value in the **number box** attached to the **message box** labeled val \$1. Note the effect on the amplitude of the sounds coming from the patch.

The combination of **jit.catch~** and **jit.release~** allows processing of audio to be accomplished using Jitter objects. In this simple example a **jit.op** object sits between the **jit.catch~** and **jit.release~** objects, effectively multiplying all three channels of audio by the same gain. For a more involved example of using **jit.catch~** and **jit.release~** for the processing of audio, look at the *jit.forbidden-planet.pat* example, which does frame-based FFT processing in Jitter.

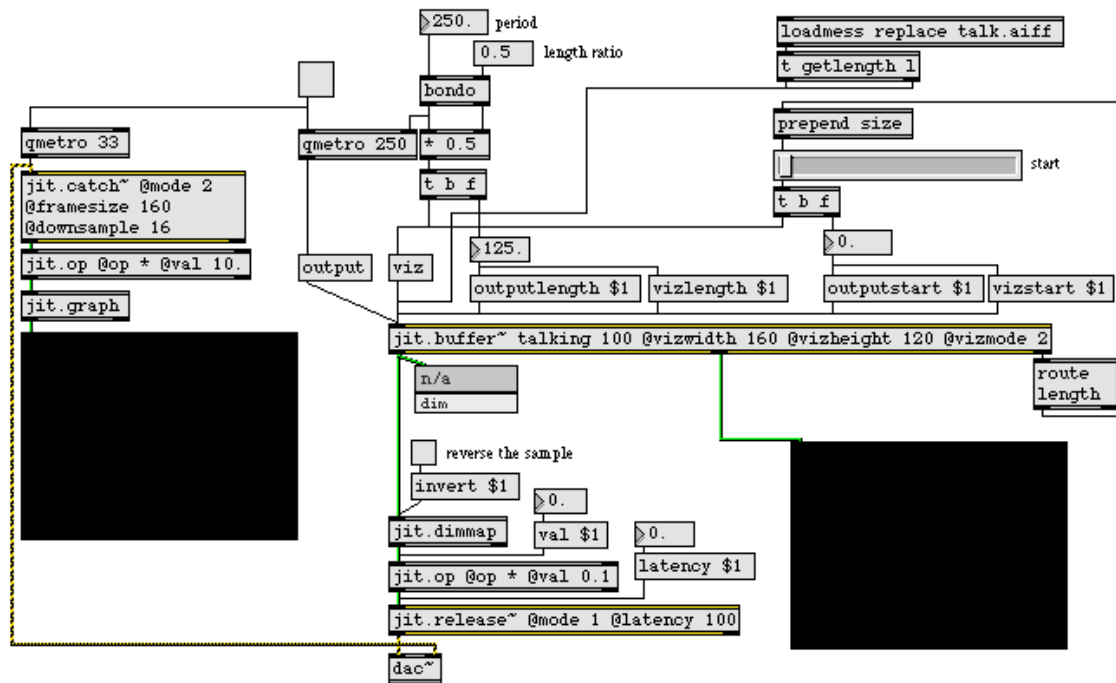
Varispeed

- Stop the **qmetro** and close the subpatch. Open the **patcher** named interpolation release~.

The **jit.release~** object can operate in one of two modes: in the standard mode 0 it expects that it will be supplied directly with samples at the audio rate – that is, for CD-quality audio the object will on average receive 44,100 float32 elements every second and it will place those samples directly into the signals. In mode 1, however, the object will interpolate in its internal buffer based on how many samples it has stored. If the playback time of samples stored is less than the length of the latency attribute, **jit.release~** will play through the samples more slowly. If the object stops receiving data altogether, playback will slide to a stop in a manner analogous to hitting a turntable platter's stop button. On the other hand, if there are more samples than needed in the internal buffer, **jit.release~** will play through the samples more quickly.

One can use this feature to generate sound directly from an event-driven network. The interpolating release~ subpatch shows a way for us to do this. The data that drives the **jit.release~** object in this subpatcher receives data from a **jit.buffer~** object, which allows us to extract data from audio samples in matrix form using messages to the MSP **buffer~** object. The **jit.buffer~** object is essentially a Jitter wrapper around a regular **buffer~** object; **jit.buffer~** will accept every message that **buffer~** accepts, allows the getting and setting of **buffer~** data in matrix form, and also provides some efficient functions for visualizing waveform data in two dimensions, which we use in this patch in the **jit.pwindow** object at right. At **loadbang** time this **jit.buffer~** object loaded the data in the sound file talk.aiff.

- Turn on the **toggle box** connected to the **qmetro** object in the subpatch. Play with the **number boxes** labeled *period* and *length ratio* and the **hslider** object labeled *start*.



Playing back interpolated audio data from a sound file in Jitter.

The rate of the **qmetro** determines how often data from the **jit.buffer~** object is sent down towards the **jit.release~** object. The construction at the top maintains a ratio between the period and the number of samples that are output from **jit.buffer~**. Experimenting with the start point, the length ratio, and the output period will give you a sense of the types of sounds that are possible in this mode of **jit.release~**.

Summary

In this Tutorial we were introduced to **jit.catch~**, **jit.graph**, **jit.release~**, and **jit.buffer~** as objects for storing, visualizing, outputting, and reading MSP signal data as Jitter matrices. The **jit.catch~** and **jit.release~** objects allow us to transform MSP signals into Jitter matrices at an event rate, and vice versa. The **jit.graph** object provides a number of ways to visualize single-dimension matrix data in a two-dimensional plot, making it ideal for the visualization of audio data. The **jit.buffer~** object acts in a similar manner to the MSP **buffer~** object, allowing us to load audio data directly into a Jitter matrix from a sound file.

Tutorial 49: Colorspaces

In order to generate and display matrices of video data in Jitter, we make some assumptions about how the digital image is represented. For many typical uses (as covered by earlier tutorials) we encode color images into 4-plane matrices of type char. These planes represent the alpha, red, green, and blue color channels of each cell in that matrix. This type of color representation (ARGB) is useful as it closely matches both the way we see color (through color receptors in our eyes tuned to red, green, and blue) and the way computer monitors, projectors, and televisions display it. *Tutorial 5: ARGB Color* examines the rationale behind this system and explains how you typically manipulate this data.

It's important to note, however, that ARGB is not the only way to represent color information in a digital form. This tutorial examines one of several different ways of representing color image information in Jitter matrices, along with a discussion of several alternatives available to us for different uses. Along the way we'll look at a simple, efficient way to texture video onto an OpenGL plane to take advantage of hardware accelerated post-processing of the video image.

Software Requirement: In order to use the `uyvy` colormode in Jitter under QuickTime version 6.5 and earlier, your media needs to be compressed with a codec that uses a YUV colorspace (discussed in depth below). QuickTime media compressed with codecs using an RGB colorspace (e.g. PICT files or Planar RGB video files) cannot be decompressed by **jit.qt.movie** in the `uyvy` colormode used in this Tutorial. QuickTime version 7 and later will allow you to work in the `uyvy` colormode regardless of the colorspace in which your media is encoded.

- Open the tutorial patch `49jColorspaces.pat` in the Jitter Tutorials folder.

At first glance, this patch looks very similar to the one we used in *Tutorial 12: Color Lookup Tables*. It reads a file into a **jit.qt.movie** object, sending the matrices out into a **jit.charmap** object, where we can alter the color mapping of the different planes in an arbitrary manner by creating a matrix (`themap`) that serves as a color lookup table. The processed matrix is then displayed.

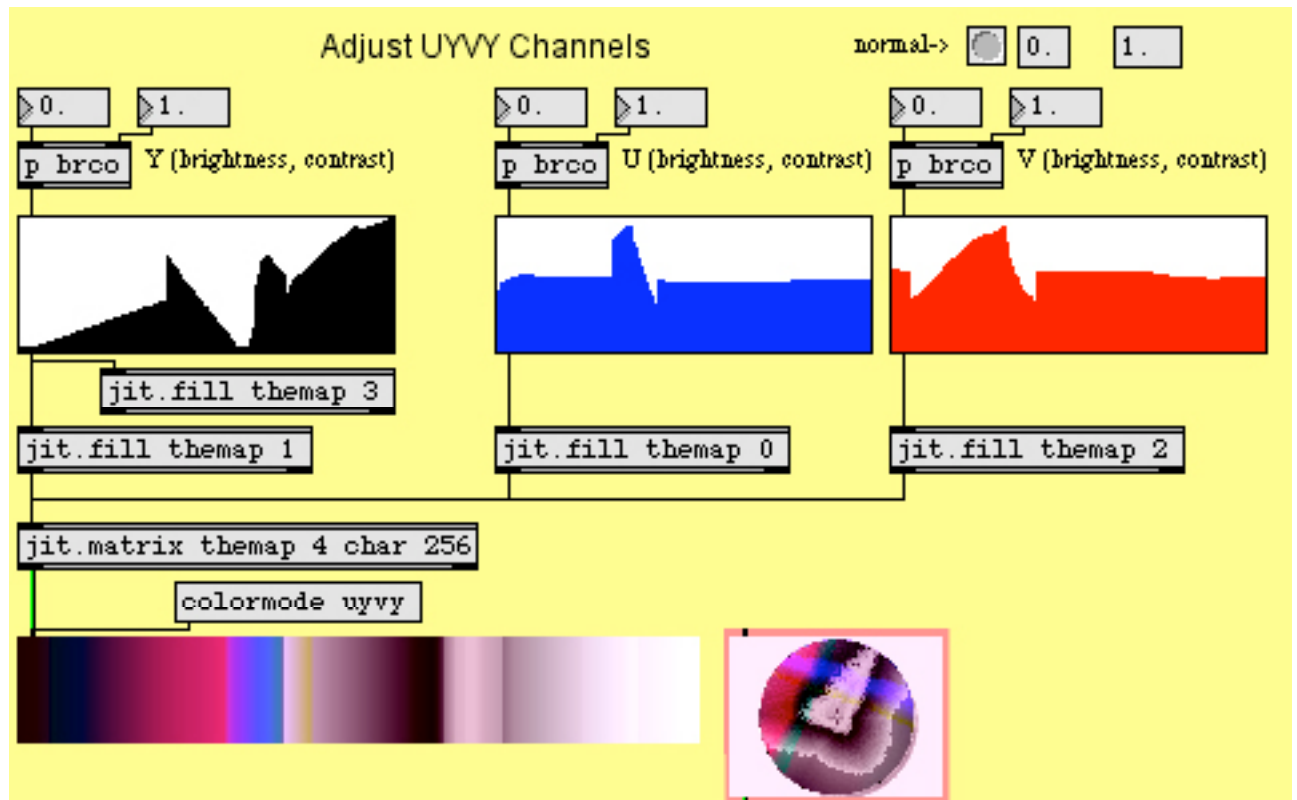
- Click the **message box** labeled `read colorwheel.jpg`. Click the **toggle** labeled *Display* connected the **qmetro** object at the top of the patch. You should see the colorwheel appear both in the **jit.pwindow** and in the window created by the **jit.window** object.

Though it isn't immediately obvious (yet), the image matrix in our patch is being generated and manipulated according to a different system of color than the ARGB mapping we're accustomed to. The **jit.qt.movie** object in this patch is transmitting matrices

using a colormap called `uyvy`. This means that the image processing chain is working with data in a different colorspace than we usually work in, called YUV 4:2:2. In addition to transmitting color according to a different coordinate system than we usually use (YUV instead of ARGB), this mode of transmission uses a technique called *chroma subsampling* to reduce (by half!) the amount of data transmitted for an image of a given size.

Color Lookup with a Twist

- Try manipulating the **multislider** objects at the right of the patch. Notice that they have an effect on how color is mapped in the image. You may get the sense that the three **multislider** objects correspond to the mapping of green, red, and blue in the matrix, respectively. Try zeroing the leftmost (black-on-white) **multislider** (i.e. set all of its values to 0). Notice that the image disappears. Click on the **button** labeled *normal* to put everything back. Try zeroing the other two **multisliders** in turn, then setting them to straight horizontal lines across the middle of their range.



Manipulating the color.

The YUV colorspace is a luminance/chrominance color system—it separates the luminosity of a given color from the chromatic information used to determine its hue. It stores the luminosity of a given pixel in a luminance channel (Y). The U channel is then created by subtracting the Y from the amount of blue in the color image. The V channel is created by subtracting the Y from the amount of red in the color image. The U and V

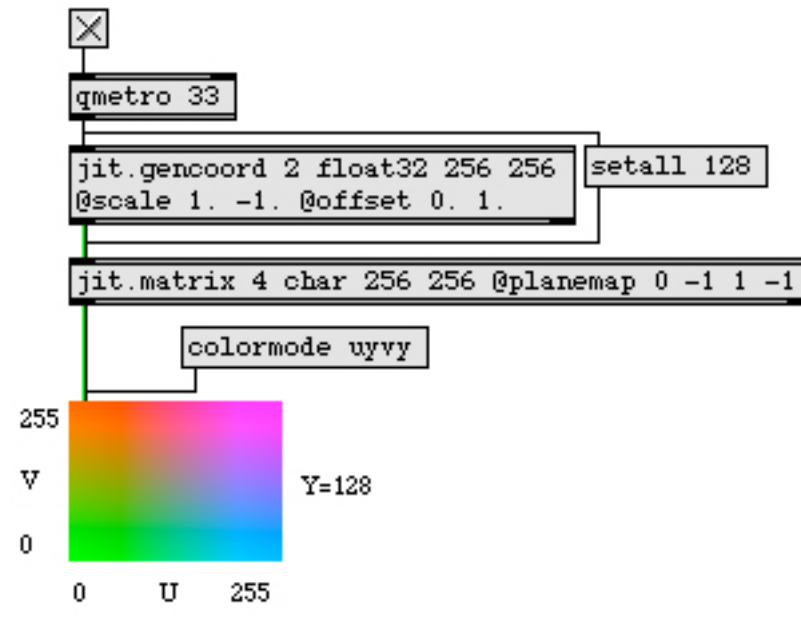
channels (representing chrominance) are then scaled by different factors. As a result, low values of U and V will expose shades of green, while a constant medium value of both will give a grayscale image. One can convert color values from RGB to YUV using the following formula:

$$Y = 0.299R + 0.587G + 0.114B$$

$$U = 0.492(B - Y)$$

$$V = 0.877(R - Y)$$

Note that the U and V components in this color space are usually signed (i.e. they can be negative numbers if the luminosity exceeds the blue or red amount, as it does with hues such as orange, green, and cyan). Jitter matrices store unsigned char data, so the U and V values are represented in the range of 0-255, with 128 as the center point of the chromatic space.



The YUV color space with a constant midrange Y value.

The specific implementation of the YUV colorspace used in our Tutorial patch is called YUV 4:2:2. Jitter objects that need to interpret matrix data as video (e.g. **jit.qt.movie**, **jit.pwindow**, etc.) can generate and display this colorspace when their `colormode` attribute is set to `uyvy`. This `colormode` uses something called *chroma subsampling* to store two adjacent color pixels into a single cell (referred to as a “macro-pixel”). Because our eyes are more attuned to fine gradations in luminosity than in color, this is an efficient way to perform data reduction on an image, in effect cutting in half the amount of information needed to convey the color with reasonable accuracy. In this system, each cell in a Jitter matrix contains four planes that represent two horizontally adjacent pixels: plane 0 contains the

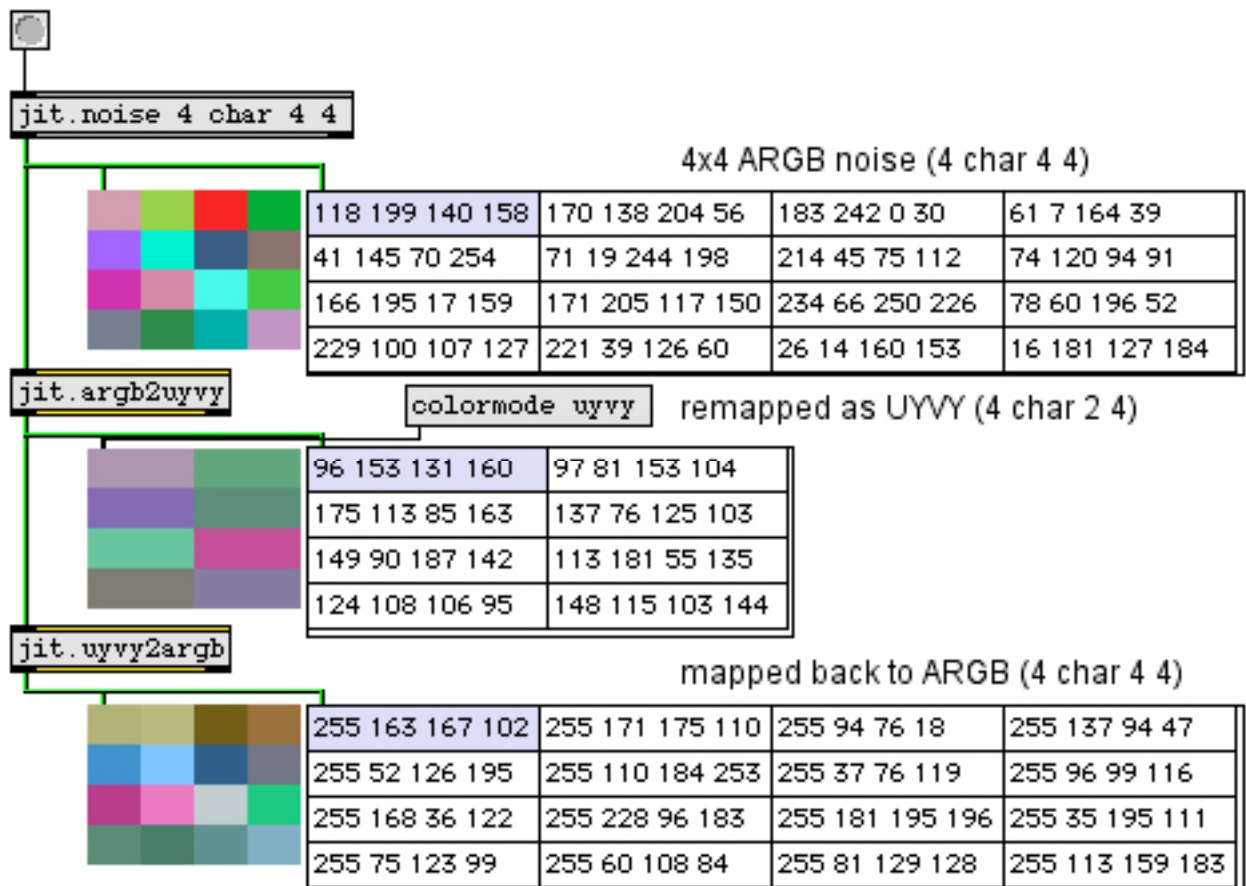
U value for both pixels; plane 1 contains the *Y* value for the first pixel; plane 2 contains the *V* value for both pixels; plane 3 contains the *Y* value for the second pixel. The ordering of the planes (*uyvy*) means that we can alter the luminosity of the image by adjusting planes 1 and 3 (for alternating pixel columns), but we can change the chrominance of pixels only in pairs (by adjusting planes 0 and 2).

Historical Note: Luminance-chrominance color encoding (where the luminosity of the image is transmitted separately from the hue or chrominance of the image) has its roots in the history of color television broadcasting. When color TV's were introduced in the United States in 1953, it was necessary to provide a means for television consumers with monochrome (black-and-white) TV sets to still be able to watch the programming. As a result, it was decided to simply add color information (in the form of a subcarrier) to the original broadcast signal, which already contained the luminosity of the image as grayscale. The result was called YIQ (for luminosity-intermodulation-quadrature), and is the colorspace used in broadcast NTSC color television. The equivalent on PAL television systems is the YUV colorspace under discussion here.

The following illustration shows how the conversion from ARGB to UYVY is handled in Jitter. Our **jit.qt.movie** object performs this translation for us when necessary (see the box below), but the **jit.rgb2uyvy** and **jit.uvyv2rgb** objects will convert any matrix between colorspace. Note that the alpha channel is lost in the conversion and that chromatic information is averaged across pairs of horizontal cells in the ARGB original, creating a slight loss in color information.

Because the UYVY color system uses a macro-pixel, each cell in the matrix actually represents two horizontally adjacent pixels in the image; for example, a 320x240 pixel image becomes a 160x240 cell matrix when output as UYVY. When processing these matrices, it's important to keep that in mind, as Jitter objects that process matrices based on *spatial* information (e.g. those that do scaling, rotation, convolution, etc.) will treat these pairs of pixels as a single unit. When working with these types of processes it may be easier to work in a *colormode* that uses a full resolution pixel, e.g. ARGB or AYUV (a full-resolution YUV colorspace also supported by Jitter).

The matrix size shrinks in half (along the horizontal axis) upon conversion into a uyvy matrix, and is doubled when converting back to argb (note that a new, empty alpha channel is also created).



A 4x4 grid of random values converted from ARGB to UYVY and back again.

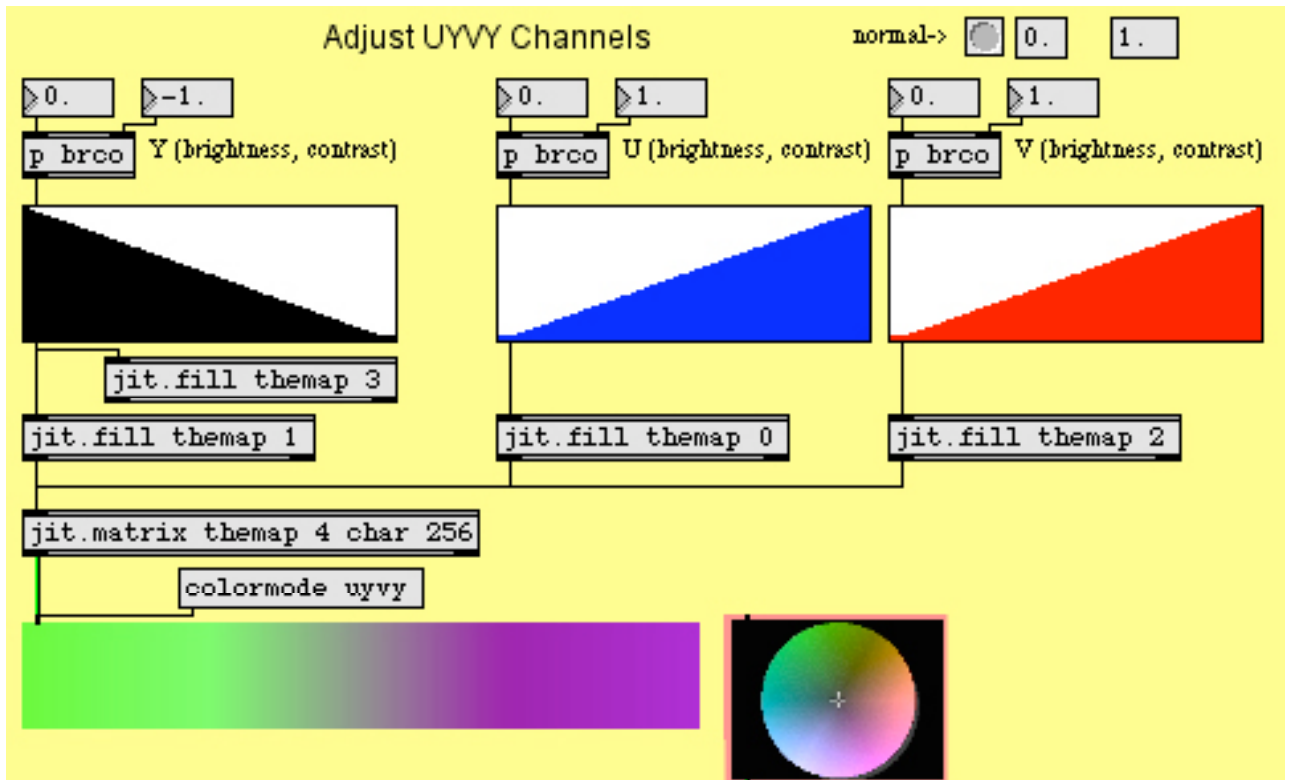
Many commercial video codecs use YUV 4:2:2 (or similar chroma subsampling systems such as 4:1:1 and 4:2:0) as their native video format. As a result, **jit.qt.movie** can decompress these files faster in the uyvy colormode than when outputting matrices in the normal ARGB format. The matrices thus generated are also half the size, giving a performance increase to any Jitter patches that can take advantage of this system. Codecs such as Photo-JPEG, DVCPRO, and NTSC DV all use some form of subsampled YUV codec as their native color format.

With this in mind, we can understand the processing going on in our patch. The **jit.qt.movie** sends matrices in the uyvy 4-plane char format to the **jit.charmap**, which processes the data and sends it onwards. The **jit.fpsgui** tells us that the dim of the matrices being sent to it are 160x240, which makes sense now that we understand the macro-pixel data reduction that accompanies the switch in colormode. In addition, we can now see why the

matrix themap has one **jit.fill** object feeding both planes 1 and 3; these correspond to the two Y values in the macro-pixel, which we want to share the same lookup table.

Color tinting and saturation

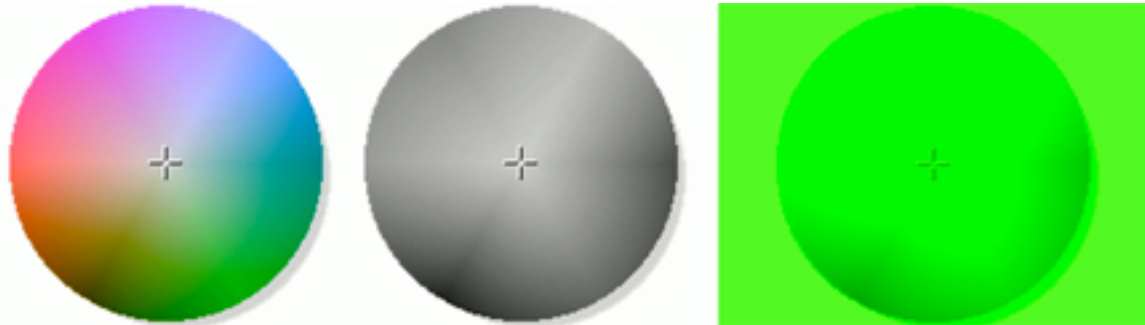
- Set the **multislider** objects to their normal curve by clicking the **button** object labeled *normal*. Above the patcher objects labeled *brco*, adjust the number box controlling the contrast for the Y channel until it reads -1.



Inverted Y color table.

Because the luminosity is separate from the chrominance in our new colorspace, it is a simple matter to invert the brightness of pixels without affecting their hue. This would be a more involved procedure if working in ARGB.

- Normalize the **multislider** objects again, and adjust the **number box** objects for the *U* and *V* channels' contrast. Try setting them both to -1, then to 0. Normalize their contrast settings and set the brightness values both to -1.



Different U and V settings.

By inverting the *U* and *V* channels' color lookup, we perform a 180-degree hue rotation on the image. Setting both of these channels to a constant value *desaturates* the image so that it appears at a constant chroma, or hue, according to the Cartesian space shown earlier in this tutorial. Values of 0 will desaturate the image to greyscale; values of -1 will make the entire image appear green.

- Do some freehand drawing in the **multislider** objects. Try to get a feel for how different ranges of the char range results in different effects on different channels.

In addition to YUV 4:2:2 (colormode uyvy) and ARGB (colormode argb, the default for most objects), Jitter objects exist that generate matrices in a number of other colorspaces. Examples include `grgb` (a chroma subsampled RGB colorspace similar to uyvy), `ayuv` (a full-resolution YUV colorspace with an alpha channel), `luma` (a 1-plane char grayscale format), and `ahsl` (alpha, hue, saturation, luminosity). Conversion objects are typically named `jit.x2y`, e.g. `jit.argb2uyvy`. In addition, the `jit.colorsapce` object supports translation to and from a variety of 4-plane char colorspaces (including approximations of floating-point colorspaces such as $L^*a^*b^*$). A good place to begin for more information on colorspaces (and the numerical representation of color in general) is the Wikipedia article on the subject:

http://en.wikipedia.org/wiki/Color_space

Videoplane

- Notice that the output of the **jit.pwindow** in our patch doesn't go directly to a **jit.window** object. Instead, it goes to an object called **jit.gl.videoplane**. Click on the **message box** labeled `read track1.mov`. The processed image should switch to a movie. Normalize the color lookup tables on the right by clicking the **button** labeled *normal*.

The **jit.gl.videoplane** object textures the Jitter matrix sent into its inlet onto a plane in an OpenGL drawing context. Our drawing context (colorspace) is being driven by the **jit.gl.render** object at the top of the patch, and is viewable through the **jit.window** object's window. If you need to review the basics of OpenGL rendering in Jitter, a look at *Tutorial 30: Drawing 3D text* will fill you in on the basics of creating a rendering system. One thing of note is that the `ortho` attribute of **jit.gl.render**, when set to 2, renders our scene in an orthographic projection (i.e. there is no sense of depth). Also note that the **jit.gl.videoplane** object, not the **jit.window**, needs to be told to interpret texture matrices as `uyvy` through its `colormode` attribute. In our patch we really aren't using OpenGL for 3D modeling; but we are taking advantage of some features of hardware accelerated texture mapping.

- Click on the **message box** containing the text `dim 16 16` (next to the **jit.pwindow** object).

Notice that the **jit.pwindow** object shows a massively downsampled and pixelated image (It's actually processed as an 8x16 matrix since we're still in `uyvy` mode).

The **jit.window** object, however, shows an image where the pixels are smoothly interpolated into one another (the effect is similar to upsampling in a **jit.matrix** object with the `interp` attribute set to 1).



Hardware interpolation of a small matrix applied as a texture.

This interpolation is occurring on the hardware Graphics Processing Unit (GPU), and is one of the many advantages to using OpenGL to display video, as it causes no performance penalty on the main CPU of our computer.

- Click on the **message box** containing the text `dim 320 240`. This will set the size of the matrices back to the normal resolution we've been using: 320x240 pixels, output as 160x240 cells per matrix because of the `colormode`. On the right of the patch, click on the **toggle** attached to the **message box** labeled `fullscreen $1` (alternately, hit the ESC key on your keyboard).

When you send a **jit.window** object into fullscreen mode, the **jit.gl.videoplane** upsamples the texture even further, giving you the smoothest possible interpolation for your display.

- Hit the ESC key to trigger the **toggle** again, taking the **jit.window** out of fullscreen.

Videoplane Post-Processing

- Adjust the red, green, and blue **number box** objects attached to the **pak** object connected to the **jit.gl.videoplane** object.

Note that even though the applied texture is mapped in YUV colorspace, the **jit.gl.videoplane** responds to color attributes in floating-point RGBA. Manipulate the **number box** labeled *rotate*.



Some examples of GPU processing on a videotexture.

We can see that the color and rotate attributes (as well as scale, blend_enable, etc.) of most OpenGL objects also apply to **jit.gl.videoplane**. As a result, **jit.gl.videoplane** is an incredibly useful object for video processing, as it allows us to apply processing to the image directly on the GPU.

Summary

The **jit.qt.movie** object can output matrices in a number of colorspace beyond ARGB. The YUV 4:2:2 colorspace can be used by setting the colormode attribute of the objects that support it (**jit.qt.movie**, **jit.pwindow** and **jit.window**) to uyvy. The uyvy colormode has the advantage of using a macro-pixel chroma subsampling to cut the data rate in half, allowing for matrices to be processed faster in the Jitter matrix processing chain. Since the data output in this colorspace is still 4 planes of char information, standard objects such as **jit.charmap** can be used to manipulate the matrix, albeit with different results.

The **jit.gl.videoplane** object accepts matrices (including uyvy matrices) as textures that are then mapped onto a plane in the OpenGL drawing context named by the object. GPU accelerated processing of the image can therefore be done directly on the plane, including color tinting, blending, spatial transformation, etc. In *Tutorial 52: Shaders* we'll look at ways to apply entire processing algorithms to objects in the drawing context, further expanding the possibilities of using the GPU for processing in Jitter.

Tutorial 50: Procedural Texturing & Modeling

In this Tutorial we will be examining different operations that can be used to construct a procedural model of a texture or some form of geometric data.

Procedural techniques are a powerful way of defining some aspect of a computer-generated model through algorithms and/or mathematical functions. In contrast to using pre-existing data such as static images or photographs, procedural models can generate visual complexity of arbitrary resolution and infinite variation. In conjunction with parametric controls, such models can be used to build a flexible interface for controlling complex behaviors and capturing a special effect.

Jitter provides a comprehensive set of basis functions and generators that are exposed through the **jit.bfg** object. Each function performs a point-wise operation in n -dimensional space whose evaluation is independent of neighboring results. This means that these operations can be performed on any number of dimensions, across any coordinate, without any need of referencing existing calculations. In addition, since they all share a common interface, these objects can be combined together and evaluated in a function graph by cross-referencing several **jit.bfg** objects.

There are several categories of functions, each of which are characterized by a different intended use. These categories include **fractal**, **noise**, **filter**, **transfer**, and **distance** operations. Functions contained in these folders can be passed by name to **jit.bfg** either fully qualified (category.classname) or relaxed (classname).

Before looking at these categories in detail, we'll first explore the general interface of **jit.bfg** and show how to create different types of procedural functions and fill a **jit.matrix** with our results.

jit.bfg

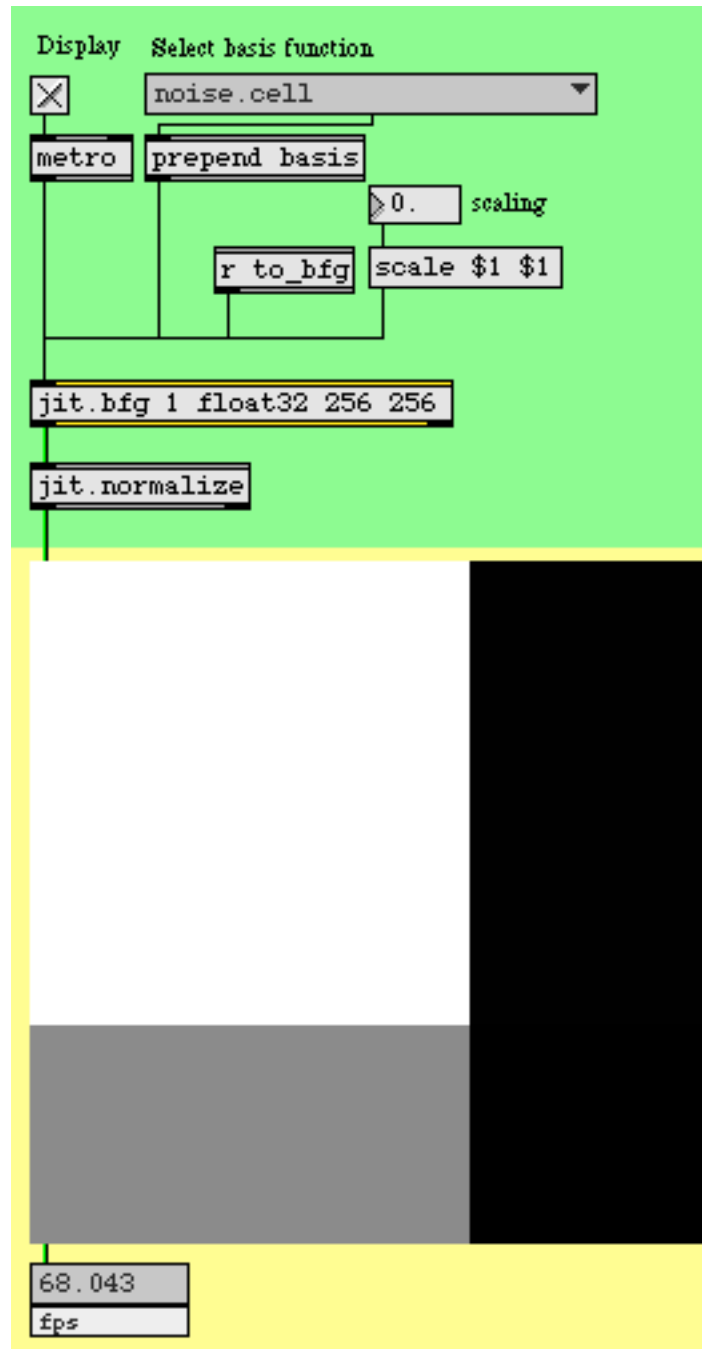
- Open the tutorial patch *50jProcedural.pat* in the Jitter Tutorial folder.

Once the patch loads, take a look at the different objects being used. Notice that we have a **metro** object attached to the **jit.bfg** object. Once activated, a bang message will notify **jit.bfg** to evaluate and output a Jitter matrix just like most other Jitter objects. In this example, **jit.bfg** has been setup to generate a single plane matrix of type float32 and size 128x128.

- Click on the **toggle box** connected to the **metro** object to begin sending bang messages to **jit.bfg**.

Notice that the **jit.pwindow** object remains solid black in color! Since **jit.bfg** has not been told what basis function evaluate, **jit.bfg** is not outputting a matrix and **jit.pwindow** remains unchanged.

- Select the noise.cell basis function from the list in the **jit.ubumenu** object.



An evaluated basis function.

Now that **jit.bfg** has been given a function to evaluate, we can see the results of its calculation in **jit.pwindow**. Internally, **jit.bfg** is generating a series of Cartesian coordinates that it passes to the indicated basis function during its evaluation.

If we wanted to, we could adjust these coordinates and have **jit.bfg** perform the evaluation over a different domain.

- Change the value of the **number box** connected to the scale **message box**.

Notice that the results of **jit.bfg** change as we adjust the domain.

- Select the distance.euclidean basis function from the list in the **jit.ubumenu** object.
- Again change the value of the **number box** connected to the scale **message box**.

Notice that positive values in the scale **number box** have little effect on the results being shown in **jit.pwindow**, whereas negative values flip the image components from white to black. What is going on? Distance should always be positive and increase outward from the origin, right?

jit.normalize

The output of **jit.bfg** goes into a **jit.normalize** object connected to the **jit.pwindow**. This object will examine an incoming matrix and scale the minimum and maximum values into a normalized range of 0-1.

When we changed the scale values being sent to **jit.bfg** for evaluating the distance.euclidean function from positive to negative, the highest and lowest values that were being outputted from **jit.bfg** switched as we crossed over the origin. Since **jit.normalize** always scales the input matrix maximum to 1 and the minimum to 0, our colors flipped.

Since the output range of **jit.bfg** may yield extremely large results, especially when evaluating unbounded functions such as fractals, we need to normalize our output in order to map the results for display.

Basis Categories

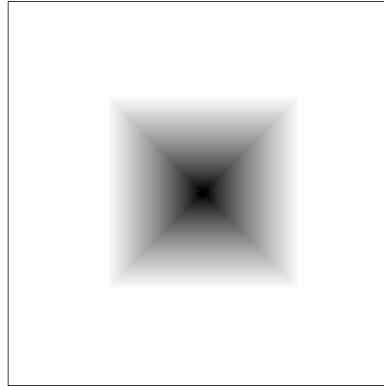
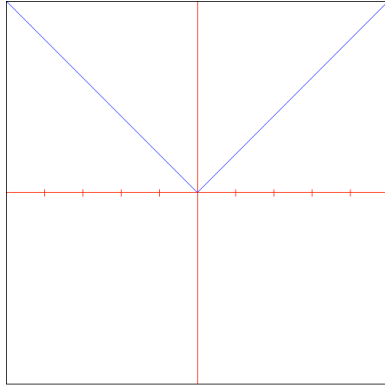
Now that we are familiar with the basic interface for setting up **jit.bfg** and specifying a basis function to evaluate, let's examine the contents of each function category.

Distance Functions

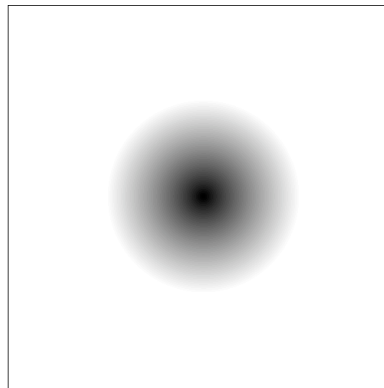
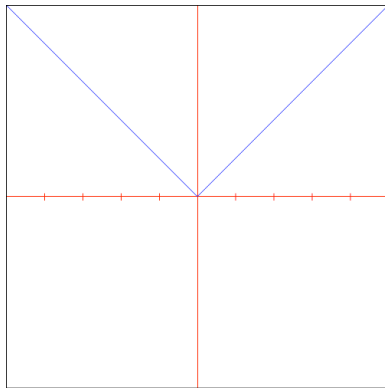
The functions in the distance category each define a unique metric for determining the positional difference from a given point to the global origin.

Descriptions of each of these functions are provided in the following list.

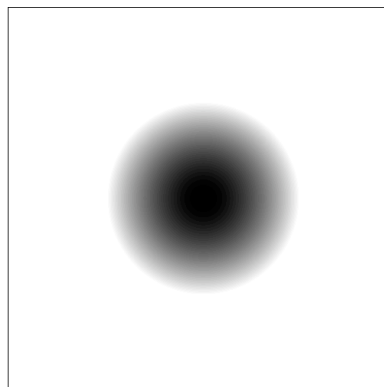
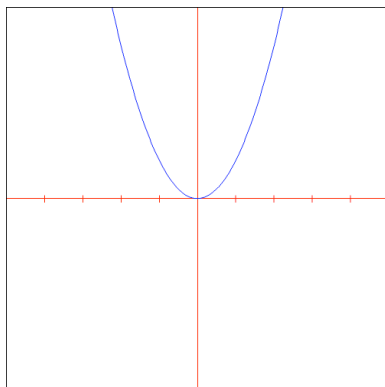
- `chebychev`: Absolute maximum difference between two points.



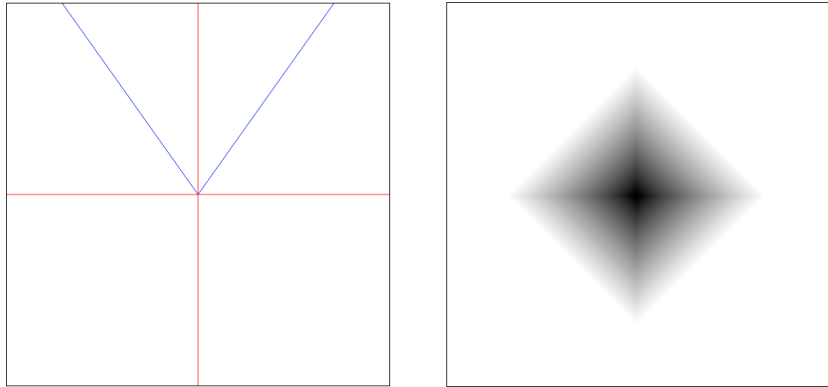
- `euclidean`: True straight line distance in Euclidean space.



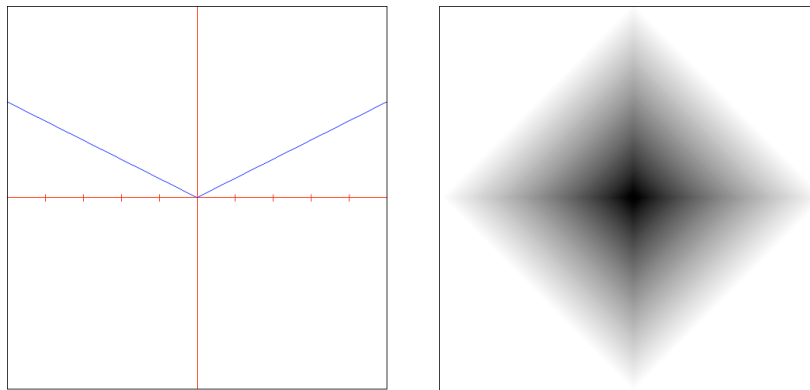
- `euclidean.squared`: Squared Euclidean distance.



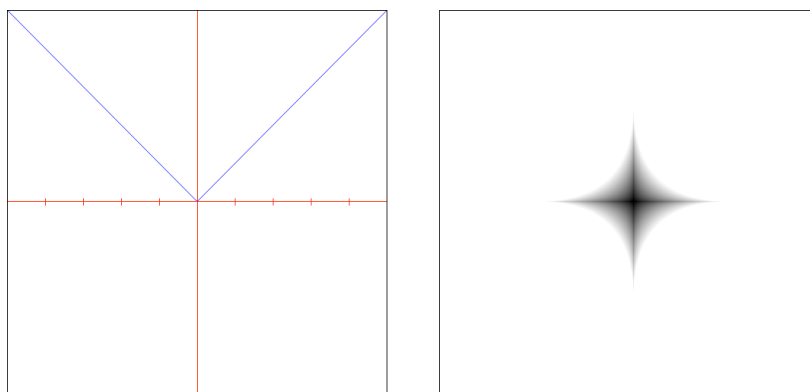
- `manhattan`: Rectilinear distance measured along axes at right angles.



- `manhattan.radial`: Manhattan distance with radius fall-off control.



- `minkovsky`: Exponentially controlled distance.



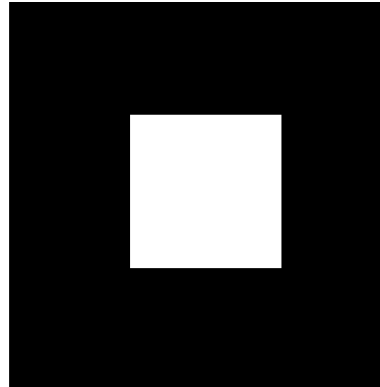
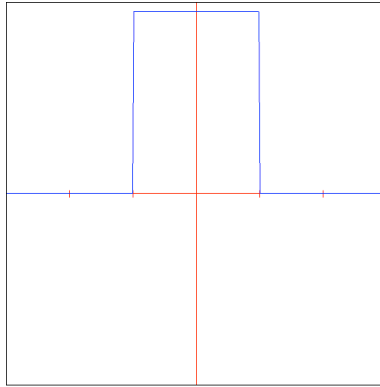
The `noise.voronoi` object requires one of these distance objects to be specified as part of its evaluation.

Filter Functions

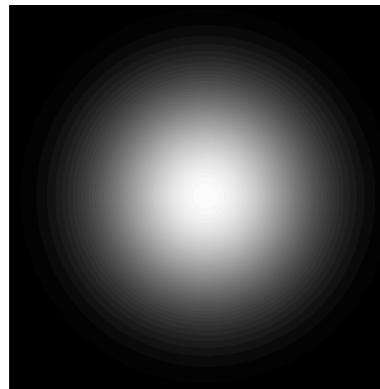
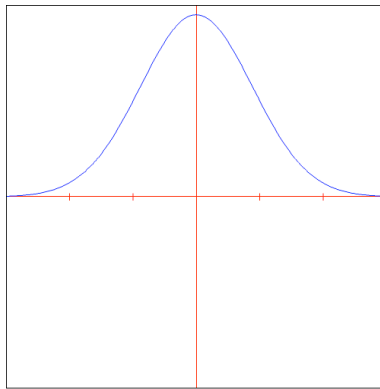
The filter category contains signal processing filters which can be used to perform image sampling and reconstruction or to create pre-computed kernels for a general convolution.

Descriptions of each of these functions are provided in the following list.

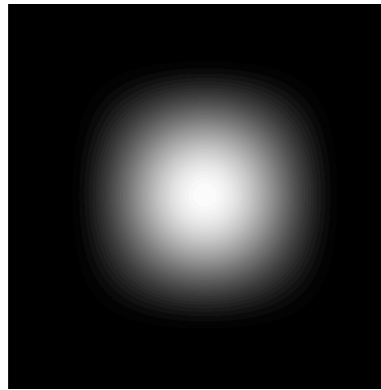
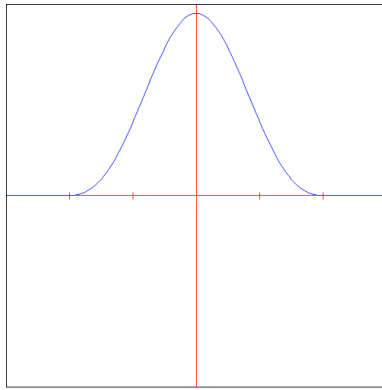
- **box:** Sums all samples in the filter area with equal weight.



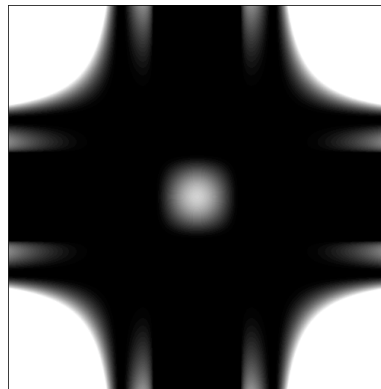
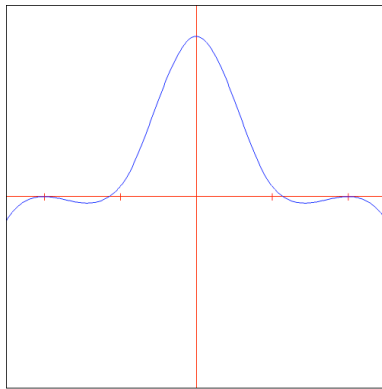
- **gaussian:** Weights samples in the filter area using a bell curve.



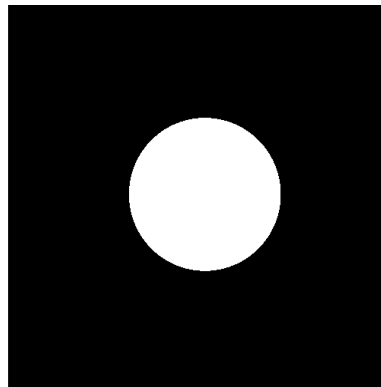
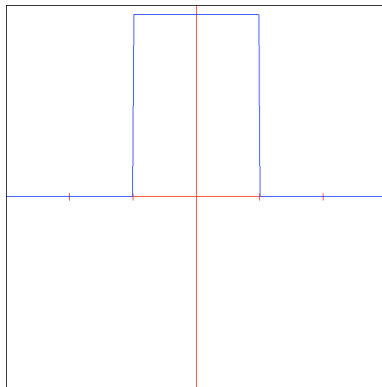
- lanczossinc: Weights samples using a steep windowed sinc curve.



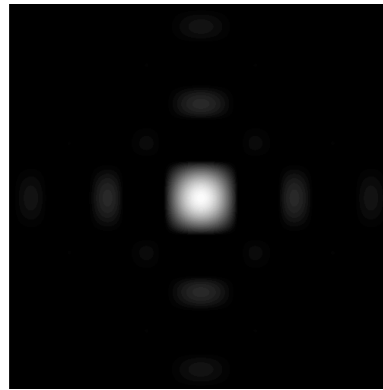
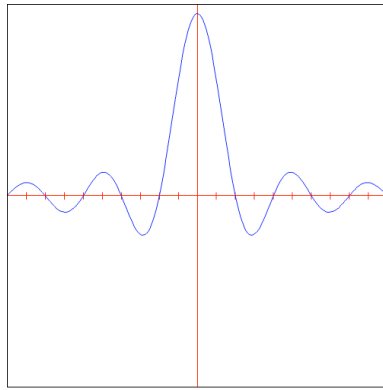
- mitchell: Weights samples using a controllable cubic polynomial.



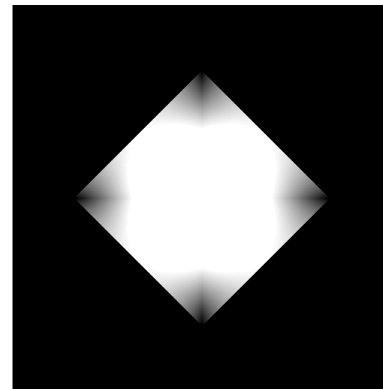
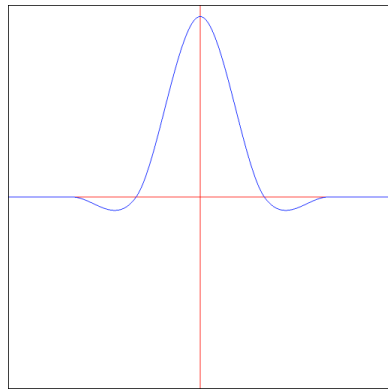
- disk: Sums all samples inside the filter's radius with equal weight.



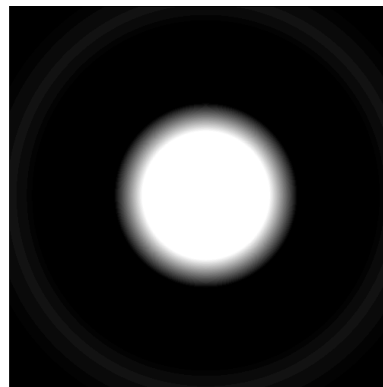
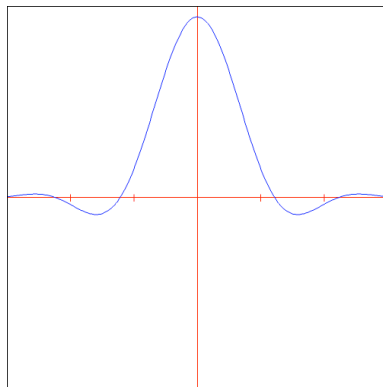
- sinc: Weights samples using an un-windowed sinc curve.



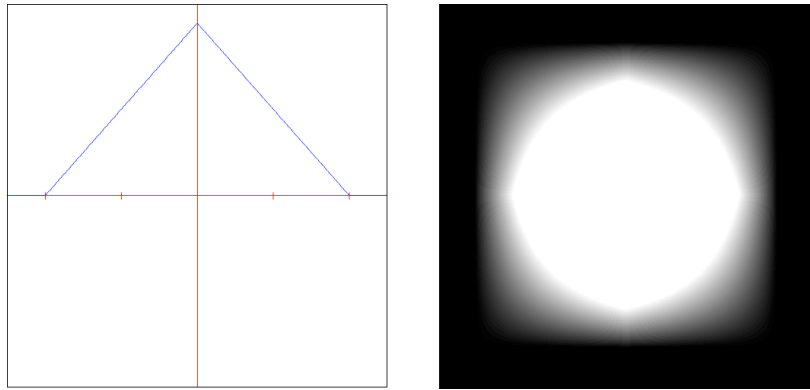
- catmullrom: Weights samples using a Catmull-Rom cubic polynomial.



- bessel: Weights samples with a linear phase response.



- triangle: Weights samples in the filter area using a pyramid.



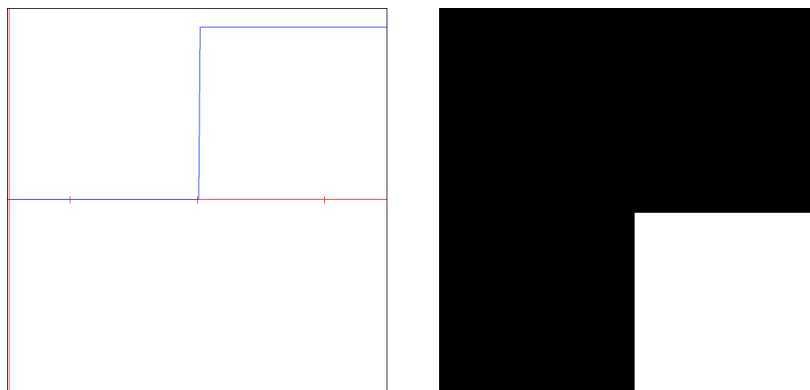
These objects are used as parameters to both the `noise.value.convolution` and the `noise.sparse.convolution` objects, which expect to be given a filter object as part of their evaluation.

Transfer Functions

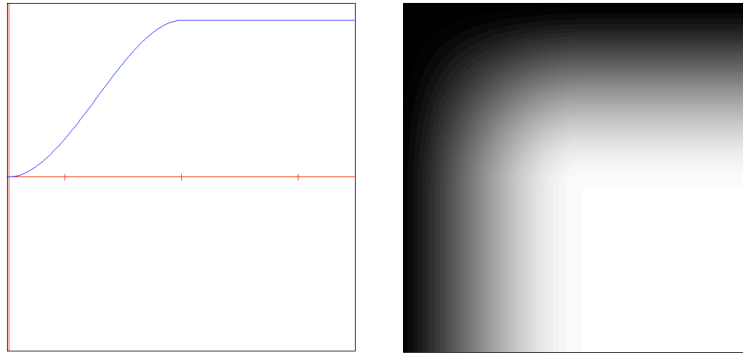
Functions that map input to a different output are contained in the transfer category. Most of these functions operate only on a single dimension within the unit interval 0-1.

A brief description of these functions is contained in the list below.

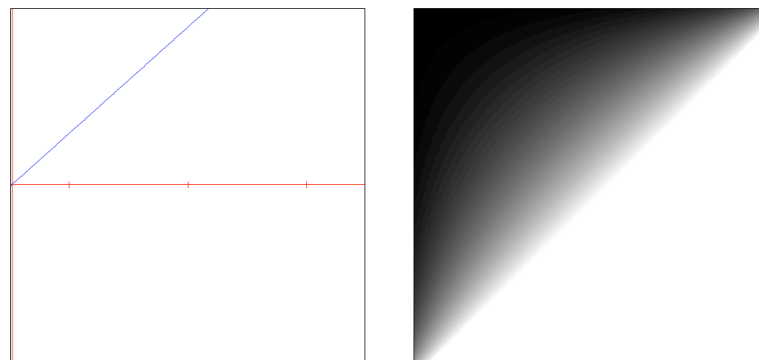
- step: Always 1 if given value is less than threshold.



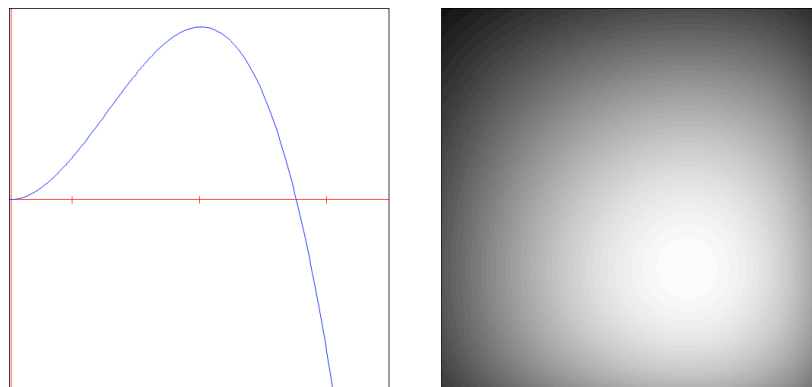
- smoothstep: Step function with cubic smoothing at boundaries.



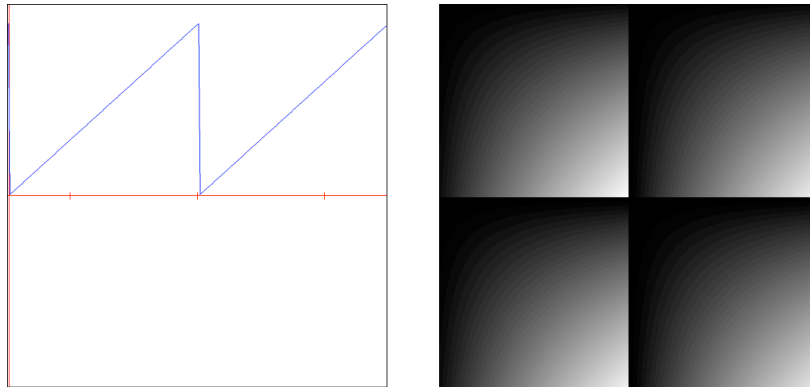
- bias: Polynomial similar to gamma but remapped to unit interval.



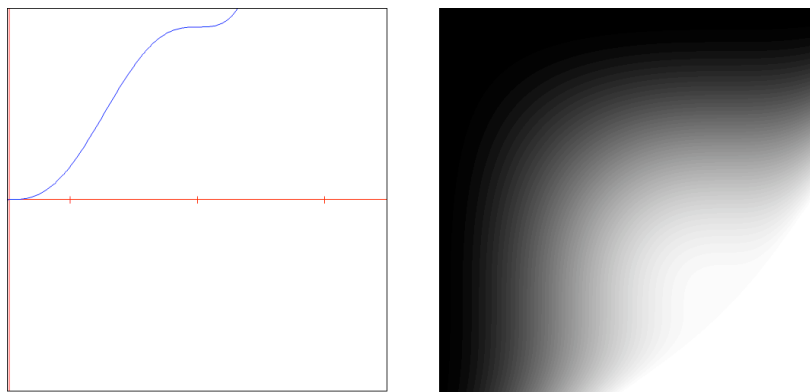
- cubic: Generic 3rd order polynomial with controllable coefficients.



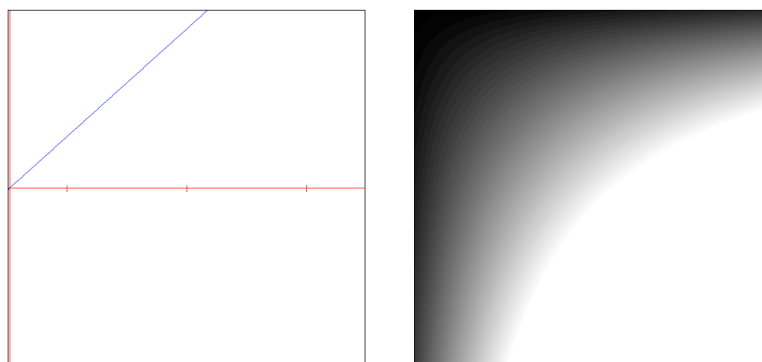
- saw: Periodic triangle pulse train.



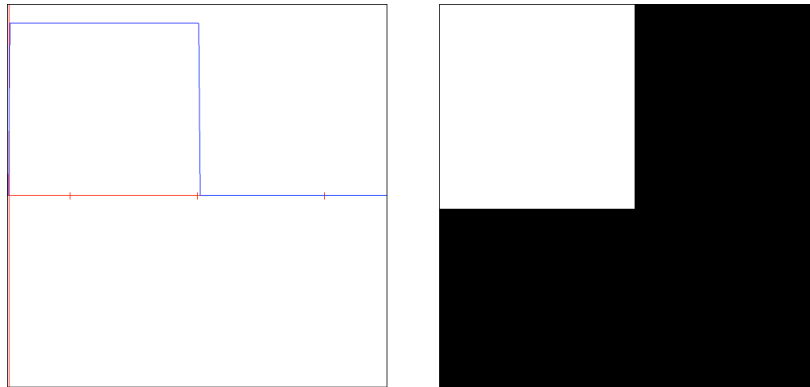
- quintic: Generic 5th order polynomial with controllable coefficients.



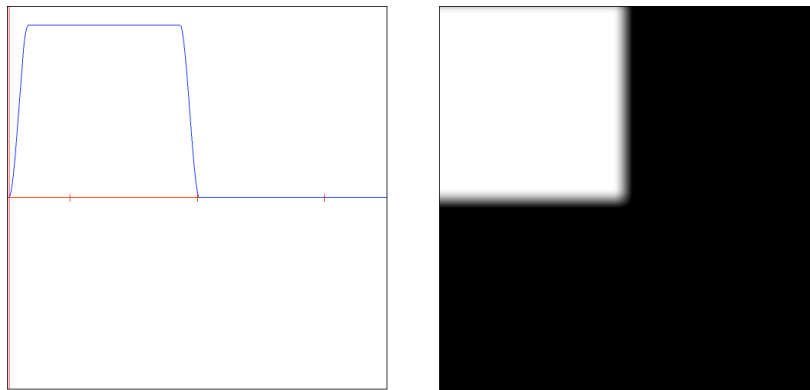
- gain: S-Shaped polynomial evaluated inside unit interval. Note: the default settings will result in a linear curve instead of the descriptive S-curve shape.



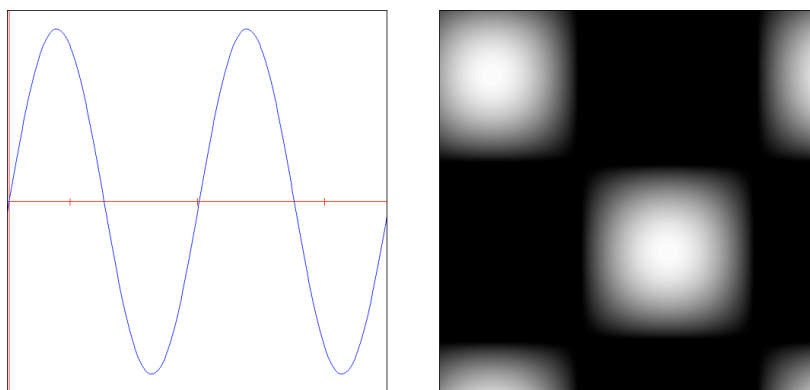
- pulse: Periodic step function.



- smoothpulse: Periodic step function with cubic smoothing at boundaries.



- sine: Periodic sinusoidal curve.



- linear: Linear function across unit interval.
- solarize: Scales given value if threshold is exceeded.

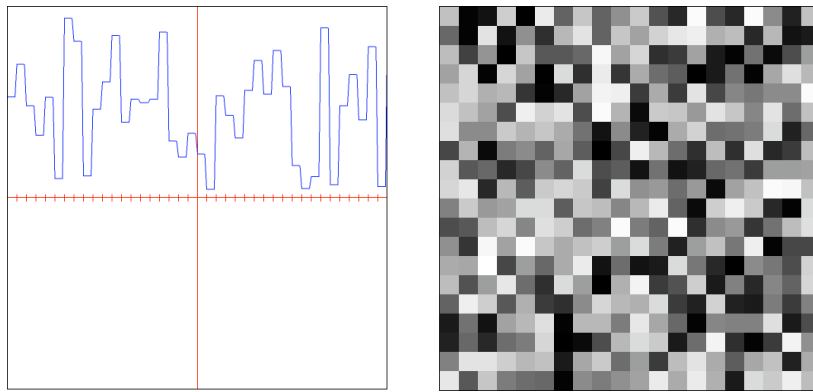
These transfer functions can be used inside of several of the noise objects to change their smoothing function and/or alter their output.

Noise Functions

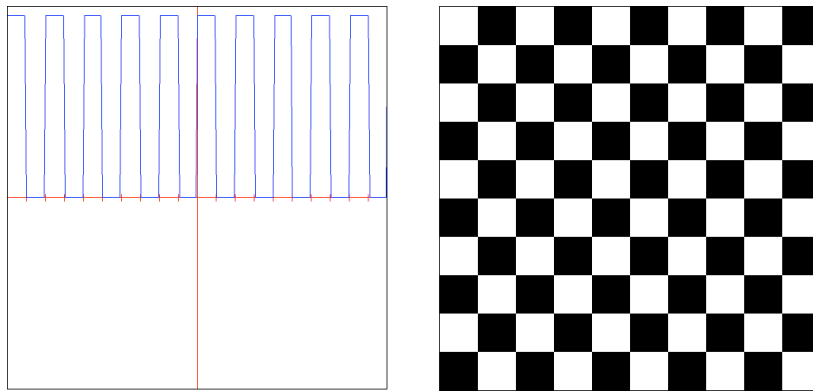
Deterministic stochastic patterns (aka pseudo-random coherent noise functions) are the cornerstone of nearly every procedural model. They allow a controllable amount of complexity to be created by adding visual detail.

A brief description of these functions is contained in the list below.

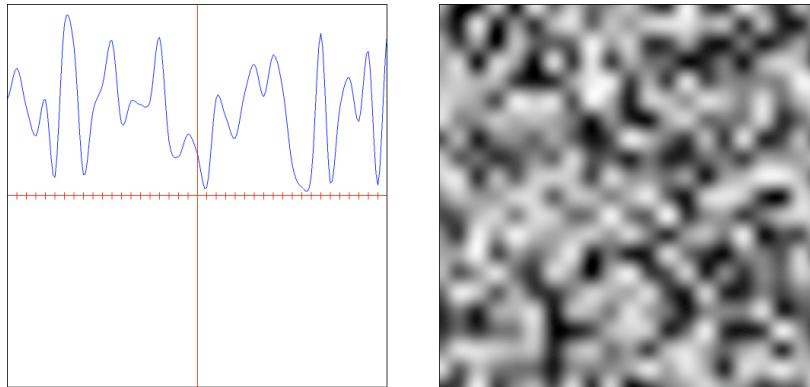
- cellnoise: Coherent blocky noise.



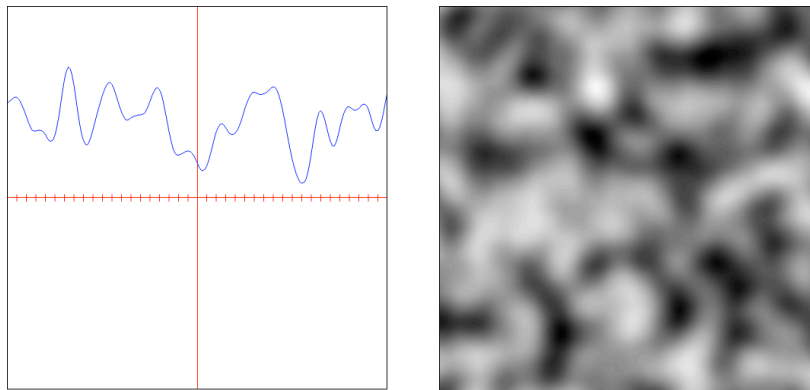
- checker: Periodic checker squares.



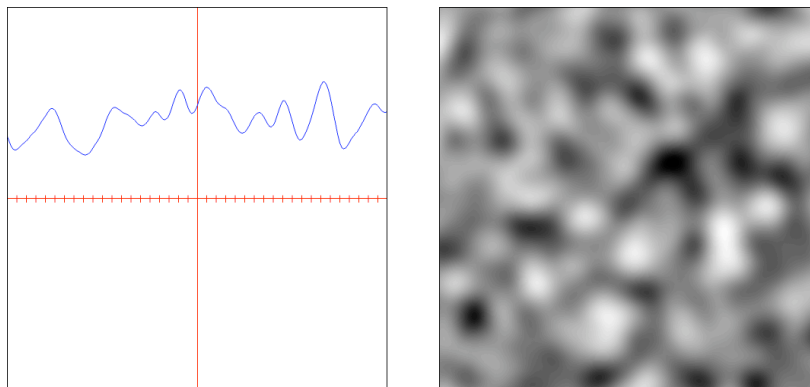
- `value.cubicspline`: Polynomial smoothed pseudo-random values.



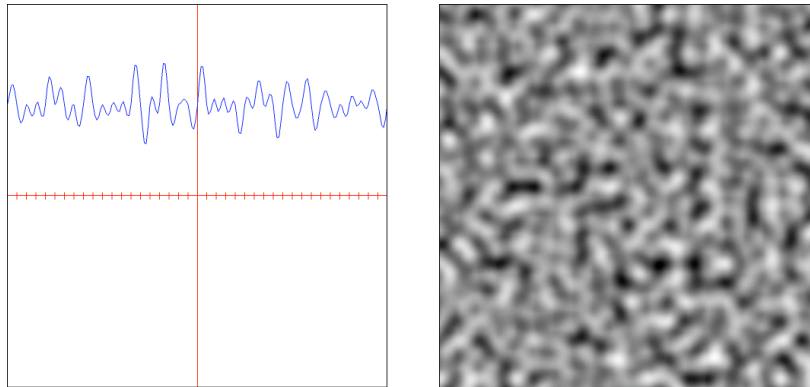
- `value.convolution`: Convolution filtered pseudo-random values.



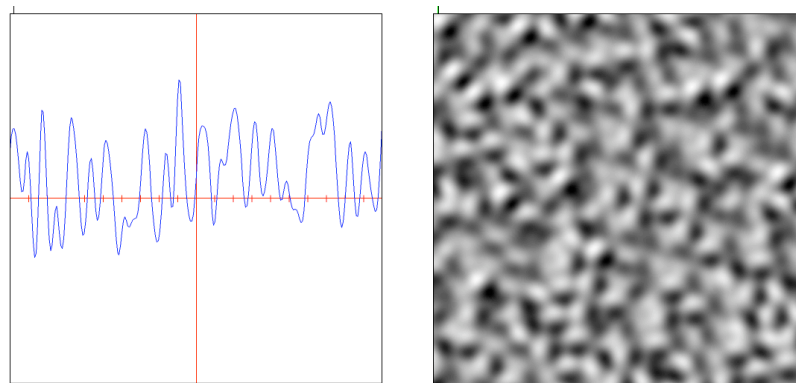
- `sparse.convolution`: Convolution filtered pseudo-random feature points.



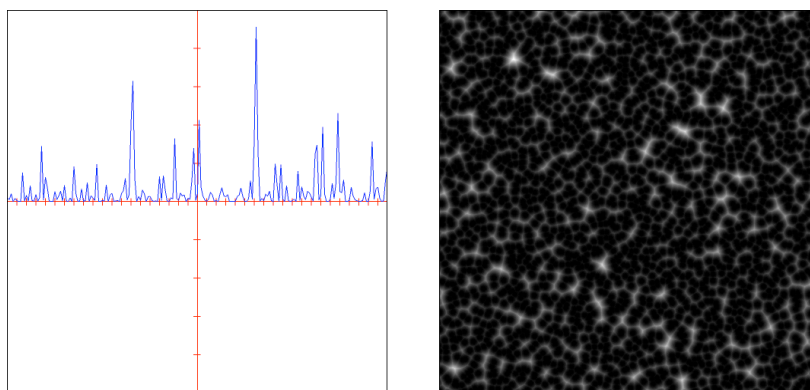
- gradient: Directionally weighted polynomially interpolated values.



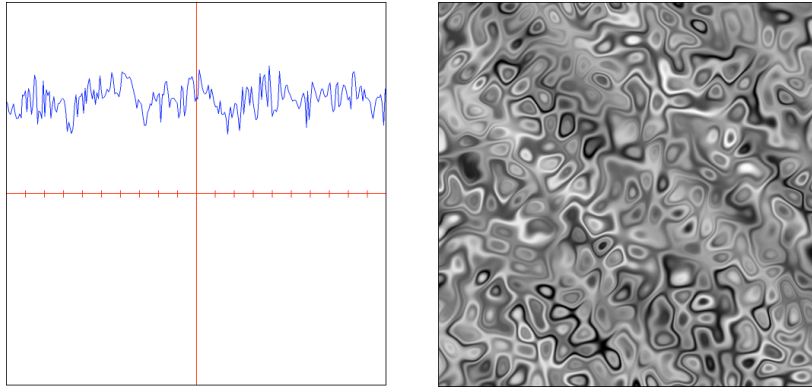
- simplex: Simplex weighted pseudo-random values.



- voronoi: Distance weighted pseudo-random feature points.



- `distorted`: Domain distorted combinational noise.



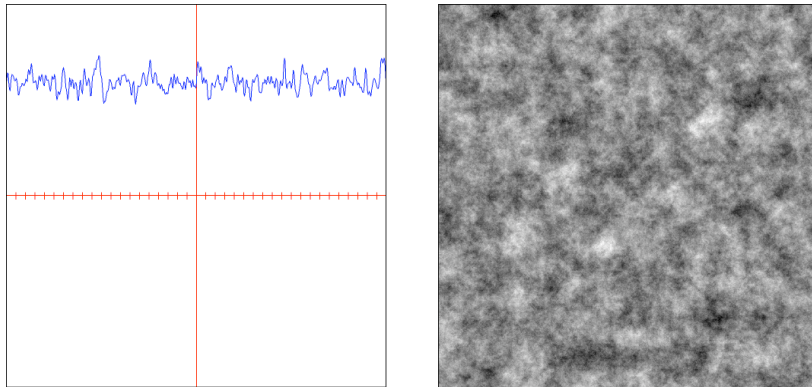
All of these functions are generators with the exception of the `noise.distorted` object, which is a binary operator and uses two existing functions for its evaluation.

Fractal Functions

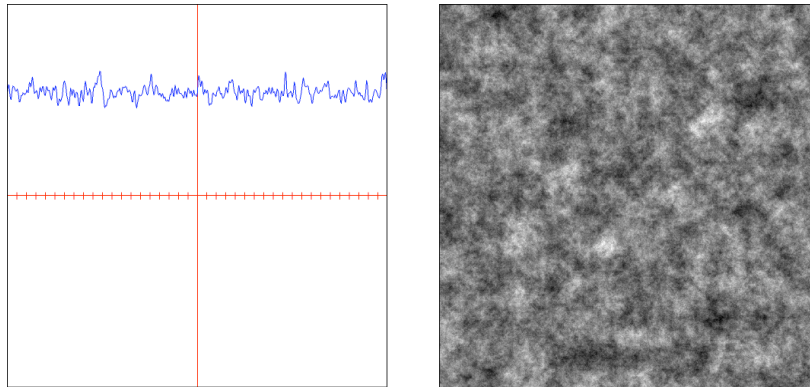
Fractals provide a specialized form of generation by combining multiple scales or octaves of another basis function. This process forms the characteristic self-similarity exhibited by all fractals.

A brief description of these functions is contained in the list below.

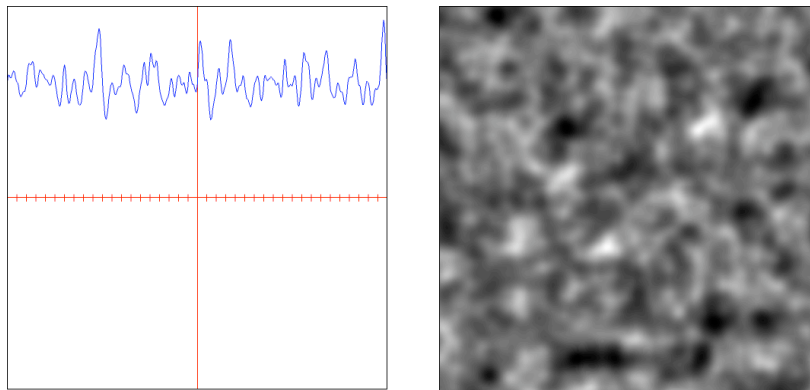
- `mono`: Additive fractal with global similarity across scales.



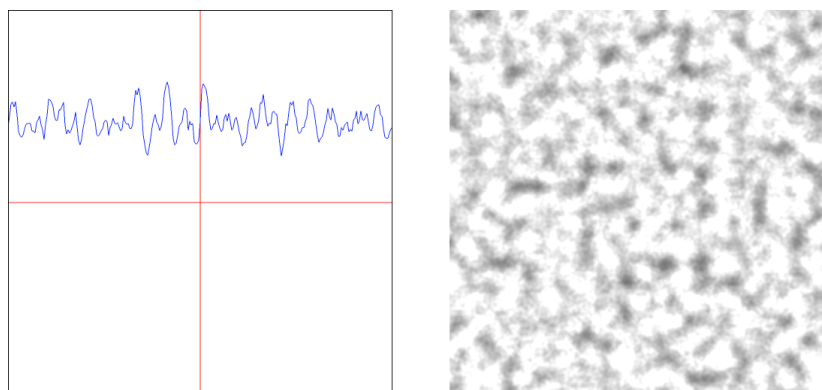
- multi: Multiplicative fractal with varying similarity across scales.



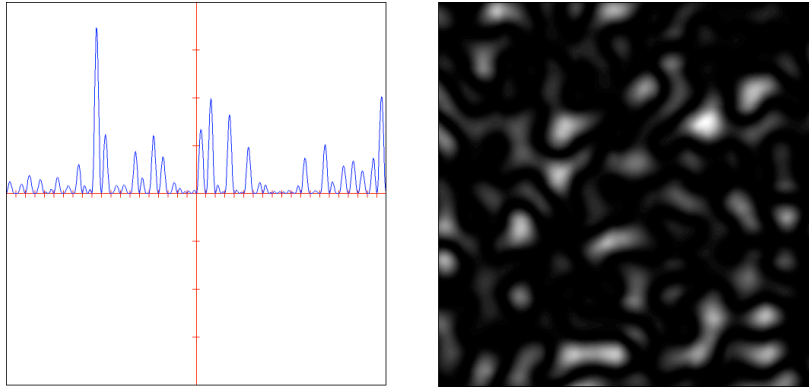
- multi.hybrid: A hybrid additive and multiplicative fractal.



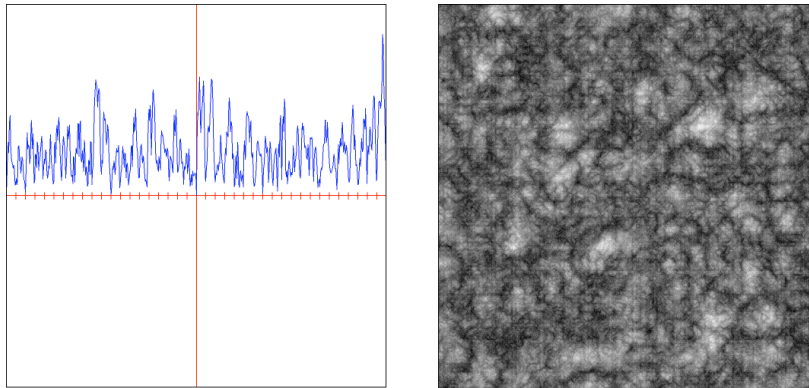
- multi.hetero: Heterogenous multiplicative fractal.



- **multi.ridged**: Multiplicative fractal with sharp ridges.



- **turbulence**: Additive mono-fractal with sharp ridges.



Other Attributes & Messages

- Select the **noise.checker** basis function from the **jit.ubumenu** object.

Notice that in addition to the **scale** message that we used previously, we can also transform the evaluation coordinates through rotation, translation (via **offset**) and by adjusting their origin.

- Change the 1st and 2nd **number boxes** connected to **origin** to change the x and y origin.
- Change the 1st and 2nd **number boxes** connected to **offset** to change the x and y offset position.
- Change the 1st **number box** connected to **rotation** to change our rotation angle about the x axis.

Notice the effect of the transform. Also notice the drop in performance when a rotation is performed – we will always get better frame rates if the rotation attribute is left at 0 for each matrix dimension.

Technical Detail: In addition to the internal coordinate generation already described, **jit.bfg** also accepts an input matrix of coordinates to evaluate (XYZ map to planes 0-2, and the input matrix must be the same dim as the **jit.bfg** output matrix).

- Change the 1st **number box** connected to rotation to 0 to disable rotation.
- Click the **toggle box** connected to autocenter to enable automatic centering.

If the autocenter attribute is set to 1, the current matrix dim sizes will be used to place the origin in the center of the output matrix, overriding any values already set for the origin.

- Click the **toggle box** connected to autocenter again to disable automatic centering.
- Select the noise.gradient basis function from the **jit.ubumenu** object.
- Click the dim 128 128 1 **message box**.

As mentioned previously, all of the basis functions that Jitter provides can be evaluated over any number of dimensions. This message has changed our output matrix to be a 3D matrix, and has correspondingly set the evaluation to be performed in 3-dimensional space. Since our display is still a 2D screen, we only need to evaluate a single slice in 3D, and thus our 3rd dim is set to 1.

- Change the 3rd **number box** connected to offset to change the z evaluation position.

Notice how our results change. We are now traversing along the z-axis as if we were moving forward/backwards through a volume aligned with the screen.

- Select float64 from the **jit.ubumenu** connected to the precision message.

The precision message can be used to change the **jit.bfg** object's internal evaluation precision. This may be desirable if we need more or less accurate results without changing the output matrix type.

- Change the 3rd **number box** connected to offset to change the z evaluation position.

Notice how the higher precision affects the frame rate reported by the **jit.fpsgui** object. We should be careful to only use float64 precision when needed.

- Select float32 from the **jit.ubumenu** connected to the precision message.

- Change the `planeCount` for **jit.bfg** from 1 to 3 to enable RGB output.

In addition to n -dimensional evaluation, **jit.bfg** can generate up to 32 planes per dimension. Each plane is offset by a pseudo-random fractional amount controlled by the `align` attribute.

- Change the **number box** connected to `align` for **jit.bfg**.

Notice how the planes separate and become more visible as the `align` amount gets larger.

- To see more specific examples for different combinations of *basis* functions, open the help patch for the **jit.bfg** object and look in the subpatches for each category of function..

Technical Detail: The output of **jit.bfg** can actually be used as an input to another **jit.bfg** to perform domain distortion, similar to the way `noise.distorted` operates. Check out the example patch in *jit-examples/other/jit.bfg.distorter.pat*.

Summary

The **jit.bfg** object gives us access to a library of procedural basis functions and generators that we can use to define a procedural model for creating textures and modifying geometry. Internally **jit.bfg** generates Cartesian coordinates along a grid. These coordinates can be transformed using the corresponding `origin`, `offset`, and `rotation` attributes, or overridden altogether via an input matrix containing evaluation coordinates.

Tutorial 51: Jitter Java

This tutorial assumes that the reader is familiar with the process of programming **mxj** Java classes. The details for how this is done are contained in the *Writing Max External in Java* document.

In this Tutorial we'll outline how **mxj** classes can operate directly on the cells of an input matrix. We also will see how to create classes in the Java programming language that internally load Jitter objects and define and execute a processing network. Finally, we will learn how to design hardware-accelerated user interface elements by attaching a "listener" to a drawing context and polling a window for mouse events.

The Jitter Java API centers around the **JitterObject** class, which provides a way for **mxj** classes to instantiate Jitter objects. The following line of code creates a **jit.op** Jitter object.

```
JitterObject jo = new JitterObject("jit.op");
```

We can send messages to objects in our Java code by using **JitterObject**'s **call()** or **send()** methods. For instance, we might set the operator in the above instance of **jit.op** as follows:

```
jo.send("op", "+");
```

We could alternatively set this attribute using the **setAttr()** method:

```
jo.setAttr("op", "+");
```

The **JitterMatrix** class extends **JitterObject** – after all, **jit.matrix** is a Jitter Object. Since **jit.matrix** is such a common object it is handy to have convenience methods dedicated to its creation and destruction. For instance, the below line of code creates a new 4-plane **JitterMatrix** of type **char** and dimensions 320 by 240.

```
JitterMatrix jm = new JitterMatrix(4, "char", 320, 240);
```

We can also create a **JitterMatrix** object by specifying the matrix name:

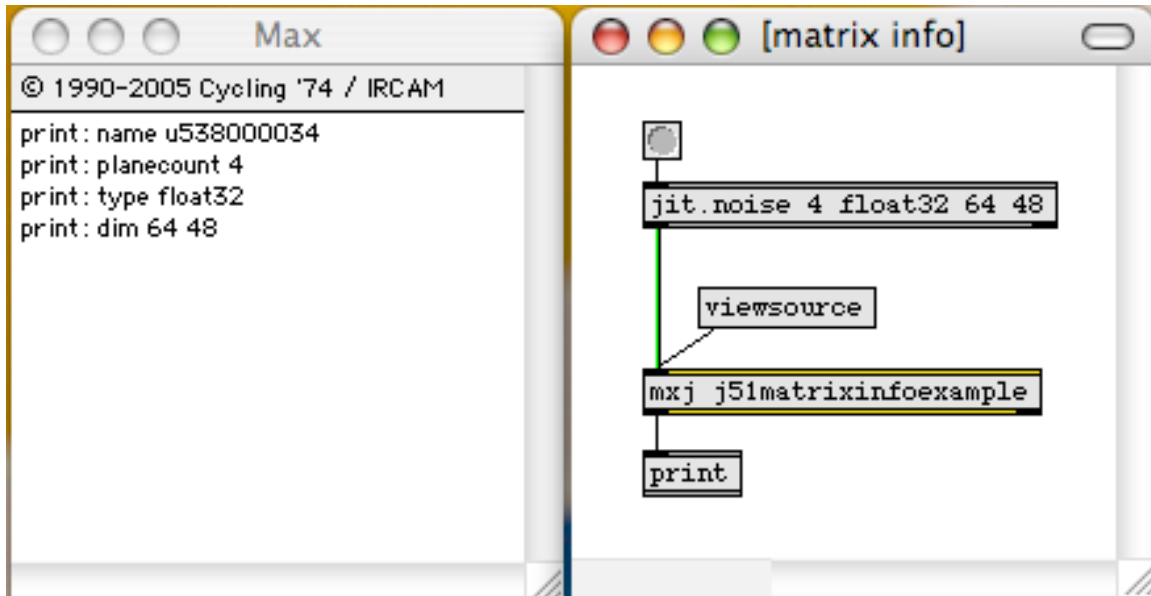
```
JitterMatrix jm = new JitterMatrix("Stanley");
```

If a **jit.matrix** object named Stanley already exists, this Java peer **JitterMatrix** object will refer to it. In this case a new **jit.matrix** object is not created.

Conveniences are nice, but the primary reason the **JitterMatrix** class exists is to provide native methods to access the data in the matrix cells. We'll see an example of this in action when we open the tutorial patch.

Accessing an Input Matrix

- Open the tutorial patch *51jJava.pat* in the Jitter Tutorials folder. Double-click on the matrix info subpatcher. Click on the **button** object and look at the output in the Max window.



Information about our matrix printed in the Max window.

In this subpatcher, clicking on the **button** object causes the **jit.noise** object to send a 4 plane 64 by 48 float32 matrix into an **mxj** object with the *j51matrixinfoexample* class loaded. This Java class sends some basic information about the input matrix out its outlet. Let's examine the code for the class:

```
import com.cycling74.max.*;
import com.cycling74.jitter.*;

public class j51matrixinfoexample extends MaxObject {

    public void jit_matrix(String s)
    {
        JitterMatrix jm = new JitterMatrix(s);

        outlet(0,"name", jm.getName());
        outlet(0,"plane count", jm.getPlaneCount());
        outlet(0,"type", jm.getType());
        outlet(0,"dim", jm.getDim());
    }
}
```

The class consists of a single `jit_matrix` method. Of course since matrices are passed between Jitter objects as named references, we think of a matrix being input as just a simple two element list, and this is how **mxj** sees it. The argument of the `jit_matrix`

message is of course the matrix name, so the first thing our method does is create a new JitterMatrix object with the name of the input matrix as the constructor's only argument. The JitterMatrix object that is created will have that named **jit.matrix** object as its peer. The next four lines of code simply output the results of some basic queries that can be made using the methods of the JitterMatrix class.

Operating on a Matrix

- Close the matrix info subpatcher. Open the striper subpatcher.

The striper subpatcher gives us an example of a class that operates on the data of an input matrix. Note that there are two classes set up to be used, with a **Ggate** object allowing us to switch between them; we will be comparing two different ways of performing the same operation to see which is more efficient.

Here is the code for *j51matrixstriperA*:

```
import com.cycling74.max.*;
import com.cycling74.jitter.*;

public class j51matrixstriperA extends MaxObject {

    JitterMatrix jm = new JitterMatrix();
    int frgb[] = new int[] {255, 255, 255, 255};
    int on = 2, off = 1;

    j51matrixstriperA()
    {
        declareAttribute("frgb");
        declareAttribute("on");
        declareAttribute("off");
    }

    //note that this method assumes a 2D char matrix!
    public void jit_matrix(String s)
    {
        jm.frommatrix(s);
        int dim[] = jm.getDim();
        int count = 0;
        boolean notoff = true;
        for (int i=0;i<dim[1];i++)
            for(int j=0;j<dim[0];j++)
            {
                if (notoff)
                    jm.setcell2d(j, i, frgb);
                if ((notoff && (++count > on))
                    || (!notoff&& (++count > off)))
                {
                    count = 0;
                    notoff = !notoff;
                }
            }
        outlet(0, "jit_matrix", jm.getName());
    }
}
```

In the `jit_matrix` method we use the `getDim()` method to find out the dimensions of our matrix and store them in the int array *dim*, which we then use as the terminal conditions in the `for()` loops of our iterative procedure. The `setcell2d` method allows us to directly set the value of any cell in the matrix. When we are finished processing, we send a `jit_matrix` message out with the name of our `JitterMatrix` as the argument.

- Toggle the **Ggate** object back and forth. Which class is faster?

The only difference between *j51matrixstriperA* and *j51matrixstriperB* is the `jit_matrix` method. Here is *j51matrixstriperB*'s `jit_matrix` method:

```
//note that this method assumes a 2D char matrix!
public void jit_matrix(String s)
{
    jm.frommatrix(s);
    int dim[] = jm.getDim();
    int count = 0;
    int planecount = jm.getPlanecount();
    int offset[] = new int[]{0,0};
    boolean notoff = true;
    int row[] = new int[dim[0]*planecount];

    for (int i=0;i<dim[1];i++)
    {
        offset[1] = i;
        jm.copyVectorToArray(0, offset, row,
                             dim[0]*planecount, 0);
        for(int j=0;j<dim[0];j++)
        {
            if (notoff)
            {
                for (int k=0;k<planecount;k++)
                    row[j*planecount+k] = frgb[k];
            }
            if ((notoff &&(++count > on))
                ||(!notoff&&(++count > off)))
            {
                count = 0;
                notoff = !notoff;
            }
        }
        jm.copyArrayToVector(0, offset, row,
                             dim[0]*planecount, 0);
    }
    outlet(0, "jit_matrix", jm.getName());
}
```

Rather than set values in the matrix one cell at a time, the `jit_matrix` method of *j51matrixstriperB* grabs an entire row from the matrix using `JitterMatrix`'s `copyVectorToArray` method, overwrites the appropriate values in the row, then writes the row back to the matrix using `JitterMatrix`'s `copyArrayToVector` method. As you may have noticed, this version of the class is significantly faster than the version that sets cells in the matrix one-by-one. This is an excellent demonstration of the following very important thing to keep in mind: it is very expensive to cross the C/Java boundary. The version of this object that uses `setcell` must go back and forth across the C/Java boundary once for every cell in the matrix. The latter version goes back and forth just twice per row. If you have tried both versions of the object out, the savings are evident.

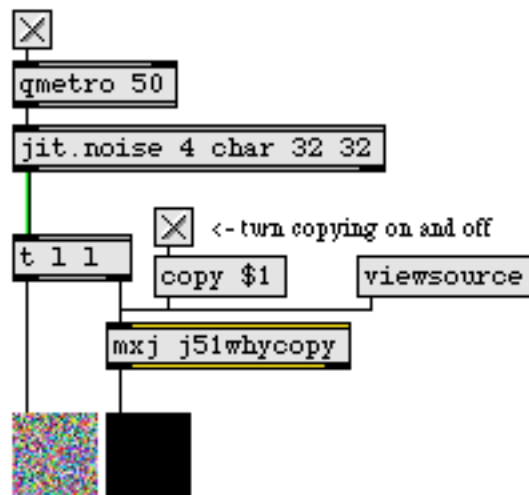
The `copyVectorToArray` and `copyArrayToVector` methods are overloaded with signatures that cover all the relevant data types. As you can see from a careful examination of the above code, these methods provide and expect the Java arrays with the planar data multiplexed so that all of a cell's data is presented contiguously. There are also `copyVectorToArrayPlanar` and `copyArrayToVectorPlanar` methods for moving a single plane's data into Java.

When using `mxj` classes to process matrices in Jitter there is no built-in `usurp` functionality, so it is important to always drive the object network using a `qmetro` object, or some other construction that defers events to prevent backlog.

Copying Input Data

In the above examples may also have noted the different way that we internally use our `JitterMatrix`. Whereas in the matrix info example we created a new `JitterMatrix` every time an input matrix was received, this time we cache one instance of `JitterMatrix` and copy the input data into it every time using the `frommatrix` method. Why do we do this?

- Stop the `qmetro` and close the `striper` subpatch. Open the `why copy?` subpatch. Turn on the `qmetro`.



Why copy?

The **trigger** object in this subpatch sends the output of the `jit.noise` object first into an instance of the `mxj` class `j51whycopy`, which outputs to a `jit.pwindow` object, and then the same matrix is sent to a different `jit.pwindow` object.

Let's examine the Java code for *j51whycopy*:

```
import com.cycling74.max.*;
import com.cycling74.jitter.*;

public class j51whycopy extends MaxObject {

    JitterMatrix jm = new JitterMatrix();
    boolean copy = false;

    j51whycopy()
    {
        declareAttribute("copy");
    }

    public void jit_matrix(String inname)
    {
        //under normal circumstances
        //we would only create this matrix once
        jm = new JitterMatrix();
        if (copy)
        {
            jm.frommatrix(inname);
        }
        else /*!copy
        {
            jm = new JitterMatrix(inname);
        }
        zero(jm);
        outlet(0, "jit_matrix", jm.getName());
    }

    //note that this method assumes the matrix is of type char
    private void zero(JitterMatrix m)
    {
        int z[] = new int[m.getPlanecount()];
        for (int i=0;i<m.getPlanecount();i++)
            z[i] = 0;
        m.setall(z);
    }
}
```

The copy attribute flips the `jit_matrix` method between two different modes: if `copy` is true, the data from the input matrix is copied into our internal `JitterMatrix` using the `frommatrix` method. If `copy` is not true, the `JitterMatrix` `jm` is created with the input matrix as its peer. In both cases our `JitterMatrix` is zeroed using the `setall` method, and then output.

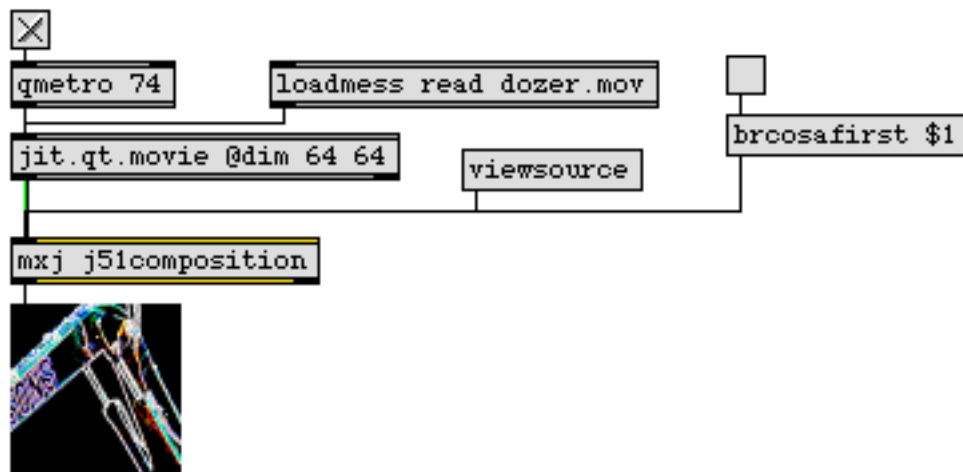
- Turn the copy attribute on and off with the **toggle box**. What happens in the left **jit.pwindow**?

When the copy attribute is off and we create a new JitterMatrix associated with the input name we operate directly on the data of that matrix. Therefore we may be altering the contents of the matrix before other objects have had a chance to see it. In this example patch, when the copy attribute is true the left **jit.pwindow** correctly shows the matrix produced by **jit.noise**. When the copy attribute is false both **jit.pwindow** objects are black because our **mxj** class alters the matrix's data directly. Therefore if we are operating on a matrix, we should make sure to copy the input data into an internal cache as we do here with **frommatrix**.

Object Composition

In this section we'll examine the use of multiple JitterObjects within a single Java class.

- Toggle the **qmetro** off and close the subpatcher window. Open the composition subpatcher. Toggle the **qmetro** on.



A Jitter processing chain executing within Java.

Let's examine the code for the *j51composition* class:

```
import com.cycling74.max.*;
import com.cycling74.jitter.*;

public class j51composition extends MaxObject {
    JitterMatrix jm = new JitterMatrix();
    JitterMatrix temp = new JitterMatrix();
    JitterObject brcosa;
    JitterObject sobel;
    boolean brcosafirst = false;

    j51composition() {
        declareAttribute("brcosafirst");
        brcosa = new JitterObject("jit.brcosa");
        brcosa.setAttr("brightness", 2.0f);
        sobel = new JitterObject("jit.sobel");
        sobel.setAttr("thresh", 0.5f);
    }

    public void jit_matrix(String mname)
    {
        jm.frommatrix(mname);
        temp.setinfo(jm);
        if (brcosafirst)
        {
            brcosa.matrixcalc(jm, temp);
            sobel.matrixcalc(temp, jm);
        }
        else
        {
            sobel.matrixcalc(jm, temp);
            brcosa.matrixcalc(temp, jm);
        }
        outlet(0, "jit_matrix", jm.getName());
    }

    public void notifyDeleted()
    {
        brcosa.freePeer();
        sobel.freePeer();
    }
}
```

Two JitterObjects are created in the class's constructor: *brcosa* controls a peer **jit.brcosa** object, and *sobel* controls a peer **jit.sobel** object. The *jit_matrix* method copies the data from the input matrix to *jm*, and then sets the dimensions, planecount and type of the *temp* matrix to that of *jm* with the *setinfo* method. After this the two JitterObjects can

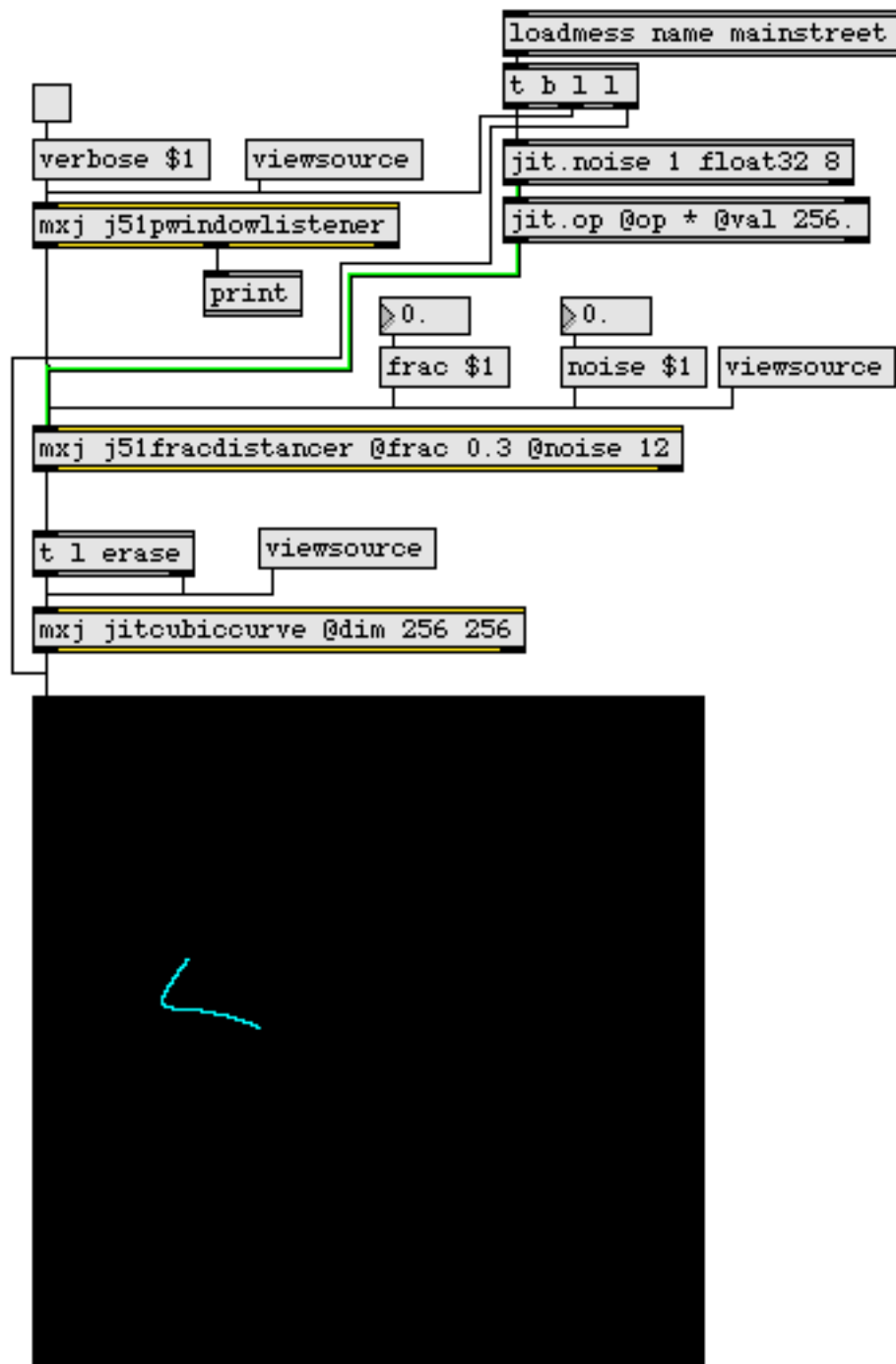
be used to process the data in either order: if the attribute `brcosafirst` is true, the `brcosa` operates first and then `sobel`, and of course vice versa if `brcosafirst` is false. We operate on the data in a matrix by calling a `JitterObject`'s `matrixcalc` method. The two arguments are input and output matrices – it is also possible to use arrays of input and output matrices if the peer Jitter object supports multiple matrix inputs or outputs. This simple example shows how it is possible to define a varying network of Jitter objects within a Java class.

- Toggle the `brcosafirst` attribute on and off to see the difference in the resulting image.

Finally, note that when the `mxj` object is deleted and the `notifyDeleted` method is called, the peer Jitter objects are freed by calling the `freePeer()` method for every `JitterObject` we've instantiated. If `freePeer()` is not called the C object peers will persist until the Java memory manager garbage collects, and this could take a while.

Listening

- Toggle the **qmetro** off and close the subpatch. Open the cubiccurver subpatch. Turn on the **toggle box** labeled *verbose*, move the mouse in the **jit.pwindow** object and observe the results in the Max window.



This section assumes you've looked at *Tutorial 47: Using Jitter Object Callbacks in JavaScript*, which covers object callbacks in Javascript. The Jitter Java API provides the same mechanism for "listening" to events generated by named Jitter objects. In this subpatcher we use this mechanism to track mouse movement within a named **jit.pwindow**. Let's examine the source code for the *j51pwindowlistener* class:

```
import com.cycling74.max.*;
import com.cycling74.jitter.*;

public class j51pwindowlistener extends MaxObject
implements JitterNotifiable
{
    JitterListener listener;
    boolean verbose = false;

    j51pwindowlistener()
    {
        declareIO(1,2);
        declareAttribute("verbose");
    }

    public void name(String s)
    {
        listener = new JitterListener(s,this);
    }

    public void notify(JitterEvent e)
    {
        //this gets the name of the listening context
        String subjectname = e.getSubjectName();
        //this gets the type of event...mouse, mouseidle, etc.
        String eventname = e.getEventName();
        //this gets the arguments of the event
        Atom args[] = e.getArgs();
        if (verbose)
        {
            outlet(1,subjectname);
            outlet(1,eventname,args);
        }

        if ((eventname.equals("mouse"))
        ||(eventname.equals("mouseidle")))
        {
            int xy[] = new int[] {args[0].toInt(),
args[1].toInt()};
            //output the x and y coordinates of the mouse
            outlet(0, xy);
        }
    }
}
```

The first thing to notice is that the class implements an interface called `JitterNotifiable`. This simple interface ensures that the class has a `notify` method which will be called when an event is “heard”. Notice that the `notify` method takes a `JitterEvent` as an argument. The `notify` method in this class illustrates the methods of these `JitterEvents` that we’ll use to extract the pertinent data: `getSubjectName` returns the name of the listening context, which is useful if we’re listening in more than one place; `getEventName` returns the name of the event that has been heard; and `getArgs` returns the arguments that detail the parameters of the event. With the `verbose` attribute enabled, this class’s `notify` method outputs these data out the second outlet and a **print** object sends it to the Max window.

The `notify` method continues by testing the value of `eventname` for equality with either `mouseidle` (for normal mouse movement) or `mouse` (for movement with the button down). If either of these Strings is a match, the first two arguments, which represent the *x* and *y* position of the mouse in the window, are output as a list.

The last thing to note about this class is that before any listening can happen the listener must be created with an existing context. In this case that happens when an instance of the class receives a name message. Unlocking the patch will show you how the **jit.pwindow** object is given a name first; the name message is then sent to the instance of *j51pwindowlistener*.

The rest of the patch uses the *jitcubiccurve* example class to render a cubic curve the follows the mouse around. The control points for the cubic curve are calculated using the *j51fracdistancer* class, whose code you may want to read through to see if you’ve understood the material in this lesson:

```
import com.cycling74.max.*;
import com.cycling74.jitter.*;
import java.util.*;

public class j51fracdistancer extends MaxObject{

    JitterMatrix ctlmatrix =
        new JitterMatrix(1, "float32", 8);
    float ctldata[] = new float[8];
    int offset[] = new int[] {0};
    float frac = 0.5f;
    float noise = 0.5f;
    Random r = new Random();

    j51fracdistancer() {
        declareAttribute("frac");
        declareAttribute("noise");
    }
}
```

```

//new x,y input
public void list(int args[])
{
    if (args.length != 2) return;
    float x = (float)args[0];
    float y = (float)args[1];
    for (int i=0;i<4;i++)
    {
        ctldata[i*2] += (x-ctldata[i*2]) * frac +
((float)r.nextGaussian()*noise);
        ctldata[i*2+1] += (y-ctldata[i*2+1]) * frac +
((float)r.nextGaussian()*noise);
    }
    ctlmatrix.copyArrayToVector(0,offset, ctldata, 8, 0);
    outlet(0, "jit_matrix", ctlmatrix.getName());
}

public void jit_matrix(String mname)
{
    JitterMatrix jm = new JitterMatrix(mname);
    if (jm.getDim()[0] != 8) return;
    if (jm.getPlanecount() != 1) return;
    if (!jm.getType().equals("float32")) return;

    ctlmatrix.frommatrix(mname);
    ctlmatrix.copyVectorToArray(0, offset, ctldata, 8, 0);
}
}

```

Summary

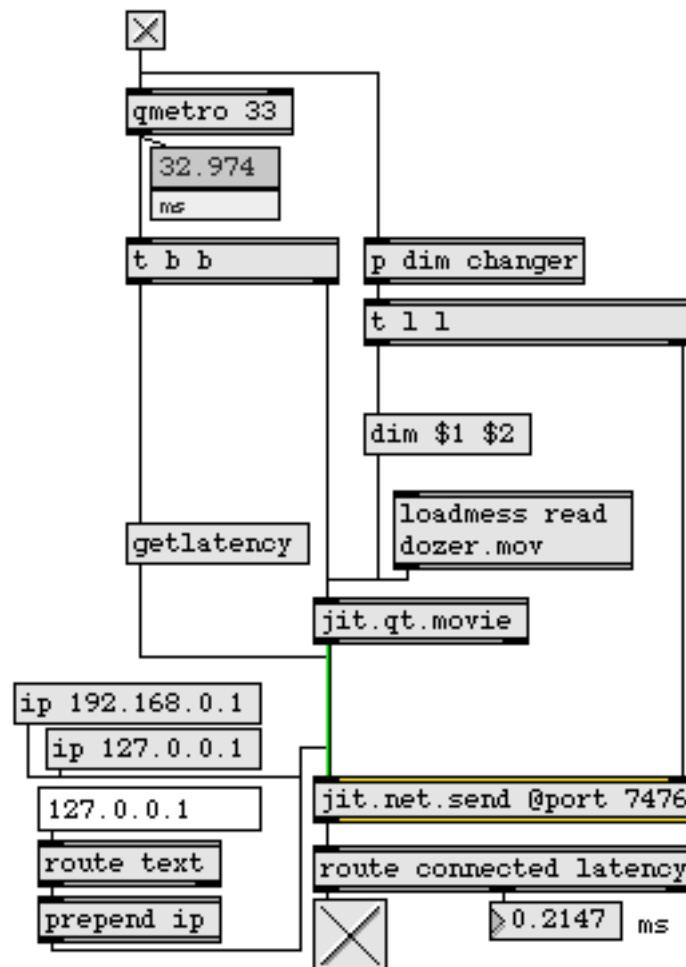
In this tutorial we've introduced the JitterObject, JitterMatrix, and JitterListener classes, and we've seen how to use them to put together **mxj** classes that operate directly on data, use composition of existing Jitter objects to define graphs of processing, and how to listen to Jitter objects for events in a way that can facilitate the building of user interface components, among other things.

Tutorial 52: Jitter Networking

The **jit.net.send** object allows us to send uncompressed matrices over the Internet to a **jit.net.rcv** object in a Max patch on a different computer.

- In the Jitter Tutorials folder, open the tutorial patches *52jNetworkSend.pat* and *52jNetworkRecv.pat*. If you like, you can open these on different computers.

The **jit.net.send** and **jit.net.recv** objects communicate using the TCP protocol. A discussion of the different types of network transmission protocols is beyond the scope of this Tutorial; it's important to note, however, that TCP is a *connection-oriented* protocol. Before anything can be sent or received the pair of objects must make a connection to one another. Both objects report any change in their connection status out the dump outlet with a `connected` message, followed by an argument of 1 when connected or 0 when disconnected.

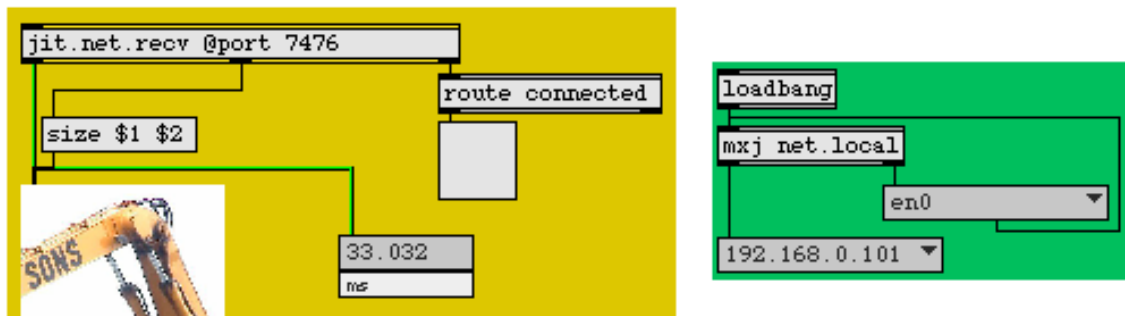


A patch to send Jitter matrices over a computer network using TCP.

To form a connection, the **jit.net.recv** and **jit.net.send** objects must be set to “listen” to the same *port*. We can set the port attribute to any positive integer value, but some of the lower numbers are reserved for commonly used network services such as FTP, web browsing, etc. A port in the range above 1024 and below 10000 will probably be safe. If no port is specified, the default for both objects is 7474.

In addition to port number, the **jit.net.send** object must know the *IP address* of the computer running the Max patch containing the **jit.net.recv** object we’d like to send to. A computer may have more than one IP address – the wireless interface and Ethernet port will have different addresses, for example – and, by default, the **jit.net.recv** object will listen to all of them for a connection request from a **jit.net.send** suitor. It is possible to specify which interface to use by sending the object an ip message with the specific IP address of the desired interface as an argument.

If we open the sending and receiving patches on the same computer, the objects may immediately form a connection. The **toggle boxes** connected to the **route** objects in each patch will be checked if this is the case. By default the **jit.net.send** object’s ip attribute is set to 127.0.0.1, which is the *local loopback* address. The **jit.net.recv** object is listening to all addresses by default, including the local loopback, so a connection should be made if the two patches are on the same computer. If you’re running the patches on different computers, you can use the **mxj net.local** object to figure out what IP address the **jit.net.send** object should be trying to connect with. Note that one can specify an IP address with the ip attribute, or alternatively one can specify a *hostname* with the host attribute and **jit.net.send** will attempt to use the Domain Name Service (DNS) to resolve the hostname to an IP address.



A patch to receive and display Jitter matrices sent over a computer network.

If you can't get your objects to connect to one another, and you're sure you've got the IP address and port correct, make sure a software firewall on any computer isn't preventing communication on the port we've selected (7474 in these tutorial patches). On a computer running Windows XP, the controls for the software firewall can be found in the *Control Panel: Windows Firewall: Advanced* tab, where double clicking any of the network connections will bring up an *Advanced Settings* window that allows you to define new "services". On an OS X machine the controls for the software firewall can be found in *System Preferences: Sharing: Firewall*, where you can click on the *New...* button to specify a port to open. If you are connecting two computers together through one or more routers, it's possible that they have filters in place that are rejecting the packets.

- Turn on the **toggle box** connected to the **qmetro** in the *52jNetworkSend.pat* patch.

In the sending patch, a **jit.qt.movie** object is sending matrices to a **jit.net.send** object. Note that the **qmetro** driving the movie playback triggers the movie frame and then sends the **jit.net.send** object a *getlatency* message. This causes a one-way latency estimate to be output from the dump outlet. We can use this estimate of the transmission latency to synchronize events on the two computers, for instance by delaying the processing or display of a matrix on a local computer by the amount of the estimate, at which point the receiving computer should be ready to process the matrix it has received.

Latency is influenced by two factors: the direct transmission time between the two computers, and the amount of data being sent. The more data that is sent, the longer the time it takes to move all the data across the network. Obviously the speed of the computer's network interface and all routing hardware is key in this respect – the transmission can only be as fast as the weakest link in the chain. One technique we might consider using to reduce latency is to transmit matrices using either of the *grgb* or *uyvy* compressed color spaces that are discussed in Tutorial 49.

In addition to sending matrices, any normal Max message input into the right inlet of **jit.net.send** will come out the middle outlet of the connected **jit.net.recv** object. The input order of messages and matrices will be preserved in the receiving patch. In this example, the **patcher** named *dim* changes contains a slower **qmetro** that changes the dimensionality of the movie every half second. These new dimensions are input into the right inlet of the **jit.net.send** object. When the two integers come out the second outlet of the **jit.net.recv** object they are used as the arguments of the *size* message to resize the **jit.pwindow** object.

The matrices in this tutorial patch are of type *char*, but `jit.net.send` and `jit.net.recv` can handle matrices of any type. For an example that uses *float32* matrices, please refer to *audio-over-network.pat* in the *jit-examples/audio* folder.

Summary

Using the **`jit.net.send`** and **`jit.net.recv`** objects, you can send uncompressed Jitter matrices to another computer running Max. You specify a `port` for the two machines to communicate on, as well as an `ip` address for the sending machine to use when connecting to the receiving machine.

Tutorial 53: Jitter Networking (part 2)

Broadcasting

In addition to sending matrix data over networks, we can also create RTSP streams in Jitter, using the objects **jit.broadcast** and **jit.qt.broadcast**. The **jit.qt.broadcast** object is only available on computers running the Mac OS, and is described at the end of this chapter. The **jit.broadcast** object is cross-platform. Both objects function more or less the same way, though, and users of **jit.qt.broadcast** should follow the entire chapter.

RTSP stands for *Real-Time Streaming Protocol*, and is a popular network streaming standard, supported by many media players such as the QuickTime Player and VLC (you can learn more about VLC, a free, open-source, cross-platform player, at <http://www.videolan.org/vlc/>). These media players are RTSP *clients*. Our Max objects are RTSP *servers*, and they properly format, in some cases compress, packetize and broadcast data for RTSP streaming. Using these objects, you don't need any external server — everything is done for you, all inside of Jitter!

Mainstream

Before we get started, you'll want to make sure that your network is properly configured. Ensure that you have a network interface active and a valid IP address. On Windows, this is particularly important, or our streaming experiments won't work.

Even if you are not connected to a network on Windows, you'll need to have at least one active network interface, as far as the OS is concerned. This might mean logging/dialing into the Internet, or attaching a powered-on router to your Ethernet port, even if you have no other computers to share with—basically, anything which causes Windows to detect that you're using the network—so that it assigns your computer an IP address.

- Open the tutorial patch *53jBroadcasting.pat* in the Jitter Tutorials folder.
- Click on the **message box** labeled *read dishes.mov* and turn on the **toggle box** labeled *Start Movie* to begin movie playback. You should see the movie playing in the **jit.pwindow** object.

Before we begin the actual broadcast, we should consider a few things.

First, what will the dimensions of our broadcast stream be? The two arguments in the **jit.broadcast** object's box determine the size of the compressed, streamed image. To change this size, the *size* attribute is used. In our case, we will stream at 160x120 pixels.

Second, what streaming video codec are we going to use? The **jit.broadcast** object supports 3 different ones—MPEG-4, H.263 and JPEG. Incoming Jitter frames will be automatically compressed into the chosen format before streaming.

- Using the **ubumenu** object, choose your desired streaming codec from the 3 available.

By default, the object uses the MPEG-4 codec, but the choice is yours. Certain codecs may be easier for some clients to receive. For instance, at the time of this writing, QuickTime Player is much better at receiving MPEG-4 video than VLC.

Third, are we streaming to a single computer (known as a *unicast*) or should our stream be made available to any client who accesses the URL of our stream (known as a *multicast*)? By default, the **jit.broadcast** object uses multicast streaming, although this can be changed using the *unicast* attribute.

- Click on the **message box** labeled *getvip* and note the output in the Max Window. When multicasting, the **jit.broadcast** object automatically generates a valid multicast IP address. Click on the **toggle** labeled *Enable Unicast* and click the **message box** labeled *getvip* again, noting the output in the Max Window. The IP address changed to 127.0.0.1 (this address is also known as *localhost*, and means "myself"—using this IP address, the computer would broadcast to itself). Using the *vip* attribute, we could now manually set the IP address of the (single) client computer (you can't change the *vip* attribute when multicasting, since it's generated automatically). For the purposes of this tutorial, we're going to stick to multicasting, but it's important to note that both methods are available.

There are a number of additional parameters which we can set to control the broadcast, such as port numbers, time-to-live (TTL), codec quality and keyframe-generation, and so on. By default, those values are all set to perfectly functional defaults, and there's no need to change them for this Tutorial.

Finally, what should we name our stream? The **jit.broadcast** object produces a URL for client computers to access, in the form *rtsp://192.168.2.2:8554/jitStream*. The IP address and port numbers are generated for us, but we could change the last part to our liking. For instance, we could name our RTSP stream *someDishes*.

- Click on the **message box** labeled *streamname someDishes*. This will customize our stream's name.
- Click on the **message box** labeled *start* to begin streaming. When streaming has successfully begun, the message *start*, followed by a 1, then the RTSP URL (e.g. *start 1 rtsp://192.168.2.2:8554/someDishes*) should appear in the Max Window. This last part is the address that client computers should connect to, in order to receive your stream.

Router Users: Note that if your computer is behind a router providing a local IP address (using NAT, for instance), the RTSP URL will refer to your local IP address, not your "real" IP address (WAN address), as would be accessed from computers outside of your local network. To learn your WAN address, you can point your web browser to <http://checkip.dyndns.org/> (merely one example from many similar websites). You'll substitute this address for the numerical portion of the RTSP URL before the colon. In the example above, the WAN address would take the place of 192.168.2.2.

Most routers have built-in firewalls. In order to properly stream from Jitter, be aware that the RTSP port (the portion of the RTSP URL after the colon, but before the forward-slash; here, 8554) will have to be unblocked, and any network traffic to that port must be directed to your computer. Consult the manual of your router if you are unsure how to do this.

- If you have a 2nd computer on your network, use a web browser or media player on that computer (such as QuickTime Player or VLC) to connect to the URL printed in the Max Window. Or call a friend and ask them to try it over the Internet. Or, if you have neither a 2nd computer nor available friends, you can fire up a copy of a web browser or media player on the same machine Jitter is running on to see your stream.

If it didn't work, check the IP address in the RTSP URL. If the address is 0.0.0.0, this means you've got a network configuration problem on your server machine. Click the **message box** labeled stop, make sure that you're connected to a network and click the **message box** labeled start again.

- When you've confirmed that it's all working, click the **message box** labeled stop. You might want to try some more broadcasting using different codecs and compare image quality and performance.

File Streaming

Using the **jit.broadcast** object, we can also stream pre-compressed Hinted Movie files as created by QuickTime. The **jit.qt.movie** object can export Hinted Movie files, so let's use it to export and hint our *dishes.mov* file, preparing it for broadcast.

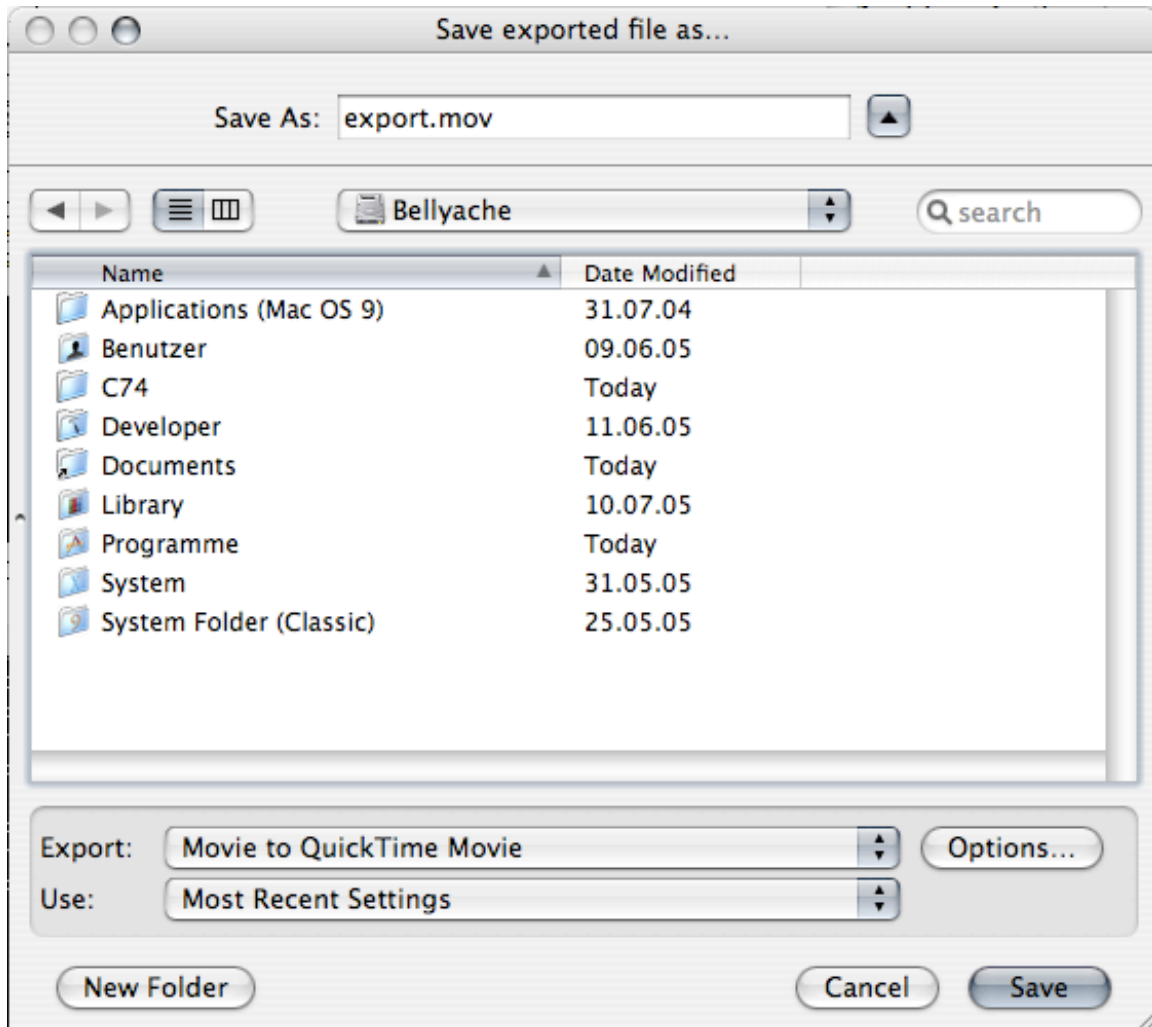
Hinted Movie files contain extra information about the contents of the movie tracks, providing "hints" to a streaming server about codecs and quality, so that the server only has to worry about packetizing and broadcasting the data. In the case of **jit.broadcast**, since the object needn't compress incoming frames, streaming a Hinted Movie might offer a modest performance improvement, depending on your media.

We first need to tell the **jit.broadcast** object that we want to stream a movie file, rather than incoming matrix data. This is accomplished by setting the object's mode attribute.

- Turn on the **toggle** labeled *Matrix / File* to set the **jit.broadcast** object's mode attribute to 1—file broadcast mode.

We use the export message to **jit.qt.movie** to create a Hinted Movie. You may want to refer to *Tutorial 20: Importing and Exporting Single Matrices* if you want to refresh your memory about movie exporting.

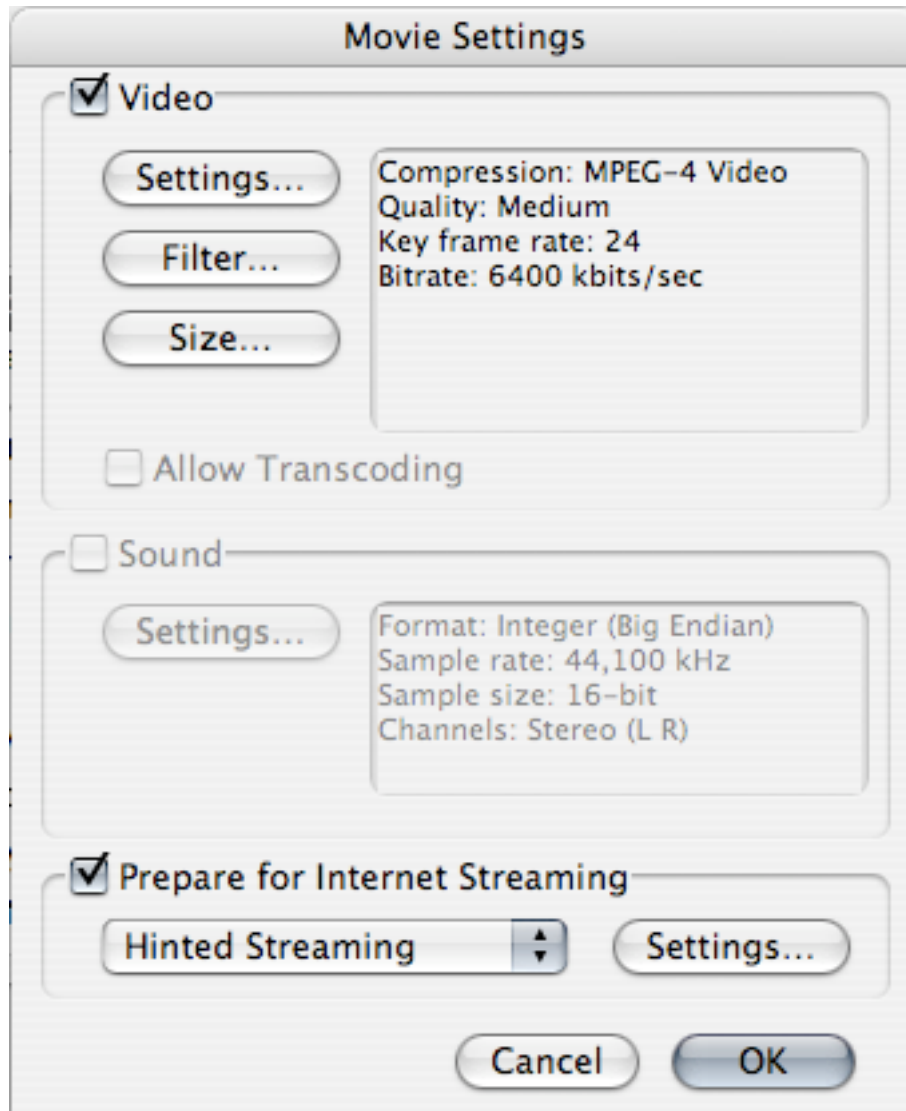
- Click on the **message box** labeled export fulldialog. This brings up the 'Export' dialog box for the movie loaded into **jit.qt.movie** (in our case, *dishes.mov*).



The QuickTime export dialog box on Mac OS X.

- Chose *Movie to QuickTime Movie* from the pop-up menu labeled *Export*, beneath the file browser section of the dialog box. Although there is a selection called *Movie to Hinted Movie*, it doesn't recompress a movie's media, which we want to do.

- Click the button labeled *Options...* to open the Movie Settings dialog box.



The Movie Settings dialog box on Mac OS X.

- Make sure that the *Video* check box has been checked and click on the button labeled *Settings...* just beneath it to open the Standard Video Compression Settings dialog box. Choose *MPEG-4 Video* from the pop-up menu labeled *Compression Type* and then click *OK* to return to the Movie Settings dialog box. Make sure that the check box labeled *Prepare for Internet Streaming* is checked, and that *Hinted Streaming* is selected in the pop-up menu directly beneath. Click *OK* when you're done, to return to the Export dialog box.
- Choose a location to save your Hinted Movie and a name, and click the *Save* button to write the movie to a file on your hard disk.

Good work! We're almost ready to stream the movie. In the case of a Hinted Movie, the `vcodec` attribute (or related information such as the `spatialquality`, `temporalquality` or `kframe` attributes) to **jit.broadcast** has no effect—all of that information is already provided by the movie itself. All we need to do is tell the **jit.broadcast** object which movie to stream.

- Click the **message box** labeled movie to bring up a standard file dialog box, navigate to the location of your newly saved Hinted Movie and open it. A message should be posted to the Max Window with the filename of your movie followed by the number 1, indicating that the movie was read successfully.
- Press the **message box** labeled start to begin broadcasting. Just like before, a message with the RTSP URL will be output from the object's dump outlet and posted to the Max Window. Verify with a media player or web browser that the movie is streaming.
- When you're done, click the **message box** labeled stop to stop the broadcast.

Using the **jit.qt.broadcast** object (Mac OS X only)

In most respects, the **jit.qt.broadcast** object functions identically to its companion **jit.broadcast** object. There are a few significant differences between the 2 objects, though:

- The **jit.qt.broadcast** uses Session Description Protocol (*SDP*) files, instead of an "automatic" session announcement, when making a stream available.
- The **jit.qt.broadcast** can stream several more codecs than the **jit.broadcast** object.
- The **jit.qt.broadcast** can stream audio and/or video from Hinted Movie files, not just video.
- The **jit.qt.broadcast** can stream audio and/or video from an attached camera.

The interested learner can consult the on-line Reference and help patch for the **jit.qt.broadcast** object to learn about streaming from a camera. For this tutorial, we're simply going to go through the process of generating an SDP file and starting a video-only broadcast based on incoming Jitter matrices.

Session Description Protocol files simply provide some basic data to RTSP clients about how many tracks are being streamed, some information about how they are encoded, and where in the network or internet the client should look for the actual streaming data (the SDP file needn't be located at the source of the data). The **jit.qt.broadcast** object automatically generates these files, based on how the broadcast's codec, size, etc. have been set in the object. By making this file available to RTSP clients—emailing the file to friends, posting a link to the file on a web site, etc.—you can publicize your stream.

- If the tutorial patch *53jBroadcasting.pat* is still open, close it now. Then open the tutorial patch *53jQTBroadcasting.pat* from the Jitter Tutorials folder. As you can see, the patches look very similar. Just as in the previous patch, the 2 arguments to our broadcast object determine the dimensions of the broadcast stream.
- Click on the **message box** labeled *read dishes.mov* and turn on the **toggle** labeled *Start Movie* to begin movie playback. You should see the movie playing in the **jit.pwindow** object.
- If you like, make a selection from the **ubumenu** labeled *Codec*. The default video codec, H.263, should be already selected. Note that we have access to all of the standard video codecs available in QuickTime.

Some codecs are much better suited for streaming than others. H.261, H.263, MPEG-4 and Photo-JPEG are reasonably common. Other codecs can be used, but may not offer acceptable image quality or performance, streamed over a network.

- Click on the **message box** labeled *write*. A standard file dialog box will open, where you can choose the name and location for the SDP file that describes your broadcast stream. Click *OK* to save the file.

The SDP file is now configured for your broadcast, as it is currently set up. If you make any changes—a change from multicast to unicast, or to a new codec—you will need to write a new SDP file.

- Click on the **message box** labeled *start* to begin streaming.
- If you have a 2nd computer on your network, use a web browser or media player on that computer (such as QuickTime Player or VLC) to open the SDP file (you will need to transfer the file to the other machine somehow). Or email the SDP file to a friend and ask them to try it over the Internet. Or, if you have neither a 2nd computer nor available friends, you can open the file in a local web browser or media player and try it out yourself.
- Click on the **message box** labeled *stop* to end the broadcast.

Summary

In this chapter, we learned how to use the **jit.broadcast** and **jit.qt.broadcast** objects to serve RTSP (*Real-Time Streaming Protocol*) streams directly out of Jitter, using different codecs and different methods of session announcement. We also learned how to export Hinted Movies from the **jit.qt.movie** object.

Appendix A: QuickTime Confidential

Structure of QuickTime movies

When you imagine a QuickTime movie, chances are good that you think of something similar to film: a single series of frames arranged linearly in time, with or without sound.

In fact, the structure of QuickTime movies is far more similar to the model you may be familiar with from MIDI and audio sequencers.

QuickTime movies are containers for tracks. Each track contains a particular type of media—such as audio or video—and defines the spatial and/or temporal boundaries of that media. For instance, a QuickTime movie might be 320x240 in size, and 3 minutes in length, while one of its video tracks is 80x60 in size, and only 30 seconds long. QuickTime movies are never longer than the end of their longest track.

Each track in a QuickTime movie might be of a different type—video, audio, sprites, MIDI (QuickTime calls these *music* tracks), and VR Panoramas are some of the available types—and might start at a different point within the movie. You can have several tracks of identical types in a single movie as well. Tracks are independent—they can be turned on and off individually, and they can all be of different lengths.

The QuickTime music track is a container for a Standard MIDI File. Using the QuickTime Settings Control Panel, you can set the output for these tracks. By default, QuickTime uses the QuickTime Music Synthesizer, which is a General MIDI software synthesizer built into QuickTime. If you have Open Music System (OMS) installed, you can have QuickTime send music tracks out via OMS to a hardware synthesizer. Or, by using the Inter-Application Communication bus (IAC), you can send them right back into Max!

The visual priority of tracks is determined by the track's visibility layer. Track layers are numbered from -32768 to 32767. Tracks with low visibility layer values will be displayed in front of tracks with high visibility layer values. Audio tracks will automatically mix, regardless of their layer value.

The **jit.qt.movie** object offers a number of track-centered functions. A brief list of some useful messages follows:

General Track Editing

`addtrack`: create a new track
`deletetrack`: delete a track
`gettrackinfo`: displays several pieces of track information (index, name, type, enabled status, visibility layer)
`gettrackoffset`: displays a track's temporal offset (set using the `trackoffset` message)
`gettrackduration`: displays a track's duration (set using the `trackduration` message)
`gettrackdim`: displays a track's spatial dimensions (set using the `trackdim` message). You can adjust a track's size further using the `mxform` message to generate a matrix transformation for a particular track.
`gettrackname`: displays a track's name (set using the `trackname` message)
`gettracktype`: displays a track's type (type cannot be set—use `addtrack` to create a new track)
`gettracklayer`: display a track's visibility layer (set using the `tracklayer` message)
`gettrackcodec`: display a track's codec (codec cannot be set)
`gettrackenabled`: displays a track's enabled status (set using the `trackenabled` message)
`getgmode`: displays a track's drawing mode—the manner in which it interacts with other tracks (set using the `gmode` message)

Important: Track edits of any sort, including adding or deleting tracks, will not be automatically saved with a movie when it is closed. Use the `savemovie`, `savemovieas` or `savemoviecopy` messages to permanently save your changes, or set the **`jit.qt.movie`** object's `autosave` attribute to 1 (on).

QuickTime VR

`vrpan`: set the pan for a VR movie
`vrtilt`: set the tilt for a VR movie
`vrfov`: set the field of view for a VR movie
`vrnode`: set the current node for a multi-node VR movie (use the `getvrnodelist` message to enumerate available nodes)

QuickTime Effect (see *Tutorial 24: QuickTime Effects* for more information)

`addfxtrack`: create a QuickTime Effects track set (1 to 3 actual tracks), by importing a .qfx file
`deletefxtrack`: delete any and all QuickTime Effects tracks created by the `addfxtrack` message

Background Color

`addbgtrack`: create a background color track
`deletebgtrack`: delete a background color track created by Jitter.

Time in QuickTime

Unlike most hardware-based media, QuickTime does not use a frame-based time system. Instead, QuickTime uses a system based on *timescale* and *time values*. Timescale is an integer value that indicates how many time values make up one second of a movie. By default, a new QuickTime movie has a timescale of 600—meaning that there are 600 time values per second.

Frame rate is determined by the concept of *interesting time*—times where the movie changes. If a movie changes every 40 time values, then it has an effective frame rate of 15 frames per second (since 600 divided by 40 equals 15). When Jitter determines how many frames are in a movie with the `getframecount` message, it scans the movie for interesting times, and reports their number. The `frame` and `jump` messages move the “play head” between different points of interesting time.

For more information on the relationship between timescale and frame rate, refer to *Tutorial 4: Controlling Movie Playback*.

In Jitter, recording operations permit you to specify a frame rate, and Jitter takes care of the calculations for you as it constructs a new movie. Editing operations, in contrast, use time values to determine their range of effect.

Optimizing Movies for Playback in Jitter

Although Jitter will gladly play any movie you throw at it, there are some guidelines you can follow to improve performance. Sadly, there is no precise recipe for perfection—performance in a real-time application such as Jitter is the result of the interaction between a movie’s track codecs (which affect data bandwidth and processor load), movie dimensions, frame rate and, to some extent, the complexity of the media being processed.

Codecs

Visual media, in particular, contain large amounts of data that must be read from disk and processed by your computer before they can be displayed. Codecs, or *compressor/decompressors*, are used to encode and decode data. When encoding, the goal is generally to thin out the data so that less information has to be read from disk when the movie plays back. Reading data from disk is a major source of overhead when playing movies.

If you have enough RAM, you can use the `loadram` message to **jit.qt.movie** to copy a movie's (compressed) media to RAM. Since accessing RAM is significantly faster than accessing a hard disk, movie playback will generally improve, although the **jit.qt.movie** object still has to decompress each frame as the movie plays. To buffer decompressed matrix data to RAM, use the **jit.matrixset** object.

When decoding, the goal is to return the data to its pre-encoded state as quickly as possible. Codecs, by and large, are *lossy*, which means that some data is lost in the process. As users, our goal is to figure out which codec offers the greatest quality at the greatest speed for our particular application.

To determine a QuickTime movie's track codecs, use the `gettrackcodec` message to **jit.qt.movie**.

Audio Codecs

Codecs are available for both video and audio tracks. For online distribution, you might want to use an MPEG 2 Level 3 (.mp3) or QDesign audio codec to create smaller files. In Jitter, however, if you are playing movies with video and audio tracks, you'll achieve the best results with uncompressed audio (PCM audio) simply because there will be no audio codec decompression overhead.

Technical Note: .mp3 files can be read by **jit.qt.movie** as audio-only movies (they are compressed with the MPEG 2 Layer 3 codec). Although you can't process the audio data with any other Jitter objects besides the **jit.qt.movie** object, you can use the **jit.qt.movie**'s `soc` attribute to send the audio to MSP via the **spigot~** object (see *Tutorial 27: Using MSP Audio in a Jitter Matrix* for more information).

Video codecs

Video codecs may be handled in hardware—you may have a special video card that provides hardware compression and decompression of a particular codec, like MPEG or Motion-JPEG—or more typically in software. *In Jitter, hardware codec support is only relevant to video output components.* Movie playback will always use a software codec, with the important exception of the **jit.qt.movie** object's direct to video output component feature (see *Tutorial 22: Video Output Components* and the *Object Reference* entry for the **jit.qt.movie** object for more information).

Technical Note: Video cards that provide hardware support of codecs usually only support them for onscreen decompression of media. Since Jitter generally decompresses media into an offscreen buffer (with the exception noted above), software codecs are used.

Video codecs generally use one or both of the following schemes: spatial and temporal compression.

Spatial compression is probably familiar to you from the world of still images. JPEG, PNG and PICT files each use types of spatial compression. Spatial compression schemes search a single image frame for patterns and repetitions that can be described in a simpler fashion. Most also simplify images to ensure that they contain these patterns. Nevertheless, more complex images are harder to compress, and will generally result in larger files. Spatial compression does not take time into account—it simply compresses each frame according to its encoding algorithm.

Temporal compression is unique to the world of moving images, since it operates by creating a description of change between consecutive frames. In general, temporal compression does not fully describe every frame. Instead, a temporally compressed movie contains two types of frames: keyframes, which are fully described frames (usually spatially compressed, as well), and regular frames, which are described by their change from the previous keyframe.

For applications where a movie will be played from start to finish, temporal compression is quite useful. Codecs like Sorenson use temporal compression to create extremely small files that are ideal for web playback. However, temporal compression is not a good choice if you need to play your movie backwards, since the order of the keyframes is vital to properly describing the sequence of images. If we play a temporally compressed movie backwards, the change descriptions will be processed before the keyframe that describes the initial state of the frame! Additionally, the Sorenson codec is quite processor-intensive to decompress. Playback of a Sorenson-compressed movie will be slower than playback of a movie compressed using a “lighter” method.

For Jitter playback, we recommend using a video codec without temporal compression, such a Photo-JPEG or Motion-JPEG (Photo- and Motion-JPEG compression use the same compression method, but Motion-JPEG is optimized for special hardware support [see note above]). At high quality settings, JPEG compression offers a nice compromise between file size and image quality. It’s also relatively simple to decode, so your processor can be put to better use than decompressing video.

If image quality is of principle importance, the Animation codec looks better than Photo-JPEG, but creates much larger files.

Different versions of QuickTime support different audio and video codecs. For instance, QuickTime 5 doesn't support the MPEG-4 codec, although QuickTime 6 does. You should experiment with different codec settings to find the best match for your particular use of Jitter.

Discreet's Codec Central provides an excellent online educational resource about codecs. At the time of this writing, it is offline, but Discreet claims that the site will be up again soon, at <http://www.codeccentral.com/>

Movie Dimensions and Frame Rate

Compared to codec, movie dimensions and frame rate are more straightforward factors in Jitter performance. Simply put, a bigger image frame or a higher frame rate indicates that Jitter has more data to process each second.

A 640x480 movie produces 1,228,800 individual values *per frame* for Jitter to process (640 times 480 times 4 (separate values for alpha, red, green and blue channels). A 320x240 movie produces a mere 307,200 individual values per frame, one quarter the size of the larger movie. Where possible, we recommend using smaller movies if you intend to use Jitter for processing. On most machines, 320x240 movies will give fine performance.

If you are working with DV media, your movies are recorded at 29.97 frames per second (in NTSC) or 25 frames per second (in PAL). Even using a 320x240 movie, Jitter has to process 9,206,784 values per second in NTSC and 7,680,000 values per second in PAL. Thinning this data, by reducing the frame rate to 15 or 20 frames per second, will improve performance significantly if you are using Jitter for processing. On most machines, 15 fps is a good choice.

Our Favorite Setting

We've found that, for most movies, the following parameters yield consistently good results:

- 320x240 frame size
- 15 frames per second
- Video tracks: Photo-JPEG codec, using a medium to high spatial quality setting
- Audio tracks: no compression

Summary

QuickTime movies can contain many tracks of different types—audio, video, QuickTime VR and text, to name a few. In Jitter, we can both get and set basic information about these tracks—including their temporal offset, spatial dimensions, duration, visibility layer and graphic transfer mode—using the **jit.qt.movie** object's track messages.

QuickTime's time model is somewhat different from the standard frame-based model, relying on a *timescale* value to determine the number of *time values* in a second of QuickTime media. The **jit.qt.movie** object allows for playback navigation using both frame and timescale models. All editing functions in the **jit.qt.movie** object use the timescale model to determine the extent of their effect.

Determining the ideal settings for a movie used in Jitter is an inexact science. Actual performance depends on a number of factors, including codec, dimensions, frame rate and media complexity. For best results on current hardware, we recommend using 320x240, 15 fps, Photo-JPEG compressed movies

Appendix B: The OpenGL Matrix Format

Matrices, Video and OpenGL

OpenGL is a standard which specifies a virtual machine for turning lists of numbers into 2D images. The lists may contain the 3D locations of vertices to connect with lines or polygons. They can also contain other data such as coordinates for texture mapping, normals for lighting, colors and edge flags.

Jitter treats video as a type of data which can be processed in its general matrix system. When manipulating video data in Jitter, each cell of a matrix represents one pixel. The four planes of the matrix store the A, R, G and B components of that pixel. This is Jitter's convention for storing video.

Jitter also has a convention, used by all the objects in the GL group, for storing OpenGL-compatible image descriptions in a matrix. Like video data, these descriptions can be read, processed, and output by various objects. This document specifies Jitter's convention for storing OpenGL data in matrices and the syntax of messages to the **jit.gl.render** object containing these matrices.

When You Need This Reference

Objects in the GL group send out matrices in the format described here when they have their **matrixoutput** attributes set to 1. In order to process these geometries through your own patches, you will need to know this format. If you want to make geometries directly using the **jit.op** object and other matrix operators, you will also need to know the format. But if Jitter's GL objects are flexible enough for your drawing needs, you can continue to let these objects draw “behind the scenes” without any knowledge of the OpenGL matrix format.

GL Matrix Reference

Message Format

OpenGL data can be passed to the **jit.gl.render** object in Max messages of the forms

`jit_matrix [geometry matrix]`

`jit_matrix [geometry matrix] [draw primitive]`

and

`jit_matrix [geometry matrix] [connections matrix] [draw primitive].`

Draw Primitive

The draw primitive specifies how to draw the connections between vertices in the geometry matrix. When a draw primitive is not passed to the **jit.gl.render** object along with the geometry matrix, the current primitive of the **jit.gl.render** object is used. The current primitive can be changed by sending a message containing just the primitive. When a draw primitive accompanies the geometry matrix, it is used only to draw the given geometry and the current primitive is unchanged.

The draw primitive can be one of the following symbols:

<code>points</code>	Draws each vertex as a single point.
<code>lines</code>	Connects every other pair of vertices with a line. Given a geometry matrix with vertices A, B, C and D, it draws line segments AB and CD.
<code>line_strip</code>	Connects each pair of vertices with a line. Given a geometry matrix with vertices A, B, C and D, it draws line segments AB, BC and CD.
<code>line_loop</code>	Like <code>line_strip</code> but a line segment is drawn connecting the last vertex in the matrix back to the first one.
<code>triangles</code>	Draws unconnected triangles. Given vertices A B C D E F, draws triangles ABC and DEF.
<code>tri_strip</code>	Draws a strip of connected triangles. Given vertices A B C D E F, draws triangles ABC, CBD, CDE and EDF. Note the order of the vertices, which is swapped so that all the triangles face the same way (see Tutorial 33: Polygon Modes, Colors and Blending).

<code>tri_fan</code>	Draws a fan of triangles. Given vertices A B C D E F, draws triangles ABC, ACD, ADE and AEF.
<code>quads</code>	Draws unconnected quadrilaterals. Given vertices A B C D E F G H, draws quadrilaterals ABCD and EFGH.
<code>quad_strip</code>	Draws connected quadrilaterals. Given vertices A B C D E F, draws quadrilateral ABCD and CDFE.
<code>polygon</code>	Draws a single polygon using all the vertices in the matrix. If the polygon is not simple and convex, the results are undefined.
<code>tri_grid</code>	If the geometry matrix has two dimensions, triangles are drawn which link each vertex to its neighbors in the matrix to form a surface.
<code>quad_grid</code>	If the geometry matrix has two dimensions, quadrilaterals are drawn which link each vertex to its neighbors in the matrix to form a surface.

The Connections Matrix

A connections matrix must be one-dimensional and contain either *long* or *char* data. If present, it specifies the order in which to connect the vertices in the geometry matrix. By leaving the geometry matrix constant and changing the connections matrix which indexes into it, a changing set of connections between the same vertices can be drawn.

The Geometry Matrix

Geometry matrices must contain data in *long*, *float32*, or *float64* format, *float32* being the most common choice. *Float64* has more precision than is normally needed for rendering to the screen. The long data type restricts values to integers, which is generally not desired when specifying coordinates.

Each cell of the matrix represents one vertex. The image is rendered by drawing connections between vertices, either along the rows or the columns of the matrix. A one cell wide matrix will be connected along columns (its only column). A one cell high matrix will be connected along rows. If the matrix is more than one cell in both width and height, the `geom_rows` attribute of the **jit.gl.render** object determines whether rows or columns will be followed. If the `tri_grid` or `quad_grid` primitives are specified, both the rows and the columns of the matrix are always connected.

Matrices with anywhere from two to 13 planes can be used. Planes 13 and up are reserved for later use and will currently be ignored. Each plane represents a different value in one of five groups: vertices, texture coordinates, normals, vertex color and edges. If enough planes are present to specify a given group, that group is used in rendering the matrix. To

be used, each group must be totally present, with the exception of vertices. If only x and y values for vertices are present, the z coordinate of all vertices is set to 0, and an 2D image on the xy plane results.

plane 0:	x coordinate of vertex
plane 1:	y coordinate of vertex
plane 2:	z coordinate of vertex
plane 3:	s coordinate of texture at vertex, if textured
plane 4:	t coordinate of texture at vertex, if textured
plane 5:	x component of lighting normal at vertex
plane 6:	y component of lighting normal at vertex
plane 7:	z component of lighting normal at vertex
plane 8:	red component of vertex color
plane 9:	green component of vertex color
plane 10:	blue component of vertex color
plane 11:	alpha component of vertex color
plane 12:	edge flag for connection to next vertex: if < 0 , no line is drawn.

Appendix C: The JXS File Format

The Jitter Shader Description File

Jitter provides a means of encapsulating shader parameters and programs through an XML based shader description file (**JXS**). This **JXS** file offers an easy way of setting up any required OpenGL state to achieve a particular effect, without any additional patcher related work.

The **JXS** file may contain a list of parameters and textures, followed by a set of language dependent implementations for the shader's vertex and fragment programs.

Each language implementation must specify both a vertex and a fragment program. These programs may either reference external files, or be stored inline within a CDATA section or an XML comment. Programs for each language implementation will be loaded and bound in the order that they are listed in the **JXS** file. If a particular language implementation fails to compile, the next implementation will be used, until a successful candidate is found.

Inside of the language implementation declaration, any shader parameter previously specified must be identified with a bind tag that indicates which shader program should receive the parameter's values.

The following listing outlines the format of a typical **JXS** file:

```
<jittershader name="myshader">

<!-- optional description -->
<description>This is my shader</description>

<!-- optional list of texture objects to bind -->
<texture file="mytexture.jpg" />

<!-- optional list of shader parameters -->
<param="myparam" type="vec3" default="3.0 4.0 5.0">
<description>This is my parameter</description>
</param>

<!-- list of language implementations -->
<language type="glsl" version="1.0">

<!-- list of binding targets for shader parameters -->
<bind param="myparam" program="vp" />
```

```

<!-- vertex and fragment programs -->
<program name="vp" type="vertex" source="sh.passthru.xform.vp.glsl" />
<program name="fp" type="fragment">
<![CDATA[

// entry point
void main()
{
    gl_FragColor = vec4(0.5, 0.5, 0.5, 1.0);
}
]]>
</program>
</language>
</jittershader>

```

Note that the XML document must be well-formed with matching opening/closing tags.

JXS Tags

Any of the following **XML** tags may be used inside the **JXS** document:

jittershader	Defines the shader object. Valid attributes are:		
	symbol	name	The name of the shader.
	text.	description	Optional descriptive text.
texture	Defines a texture object to use. Any jit.gl.texture attributes can be specified (see the html reference for jit.gl.texture for a complete list).		

param	Defines a shader parameter. Valid attributes are:		
	symbol	name	The name of the shader.
	text	description	Optional descriptive text.
	symbol	type	The name of the datatype for the parameter values. Valid types are: int, float, vec2-4, float2-4, mat2-4
	list	default	The default values for the parameter. Values will be converted to parameter's datatype.
	symbol	transform	The type of matrix transform to perform. Only valid if type is mat2-4.

	symbol	state	<p>The type of GL state to automatically reference. Valid state binding symantics are:</p> <p>EMISSION</p> <p>CAMERA_POSITION</p> <p>CAMERA_DIRECTION</p> <p>LIGHT0-7_POSITION</p> <p>LIGHT0-7_DIRECTION</p> <p>LIGHT0-7_AMBIENT</p> <p>LIGHT0-7_MODELVIEW_MATRIX</p> <p>PROJECTION_MATRIX</p> <p>MODELVIEW_PROJECTION_MATRIX</p> <p>NORMAL_MATRIX</p> <p>TEXTURE_MATRIX</p> <p>TEXTURE0-7_MATRIX</p> <p>POSITION</p> <p>NORMAL</p> <p>TANGENT</p> <p>BITANGENT</p> <p>EDGEFLAG</p> <p>AMBIENT</p> <p>DIFFUSE</p> <p>SPECULAR</p> <p>_DIFFUSE</p> <p>LIGHT0-7_SPECULAR</p> <p>LIGHT0-7_CUTOFF</p> <p>LIGHT0-7_EXPONENT</p> <p>TEXDIM0-7</p>
language	<p>Defines a language implementation. Must contain bind targets for any previously specified shader parameter, as well as both a vertex and fragment program.</p>		
	symbol	name	The name of the shading language.

	float	version	The optional version identifier for the shading language.
bind	Defines a bind target within a language implementation to associate a parameter with a particular shader program.		
	symbol	param	The name of the parameter to associate.
	symbol	program	The name of the program to associate.
	symbol	target	The name of the binding target (register). Only needed for assembly programs. Valid targets are:env[0-256], local[0-256]
program	Defines a shader program within a language implementation.		
	symbol	name	The name of the shader program.
	symbol	source	The name of the external file containing the shader program source code. Only necessary if program is not defined within a CDATA section or an XML comment

Appendix D: Building Standalone Applications with Jitter

If you want to create a standalone application that uses Jitter, you will need to copy a number of files to your standalone application folder (for Windows applications) or include them in your Macintosh applications package. The appendix describes the files you will need to copy if you intend to include Jitter, Jitter shaders, Java, or Javascript support in your application.

For more information about building standalone applications, see the chapter on Collectives in the Max Tutorials and Topics Manual.

Windows Standalone Format

A standalone application on Windows is actually a folder containing MaxMSP Runtime executable plus a *.mxp* collective file containing your patches and files. In addition, the standalone folder contains a support folder with files necessary to run your application.

Jitter

To include Jitter components in your standalone application:

- Copy the file */Program Files/Cycling '74/MaxMSP 4.5/support/jitlib.dll* to {standalone folder}/support/jitlib.dll
- Copy the file */Program Files/Cycling '74/MaxMSP 4.5/support/cg.dll* to {standalone folder}/support/cg.dll
- Copy the file */Program Files/Cycling '74/MaxMSP 4.5/support/cgGL.dll* to {standalone folder}/support/cgGL.dll

Jitter Shader

To include Jitter shader components in your standalone application:

- Copy the file */Program Files/Common Files/Cycling '74/jitter-shaders* to {standalone folder}/support/jitter-shaders
- Copy the file */Program Files/Cycling '74/MaxMSP 4.5/support/cg.dll* to {standalone folder}/support/cg.dll

- Copy the file */Program Files/Cycling '74/MaxMSP 4.5/support/cgGL.dll* to *{standalone folder}/support/cgGL.dll*

Java

To include Java components in your standalone application:

- Copy the file */Program Files/Common Files/Cycling '74/java/lib/max.jar* to *{standalone folder}/support/java/lib/max.jar*
- Copy any other necessary jar files (e.g. jitter.jar) to *{standalone folder}/support/java/lib/max.jar*
- Copy */Program Files/Common Files/Cycling '74/java/classes/*.class* to *{standalone folder}/support/java/classes/*.class* (all your necessary class files)

Javascript

To include Javascript components in your standalone application:

- Copy the file */Program Files/Cycling '74/MaxMSP 4.5/support/js32mt.dll* to *{standalone folder}/support/js32mt.dll*
- Copy the file */Program Files/Common Files/Cycling '74/jsextensions/* to *{standalone folder}/support/jsextensions/*

Macintosh Standalone Format

On Macintosh, the standalone is an application package, which is also a folder that looks like a file in the Finder. You can peek into the package by control-clicking on it and choosing **Show Package Contents** from the contextual menu. As with the Windows standalone, inside the Contents/MacOS folder you will find a *.mxfl* collective, a runtime application, and a support folder.

Jitter

To include Jitter components in your standalone application:

- Copy the file */Applications/MaxMSP 4.5/jitlib* to *{Package}/Contents/MacOS/jitlib*

Jitter shader

To include Jitter shader components in your standalone application:

- Copy the file */Library/Application Support/Cycling '74/jitter-externals/Cg.framework* to *{Package}/Contents/MacOS/support/Cg.framework*
- Copy the file */Library/Application Support/Cycling '74/jitter-shaders* to *{Package}/Contents/MacOS/support/jitter-shaders*

Java

To include Java components in your standalone application:

- Copy the file */Library/Application Support/Cycling '74/java/lib/max.jar* to *{Package}/Contents/MacOS/support/java/lib/max.jar*
- Copy any other necessary jar files (e.g. *jitter.jar*) to *{Package}/Contents/MacOS/support/java/lib/*
- Copy */Library/Application Support/Cycling '74/java/classes/*.class* to *{Package}/Contents/MacOS/support/java/classes/*.class* (all your necessary class files)

Javascript

To include Javascript components in your standalone application:

- Copy the file */Library/CFMSupport/JSRefThreadsafe.shlb* to *{Package}/Contents/MacOS/JSRefThreadsafe.shlb*
- Copy the file */Library/Application Support/Cycling '74/jsextensions/* to *{Package}/Contents/MacOS/support/jsextensions/*

Jitter Bibliography: For Your Further Reading Pleasure

Jitter Reference List

The following is a list of books, articles, and web sites you might find helpful for learning more about different Jitter-related topics.

Video

Jim Blinn, *Jim Blinn's Corner: Dirty Pixels* (San Francisco: Morgan Kaufmann, 1998). ISBN: 1558604553

An excellent introduction to a diverse array of topics including analog and digital video formats, aliasing, compositing, tensor arithmetic, and the world's largest Easter Egg.

A. Bovik et al., *Handbook of Image and Video Processing* (San Diego: Academic Press, 2000). ISBN: 0121197921

Richard Rickitt, *Special Effects: The History and Technique* (London: Virgin Books, 2000). ISBN: 1852278005

A very readable introduction to the history and technology of special effects in film.

Image Processing

W. K. Pratt, *Digital Image Processing* (New York: John Wiley and Sons, 2001). ISBN: 0-471-85766-1

A. S. Glassner, *Principles of Digital Image Synthesis* (San Francisco: Morgan Kaufmann, 1995). ISBN: 1558602763

Video Codecs

Peter Symes, *Digital Video Compression* (New York: McGraw-Hill, 2003). ISBN: 0071424873

M. Ghanbari, *Standard Codecs: Image Compression to Advanced Video Coding* (IEEE Publishing: 2004). ISBN: 0852967101

OpenGL and 3D Graphics

E. Angel, *OpenGL 1.2: A Primer* (Reading: Addison-Wesley, 2001). ISBN: 0201741865

David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley, *Texturing and Modeling: a Procedural Approach* (AP Professional, July 1998). ISBN: 0122287606

This book gives the reader a working knowledge of several procedural texturing, modeling, and animation techniques, including two-dimensional texturing, solid texturing, hypertextures, volume density functions, and fractal algorithms. Emphasis is placed on practical applications with many code examples.

A. Watt, *3D Computer Graphics* (Essex: Addison-Wesley, 2000). ISBN: 0201398559

M. Woo et al., *OpenGL Programming Guide* (Reading: Addison-Wesley, 1999). ISBN: 0201461382

<http://www.sgi.com/grafica/>

Shaders

Randi J. Rost, *OpenGL Shading Language* (aka "The Orange Book") (Reading: Addison-Wesley, 2004). ISBN: 0321197895

Randima Fernando, Mark J. Kilgard , *Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics* (Reading: Addison-Wesley, 2003). ISBN: 0321194969

Randima Fernando, *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics* (Reading: Addison-Wesley, 2004). ISBN: 0321228324

Procedural Texturing and Modeling

Ebert, Musgrave, et. al., *Texturing & Modeling: A Procedural Approach* (San Francisco: Morgan Kaufman, 2002). ISBN: 1558608486

Anthony A. Apodaca and Larry Gritz, *Advanced RenderMan: Creating CGI for Motion Pictures* (San Francisco: Morgan Kaufman, 1999). ISBN: 1558606181

2D Graphics and Vector Animation

A. S. Glassner et al., *Graphics Gems* (San Diego: Academic Press, 1990). ISBN: 0122861655

<http://www.esil.univ-mrs.fr/~elingsys/cours/black-book/>

<http://www.faqs.org/faqs/graphics/>

Video Art

Gene Youngblood, *Expanded Cinema* (New York: Dutton, 1970). ISBN: 0525101527

An overview of "synaesthetic cinema" techniques in film and video with an introduction by R. Buckminster Fuller. Covers the work of the Whitneys, Carolee Schneeman, Jordan Belson, Stan Brakhage, Nam June Paik, Michael Snow and others. The first book on video and computer graphics as an art medium. An essential historical connection, unfortunately out of print.

Generative Art

<http://www.red3d.com/cwr/>

Linear Algebra and Mathematical Matrix Operations

G. Stephenson, *An Introduction to Matrices, Sets, and Groups for Science Students* (Toronto: Dover, 1986). ISBN: 0486650774

<http://archives.math.utk.edu/topics/linearAlgebra.html>

<http://www.math2.org/index.xml>

<http://mathworld.wolfram.com/>

<http://www.sosmath.com/matrix/matrix.html>

General Programming

David Flanagan, *Javascript: The Definitive Guide* (Cambridge: O'Reilly & Associates, 1998). ISBN: 1565923928

Colin Moock, *Actionscript for Flash MX: The Definitive Guide* (Cambridge: O'Reilly & Associates, 2001). ISBN: 059600396X

David Flanagan, *Java in a Nutshell* (Cambridge: O'Reilly & Associates, 2005). ISBN: 0596007736

Patrick Niemeyer and Jonathan Knudsen, *Learning Java* (Cambridge: O'Reilly & Associates, 2005). ISBN: 0596008732

Miscellaneous

Don Ritter, *E's for Artists: a handbook for New Media Artists* (forthcoming, publisher to be determined.)

Randall Packer and Ken Jordan, eds., *Multimedia: From Wagner to Virtual Reality* (New York: Norton, 2001). ISBN: 0393049795

Stephen Wolfram, *A New Kind of Science* (Champaign: Wolfram Media, 2001). ISBN: 1579550088

A wide-ranging exploration of the potential of cellular automata for solving problems in mathematics, biology, and theoretical physics. Most of the concepts and algorithms illustrated in this book could be easily realized using Jitter.

