

WORKING WITH VIDEO

As we learned earlier, Jitter is a library of about 140 Max objects oriented mostly towards working with video, still images, and 3D images. Jitter is useful for realtime video processing, as well as audio/visual interaction. Since it is optimized for these functions, it handles video data robustly and fast. Jitter is designed with extensive support for Apple's QuickTime architecture, both its video and its audio capabilities. While Jitter is best for handling video, QuickTime audio can be routed into MSP functions for audio processing. Jitter supports live digital video (DV) camera control as well as output via multiple monitors, data projectors, and FireWire devices; the output components only work on the Macintosh in the version we are working with.

MATRICES

Jitter video handling is based in a structure called a matrix. We'll learn a bit about matrices, but it's a subject you can delve into in more depth on your own. As with other kinds of numerical handling in Max, Jitter matrixes can store several different kinds of information: namely, **char** (8 bit unsigned int), **long** (32 bit signed int), **float32** (32 bit floating point), or **float64** (64 bit floating point). Matrices can have up to 32 dimensions, and up to 32 planes.

One of the things matrices make possible is what is called **transcoding** of information, that is, taking information of one kind (e.g. text) and converting it to a different kind of output (e.g. sound); or video as audio; or audio as text.

Familiar example: the sentence A BEADED BAG FED A FADED FACE can be output very easily as music in familiar western notation because the letters are all used in both systems. With a few extra rules, any English sentence can be transcoded into music. For instance, A B C D E F G (h i j) (k l) (m n o) (p q) (r s t) (u v w) (x y z) all the other letters can be assigned to one of the primary 7; or to sharps and flats; or to different octaves; or to different rhythms (quarter vs half note).

PLAY RAT BRAIN MUSIC HERE.

A **matrix** is most easily thought of as a **grid**, usually of two dimensions or more. (A one-dimensional array is what programmers call an **array**, which is basically just a **list**.) We'll start with 2D grids; think of a chessboard. The crucial elements of a grid are these:

1. each location (cell) in the grid contains information.
2. each location has a unique address.



In chess, as those of you who play know, different forms of address notation are used, like "KB3" to designate one location (King's Bishop 3). We'll use plain alpha-numeric addressing for our examples. In the above example, there is a white pawn "stored" at 4e. Now, note that the information stored at 4e is multidimensional: it is an object that we understand to be a "white" "pawn". Digital matrixes store much simpler information, usually numbers. Thus, there must also be a **translation table** as part of any program that tells the computer what to do with the information at location 4e. In the chess example, the information at location 4e is interpreted through the rules of the game (what a "white pawn" is and what it can do). But now suppose we store the number 11 at location 4e in our digital matrix. One possible translation table in computer program X is as follows:

- 01 pawn
- 02 knight
- 03 bishop
- 04 rook
- 05 king
- 06 queen
- 07 white pawn
- 08 black pawn
- 09 white knight
- 10 black knight etc.
- 11 white bishop**
- 12 black bishop etc.

Another very different translation table in computer program Y would work thus:

- 1-6: the base pieces
- initial modifier 0: no effect
- initial modifier 1: white
- initial modifier 2: black
- then **11 = white pawn**, while 22 = black knight, etc.

So you can see that the two programs will generate very different information from the same stored number (11). And you can see also that the transcoding I referred to earlier is not something exotic, but an affordance of the way computers store numbers and the way computer programs work.

VIDEO MATRIX

Matrixes matter in Jitter, because the program stores the information about what is visible on your computer screen through matrixes. A screen is made up of a number of pixels, say $1024 \times 768 = 786,432$ individual cells or locations. And each of these pixels has a color that is determined by three values signifying the precise mixture of red, blue and, green light on a scale of 0-255. Jitter, like Photoshop, stores a 4th piece of information which is the equivalent of Photoshop's alpha channel and is used similarly for masking and mixing effects (transparency/opacity).

(Note that there is more information about the difference between computers and tv monitors, and between different aspect ratios, in the Jitter Tutorial Manual.)

Jitter stores all four numbers in a 2-dimensional matrix. Each cell of the matrix holds all 4 ARGB values in a strict order.

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.

For example, imagine that cell 0,0 in the matrix corresponds to the top leftmost pixel on your computer screen. In that cell is the following number: 000255100110. This can be parsed as 000A, 255R, 100G, 110B. The Max/Jitter program knows that the first 3 digits of the 12-digit number stored in this array will always be the alpha value for that pixel. This organization makes it easy to change, say, all the red values in the array by a certain amount, making the overall image more or less red.

Jitter refers to these multiple bits of information stored within the video matrix as **planes**, i.e. the alpha plane is the alpha information across the entire matrix. Each plane has its own number; alpha is 0, red 1, green 2, blue 3.

DATA TYPES

As I said earlier, Max/Jitter can handle several data types, including integers and fractions. Programmers prefer to store the most compact kind of data possible, to save on memory. A fraction takes much more memory space than an integer. Now, as we saw, video requires four 3-digit numbers for each pixel, and each of those numbers falls in the 0-255 range. That is, they are 8-bit numbers. That is, it only takes 3-digit numbers to encode the millions of possible colors of a computer screen. This is very compact.

Max/Jitter calls such 8-bit numbers **char** (because the set of ASCII characters can be represented in just 8 bits as well). Because video manipulation is a primary function of Jitter programs, most Jitter matrix objects used the **char** storage type as their default. Even though the values being stored in Jitter patches are usually numeric (not alphabetic characters), we only need 256 different possible values for each one, as we saw above, so the 8 bits of a char are sufficient.

PLAYING A QUICKTIME VIDEO

If you haven't already done so, launch MAX now so you can follow along.

To play a QuickTime Video, we will need two objects. The first is the **jit.window** object, which automatically opens a window on your computer screen. The second is the **jit.qt.movie** object, which allows you to open and play a QuickTime movie.

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.

By default a **jit.qt.movie** object will begin playing a movie as soon as it opens it. (You can change the default by sending a **jit.qt.movie** object an `autostart 0` message before opening the file.)

Notice, however, that even though I've said that the **jit.qt.movie** object is playing the movie, the movie is not visible in the Movie window. Here's why: Each object in Jitter does a particular task. What we 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. Most Jitter objects send out a `jit_matrix` message when they receive the message *outputmatrix* or *bang*, or when they have themselves have received such a message and have modified their matrix data in some way.

In other words here is the flow:

`bang >> into Jitter object inlet >> object sends jit_matrix message out its outlet`

`jit_matrix message >> into Jitter object inlet >> object does something to matrix >> sends new jit_matrix message out its outlet`

Video in Max/MSP/Jitter

In the tutorial example, clicking the "read" message causes the `jit.qt.movie` object to read the QuickTime movie into RAM, frame by frame. However, there is as yet no message passed to the `jit.window` object telling it to display the frames in RAM. To do that, `jit.qt.movie` needs to receive a bang.

Test the difference between clicking the bang right above the `jit.qt.movie` object, and clicking the toggle right above the metro object. The first sends a command to display just the frame currently in RAM. The metro with its repeated bangs, reads frames in succession.

Note what happens when you change the metro attribute to 500 (half second).

From the Jitter Tutorial Manual:

"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.... 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."