Messages and Structure in Max Patches

This essay addresses some of the larger issues of design in Max/MSP/Jitter. Most tutorials cover specific problems and how to solve them– this essay examines the issues involved from a distant perspective, and will discuss the structural elements required in all patches without going into details of specific applications. I will also discuss features often lacking in patches but essential to long term utility: these are robustness, and maintainability.

Every Max patch consists of object boxes connected by lines. We will look first at what the lines represent.

The Concept of Message

One essential difference between Max and other programming languages[1] is that all action is based on the passing of messages. Most other languages have a list of instructions that are executed in an endless loop. This means code is loaded and unloaded even when there is nothing to do. If the program is well written, the looped code mostly checks for the need for other code to run, but even this takes time. A worse problem is that such programs work as a polling system, meaning that events in the outside world do not affect the program until the loop gets around to looking for them.

What does it mean to pass a message? In the typical language, when a piece of code finishes, any results are parked in memory (in locations called "variables") and sit there until the program loop calls a routine that is interested in these results. In Max, the code resides in objects. When an object produces a result, a new message containing the result is passed to the next object in line. More importantly, the receiving object is activated immediately to process the message. This means any event triggers a cascade of calculations which for the most part execute at the computer's maximum speed. No extraneous code is invoked, only routines relating to the message path are run.

The Content of Messages

The possibilities of Max are ultimately limited by the possible content of the messages. One of the first things we need to learn in our mastery of Max is the types of message available.

Bangs

As Chris Dobrian wrote in the very first Max tutorial, bang means "do it". There is no new information in a bang, only an order to execute code. All objects respond to a bang somehow; if nothing else, they repeat the last output. The obvious function of bang messages is to control timing. There are a variety of objects that create bangs in predictable rhythms, and we often use bangs to determine the order of operation within a complex patch. Max has no concept of a rest– there is only time between bangs.

[1] Most other languages-- any language you are likely to study, anyway.

Numbers

There are two types of number in Max-- floating point numbers, which have a decimal point, and integers, which do not. These are generally called ints[2] and floats. The distinction is left over from the not so distant days of computing, when computation with floating point numbers took about 1000 times as long as computation with integers. The distinction is hardly necessary in modern processors, but is continued to provide backwards compatibility[3]. Translation from one number type to the other is more or less effortless, but when a float is converted to an integer the fractional part is lost. A more subtle and often troublesome aspect of this is that some of the older Max objects take their operational mode from the type of the arguments. If there is no decimal point in the argument (or no argument), the math will be done with only integral precision. Thus 3 / 2 = 1, whereas 3 / 2.0 = 1.5. This type of problem is high on the list of most common gotchas[4] in Max.

Numbers are currently represented by 32 bits[5]. This means they can represent 4,294,967,295 distinct values. An integer can represent values from -2,147,483,648 to 2,147,483,647. Floating point numbers range from 10^{-308} to 10^{308} , but many numbers are skipped within that range. Most of the numbers in the mid-range are available, but very large ones are only approximated. Small numbers skip around in a peculiar way. Only those that are some power of two can be represented exactly. This means that 0.2 is really something like 0.20000001. Max will usually display that as 0.2, but eventually the difference shows up.

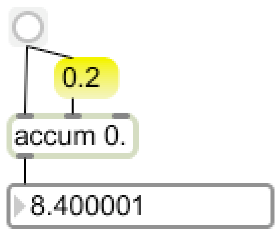


Figure 1.

Jitter introduced a new number type known as char. This is actually the most ancient number type, the unsigned 8 bit number or byte. The name char comes from a common use of these to represent printable characters[6]. A char can represent 256 values, ranging from 0 to 255. The utility of such a puny number may seem limited, but it's probably the most common type. This is based on the nature of computer memory (and other

[2] Integers may also be called longs- this refers back to a distinction between 16 bit ints and 32 bit longints. Floats are distinguished as float32 and "double" or float64.
[3] Not to mention old habits of thought.
[4] A gotcha is defined in Wikipedia as "an unexpected, or unintuitive, but documented, behavior ins a computer system (as opposed to a bug)."
[5] Soon to be 64.
[6] In the days when computers only knew English.

hardware) which is usually built in modules of bytes. Thus a byte is used to represent the intensity of a red pixel in a computer display. Chars only exist inside jitter matrices. When a value is extracted by getcell, it becomes an int. In some situations, Jitter objects actually convert chars into floats ranging from 0.0 to 1.0.

Symbols

When Max was originally invented, most programs used a string of chars to represent text. Max does this at the deepest level, but the passing of long text strings is pretty inefficient. Strings are represented in Max messages by symbols. Every time you enter a new word into a patch, Max adds the word to a master list of symbols and assigns a unique number to it. This number is passed in the message. Objects have access to the symbol table and can reconstruct the original string if necessary. The advantage of this approach is the speed with which symbols can be passed around. The downside is the ever-growing symbol table, although with multi-gigabyte memories, it's a minor problem. Since Max uses spaces to determine what characters to include in a symbol, you need to type quotes around phrases that include spaces. We most often encounter this with file names. Symbols make Max kind of awkward for writing poetry. If you need serious manipulation of text, you should use spell and related objects to manage lists of numbers that are interpreted as ASCII characters.

The fundamental use of symbols is to define messages. The bang message is literally the symbol "bang", integers are prefaced with "int" and lists are prefaced with the word "list". You seldom[7] see these, as Max hides the standard message indicators. Other symbols, such as "open" define messages for specific objects. If you send a symbol to an object that has no code to respond to it, you will see an error in the Max window. Many symbols are followed by one or more numbers, which are the arguments of the message.

Lists

A list is a single message containing several numbers and/or symbols. Originally, a list was required to begin with a number, but that requirement has been relaxed somewhat in recent years. A list that begins with a symbol is a message for most objects, but some will accept anything without complaint. The length of lists is limited by most objects: 256 in some cases, 2048 in others, endless in a few[8]. This has been a source of constant complaints in the Max user community. Someone always seemed to need a list twice as long as the current standard. Long lists are awkward to manipulate, and are essentially replaced by Jitter matrices.

Objects without a defined list method will process the members as if they were received in consecutive inlets. A common gotcha is to accidentally send a list to a math object. This will (for instance) add the first two members, but it also leaves the internal operand of the object set to the second member of the list.

[7] They show up occasionally, including that particularly enigmatic symbol "symbol".
[8] It was 64 in the very beginning.

Signals

Signals are not really messages at all. The patch represented by the striped lines is actually set up independently of all regular Max objects and runs continuously. You could almost consider it a separate program. The most important concept to be aware of is the signal vector size. Samples are handled in small batches called vectors. An MSP object receives a vector and will deal with it in one uninterruptible process. The only time an MSP object can interact with the regular Max world is in the space between vectors. Thus small vectors will make audio patches more responsive. They also make audio processing less efficient, and at some point the CPU will not be able to keep up and you will hear alarming crackles and pops.

Any MSP object can interpret Max messages, using floats to set frequency and so forth, but the only road back is through certain hybrid objects such as sah~ and number~. Note that some of these objects need a bang to prompt output, others output periodically, as if they had an internal source of bangs.

Matrices

Matrices are massive collections of data. Matrices are not actually passed around. The matrix message produced by jitter objects is the name of a matrix for the following object to operate on. Since this process often includes copying the entire matrix, it is important to only send them when necessary. Any right or middle inlet will "remember" the last matrix applied (just as regular objects do), so avoid routinely banging all matrices in your patch.

Objects and Messages

Every object has a specific set of messages it can respond to. In some cases, Max will not let you connect an inappropriate message type. For instance, you cannot connect a metro to the right inlet of an add object. The main convention of patching is that the left inlet of the object triggers output. Anything received in the other inlets can be processed internally, but there should be nothing passed down the chain until the left inlet is activated. There are of course exceptions such as pak, but the general rule is input in left triggers output[9].

Of course, the first message must originate somewhere. There are really only three types of object that can start the chain of computations. These are user input objects, system input objects, and scheduled objects.

User Input Objects

Users can interact directly with Max patches by manipulating certain objects with the mouse. There are quite a few UI objects, each with unique look and action. In most cases, UI objects produce numbers as their output, but bangs and arbitrary messages are also available. As far as the end user is concerned, good patch design consists of choosing appropriate UI objects and giving them informative labels and a rational layout.

[9] Since MSP objects run continuously, there is no left/right inlet distinction.

When the user manipulates some control, say a slider, messages are generated at fairly even time intervals, which means the faster the mouse moves, the fewer messages are generated. If a slider is moved from 0 to 127, many of the intermediate values will be skipped. Keep this in mind if you are watching for specific values.

The initial value a UI object will output is somewhat unpredictable. It will display 0 when the patch is opened, but no message of 0 is generated and following objects will be in their default state. This leads to a common gotcha. A patch may work fine to start with, but once closed, it will refuse to work again until every control has been operated. The cure is a liberal use of loadbang and loadmess to initialize all UI objects to default values[10].

System Input Objects

Several types of objects connect with the operating system and respond to data sent to the computer. These include the various MIDI inputs, the key and text objects (for typed input), serial, hi and udpreceive for input ports, and inputs for audio and video.

The MIDI objects have privileged status. If the overdrive preference is on, incoming MIDI data will generate Max messages immediately, interrupting any other action that may be going on. This privilege derives from the musical heritage of Max, but it is made possible by the nature of the computer hardware. Usually this is unimportant, but if you are building a graphic patch that uses MIDI for outside control, you may want to turn overdrive off to get a consistent frame rate.

Audio and video inputs arrive periodically according to their format. For audio, the controlling factor is the I/O vector size, which is generally larger than the signal vector size. When an audio vector turns up, all other processing ceases until it is dealt with. The I/O vector size affects latency. Smaller vectors are more responsive, but the practical limit is established by the disc access rate. Video arrives in frames at a relatively leisurely rate, but you will find input hardware differs significantly in the lag between real time and Jitter input. Video frames are only grabbed when the appropriate jitter object is banged.

Other types of input are polled periodically, often by bangs. The necessity for polling is built into the hardware. It's generally a good idea to tweak the polling rate to match the specific situation. If you poll too fast, you may get garbled data.

Scheduled Objects

Scheduled objects can create a message at a specified time. The archetypical scheduled object is metro, which produces a bang when it is turned on, then at intervals of specified milliseconds. There are two variants of metro. The normal object produces a bang that interrupts all other computation. As with MIDI overdrive, this is designed to maintain musical accuracy. The bang produced by qmetro is less pushy. It is held back (queued)

[10] This can also be managed with preset and pattr.

until all high priority activity has completed. If the computations triggered by a metro take longer than the period of the metro (not uncommon with Jitter), new bangs will continually interrupt the last one and the process will never finish. Max will lock up. The only way out of this situation is to force quit Max, losing unsaved changes in your patch. This is a major gotcha.

It is easy to forget that line is a scheduled object. Line has two associated time intervals: the ramp time is the overall time to target, and the grain time is the interval between outputs. The grain time should be set as long as possible while producing a smooth output. There is no point in outputting duplicate numbers from a very slow ramp, or in repeating a calculation more than once per video frame.

Message order in a Patch

I have already mentioned that execution of an object's code is triggered by the arrival of a message in the left inlet. This implies that any other data required must have arrived in the other inlets[11] before the triggering message. Usually, the designer of an object will have given this careful consideration when deciding which part of the computation should be the trigger for output. In some cases, there are near duplicate objects– for instance the / object and !/ objects both do division, but differ in whether the divisor or dividend is the trigger. A simple guiding principle will work here: the data that changes most often should trigger the action.

Right Outlets First

The default order of message passing helps data arrive in the proper order. It's another simple rule: all objects with multiple outlets shall output data from right before left. Thus notein with outlets for (left to right) note number, note velocity and channel will output channel, then velocity, then note number. The calculations that depend on channel and velocity can be made before the note number triggers the action. It is not a coincidence that noteout has inlets that match the notein outlets in order.

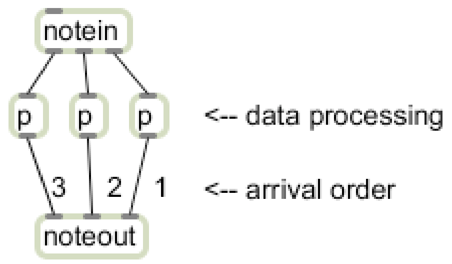


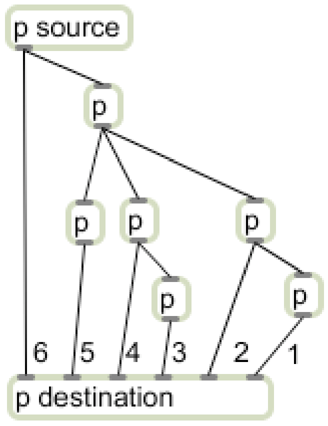
Figure 2.

There are exceptions to the right to left outlet rule. The most common are with objects that don't necessarily fire all outlets, such as select and route. But if two outlets do fire, the right will be first.

[11] If an object has more than two inlets, the order in which data arrives in the non-left inlets does not matter.

Right Destinations First

The second rule of message passing also enforces proper order. If two cords are connected to the same outlet, the cord with the rightmost destination transmits its message first. This includes all calculations that are triggered from that message. Thus the messages that are derived from the source in figure 3a will arrive in the proper order as indicated by the numbers[12].



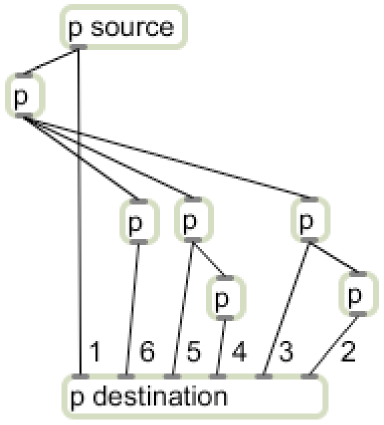


Figure 3a Figure 3b.

This only works out if the lines do not cross. In figure 3b, an object has strayed too far left with disastrous consequences. The parameters are set after the output has been triggered. If this is a music application, everything is calculated one beat late.

You would be amazed how often message order gets scrambled in complex patches. We can guarantee the order of message passing by using trigger objects as shown in figure 4. The trigger object, which is usually abbreviated t, requires a token argument for each outlet desired. The token determines the type of message to output from each: b for bang, i for int, f for float, s for symbol[13], or l for list.

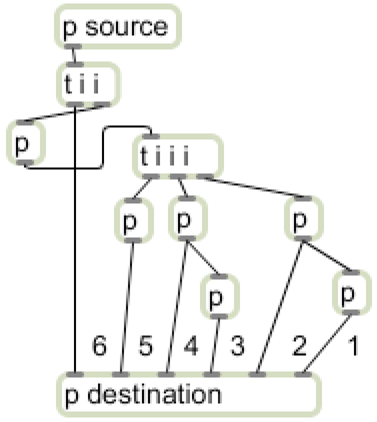


Figure 4.

[12] Of course when you encapsulate part of your patch like this, you are free to violate the left triggers output rule. Do yourself a favor and follow the rule religiously.

[13] It will only pass a symbol, not create a new one. You can also specify a specific symbol to send no matter what comes in.

Once triggers are in place, the patch will survive nearly any mangling. Inadvertently moving an object and getting the messages out of order is a major gotcha. A good rule of thumb: if an outlet has more than two cords attached, use a trigger.

In cases where the data arrival conflicts with the object design, we must artificially trigger the calculations. For instance, we may wish to plot a circle by feeding a series of angles to the poltocar object while keeping the radius constant. Unfortunately, poltocar requires the radius in the left and angle in the right. Figure 5a shows one method of dealing with this. When a new radius is entered, the value is passed to the first trigger and is immediately output from the right outlet. The message uses this value to trigger the poltocar object with an initial angle of 0.

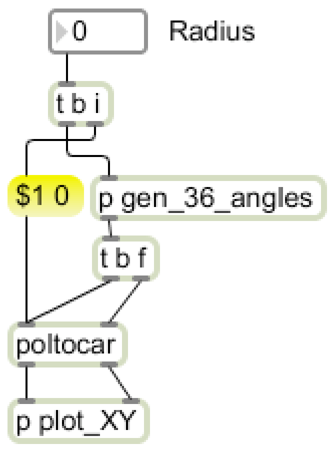


Figure 5a.

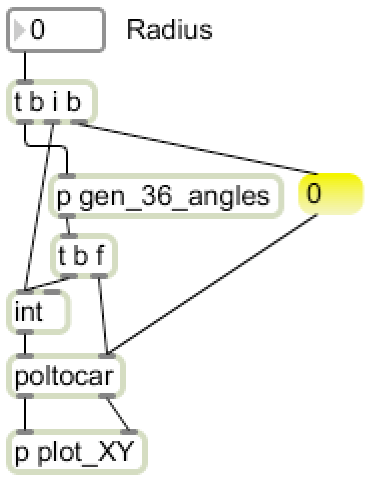


Figure 5b.

The subpatcher labeled gen_36_angles does what the label implies. A bang input triggers a sequence of angles to sweep around a circle. The following trigger applies each to the right inlet of poltocar, then bangs the left. A bang to poltocar will produce output with the most recent data, which is the new angle and the original radius.

Not all objects will recalculate when banged. Figure 5b illustrates how the problem would be addressed if poltocar did not. The radius message is stored in an int object and repeated as needed[14].

Send and Receive

Send and receive pairs are used to transmit messages without a connecting patch cord. A send may address as many receives as you like and a receive may be fed by an unlimited number of sends. Send is usually abbreviated s and receive is abbreviated r. There are many situations where they solve intractable problems, but I find them most useful in clarifying timing. For instance, I have built many patches with the general structure shown in figure 6.

[14] The int and float objects perform the function served by variables in text based languages-- they provide a temporary place to store numbers.

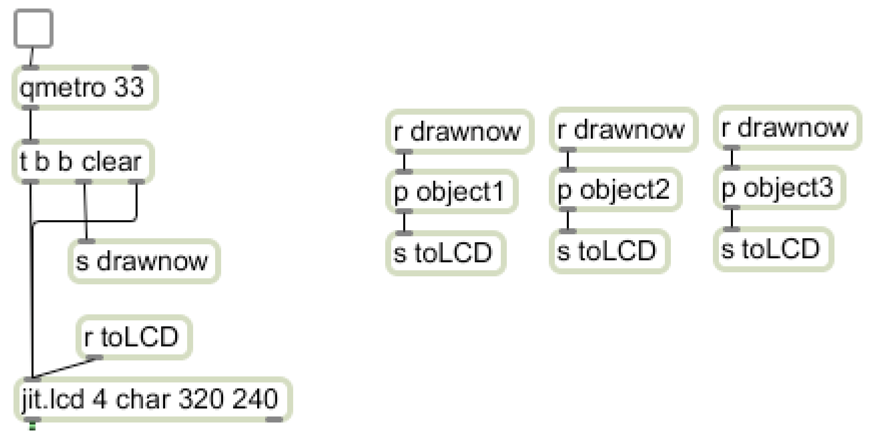


Figure 6.

In figure 6, the driving object is the qmetro. This commands a trigger object to generate a clear message and two bangs. The clear is applied to the jit.lcd, as is the second bang. All drawing commands for jit.lcd must be issued in the interval between the clear and the bang. The intervening bang is sent under the name drawnow to as many draw routines as I like. The actual commands from the drawing routines are returned via the toLCD receive object. No matter how complex this drawing becomes, the update order is correct, because all consequences of the send drawnow are completed before the final bang is emitted from the trigger object.

Figure 6 does not control which of the three object subpatchers will be executed first. The order in which receive objects get a message from a common send is not predictable. Even if a patch with several receive objects seems to behave in a consistent way, that behavior may change when the patch is saved and reopened. If the order of actions triggered by a send object is important, it must be explicitly defined– either by using multiple sends or by a common receive and trigger for critical sections. Figure 7 illustrates.

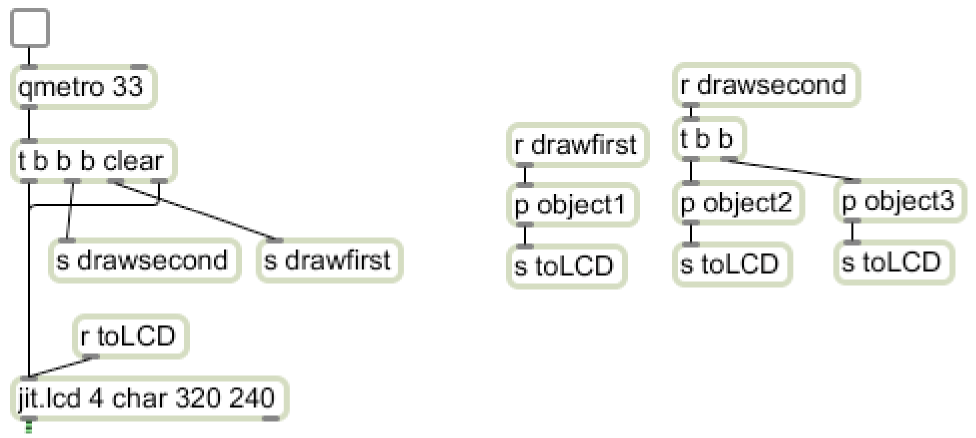
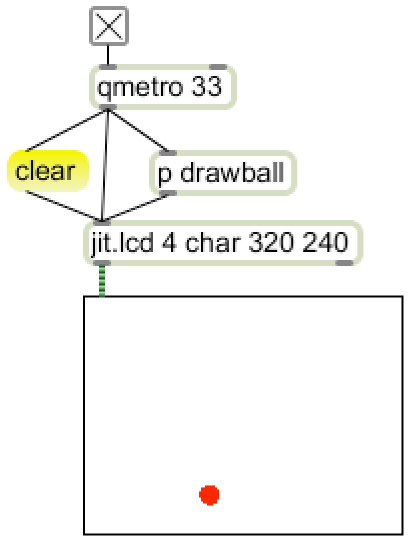


Figure 7.

You should use some care in the naming of send and receive pairs. The names should describe the associated action in a fairly specific way. I use the construction send to_something fairly often, even though it means the matching receive is grammatically awkward. I use it when there are multiple sends for one receive. Likewise, a single send from_somewhere will usually match several receives. It's also a good idea to use unique send and receive names in each patch. The habit of repeating send names is easy to get into, especially when one patch is derived from another. This can cause problems when more than one patch are open at the same time. I won't go so far as to recommend some arbitrary and unwieldy naming scheme, but something derived from the patch name and associated action should do. If the patcher of figure 7 were called lines, the name linesDraw2 might be appropriate for a send.

Scheduling and Deference



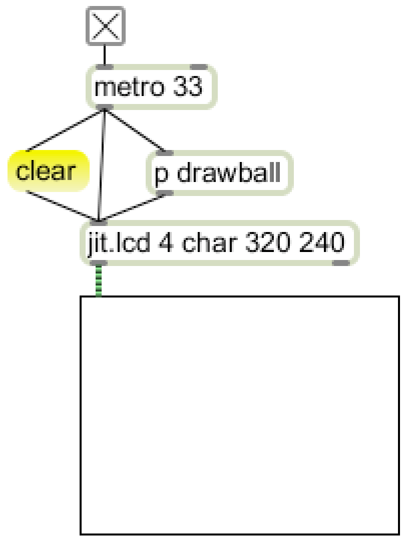


Figure 8a Figure 8b

The patches shown in figure 8 illustrate a particularly perplexing feature of Max. I often use patches like these to illustrate the right to left ordering phenomena discussed with figure 3. By moving the clear message to the left or right I can instantly break or repair the patch, an effect students usually find dramatic. Of course the proper and usual place for the clear message is to the right of the drawball subpatch-- if it is between the subpatch and the jit.lcd inlet, neither version works. Surprisingly, the leftmost location works when the source of bangs is a qmetro, but not with a metro. The reason why gives a peek into a realm of extremely knotty timing issues.

I have already mentioned that Max places a higher priority on some activities than others. If overdrive is on, MIDI input and metro bangs can actually interrupt other things that are going on. When it is off, computations in progress are allowed to finish before the MIDI data is processed-- MIDI is deferred for a bit. The mechanism that manages this is called the scheduler, and it obviously must maintain a list of things to do– this kind of list is called a queue in computer-speak. MIDI is still pretty special when overdrive is off, so MIDI and metro inspired actions may be deferred, but are placed at the head of the queue.

What goes on the end of the queue? User actions, screen updates[15], and anything to do with qmetro. This explains figure 8. The drawing of jit.pwindow is always one of the last things on the queue. The actual action of the redraw consists of looking for the matrix that was most recently passed in (by name) and transferring that data to a buffer for the operating system. The clear message resets the output matrix of jit.lcd to the background color. When the clear message is connected to a metro, it will always execute before the jit.pwindow update, so all jit.pwindow gets is a blank. When the clear message is connected to qmetro, it takes its turn after the pwindow update.

The moral of all of this is that there is a definite pecking order to Max actions. In broad terms, it goes like this:

- Audio processing interrupts everything.
- Midi and metro can interrupt most actions if overdrive is on.
- MIDI and metro actions go to the head of the queue if overdrive is off.
- User actions, screen updates and qmetro actions always go to the end of the queue.
- File actions have an even lower priority.

Note that user action is a higher priority than screen redraws. If you change a number box, all of the code connected to the number box will execute before the number in the box changes visibly. If that code is time consuming, the number box (and everything else that is visible) will freeze. If this is a problem, use a deferlow object to lower the priority of this particular action. There is also a plain defer object that will move processes from interrupt level to the head of the queue if overdrive is on. This may be useful if overdrive is needed for some types of MIDI message such as notes, but not others, such as control changes that interact with graphics.

The Structure of a Simple Patch

The rules of message passing help determine the basic layout of all Max patches. If everything is visible in one window, you should see these tendencies:

- The instigator of actions should be in the upper left corner. This would be a metro object, or notein or something similar.
- All output objects should be at the bottom of the page.
- Code that processes input will be at the left of page.
- Code that sets up processes (including most UI processing) will be toward the right of the page.

Figure 9 illustrates these in a generic way. Note that I said tendencies, not absolutes. The actual placement of things on a page is unimportant if you are using trigger objects, but if

[15] This is actually only a notification to the operating system that the program window needs updating. The actual screen refresh rate is something between 30 and 120 times a second, which is why there's no point in running Jitter routines too often.

you stay close to this layout the patch will be easy to read[16] and maintain. It is not necessary to hide all of the code in subpatchers. These may just represent groups of objects that are closely associated.

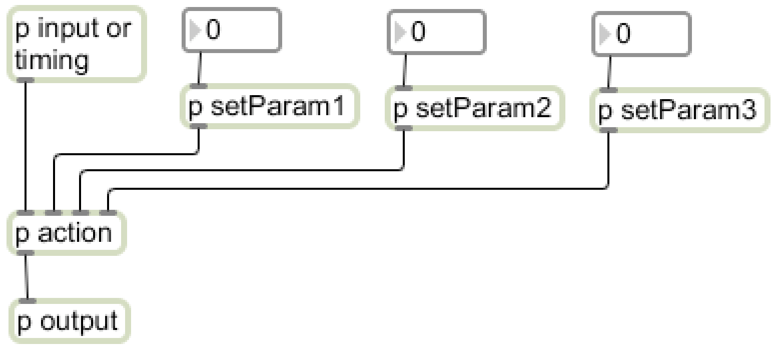


Figure 9.

The contents of the metaphorical subpatchers bear a strong similarity to the patch overview. (Figure 10.) The input is at the left of the patch, with occasional actions and initial setup toward the right. And yes, the similarity extends deeper, to the subpatches of the subpatches. This means the basic structure of Max patching is fractal– any level you look at will be similar to the level above.

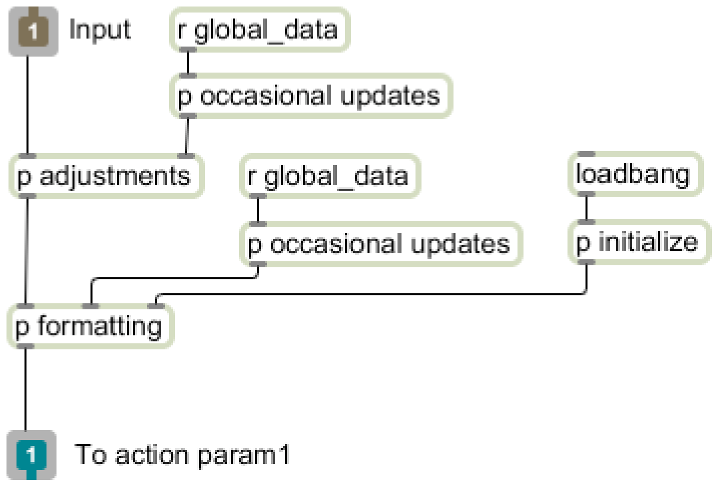


Figure 10.

Approaches to Complex Patches

Of course the model of one input, one result, is only sufficient about half the time. Many applications are more complex than that. As we encounter in most aspects of life, complex patches are a combination of simple patches. The most common combinations

[16] I always say you should make your work understandable by a stranger. The stranger most likely to look at your patches is you, in about six months.

involve doing something several times, or choosing and performing one of several options.

Branching

There are many situations in which a decision must be made and action taken based on that decision. The process is best illustrated by an old fashioned programming flowchart. Figure 11 illustrates the "branch", a basic element which chooses alterative actions.

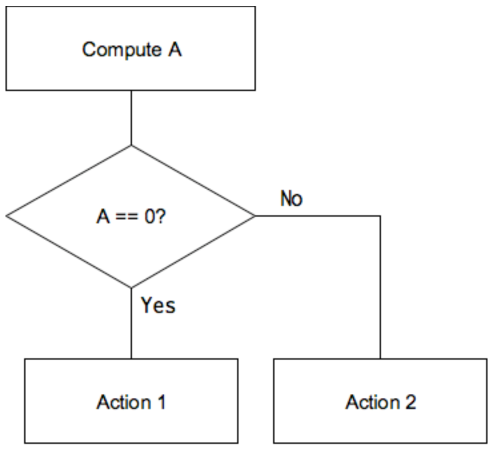


Figure 11.

The first step to making a decision is to compute the basis for the decision. Next, a comparison is made. If the comparison comes up true, some action is taken. If the comparison comes up false, a different action is taken. Comparisons in Max are usually based on numerical relationships:

==	equal[17]
!=	not equal
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal.

Figure 12.

There are objects that perform each test, comparing the input to the stored operand. The output of such objects is 1 if true and 0 if false. It is important to remember that something is output every time there is an input message. If there is a series of false tests, the result will constantly, show 0, which can lull the unwary into thinking nothing is going on. Nonetheless, the output is a steady parade of 0s. It is often a good idea to follow the comparison with a change object to avoid repeated actions.

Comparison objects are sensitive to integer and float arguments. If there is no decimal point in the argument, input is converted to integer form before the comparison is made.

[17] Yes, two equal signs. In programming, a single = usually means something else.

Thus 0.5 > 0 is false, whereas 0.5 > 0. is true. Another comparison gotcha is trying to match a float number with ==. If the numbers are generated by user actions, the specific match value may be skipped. You should always use <= or >= when comparing floats.

Select

The select (abbreviated sel) object uses == to perform comparisons. When the input is equal to an argument, the outlet associated with the argument bangs. This is a very efficient way to detect particular values out of a group of possibilities[18]. Select can match symbols as well as numbers. You often see select 1 attached to a comparison object to trigger action when the comparison is true.

If

The if... then statement pair is the basic decision mechanism of most languages. Figure 18 shows the Max version.

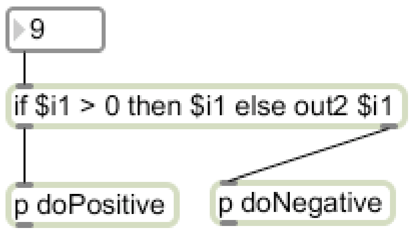


Figure 13.
There is a comparison after the word if. If this comparison comes up true, the message after the word then is output. If false, the message after the word else is output. Either output can be steered to the right outlet by the word out2. Note that the only math in an if statement is the comparison (which can be quite complex and involve different inlets). No calculation is done in the output side of the statement, except the tokens $i1, $i2 represent the input from different inlets. The i in $i1 converts input to an integer, $f1 will calculate with a float.

Complex if conditions can include Boolean operations on the outcome of two of more comparisons. (Use parentheses to group the operations.) The Boolean operators that are useful here are AND and OR. AND returns true if both of its arguments are true. OR returns true if either of its arguments are true. The operators are two ampersands (&&) for AND or two pipes (||)for OR. If you forget and use just one, entirely different math operations are performed.

[18] Experienced programmers will recognize select as a case function.

Op	in1	in2	result
AND &&	0	0	0
	1	0	0
	0	1	0
	1	1	1
OR ||	0	0	0
	1	0	1
	0	1	1
	1	1	1

Figure 14.

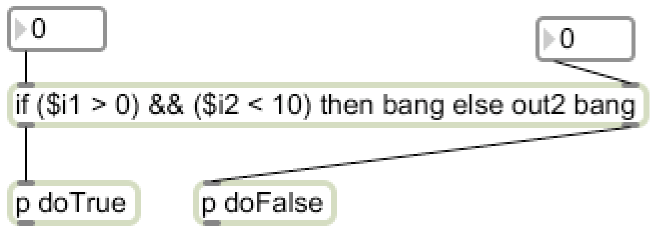


Figure 15.

Repetition

There are many situations where a single input must create multiple outputs. An example is the angle generator of figure 5, which must produce a series of numbers. The simplest way to supply a few values is with a divided message. Any of the buttons in figure 16 will produce five messages on the same scheduler tick.

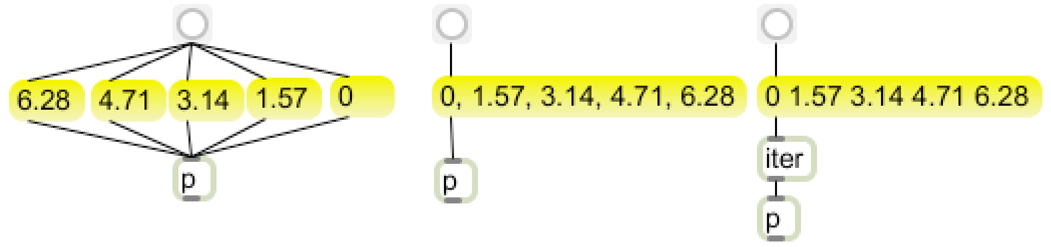


Figure 16.

This trick is reliable and effective as long as the number of actions is small. The first variant usually arises gradually, as the development of a patch reveals more iterations of some process are needed. The message with commas is compact and easy to read. Iter is most useful when the data coming in is changeable. When the number of repetitions is large, or the output is based on calculation with several variables, we usually turn to uzi.

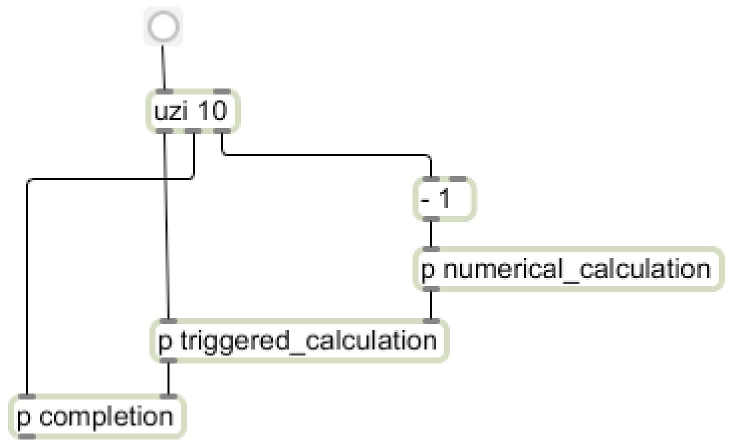


Figure 17.

Uzi is an engine that drives repetition.[19] The output of uzi consists of cycles– a number from the right outlet followed by a bang from the left. The cycle is repeated as many times as the argument specifies. When all cycles are complete, the center outlet bangs. This violates the strict right to left rule, but no harm seems to come of it. Uzi runs as fast as possible, but all calculations connected to the right outlet will complete before the left outlet bangs.

The three outlets of uzi enable the most common applications of looped code.

Numerical calculations depend on the "index" which is the number of the current iteration. For instance, to calculate 36 angles, the index would be multiplied by 0.17453. The index numbers provided by uzi begin with 1 and include the number in the argument. The majority of numerical series that are calculated from an index begin with 0, so the - 1 object on right output is usually required. This means the last value will be the argument minus 1, but the argument still sets the number of calculations.

Triggered calculations don't depend on the index– an example might be producing 6 random numbers to add together for a Gaussian distribution. Of course, since the index outlet and the bang outlet are synchronized, a fresh index is available on each bang.

The **completion** action is triggered by the center outlet. This is usually a notification of the following sections of the patch that uzi has done its thing and operation may continue.

[19] Most programs use loops, and many programmers have trouble letting go of them. If you are one of these, just think of uzi as an encapsulation of this code:

```
for(i=1; i <= n; ++i){
outletR(i);
outletL(bang);
}
outletM(bang);
```

Nesting uzis

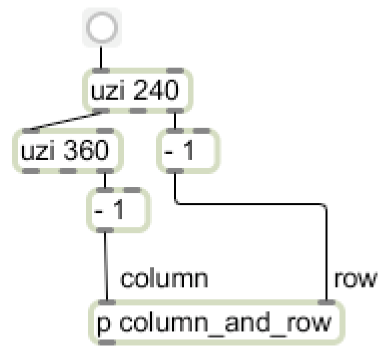


Figure 18.
We often need loops within loops. A common example is calculating values for a 2 dimensional matrix. We need to generate the column numbers for row 0, then for row 1 and so on. We do this with two uzis: one is sized for the number of rows, another for the columns is triggered by the bangs outlet of the first.

There are some gothcas to be aware of when using uzi. One, which is rather rare, happens when the index output of uzi triggers a deferred action. At the least, this will cause a loss of synch between the indices and bangs, and in the most severe case will cause the indices to be output in reverse.[20] The more common gotcha is simple overwork. If an uzi is connected to an uzi, the total number of operations will be the product of the arguments. There are many applications where this is a good idea, but it is easy to bog Max down, even lock it up entirely. The patch in figure 18 generates 86,400 calculations.

Interrupting uzi

There are many situations where the loop end depends on more than one condition. This may be the result of some calculations, or in response to an outside input.

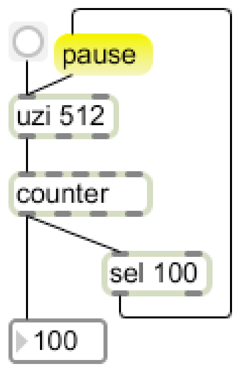


Figure 19.

[20] Remember, the defer mechanism puts the deferred action at the beginning of the queue. If the index numbers from uzi each go to the head of the line, the last will be executed first.

Figure 19 shows a mechanism for interrupting an uzi with a pause message. Once the uzi has paused, it may be restarted with a bang or continue message. With a bang, it will start from the beginning, output of 1, with continue, it will complete the interrupted series.

Slower Counting

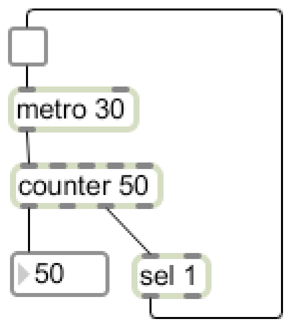


Figure 20.
Sometimes we need repetitions at a slower rate. The patch of figure 20 will step from 0 to 50 (inclusive) at 30 ms intervals. Action is instigated by the toggle. When the counter hits maximum, the select 1 object will bang and shut the toggle off. This mechanism can form the core of any patch that needs to run for a specified time (the metro argument times the counter argument) or a desired number of reps. I often combine this with an uzi for nested loops that don't slow down the patch[21].

The Gotchas of Feedback and Recursion

There are two common programming structures that don't work in Max. A feedback patch exists when the result of a calculation is brought back up to the top of the patch to use in the next iteration. This is fine, as long as the fed back value is never allowed to trigger operation.

[21] Lobject plug. I generally use Lcount here. It counts from 0 to one les than the argument, so the argument indicates the number of times the event will happen. I also directly outputs bangs, and will do parlor tricks like count by 2s or count up to 2PI.

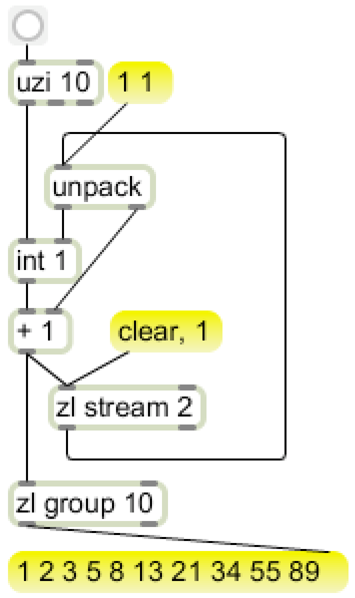
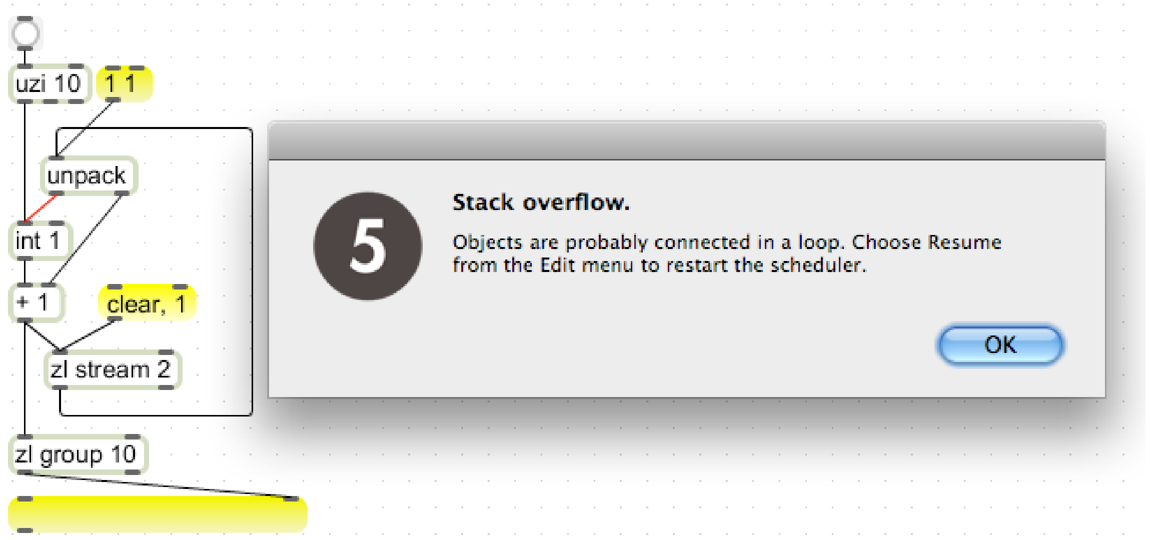


Figure 21.
Figure 21 will calculate Fibonacci numbers in batches of ten. The mistake in figure 22 brings everything to a halt!



Stack overflow.

Objects are probably connected in a loop. Choose Resume from the Edit menu to restart the scheduler.

Figure 22.

Note that the output of zl stream (the last two values calculated) is returned to an unpack, which is connected to the **left** inlet of an int box. This will produce a race condition where the output triggers output, which triggers output..... Max will usually detect this and stop the scheduler until you can sort things out. In the cases where Max does not detect this, it will lock up and you will have to force quit, losing any unsaved work.

The second big structural gotcha is recursion. It is perfectly legal in many languages for a named function to call itself. This works because a function name just refers to a section of code, which can be used recursively with different values as long as the ending conditions are properly met. In Max, although it is good practice to type the name of a saved patcher into an object in order to access the code, a patcher must never refer to its own name. That is because Max loads each file contained in a patcher, along with all of the files any subpatcher may refer to. If a patcher refers to itself, the loading process will go on forever. This will hang up max and you will have to force quit. Worse, it can be very difficult to recover your work in the contaminated patcher, because you can no longer load it[22]. It is possible do this inadvertently by loading a patcher that refers to the patcher you are working on.

The closest you can get to recursion is to encapsulate an operation, then copy that encapsulation and build a string of them. It takes up a lot of space, but it works.

Modular Programming

Many of our projects are variations on similar procedures. For instance, when I was investigating video feedback (see the Feedback Jitter tutorial), I built many patches with feedback though a rota object with external control inputs. I built most of these as two patchers-- a front end that generated matrices using one of a dozen techniques, and a back end with the feedback processing and display. These were connected via send/receive pairs. There was a master qmetro in the back end that broadcast bangs to receive objects named "tick" in each front end. The front end patchers would respond to the tick by generating an image for a send named "todisplay". Thus I cold do a set by loading in my choice of front ends while the back end remained active on the screen. I eventually added a third patcher to the setup. I controlled the performance with a variety of hardware controllers, and it seemed I had a different one available for each show. The new window would accept input from a particular controller and translate it to a defined set of command messages such as fbk_lvl. Thus I could easily switch between a MotorMix module, a microKorg, module, or a joystick module.

All of this is illustrated in figure 23. Each grey box represents a patcher. The feedback and display patcher was loaded first and remained operational for the entire set. So was one of the controllers, although it is possible to use two or more at a time.

[22] The only recourse is to open the file as text and edit the text, which requires detailed knowledge of the text format.

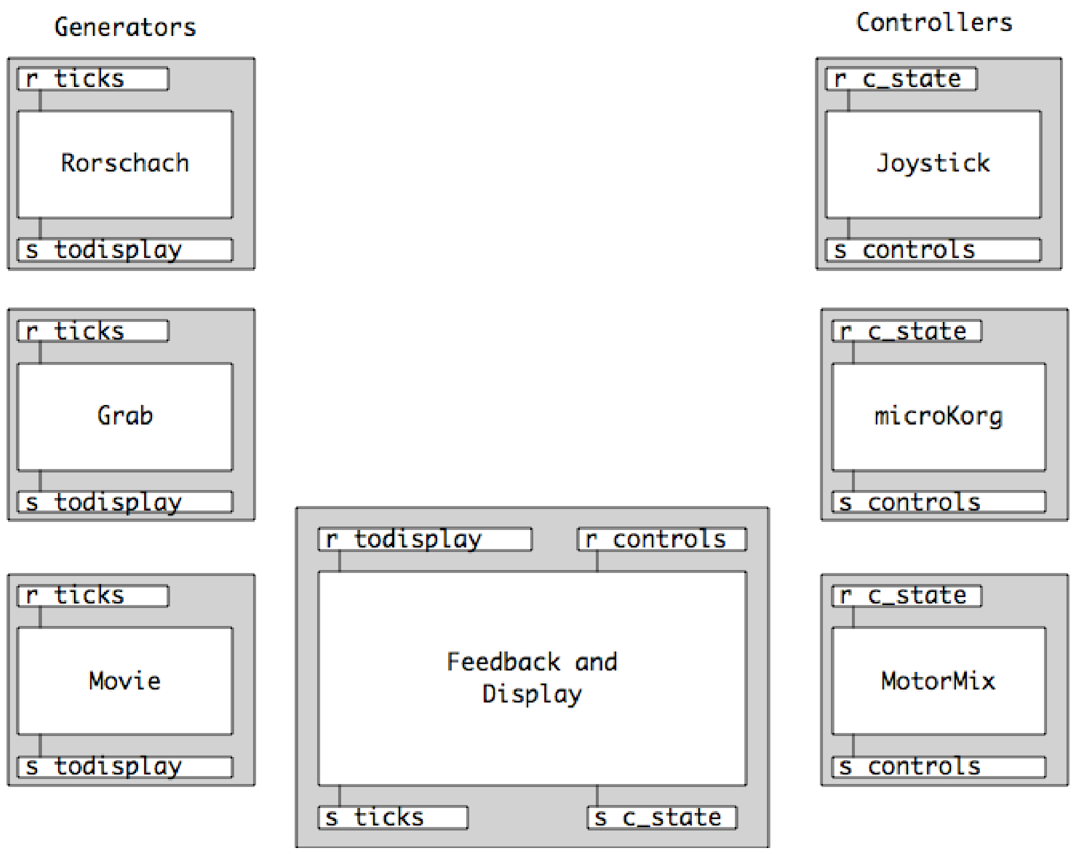


Figure 23. A modular performance system
The generators are swapped for each section, or could all be loaded and enabled as desired. In fact, the most recent version includes a four way mixing module.

The advantages of the modular approach should be obvious. It's easy to add new generation schemes-- I have modules that do waveform display and lissajous generation, animated motion, and fractal images. When I updated the feedback system to use jit.slab, only one module needed to be changed. It's also easy to add new controllers-- an Arduino based device fits right in.

The Robust Patch

The reliability of any computer program diminishes exponentially with its size. In other words, if a patch is doubled in complexity, it is four times more likely to fail. When a patch is functional, you are only half done. There is still a lot of work to do to make the patch give long term, reliable service. The following sections describe the remaining parts of the job.

Our patches usually begin looking like a colony of drunken spiders. I am no exception, but I hope the patches that illustrate these tutorials give a different impression. I revise my patches to improve readability, since the explanations will make no sense if readers can't follow the patch. If a patch is difficult to understand, any attempt to extend it (or repair it) in the future will likely break something instead.

Once you have a working patch, copy it, and clean up the copy. By "clean up" I mean:

- Ensure the patch flows from the top to the bottom of the window.
- Group related objects together.
- Move groups to proper right to left order.
- Remove dead end or redundant objects.
- Encapsulate related objects when that makes sense.
- Guarantee message order with trigger objects
- Replace very long cords with send and receive.
- Segment long cords so they never cross objects.
- Adjust cord crossings to clarify destinations.
- Color code cords that originate from popular sources.
- Comment everything.

At each step, test the copy and compare it with the working version. Moving objects around very often breaks a patch.

Figure 24 and 25 illustrate the benefit of orderly patching. They are exactly the same, and both have the same bug. The patch is an arpeggiator controlled by a k-slider in polyphonic mode. Clicking on a key enters a note into the arpeggio, clicking it again will remove it. The bug is in the velocity which somehow becomes attached to the wrong note. If you click a strong C, then a quiet E, you hear the E but the C becomes quiet. If you study figure 24, you will eventually spot the mistake, but I bet it is immediately apparent in figure 25.

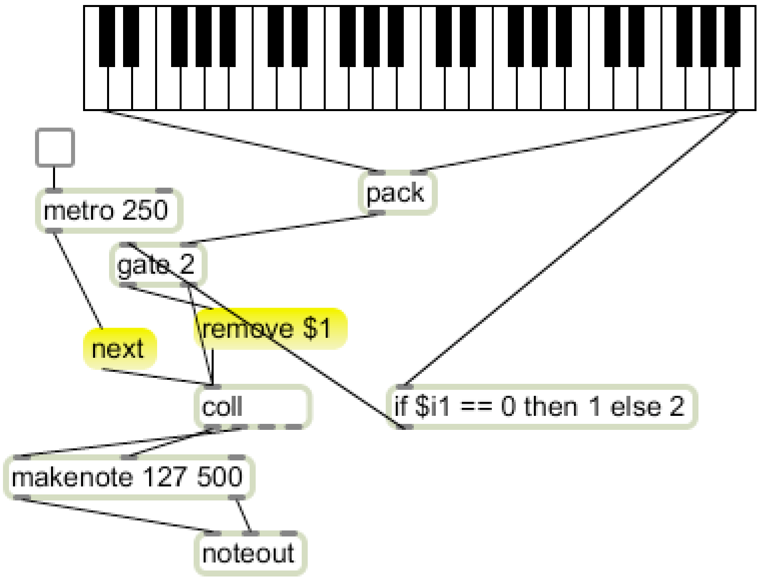


Figure 24. A messy, buggy patch.

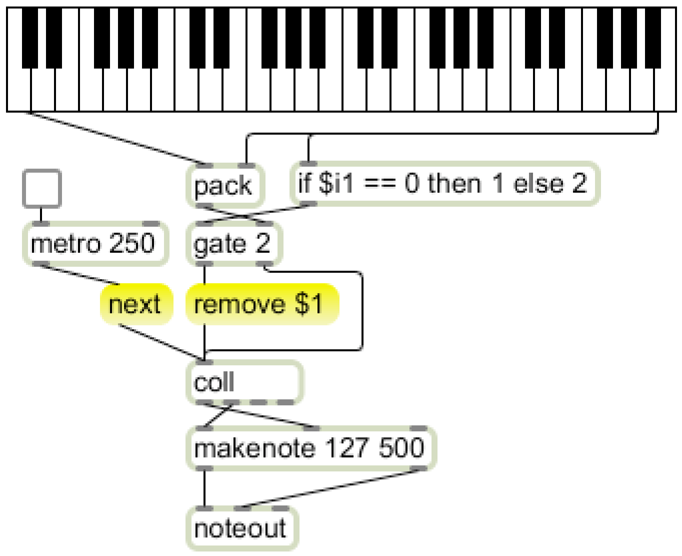


Figure 25. Cleaning up the patch makes it easier to see the bug.

The list generated by the k-slider is stored according to note number, but the coll object emits address before data- thus the pitch will be played by makenote before the velocity arrives. The velocity of the previous note is used. The problem is easily fixed with a swap object between the coll and makenote. Figure 26 shows the correction and comments that will make the operation of the patch clear.

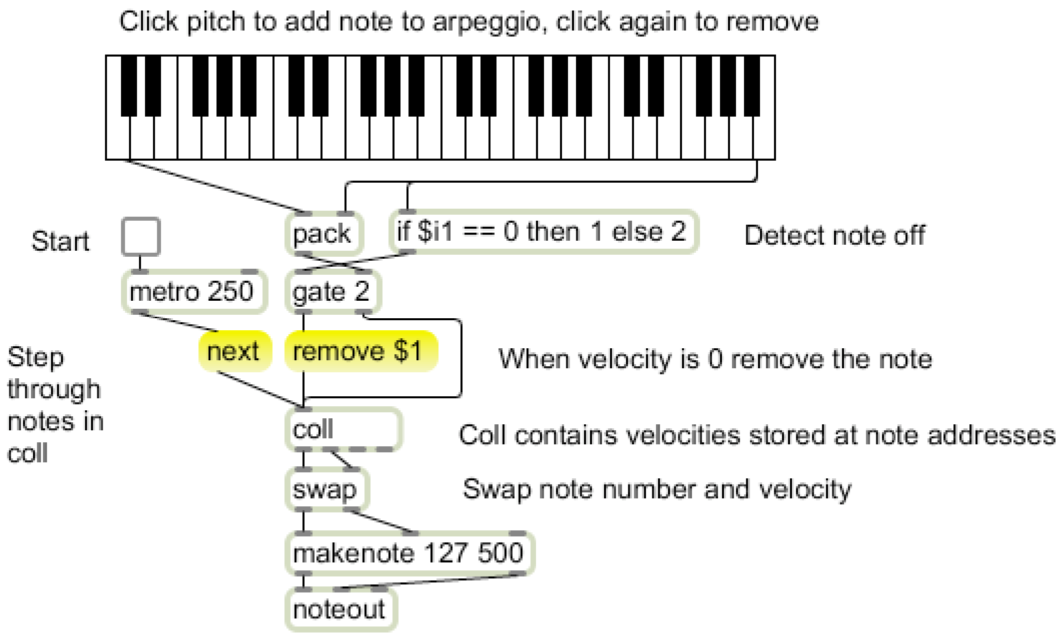

Click pitch to add note to arpeggio, click again to remove
Start
pack
if $i1 == 0 then 1 else 2
Detect note off
metro 250
gate 2
Step through notes in coll
next
remove $1
When velocity is 0 remove the note
coll
Coll contains velocities stored at note addresses
swap
Swap note number and velocity
makenote 127 500
noteout

Figure 26.