Notes on Program Design with Max

The most common question I hear from Max learners is "How do you design a patch?". This is probably the most common question in any programming language. It comes from beginners and advanced students alike. There are books on the topic (none specifically about Max as far as I know) and expensive workshops at luxury resorts. Of course there's plenty of advice on the internet[1].

Traditions of Programming

Most of the books and advice boil down to "This is how I design programs", but it never hurts to look at what people are doing and teaching. Some common themes you will encounter[2] are "Algorithms vs. Overhead", "Top Down Programming", and "Design Patterns".

Overhead

The majority of the code of most programs is concerned with mundane jobs like getting data from key presses and saving data in files. The algorithms are the meat of the program, the code that does the actual work. Max takes care of most of the overhead– that's why we use it.

Top Down and Bottom Up

Top Down Programming is nothing more exotic than designing the whole program on paper before starting to code. The alternative is Bottom Up , where you jump into the hardest part of the problem and get some algorithms working on selected data before worrying about how to put the whole thing together. Top Down is favored in corporate situations where the process has to be parceled out to many people[3]. Some of these people are called designers and just work on scratch paper and whiteboards, others are coders and have to know how the language works. If you just hand a problem to an independent programmer, he will probably work bottom up.

Design Patterns

There have been millions of programs written, enough for scholars to start analyzing them to look for similarities in structure. It turns out that the majority of programs fit one of about six models. Max patches don't fit the categories very well, primarily because the scholars are mostly interested in business applications. However, let's see if we can start with these to identify some typical Max patching patterns.

[1] A good example is http://sites.google.com/site/yacoset/Home/how-to-design-a-computer-program by C. Lawrence Wenham.
[2] In 2011, anyway. The field is subject to fads, and some fads stick around longer than others.
[3] It's also favored in schools, where Bottom Up is just too messy to teach.

One-Shot

This is a program that only does one thing once. "Hello World" is the archetype in most languages. The closest thing I can think of in Max is the function that converts tempo into milliseconds per beat:

Figure 1.
We don't often use Max this way, but such operations may be part of a larger program.

Batch

A batch process performs a set of operations on a whole load of data. The user gathers all input, the program runs, and the output is presented together.

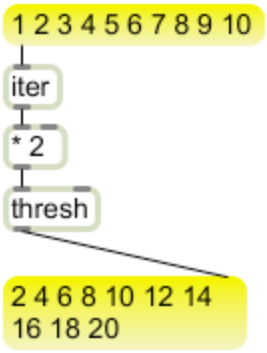

Figure 2.
Again, in Max this normally would be part of something else, but I have seen a few examples. Luke DuBois used Max to process copious numbers of data files from surveys and the US census to create the maps in "A More Perfect Union" http://perfect.lukedubois.com/

Processing loop

The loop performs an operation repeatedly. The operation always starts by checking for input (this is called "polling"). If there is no new data to work with, the operation is skipped. Many external interfaces are handled this way.

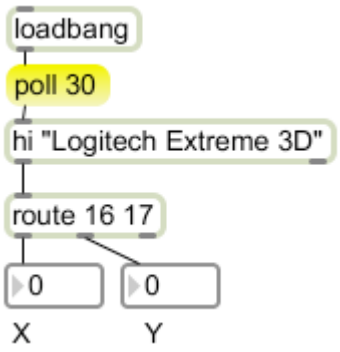

Figure 3.
Some interface objects, such as the hi object in figure 3 manage the polling on their own. Others, such as the serial object, must be polled by a metro. That the responsiveness of a polled object is limited by the polling rate. Figure 3 gathers data every 30 milliseconds, so there might be that much delay from action to result.

Pipeline

A pipeline consists of a line of processing objects where data is shunted through continuously. The key is that the first object gets started on the second data item while the first data item is being processed by the second object. Figure 4 emulates the system using buckets[4]. The counter is producing a series of numbers, the most recent being 4. the previous value 3 has just been added to ten. The 2 before that was added to ten and the resulting 12 multiplied by 5. Finally we see that (1 + 10) * 5 -5 is 50.

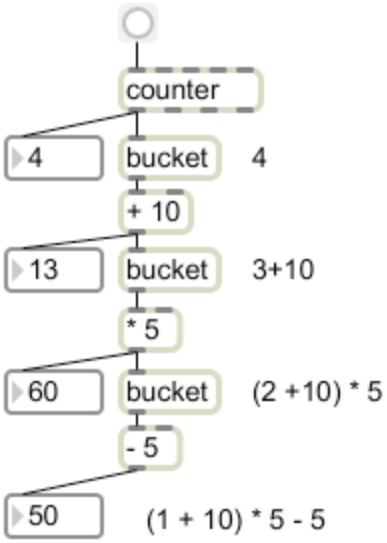

Figure 4.

[4] A bucket stores a value until a new one comes in to take its place. The stored value is then sent along.

Figure 4 isn't much good except as an illustration, but this is what happens all the time in the audio chain and in the GL objects of jitter. Pipelines are very efficient in processing data.

Workflow

A workflow is like a pipeline, except for the overlapping operations part. Data comes in from a continuous source, but all operations are performed on each input before the next is looked at. Jitter works like this, which is why visuals can slow down but audio doesn't.

Figure 6.
A frame is fetched from the movie and processed by the tiffany and robcross objects before showing up at the pwindow. Even if the qmetro fired processing would not start on a new frame before the current one is finished.

Model-View-Controller

The model-view-controller paradigm (M-V-C) is very in right now. The basic premise is the data is contained in a code module called the model, and the user interacts with a different module called the view. The controller is the section that connects the view and model together. A DAW program like Logic is a good example. The model is the tracks of audio or MIDI events, and the various editing windows are views. This is a good way to manage huge programs, but few Max patches will be so ambitious.

Stimulus-Response

While the foregoing are usually covered in standard program design texts, they aren't very common in Max. The essence of Max is message passing, which means Max is essentially a reactive system. Don Buchla described this as the stimulus-response model[5]. This views the patch as an organism with behavior triggered by user input. This mimics the operation of musical instruments. The activity can be summarized as figure 7.

There are two kinds of input to a stimulus response system. Some input does not cause a response, it merely affects what will happen when a stimulus comes in. As an example, you may choose a key for a harmonizer. This kind of input sets context. The stimulus is any action that elicits an immediate response. The system may have more than one response available, chosen according to context or details of the stimulus.

[5] I saw this in the manual for a program for a Buchla computerized synthesizer from 1980.

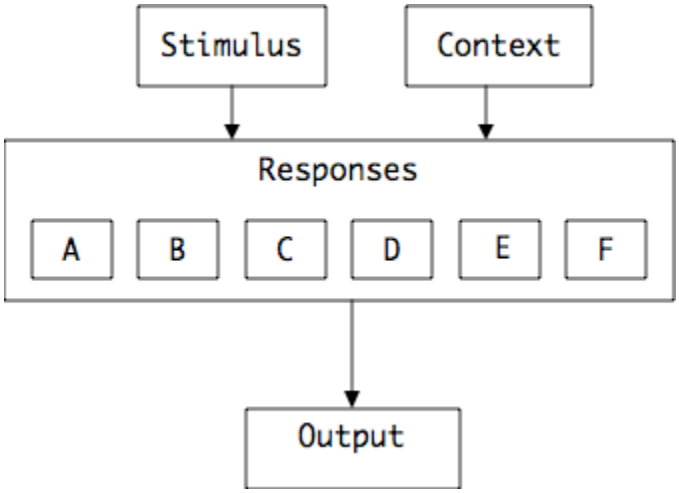

Figure 7.
If the stimulus is MIDI data, the system can respond instantly. This is because the MIDI hardware can interrupt ongoing computations without waiting for a poll[6].

Combinations

Most programs are some combination of any of these design patterns.

Program Design as a Process

Computer programs are based on a need to get a task done. It should go without saying that the better the task is understood, the better the program will work. Describing a task is more difficult than it seems at first. There are many methodologies, but here are the things that must be known before design can even begin.

Goals

The purpose of a program is always the first thing to define. This can be conveniently stated as output, or what you have after the program has run that you didn't have before. This is complicated if the program is to do more than one thing, and most programs do. For instance, you may want a program to balance your checkbook, but that probably includes displaying the balances on a screen, printing them out, and storing them in a file for the next run.

In the computing world, goals are called specifications, and are written up in great detail. The goals are often developed by interviewing potential users or looking at similar programs that already exist.

Input

Input is also known a user experience. This is what a user or the world supplies as the program runs. You can get advanced degrees in designing user experiences and the interface that connects the user to the program. One common approach to specifying the

[6] If overdrive is on. Actually there is now a polling loop in the MIDI process, but it is so short that delays are imperceptable.

user experience is to write a script or draw a storyboard showing what a user does. If input comes from the world, the specification must include details of the form that input takes.

Dependencies

Dependencies describe the relationships between the data variables in a program. A program to convert currency values between dollars and euros will have dependencies between the exchange rate, the value input and the direction of change. This includes a lot of data that is internal to the program such as the relationship between dollars and cents.

Algorithms

Once the desired output, input and dependencies are known, it is time to explore methods of getting from the latter to the former. These are also known as algorithms or formulas. The source of algorithms is the great mystery of program design. In most cases, we simply copy what someone else has done. We can find algorithms in textbooks, in online libraries, and in other people's code[7]. Somebody somewhere invented these things, and we honor them by extending their work to a new application[8]. If you can't find an algorithm to use or modify, here are some approaches to coming up with a new one:

Take the problem apart into its smallest parts.

David Cope calls this "divide and conquer". One good way to do this is write a set of instructions for doing the task. What this does is force you to look at the problem in detail– once you have isolated all of the steps, you will probably discover each is solvable with a standard algorithm and all you need to do is combine these in the right order.

Look at related problems and learn how they work.

There are surprising similarities between completely different fields. Suppose you are trying to work out an algorithm for generating melodies from a desired set of pitch classes. It might help to think of the pitches as destinations on a map, and the melody as a route that visits these destinations. You can find a lot of material on map routing, including the classic "traveling salesman problem", where various ways are presented to create the shortest path or cheapest path through selected cities.

You should always be on the lookout for methods of solving problems, no matter what the data actually represent.

---

[7] The forums at Cycling74.com have lots of examples and pointers to users' web sites where you will find more.

[8] Some algorithms are patented or copyrighted. Usually, if you can find an algorithm in a book or from more than one online source, it is "prior art" and the patent would not be enforceable. Ideas can't be copyrighted, but the form of the idea can be, so always rewrite code in your own terms. In any case, applications you make for yourself are exempt and you do sell products, you won't be sued until you have enough sales to make it worth the lawyer's time.

Imagine doing the problem backwards.

If you start with a desired output and try to get back to the input, you will learn a lot about the problem. Don't get too hung up on this however– some processes are irreversible.

Explain the problem to someone else.

You aren't asking them for the answer, it's just that the act of formulating a description of the problem for someone else will force you to think from a new perspective.

Use brute force to work around the problem.

It's possible to get hung up on a small part of a program to the point that the whole program never gets written. But there's one approach that will work with any problem. List all expected inputs and write a possible good result for each. The results are put in a table that is indexed by the input. This is completely arbitrary, but it will allow you to move on to other parts of the program and return to this later. The number of cases you can handle is determined solely by your patience. Of course, as you are working the table out, you may get an insight that suggests an algorithm after all.

Style

There are a lot of elements to a program that are not directly related to getting the job done. This is intimately connected to the user experience, but there's a lot of the personality of the designer here too. Style is both external and internal. The external style is what the world sees– aero effects, rounded patch cords, decorative fonts and so on. The internal style is visible only during the programming process, but has a lot to do with the efficiency and maintainability of the program. This includes details of layout and encapsulation, comments, and the choice of methods where there are alternatives.

The role of Napkins in Software Design

Most designs get sketched out at some point in the process. This might be a formal flowchart on a PowerPoint slide or a scribble on the back of a napkin. (One café near the Apple campus in Cupertino used to provide napkins with a graph paper design.) This is a great way to visualize the flow of the program and spot any holes in the concept. You draw a box for each process, and make connections from box to box, checking as you go that what comes out of a box is appropriate for the needs of the box below. The fact that this looks a lot like a Max patch is not accidental.

Actually Writing Code

When most of the above is sorted out, you are ready to write some code. Of course it is perfectly OK to write code before the entire program is designed. Good reasons to do this are to try out algorithms or test interface layouts. Just don't be afraid to throw out what you have written.

You will get conflicting advice on what to write first. Top Down is usually taught in schools, but each coder has his own preference. Actually, the program design itself will often suggest the best point of departure.

I like to start somewhere in the middle or the bottom of the patch, building the heart of the code first. If I am following the stimulus-response model I will make the basic response work with test input. Once I have a reliable response, I will connect it to the output, then begin the interface.

A Sample Design

To illustrate the process, I'll develop a simple synthesizer. You've seen something like this in class, but here I'll concentrate on my thinking during the design process.

Specifications

I start with specifications. This is terse, often just a list.

Goal

A one note at a time MIDI synthesizer with ADSR control of amplitude and a filter that is adjustable relative to the played pitch.

Input

MIDI or an onscreen keyboard. ADSR and filter will be controlled by number boxes.

Dependencies

MIDI note numbers will be converted to frequency. The patch amplitude will be velocity sensitive. The filter resonance will be fixed at 0.75. The filter frequency will be some multiple of the note frequency.

Method

Signal generated by a [tri~] object with be applied to a [lores~] and a [*~], with an [adsr~] controlling the [*~].

This is not the first set of specifications I came up with. There was a fair amount of back and forth between this and the following step, as the diagram showed I had left things out.

The Napkin Version

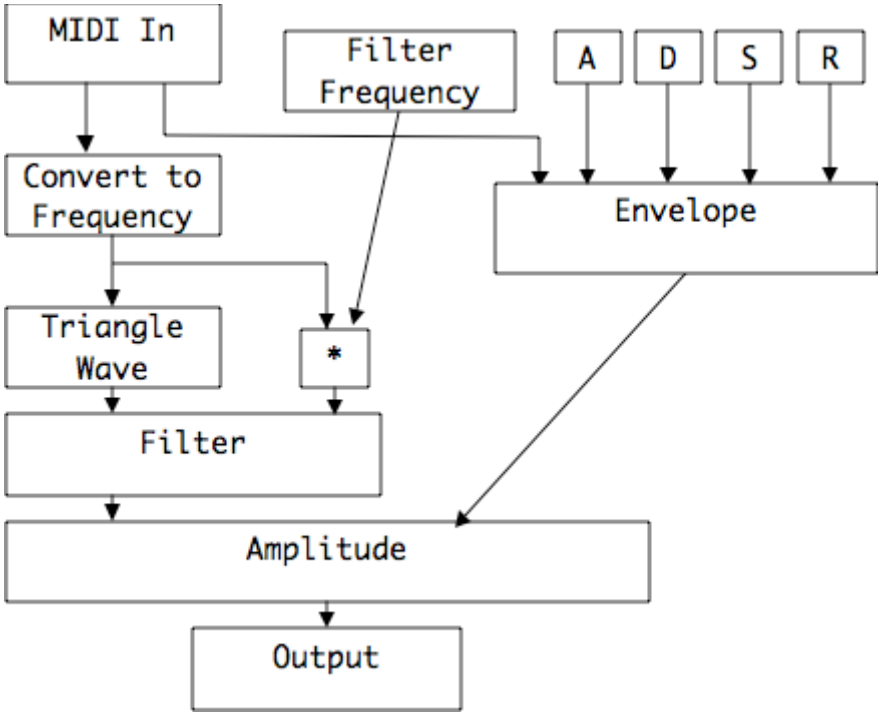

Figure 8.

Figure 8 shows the diagram of this program. It's tempting to skip this, since the Max patch is so similar, but it really does tell you things. For instance, I revised this several times to get the best control of filter frequency.

Test as You Go

Once I start coding, I build in small sections and test each as soon as I can. Figure 9. shows the first part I put together.

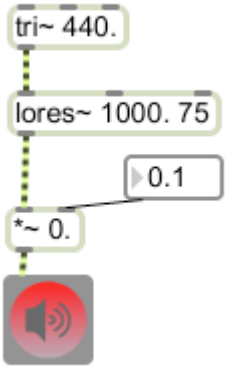

Figure 9.
This is the heart of the patch. The number box controlling amplitude is temporary– it lets me test this part of the patch on its own. From this I notice that the resonance setting was too high. That was supposed to be a 0.75 in the lores~ box.

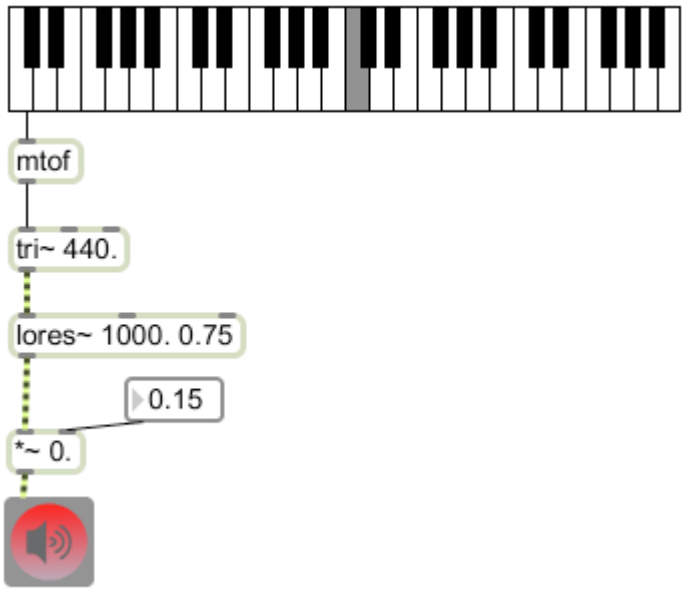

Figure 10.

The next step was to set up frequency control of the pitch. I used a k-slider (the keyboard widget) to give me quick and easy input. I put it in polyphonic mode (click-on, click-off). Mtof is very reliable if you are OK with equal tempered scales. Otherwise, you need something else.

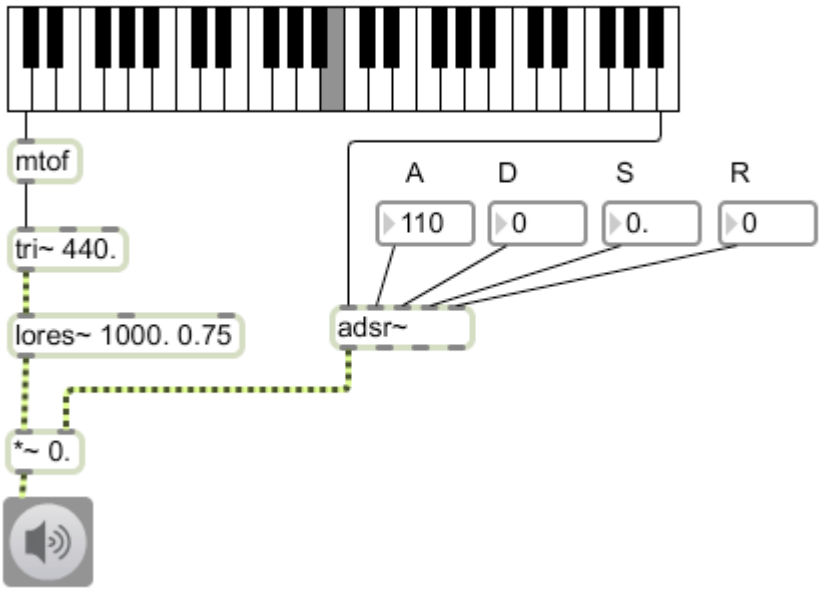

Figure 11.

Next I tackled the envelope problem. This showed another mistake. I forgot that the adsr~ object needs an input in the range of 0 to 1 if it is to control amplitude. I should have caught that back in the design stage, but you never get them all. I'll stick a [/ 127.]

between the k-slider and the adsr~. Figure 12 shows a working version, with the skeleton interface.

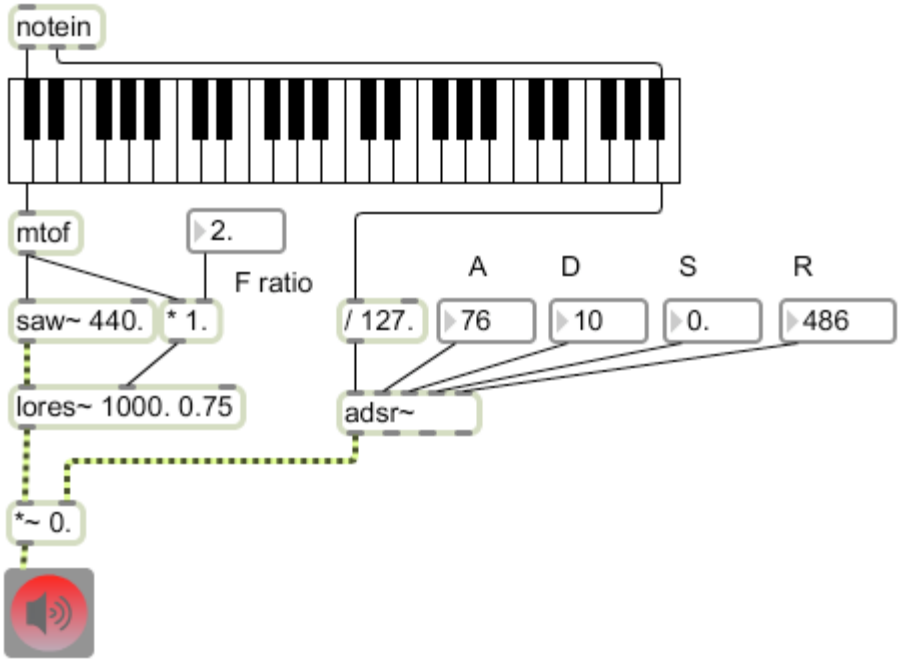

Figure 12.
After I had played with this a bit, I decided I preferred a saw for the basic waveform. I also connected a notein directly to the k-slider. That will give me a display of what has been played.

Once this is working, I can play with interface layouts in presentation mode. Interface design is another study I plan to tackle in a separate tutorial.

Debugging

The next step in the process is testing the patch thoroughly. You must play it to all limits- the full range of pitches, the full range of velocities and confirm that you are getting the results you expected. Exercise the adsr and filter settings in the same way, trying to hit as many combinations of values as you can. Save the patch and reopen it to make sure all of the initial values are appropriate. Finally, get someone else to play it.

It is inevitable that this process will reveal problems. I will discuss how to find these problems in a separate debugging tutorial.
pqe