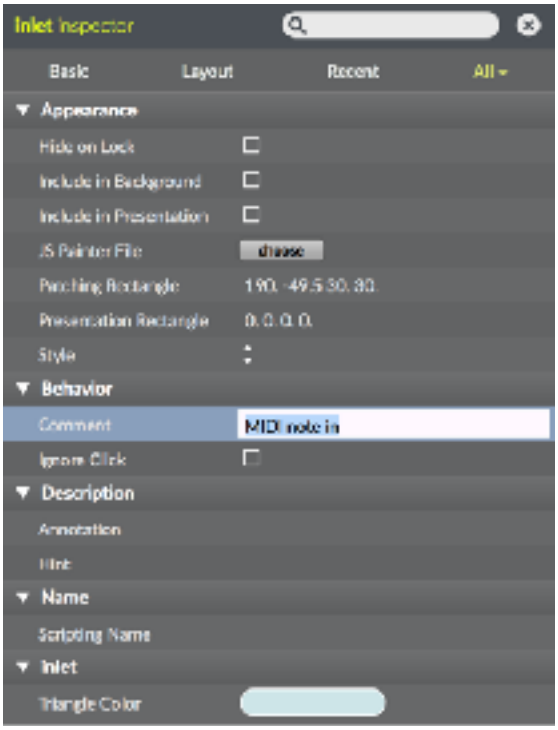
CONNECTING IT TOGETHER

We've seen how to create dynamic, evolving synthesizers, sequencers and effects in Max/MSP, and have started linking these together to create increasingly complex and layered music machines. When patches start being combined, the result can be tangled and messy and it can be hard to navigate many screens at once. This lesson we'll look at integrating our patches into one system, develop GUIs, explore concepts of connectivity and encapsulation, spruce up some patches and come away with some key ideas and crunk patches.

CREATING OUR OWN OBJECTS

As we saw last lesson when we made our **xfade~** object, we can create our own objects by saving our patch file and calling that file name inside our patch. Let's do that with the last synth we created during our subtractive synthesis lesson.

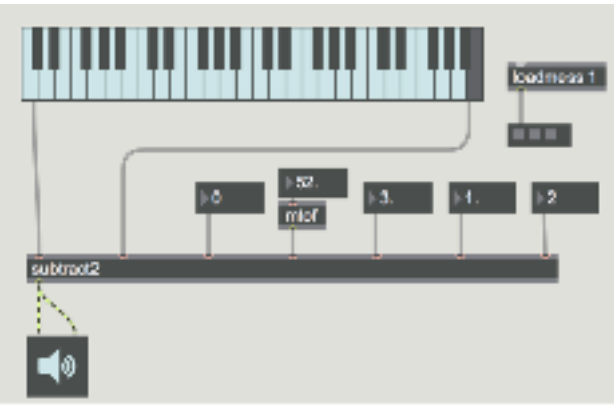
PATCH: subtract1.maxpat



If we wanted to make this into a standalone object, what inputs should we have? Note, velocity, oscillator type, filter freq, gain, Q and type all seem like relevant inputs. What objects do we use for inputs and outputs into patches (review the xfade~ object; the inlet & outlet object respectively). Add the necessary inlets. If we want the inlets to give us a 'hint' when we hover over them, we can use the 'comment' parameter when we look at the inlet object's inspector window. Note that our inlets will appear in our object in the order that they are created.

PATCH: subtract2.maxpat

Let's create a new patch and place our subtract2 object inside of it. If your object is not showing up, be sure that the new file and your subtract.maxpat file are in the same folder. You can also create your own file repository for Max patches by going to **Options->File Preferences** and adding the folder where you keep all of your patches.



In our new Max patch, add in a kslider object and some number boxes so that we can manipulate the various parameters for our subtract2 object. Now our patch's functionality is neatly contained within the object, dramatically reducing clutter and making our patch more user-friendly. But what if we want to change the function graph for our

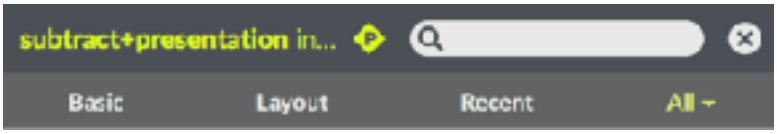
amplitude envelope? Or graphically change the filter parameters using the filtergtaph object inside our patch? Well, it's possible to create an encapsulated GUI form of our object.

PATCH: subtract3.maxpat

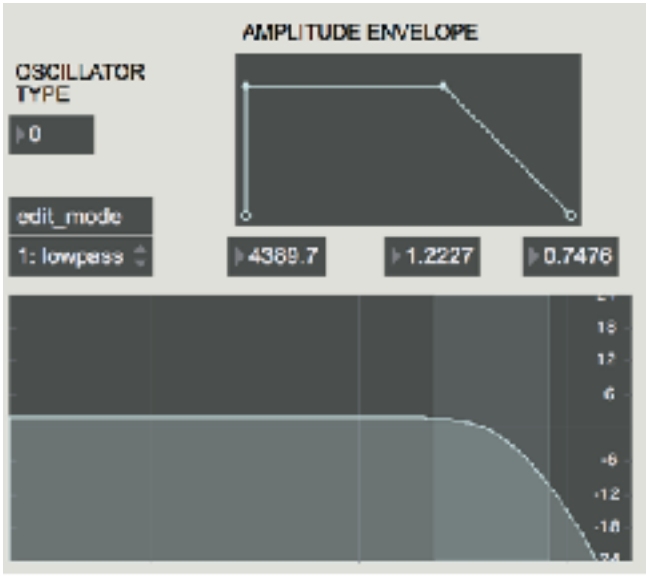
PRESENTATION MODE & BPATCHER

Let's return to our subtract2 patch, and select all GUI elements - the oscillator type, the filter graph and filter parameters, and the amplitude envelope. Then go into **Object->Add to Presentation** (there is also shortcuts for this, which you can see next to the 'Add to Presentation' option). In the bottom left corner of our screen, click the box that looks like a projector screen, you will see all objects disappear except for the ones selected. This is our **Presentation Mode**. Everything can be graphically arranged in presentation mode and it will not affect the signal flow in our **Edit Mode** (the normal mode for patching things together). Set these elements to an arrangement that makes sense for you.

In the **Patcher Inspector** (which can be found by going into our inspector window and selecting the P inside the diamond at the top), select "All" at the top, then scroll to the 'View' section. Look for the option 'Open in Presentation'. This means each time we open the patch, it will automatically open up in presentation mode.

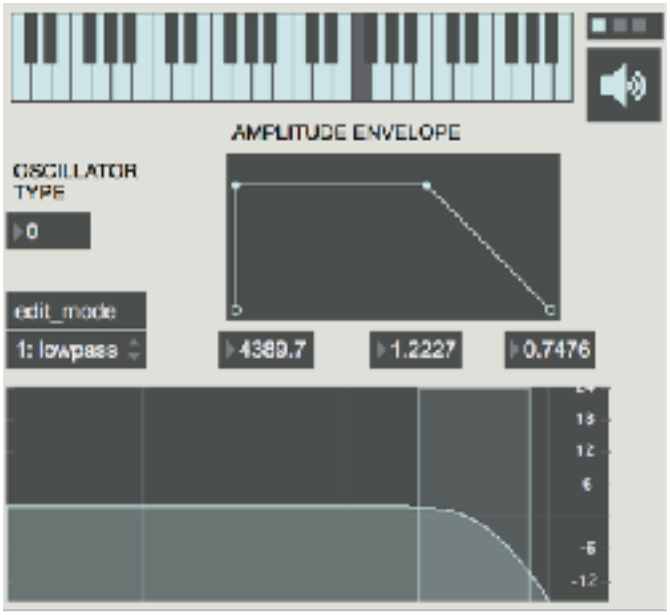


PATCH: new_subtract.maxpat



We will now incorporate this new subtract patch into our subtract3 patch by using the **bpatcher** object. This object lets you see into a patch inside another patch. When creating our patcher object, include the **attribute** @name new_subtract (or whatever the name is of your subtract synth patch.) Attributes are like including messages with specific values into your object as its instantiated. We can even just click our 'subtract2' patch, and change the name to 'bpatcher @name new_subtract'. Since both objects have the same inlets/ outlets, nothing else has to change. Then arrange your parameters as needed, possibly deleting some of our previous GUI elements since our new bpatcher contains

controls for some of those parameters. We can then include our bpatcher and the remaining GUI elements into a new presentation mode. This could of course be extended to any of the patches we've included, to create clean and intuitive interfaces for any set of effects, sequencers, etc. Using the patcher object, these units of sound design and control can then be freely dropped into our patches, giving us a personal library of distinct audio tools.



PATCH: bpatcher+subtract.maxpat

To explore this, spend a little bit of time creating a patcher interface for the degrade~ object and incorporate that into your patch.

PATCH: bpatcher+subtract+degrade.maxpat

PATCH: dist_bpatch.maxpat

POLY

You may have noticed in our synthesizer that retriggering a sound before the previous sound has finished playing leads to a slight click. Why do you think this is? If we imagine the signal a sine wave moving continuously, then we suddenly change the amplitude of that sine wave, there will be a non-continuous jump from one value to another.

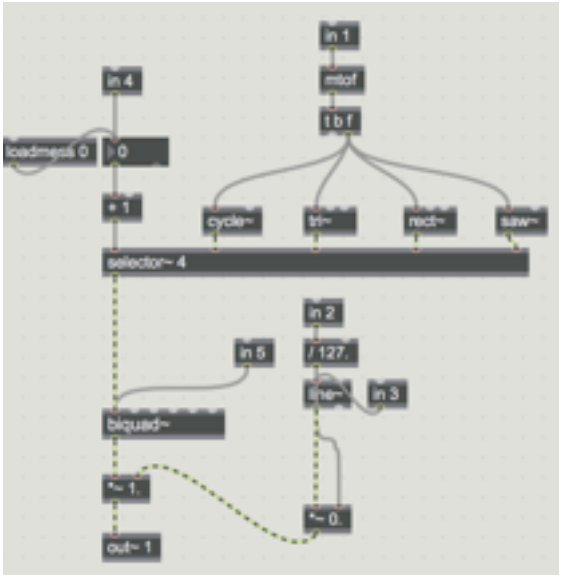
PATCH: click_example.maxpat

In this patch, we can both see and hear how the waveform changes when we send a new amplitude value. If we change our frequency to 1 Hz, we can see even more clearly in the **scope~** object the discontinuous leap in sample values when we retrigger the amplitude envelope. How could we solve this? There are numerous methods; the method we will explore today is use of the **poly~** object, which creates a variable number of instantiations of our patch, allowing us to have each **voice** for our synthesizer play the entirety of its envelope uninterrupted.

Create a poly~ object and open its helpfile. As you can see, poly~ has two arguments: the first is the patch which it will create multiple instances of, the second being the number of instances that will be created. Let's modify our subtractive synth to make it suitable for use within a poly~ object. Since we will have multiple instances of our subtractive synth, but only one GUI, we need to separate the signal objects from the GUI objects.

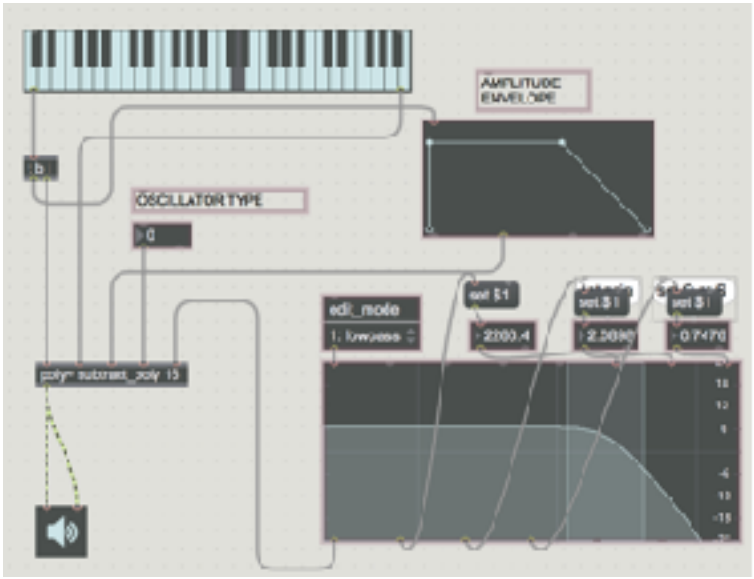
Going back into our subtract2 patch, what objects are the GUI, what objects are for the signal? Wherever we have a GUI connected to a signal, let's create an **in** object. The **in** object

is how you get input into a poly~ patch. The argument for the in object determines its order of appearance (so it doesn't have to be arranged graphically in order, unlike the inlet object). Absent its GUI elements, our patch appears much smaller and reduced.



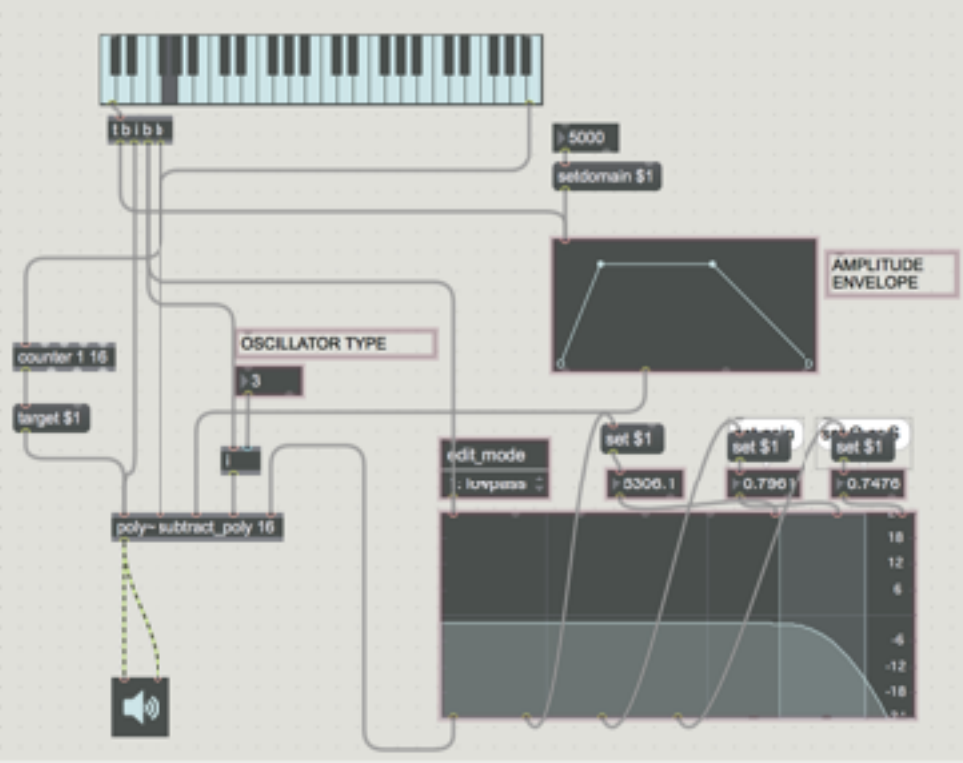
PATCH: subtract_poly.maxpat

Lets create a poly~ object with the name of our patch, and then connect our GUI elements from our subtractive synth bpatcher to our new poly~ object's inlets.



PATCH: poly_example1.maxpat

However, it seems our synthesis patch is still making the click sound! This is because we need to use a **target $1** message to our poly~ object to indicate which voice we are selecting. How might we do this? We can use a counter for each incoming note, and have the counter's int be sent to the target message. We can connect the output from our kslider object to a **trigger** object - this is 't' object you have seen before. The trigger object allows an incoming event (number, bang, etc) to 'trigger' multiple events, such as the output of int (i), float (f), bang (b), list (l) or numbers (the number written directly inside the t object). Since Max control flow processes information **from right to left**, we put the first thing we want to happen to the right. What is the order of our trigger object? Why do we have our trigger object with this specific order (t b i b = bang int bang; first select the voice using the bang to the counter, then send the note, then trigger the amplitude envelope. We first need to make sure we have the right voice selected. Then we send the synthesis parameters, such as the oscillator type and the filter graph information. Next ,we want the frequency to be correct before we hear the sound from the amplitude envelope, otherwise we might hear the frequency change briefly at the onset of the note).

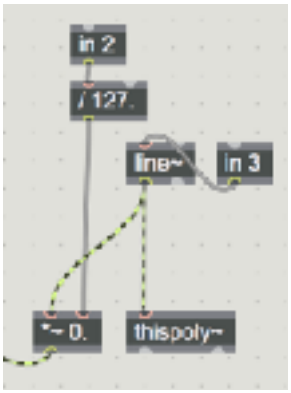


PATCH: poly_example2.maxpat

Our patch might look a little complicated, but by moving through the order of events step by step, we can understand how information moves through the patch and design machines accordingly. Once our flow of information is arranged, we can select our GUI objects and go into presentation mode.

POLY CONTINUED

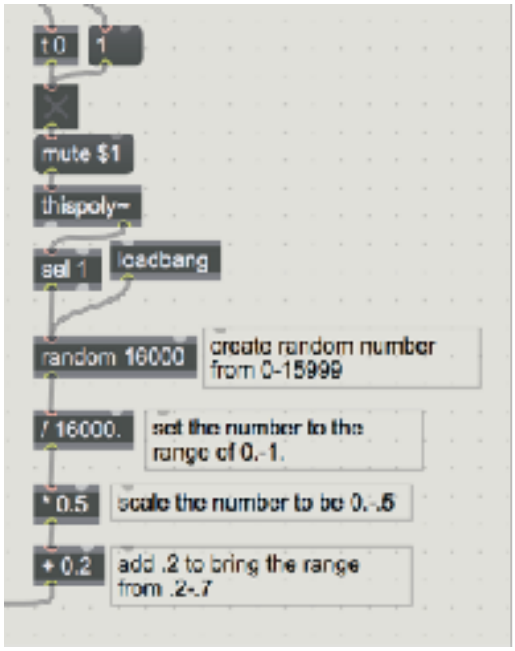
Poly~ is quite useful for layering voices, but if we are working with a computationally intensive synthesis algorithm, then poly~ can be heavy on the CPU. We can use the object **thispoly~** to indicate whether the poly~ voice is being used. If it is not being used, thispoly~ will 'mute' the poly~ instance and save CPU power.



Going into our subtract_poly patch, lets add the thispoly~ object. The thispoly~ object can be muted/unmuted by sending it **mute** messages - mute 1 for muted, mute 0 for unmuted. When our most right inlet (our in 5 object) receives a value, we can have that trigger a value of 0 for our mute mesage. The right outlet of our line~ object will output a bang when it reaches the end of its destination. We can use that bang to send mute a message of 0. value other than 1 when the voice is being played, and a value of 0 when it is not? That's right! The amplitude envelope. So you can simply connect the line~ object (which is controlled by our amplitude envelope) to your thispoly~ object.

The thispoly~ object also gives us an instance of our patch number, as well as a mute on/off flag, which outputs a 0 or 1 to indicate whether the instance is turned on. Since each voice is individual, we can give every voice individual parameter values. Let's take advantage of this by producing a random value for sample rate reduction every time the voice is muted. How might we do this using the **random** object?

Random only outputs integers, so the output must be divided, then scaled and offset according to the range desired. Lets take a listen to our synth now that each voice is behaving independently.



PATCH: subtract_poly2.maxpat
PATCH: poly_example3.maxpat

Perhaps we don't want some synthesis parameters to work independently but rather to be controlled all together. What means could we use to implement this? Yep! Our trusty send and receive objects can even extend into the depth of poly~ instances. Let's disconnect our filter parameters and instead send those to all of our poly patches. Notice how moving your filter now applies to all voices rather than just one.

PATCH: subtract_poly3.maxpat
PATCH: poly_example4.maxpat

We've learned a lot of developing our patches into workable musical instruments! Time to dive in and make your synthesizer dreams come true.