CNMAT Max/MSP Summer Class 2011
Lab Assignment 3

September 15, 2011

Summary

In this lesson, we will create a data storage and retrieval mechanism. We'll use the [function] object as our model, so that we can build up envelopes. These envelopes will eventually help us to make interesting sounds, but for now let's focus on building an interesting function storage/recall tool.

Topics

transfer functions, data storage, data recall, encapsulation, abstraction.

Objects Introduced

[function], [textedit], [line], [route], [line~]

Relevant Tutorials

Basic

1. Abstraction
2. Data Collections
3. Procedural Drawing (using line)

MSP

1. Additive Synthesis (including Envelopes)

1 Data Collection, Storage, and Recall

1.1 Data Collection

1. Open up a new patch window

2. Place a new [function] object into the patch (double-click in blank space to reveal the object palette, click on the data section, mouse over to the item labeled "function", and click on it. You should see a new box appear in your patch).

3. Lock the patch, and try drawing some shapes by clicking in the box with your cursor. To erase the points you have made, create a ⌜clear⌟ message, and patch it to the inlet of the [function] object.

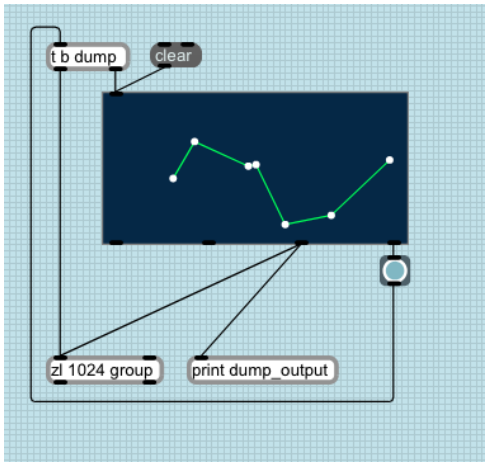
4. Place a new [trigger bang dump] object into the patch (remember that "trigger" can be abbreviated as "t"), and connect its rightmost outlet (the one associated with "dump") to the inlet of the [function] object.

5. Place a new [button] object in your patch, and connect the rightmost outlet of the [function] object to the inlet of the [button] object (this outlet will output a ⌜bang⌟ when we mouse up after a click). Then, connect the outlet of the [button] object to the inlet of the [trigger] object.

6. Create a [zl 1024 group] object[1], and connect the leftmost outlet (bang) of the [trigger] object to left inlet of the [zl 1024 group] object.

7. Now connect the 3rd outlet of the [function] object to the left inlet of the [zl 1024 group] object. When the [function o]bject receives a ⌜dump⌟ message, the third outlet will output all of the points that it contains in successive lists of (x,y) pairs. We want to collect these into a single list which we can accomplish with [zl 1024 group]. Once the [function o]bject is done outputting our coordinates, we will send a bang to [zl 1024 group], which will output them all into one long list.

Your patch should now look something like this:



Let's pause for a second to review what we've just done:

- We click the [function] object to add a new point (or to modify an existing point)
- We let go of the mouse button (this is called a "mouseup" event)

[1] [zl] accepts an optional integer as the first argument that sets the size of its internal buffer.

- A bang is issued from the [function] object's right outlet as a result of the mouseup
- The bang is sent to the [trigger] object's inlet
- [trigger] first tells the [function] object to dump its contents, then once ALL of the contents are dumped, it tells [zl 1024 group] to output its contents

1.2 Data Storage

Since our data is now bundled together via [zl 1024 group], we can easily send it to [coll] to store it as the values in an index but there's one issue: how do we create indices for these data? Let's use a handy mode of [zl] called "join", which joins two lists together. We'll join our data together with a one element list that describes the data. Something like ⌜mydata 0.1 0.3 44.41 33.1⌟ . This functionality is very similar to that of [pack], [pak], and [prepend]. How would you achieve the same basic functionality using the prepend object instead of [zl join]?

1. Create a [zl join] object.
2. Connect the lefmost outlet of [zl 1024 group] to the right inlet of [zl join].

   Now we need a way to get the names of our indices (preset names) into the left inlet of [zl join]. For this task, we can use a [textedit] object.
3. Create a new [textedit] object.
4. Create a new [route] object, with the argument "text", as in [route text].
5. Go into the [textedit] object's inspector, and check the attribute labeled "Return Enters Text".
6. Connect the left outlet of the [textedit] object to the inlet of [route].
7. Connect the left outlet of [route text] to the left inlet of [zl join].

   Each time we click, a new set of points are sent into [zl 1024 group], and then into [zl join]. After a list leaves [zl 1024 group], the object is cleared and ready for a new set of points. In this way we are building up lists to store into the right inlet of [zl join] as we edit functions, and once we have finished editing, we can enter in a descriptor for that particular [function]. When we enter in this text, it gets output together with the larger list.

   Now we just need to make sure that we give the correct storage message to coll. Our current messages would read ⌜myPresetName 0.1 0.3 44.41 33.1⌟ , but we really need ⌜store myPresetName 0.1 0.3 4 . If you'd like to know more about guidelines for storage of data in [coll], refer to the object's help file.
8. Create a new [prepend] object, and give it an argument, so that it reads: [prepend store]. Also create a new [coll] object.
9. Connect the outlet of [zl join] to the inlet of the [prepend store] object.
10. Connect the outlet of the [prepend store] object to the inlet of the [coll] object.

11. Create a new [zl iter 2] object. This is going to allow us to break our singular list back up into xy pairs, which can then be sent back to the [function] object to restore our saved preset.[2]

12. Connect the left outlet of the [coll] object to the left inlet of the [zl iter 2].

13. Connect the left outlet of [zl iter 2] to the left inlet of the [function] object.

    Now we are ready to test our preset system.

14. Draw a function in the [function o]bject, then give it a name. Create a few more functions and give them unique names (you can clear the [function o]bject by sending it the ⌜␣clear⌝ message).

15. Double-click on the [coll] object to see its contents. Do you see your two presets? If not, go back to the previous steps to review.

1.3 Data Recall

Stop and take a moment to make sure everything is functioning as expected. If so, let's move on to recalling our presets.

1. Create two new message boxes containing two of the function names you have chosen.

2. Create a new [trigger] object like so: [trigger s clear], and connect the outlet of the ⌜␣mypreset1⌝ message to the inlet of [trigger s clear]. Do the same for the other.

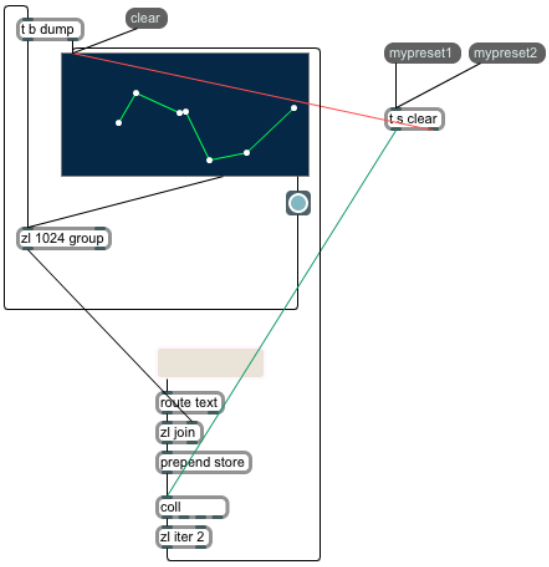
3. Connect the right outlet of [trigger s clear] (the ⌜␣clear⌝ message) to the left inlet of [function].

4. Connect the left outlet of [trigger s clear] ("s" stands for "symbol" and will pass the preset name through) to the inlet of [coll], which will call up the data for that particular preset name.

5. Lock the patch, and notice that when you click back and forth between presets that the [function] object clears properly before accepting new data, and that our presets are restored properly.

Your patch should now look something like the following:

[2] Our [function o]bject takes pairs of numbers representing (x,y) coordinates, and, upon receipt of a ⌜␣dump⌝ message, outputs all of its stored points as a sequence of (x,y) pairs. However, we want to store our data in a [coll o]bject as a single list of the form (x1 y1 x2 y2... xn yn) which means that we will have to build a mechanism to group the coordinates that [function o]utputs into a single list for storage, and a similar mechanism that breaks that list up into a sequence of (x,y) pairs that we can send back to [function]

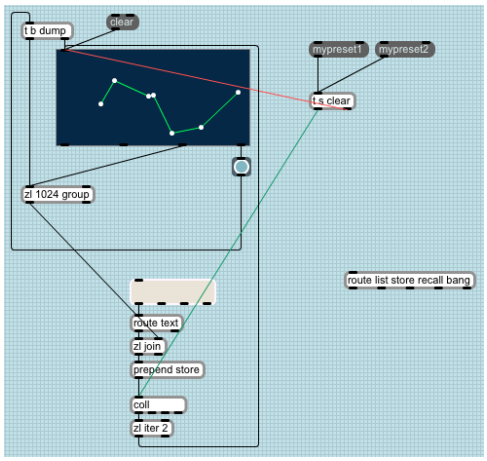


2 Encapsulation

So how can we encapsulate this? Let's start by figuring out what portions of our patch we would like to be within a subpatcher. Here are the relevant objects:

- [zl join]
- [zl group]
- [prepend store]
- [trigger s clear]
- [coll]
- [zl iter 2]

One way to make the process of encapsulation easier is to program the encapsulation *depth first*; that is, to patch as though you're already inside of a subpatcher. In this case, we might want a [route] object to act as a catch-all for our various instructions and data types. Something like [route list store recall bang] should do it. As we'll see momentarily, this handles the various types of input we'll want all in one inlet. This can be advantageous in order to not have too many inlets to keep track of:



Now, break all of the previous connections that were in your patch to the various objects, and instead format them for usage with [route]. What we'll emphasize here is twofold:

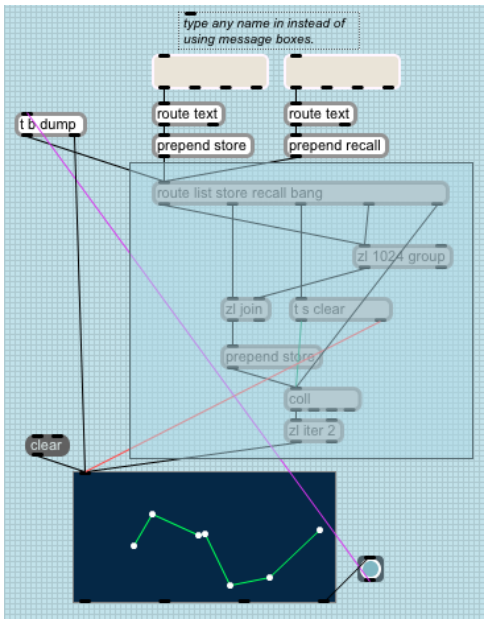
- We'd like our module to respond to special messages like ⟦store⟧ , ⟦recall⟧ , and ⟦bang⟧ . Let's also include lists.
- If none of these are matched, let's have the default behavior go into the "engine" of our module, which is the [coll] object.

Reconnect all of your connections so that they are either reformatted or go to the original desinations:

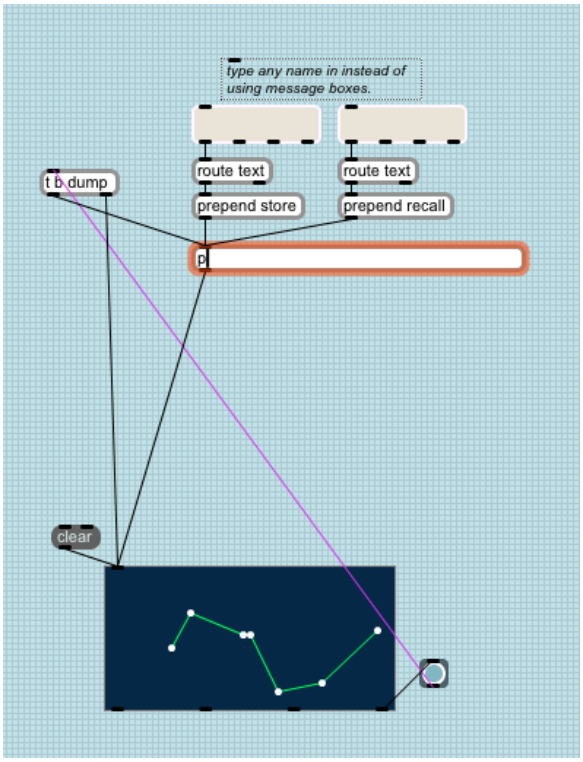
1. Create a new [prepend store] object.
2. Connect your [route text] object's left outlet to the inlet of this new [prepend store] object.
3. Duplicate this object chain ([textedit], [route text], [prepend store]).
4. Rename the [prepend store] to [prepend recall], and connect [prepend recall] to routelist store recall bang.
5. Connect the left outlet of [trigger bang dump] to [route list store recall bang].

   Now make the new [route] connections:
6. Connect [route]'s bang outlet to [zl 1024 group]'s left inlet
7. Connect [route]'s list outlet to [zl 1024 group]'s left inlet
8. Connect [route]'s store outlet to [zl join]'s left inlet
9. Connect [route]'s recall outlet to [trigger s clear]'s left inlet
10. Connect [route]'s *unmatched* outlet (rightmost) to the [coll] object's left inlet. This allows us to reacll any preset name that we want, and also to issue any command to the encapulated [coll]'s contents.

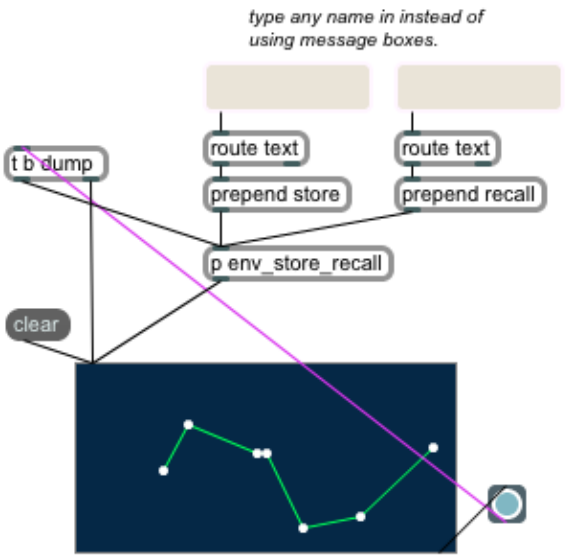
Now select all relevant objects for encapsulation, like so:



Choose "Edit / Encapsulate" to encapsulate the contents:



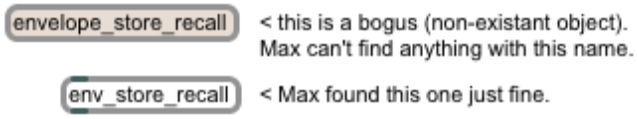
Let's rename this subpatcher to have a name that we'll remember. In our case, we should name it "env_store_recall", as we'll be using this particular filename later:



3 Abstraction

Take a minute to make sure that everything is working with your new subpatcher. If you are satisfied, move on to making this handy utility an abstraction so that we can use many of them easily:

1. Double-click on [patcher env_store_recall].
2. Choose "File / Save-as...", and save the file to disk inside of your max enabled folder (Notice that the filename is drawn from the edited subpatcher name).
3. To test whether or not the file is now seen by Max, name a new object and type "env_store_recall" as the name:



If you'd like, feel free to copy the "cnmat_function_manager.maxhelp" file and replace the abstraction's name there with our new one. Now you're ready to create, manage, and use many envelopes. Think about how this lesson might relate to other objects, in terms of managing their state.