The M Orchestra Language

Miller Puckette
M.I.T Experimental Music Studio

Introduction

Orchestra languages in the traditional software synthesis programs (Music V, Music 10, Music 11, etc.) are all object oriented, in the sense that instruments defined in them have *instances*, which are instantiated and passed exactly one message in their lives. This message is the "score card," whose "parameters" take on different meanings for different instruments. This is a good thing, but we would have more of a good thing if we could pass an instrument messages while it was playing. Then we can handle complex behavior such as slurring by implementing not only an "init" message (the message which starts the note) but a "slur" message as well. If we want to debug an instrument we can send it "print" messages, and so on.

Languages we call "high-level" universally share the features of *functionality* and *abstraction*. Functionality means that we can say "f(g(x))" instead of "y=g(x); f(y)". Abstraction means that we can define a class of objects to have some state and/or functionality which we will use repeatedly. This author has never seen an orchestra language which offers these rather fundamental-sounding features.

The m orchestra language is an attempt to provide these sane, up-to-date tools to instrument writers. The result is instruments which are easier to read, write, and debug than instruments written in less powerful languages. M is the orchestra language in the Music500 system and its extensions; the "mc" compiler turns m code into an intermediate code which music500 interprets.

An m instrument to look at

An instrument consists of a pile of tokens grouped by other tokens. Voici:

```
instr zing()
{
        out1( phasor( 100.));
}
```

The instrument above makes a sawtooth of frequency 100 Hz. and sends it to the channel 1 speaker. The result has amplitude 1. The active ingredient is the `phasor` unit generator. Its output is a vector of floating-point numbers (whose length is a default we may set in the mc command line.) The result is turned into sound by the `out1` unit generator. The phasor was called with 100. as its argument, which sets its frequency. The `out1`'s argument is the output of the phasor.

A more interesting instrument:

```
instr bang ( cps:float, amp:float)
{
        out1( fred:envlp( amp, 5., 1.)
                * oscil( cps, t_sine));
}
```

This instrument outputs an envelope-controlled sine wave; the `oscil` puts out a sine wave of amplitude one, the `envlp` puts out a transient with height amp, rise speed 5, and decay speed 1. The two outputs are multiplied and sent to channel 1 by the `out1.` The oscil's second argument specifies its wavetable; I am assuming here that someone has put a sine wave in the table `t_sine`.

A similar music11 instrument follows:

```
        instr 1
a1      envlpx  p5, 1, p3, 0, 2, -.8, 1
a2      oscili  a1, p4, 1
        outs1   a2
        endin
```

We see immediately that m's facility for expressions with unit generators makes m code much simpler and more readable. Also note that the m code is much more symbolic: the table is named `t_sine` instead of "1"; the parameters are named `cps and \verb`amp instead of `p4` and `p5` and the instrument itself is named `bang` instead of "1."

Message-passing

The example above defines the class `bang`. An object of class `bang` is instantiated in an orchestra by sending the orchestra a `start` message. After instantiating the object the orchestra sends it an `init` message. The orchestra sends any extra parameters in its `start` message to the instrument as `init` parameters. Hence, if `bang` is as defined in the last example, we could tell the orchestra, `start bing bang 440 .5`, which would instantiate an object of class bang named `bing`, and start it with a frequency of 440 and an amplitude of 0.5.

The `init` message is passed only when the instrument is turned on. The unit generators (`phasor`, `envlp`, `out1`, and `*`) are left running as long as the instrument is left on. Bing is turned off by telling the orchestra to stop it.

Messages other than "init" are handled by *methods* we write into the instrument. Suppose we want to teach `bang` to slur. We write:

```
instr bang (cps:float, amp:float)
 mycps:float;
{
        mycps = cps;
        out1( fred:envlp( amp,5.,1.)
                * oscil( mycps, t_sine));
}
to tie(cps:float, amp:float)
{
        mycps = cps;
        trigger(fred,amp,5.,1.);
}
```

In the above example, `bang` has an *instance variable*, named `mycps.` The line, `mycps:float` declares it to be a floating-point number, and it is used as the input to the phasor. It is set not only in the body of the instrument (when the `init` is sent) but in the `tie` method under it. Since the unit generators run continuously, when `mycps` changes the phasor's frequency changes. We might start this instrument at time 1, then at time 2 send it the message, `bing tie 220 .3` to reset the frequency. Note also that `fred` is retriggered in the `tie` method. The phasor in the instrument was named `fred` by invoking it as `fred:phasor.` We may now pass fred messages in any method for the class bang. For example, envlps have methods to `trigger`, which are analogous to triggering envelopes in an analog synthesizer. The line, `trigger(fred,amp,.5,1.)` in the tie method passes fred the message, `fred trigger .5 1.`

The name `fred` is internal to the instrument `bang.` We can't pass it messages from outside bang. More generally, everything defined in bang except `bang` and `tie` has a scope which lies within the body of `bang`, and variables defined within methods (including the init method) are only defined in those methods. Scoping thus looks like that of the c language except that `mycps` is inaccessible from outside the instrument and unit generator names such as `fred` may be accessed in any method of bang.

Defining unit generators

Some unit generators in m are primitives, which are interpreted directly by the orchestra interpreter. Others, such as `envlp,` are themselves defined in m. Here is envlp, with one minor simplification.

```
ugen envlp(atarget:float, rise:float, dec:float, outp:&float[])
 up:float[];            !output signal of rise expon
 down:float[];          !and of fall expon
 starget:float;         !current target value
{
        starget = atarget;                              !copy target
        up = target - risex:expon(atarget,rise);        !rise expon
        down = decex:expon(1.,dec);                     !fall expon
        outp = up*down;
}
to trigger(atarget:float, rise:float, dec:float)
{
        wherewas:float;                         !would-be next-output
        wherewas = get.value( decex) *          !figure out what next
                ( starget - get.value( risex)); !output should be
        starget = atarget;                      !copy in new target
        set( risex, atarget - wherewas, rise);  !rise expon decays away diff
                                                !between actual and target val;
        set( decex, 1., dec);                   !and dec envelope eats away at
}                                               !that.
```

The difference between a `ugen` and an `instr` is just that a `ugen` does not have some methods which an `instr` has, which allows them to be started and stopped from a score. We see the data types, `float[]` and `&float[]`, which define a floating-point array and a pointer to a floating-point array, respectively. An envlp is passed its output as its last argument, named `outp`, and its inputs as they are given in the calling instrument; in the case of `bang`, they are `amp`, .5, and 1, which are put in `atarget`, `rise`, and `dec` in the `envlp`.

When an instance of `bang` is told to slur it tells `fred` (its included instance of `envlp`) to trigger. This message causes `envlp`'s `trigger` method to be run, passing it as parameters a new amplitude, rise speed and decay speed.

As another example, here is the oscillator:

```
ugen oscil(freq:&float[],tabp:&float[],
                outp:&float[])
{
        outp = table(phasor(freq),tabp);
}

sine:&float[];

instr bang (cps:float, amp:float)
 mycps:float;
{
        mycps = cps;
        out1( fred:envlp( amp,5.,1.)
                * oscil( mycps, sine));
}
to tie(cps:float, amp:float)
{
        mycps = cps;
        trigger(fred,amp,5.,1.);
}
```

The sine wave table is loaded from the score; all the orchestra needs to allocate is the pointer. Wave tables are indistinguishable from other floating-point arrays. The definition of `sine` is located outside any instrument, hence it is global and can be accessed through the score. A sample segment of score for `bang` (written in terms of messages to the orchestra and ignoring timing) would be

```
loadtable sine sine.f
start fred rats 440 .5
mess fred slur 220 .4
stop fred
```

The `loadtable` reads in a binary floating-point array from the file `sine.f` (ascii is also available) and sets the global variable `sine` to be a pointer to that array. The `start` message sets `fred`'s unit generators running until the `stop` is encountered. The `mess` instructs the orchestra to send the message `fred slur 220 .4` to `fred`. If we have more than one instance of `bang` around, both read the same table but they receive messages independently.

Calling by reference

The code for `oscil` above demonstrates a current confusion in m between things and pointers to them; unit generators inside `oscil` are called with pointers to arrays instead of arrays. This reflects the fact that in fact all arrays are silently passed by reference to unit generators. A better situation would be:

```
ugen oscil( freq:float[], tabp:float[],
        RETURN:float[])
{
        RETURN = table(phasor(freq),tabp);
}
```

This reflects two changes. First, arrays are always passed by reference; and declaring an array argument actually declares a pointer to it. Hence, there is no "&" in the argument definitions. Then since the table and phasor take pointers anyway, we simply pass the pointers to them. This eliminates the ambiguity in the source code in which we sometimes pretend to pass arrays by value but not always. Second, "RETURN" is taken to be a special argument name implying that `oscil` is to be called with a "return value" which is implemented by sending a pointer as the `oscil`'s last argument. This is simpler than trying to add a real `return` to m. Both of these changes are forthcoming.

Conclusion

We have seen that it is indeed possible to write a more elegant and usable orchestra language than in the past, and to implement it in a software synthesis system. It is hoped that this style of language design will catch on among system designers in the future.