Remembering Performance Gestures

Curtis Abbett
Lucasfilm Ltd
20 August, 1982

ABSTRACT

This paper considers the problem of memorizing and editing gestures made by performers interacting with digital audio equipment. In particular, we present the solutions currently incorporated in the Lucasfilm Audio Signal Processor (ASP). These solutions draw on our experience with the needs of electronic musicians in developing pieces, and on the needs of practicing professionals in the audio recording and mixing field.

1. Situating the Problem

The Lucasfilm ASP [1] is designed to allow interfacing of up to 7 controlling computers, one configured as the Master Computer, handling global control such as clocking, and managing the timed update queue built into the hardware of each DSP in the ASP. The other controlling computers are conventionally called "Console Computers." From the point of view of the ASP, their role is to supply update values for immediate execution. They can be used to affect values in any of the ASP's update-accessible memories: scratchpad register banks A and B, and the microcode memory. Each connection of a controlling computer to an ASP is mediated through an update interface.

Since rather powerful computers (in our case, Motorola 68000's) can be placed "between" the physical performance input devices and these update interfaces, it is practical to attempt a good deal of processing at this stage. In this context, it can become difficult to decide what processing to do and how to organize it for maximum flexibility and utility. Such decisions depend, of course, on what the perceived needs are.

A primary design goal of the ASP was to have a system which could be used for film sound mixing and editing which would perform better than current (non-digital) hardware for this purpose. The people who will be using the ASP for film production purposes are doing a job which sometimes requires creativity, and which always requires attention and precision, but which is at the same time, very well defined. As a rule, these people are highly focused and require tools which are optimized for their needs. They have evolved ways of thinking about memorizing and editing gestures, and their attitude to new ways of doing things ranges from enthusiastic to rather hostile. At our site, we have had the good fortune to be working with people who are interested in doing things in new ways.

While the ASP must serve the film production application, we also want it to be as general-purpose a system as possible, which can address the needs of computer musicians in particular. The needs and attitudes of computer musicians are quite different. By definition, they are interested in new ways of doing things. While most computer musicians dream of having a great deal of functionality, they usually want it presented so that they can define themselves how it is to be used.

Thus, we confront a familiar dilemma: "optimized" presentation of highly developed but essentially canned features versus an approach wherein more primitive features can be composed by users so that they can develop their own set of highly developed ways of doing things. Our
2. Basic Concepts

Let us start by stating a very general problem which we call the parameter definition problem defined as "what kind of processing do we put between the input devices (and here we can view scores as input devices) and the synthesis hardware?" It is reasonably natural to divide this into two subproblems, which we will call the gesture objectification problem and the parameter mapping problem.

By gesture objectification we refer to the problem of getting a handle on 'what a performer is doing. A nice abstract way to model things like gestures is to use the notion of time-varying functions. A classic example of this way of objectifying gestures is found in the GROOVE program [2]. We will adopt a similar approach.

The parameter mapping problem amounts to this: varying what gestures mean in terms of synthesis parameters. In this general form, it is clear that it is a problem of the kind that programming languages are usually designed to address. It has not yet been dealt with in a completely satisfactory way in computer music, and we will deal with it only to a limited extent in the current paper.

This paper is mainly concerned with our solutions to what we call the gesture objectification problem. The basic idea is to treat gestures as time-varying functions, which we call Console Functions. We will often refer to these as CF's. Each knob, slider, etc. is accessed in the control window by an associated CF. We will sometimes call these physical CF's. It is also possible to create CF's which are not directly associated with a physical input device, essentially by "spinning them off" from physical CF's - these are also known as detached CF's.

Now, time-varying functions are a good way of modelling gestures, but we still need efficient means of remembering and editing these functions, and of switching between memorized and live performance conveniently. We have adopted the methods used in commercial music and film recording and mixdown - known as automation in the industry. Basically, the idea is that a physical CF can be in three states: read, write and update. In the read state, the "active version" of the function is memorized one. In write state, the active version is the live performance on the input device, and that version is remembered for possible later use. In update state, the live and memorized versions are combined, with the live version being an offset.

The memorized version of a CF is stored in a data structure called a gesture memory buffer, abbreviated GMB. Each physical CF consists of a GMB and the input from the performance device, and each detached CF consists solely of a GMB. Thus, detached CF's can be thought of as saved versions of a physical CF (although in principle, they may be defined in other ways as well). In terms of the automation function, a detached CF is always in read state.

The information stored in a GMB is noise or less directly due to the actions performed on a physical input device. As a practical matter, we want to roughly match the quantization of that information, both in time and in value, to the characteristics of human performers. In the current implementation we aim for 3 time quantization of around 5 milliseconds and we store a value in 10 bits, measured linearly with respect to device motion and in a fixed point 0.5's complement representation.

3. How Console Functions Are Used

Before going into more detail about how CF's work internally, and how performers deal with them, we consider how they are used to affect synthesis and processing parameters. The program which controls an ASP is called FMX. One of the basic abstractions in FMX's is the notion of plug, which produce or accept time-varying functions (signals). A command is provided to convert source plug to sink plug and this is how most things get done in FMX. Naturally enough then, CF's are modeled as source plugs. Typically they are connected to sink plugs.
which act as control parameters to some synthesis or processing algorithm realized as a micro-
coded function in a DSP.

This is the simplest case: the CF information, interpreted in simple units such as Hertz [for
in any case, the internal numeric scaling of CF's is unlikely to be of any direct value], is sent
directly to one or more mix plugs in one or more DSP's. This case addresses the parameter map-
ning problem essentially by ignoring it, because the meaning of the CF is taken to be related to
the synthesis parameter by a simple multiplicative scaling which amounts practically to an iden-
tity.

Even in the film production application, this is not enough. For this reason, a more compi-
cated mechanism for mapping CF's to parameters is built into FMX. This mechanism allows
several inputs to affect several outputs according to functions computed by any of the standard
arithmetic operators plus table lookup. This is approximately the level of complexity, in paramete-
ring mapping, that could be obtained in 4CED, and it is somewhat less than can be obtained in
something like MUSIC 60's LONLE code. I would not claim that it represents a complete answer
to the parameter mapping problem, but it does give enough power to keep life interesting. We
will examine it in more detail later.

4. More About How Console Functions Work

We have already said that Console Functions typically have a GMB, a sequence of values
from the input device, and a state which says how these are combined to determine the value of
the Console Function. We have also said something about the quantization of time and value. In
this section, we give more detail about all this.

It is especially important to consider what happens when the state of a CF changes during
its execution. This is complicated by the fact that several kinds of transitions involve the possi-
bility of discontinuous changes, which we probably want to avoid. Let us consider what each
kind of transition means:

• read--write means we go from using the GMB value to using the live value, and start
  memorizing the live value.

• read--update means we go from using the GMB value to using the GMB value offset by
  the live value, and start memorizing the combined value.

• write--read means we go from using the live value to using the GMB value, and stop
  memorizing the live value.

• write--update means we go from using the live value to using the GMB value offset by
  the live value.

• update--read means we go from using the combined value to using the GMB value and
  stop memorizing the combined value.

• update--write means we go from using the combined value to using the live value alone.

In the following discussion, we will sometimes abbreviate these transitions by the first letters
of their components: read--write becomes RW, etc. The RW and RU transitions and their
inverses, WR and OR, are the most useful. WU and UW are somewhat strange in that the
meaning of the live value (that is, of the performer's gesture, in the most literal sense) suddenly
changes: in write state, it is used directly, while in update state, it is used as an offset.
Nevertheless, we allow all the possible transitions. The RW and WR transitions involve possible
discontinuities since read and write states imply selection from one of two possible data streams.
The obvious strategy is used to smooth these discontinuities: we interpolate at a given rate from
one stream to the other. The rate is a parameter: if it is zero, that corresponds to one of the
usual strategies in conventional analog consoles, in which the actual transition is delayed until the
values of the two streams match. It is also possible to set the interpolation rate to infinity—this
effectively turns off interpolation altogether.
Let us consider the form of the transition interpolation algorithm a little more carefully. We have a time-varying function $M$ (the memorized function) and another $L$ (the live function). Say the transition begins at time $t_1$. We wish to compute a function $D$ (the derived function) which is $\text{Interp}(t_1,t,M)$ for the WR transition, and $\text{Interp}(t_1,t, M, r)$ for the RW transition: the linear interpolation of $L$ and $M$ starting at $t_1$ at rate $r$. This function is further complicated by the proviso that the interpolation proceeds only until the function values cross for the first time. It is straightforward, if a little tedious, to compute this function.

Now consider the $\text{RU}$ and $\text{IR}$ transitions: these are the expected transitions when we wish to slightly modify a previously recorded gesture. In the $\text{RU}$ transition, we take the zero point of the offset to be the value of the live input device at the moment of the transition. This policy implies that no discontinuity can occur on a $\text{RU}$ transition. On the other hand, a discontinuity can occur on a $\text{IR}$ transition if the live input device is not at its zero point at that time. The strategy described for $\text{RW}$ and $\text{WR}$ transitions can also be used here. If the interpolation rate, $r$, is zero, we wait until the live value reaches its zero point.

An important consequence of the algorithm just described for avoiding discontinuities associated with state transitions is that the time of the actual transition may be indefinitely delayed. Typically, the performer would indicate that a transition is desired, by pushing a button (the interpolation rate parameter, $r$, is set globally). Since the transition may not take place immediately, it is important to tell the performer when it does take place, and perhaps to give more feedback that this.

It is difficult to provide general solutions to these feedback issues, so we will only mention how we are doing it. The console which we plan to use for film production work [53] has two buttons and four LED's physically associated with each knob and slider, any of which can be dedicated to state transitions and feedback for the $\text{CF}$ associated with that device. We use one button to toggle between read and write states, and the other to toggle into and out of update state. Of the four LED's, we use one for each button to indicate its current state (since the buttons are momentary contact, this is essential) and another to indicate that a transition has been ordered but not yet happened. The fourth LED is unused, for the moment.

We also plan to optically make use of the bar graph displays on the console to show performers the $L$ and $M$ functions during transitions. This will be especially useful when interpolation is disabled.

To finish our discussion of how Console Functions work, we briefly consider how GMB's are stored. This is fairly important because they can take up a lot of space, and we would like to be able to keep a lot of them around. The value of a $\text{CF}$ typically changes quite slowly -- this suggests a differential (delta modulated) encoding for values, which is what we use. Encoding the time field is a little more interesting: the interval between changes in the value of a $\text{CF}$ is typically either very short or very long -- short when the $\text{CF}$ is being actively used, long when it goes through a period of inactivity. If we thought that $\text{CF}$'s were going to be active all the time, the right thing to do would be to sample the function at regular intervals so that we wouldn't have to store the time at all. Another possible approach would be to try to represent the $\text{CF}$ as a list of sampled functions with associated begin times. Making this work would involve noticing when the $\text{CF}$ was actively changing and doing some post processing to sort it all out.

We feel that somebody should do the experiments that would tell us what the best data reduction scheme is for this particular application. Meanwhile, we have proceeded on the basis of intuition. We encode both value and time explicitly. Value and time are both delta modulated in 8 bits. Thus, normal changes fit into 16 bits. Occasionally, either the value or time delta will overflow (this should be very unusual for the value delta). Escape values are reserved for this case, which allow 16 bits for value and 32 bits for time. These values are padded for several reasons: padding allows improved management of the memory reserved for GMB's, and each page can have a header containing useful information. For example, we store absolute time and value for the start of page in each page header, which allows us to step quickly through a GMB and start in the middle without laboriously going through all the delta values.

191
More About Parameter Mapping

In this section, we describe how we are currently doing parameter mapping. One design consideration is that since the control program, PMX, views things as flags, we will still want parameters which are subject to some kind of relatively complicated mapping to appear as flags. Another is that we wish to gracefully handle the situation where a number of synthesis parameters depend on a number of input streams, and the dependencies are non-rectangular. A simple but pervasive example of this kind of situation occurs with filter coefficients: in a standard 2-pole filter section, we have two user level controls within which we can call $f_c$ (cutoff or center frequency) and $f_r$ (choke to the null ring, in filtering terminology - this corresponds roughly to the steepness of the rolloff). In the code that realizes the filter section, we have two related coefficients which we will call $a$ and $b$. The relationship is the following: $r = \frac{a}{2b} f_c$ and $f_r = f_c$. We cannot compute a from either $f_r$ or $r$ independently - we must have them both. An implication of this is that whenever either $f_c$ or $r$ change, $a$ is liable to change as well.

We currently handle this kind of situation with a mechanism called ganged plug. A ganged plug is a collection of zaps plugs, along with a program which tells how to calculate all of the outputs whenever any of the inputs changes. This program is called a scaling transform and is written in a kind of calculation language, details of which are given next.

A scaling transform definition names its inputs and outputs, then gives a program for a stack machine which computes the outputs based on the inputs. One possible operator is table lookup, and a scaling transform must also declare which lookup tables it is going to use. The three kinds of declarations are introduced by the keywords input, output, and requires, respectively. The operators are as follows: add, sub, mul, and div pop the top two items from the stack and perform the indicated arithmetic, putting the result back on the stack. Two stack manipulation operations are provided: dup duplicates the top stack item, and swap swaps the top two items. Two operators take explicit arguments: ldup takes the name of a lookup table in parentheses, and uses the top stack item as an address in that lookup table, replacing it with the value at that address; addl takes the same of an input item in parentheses, pops the top stack item and places it in the indicated output slot. As input data or constant is placed on the stack simply by mentioning it. A few notes about numerical considerations: input and output items are always 24 bit fixed point numbers, but internally all arithmetic is 32 bit (IEEE standard) floating point. Lookup tables consist of 512 32 floating point numbers. The address argument to ldup is, of course, a floating point number, but which is converted to a 24 bit fixed point number which is interpreted as is the ISP's; the high order bits in this case address the table directly, and the low order bits are used for a linear interpolation.

A simple example should make this description clearer. We give a scaling transform for the example of filter coefficients discussed above.

```
input cf, rad;
output a, b;
requires cosw;
```

```
0 "Log(cosine)" rad 2 mul mod setout  // first operation negates
out(a) rad rad rad out(b)                // ganged plugs and scaling transforms can only be defined when the control program, PMX, is started, and cannot be modified subsequently. Thus, they are part of the static environment, in a manner of speaking. Since a general solution to the parameter mapping problem must make it convenient for mappings to change dynamically, this could be seen as a lost. Still ganged plugs seem to be a good design for fairly simple parameter mappings which don't change very often. We hope to report more fully on our experience with them at a later date.

192
6. Using Buttons

Although it is possible to think of buttons as input devices which generate functions of time taking only two possible values (on or off), they generally serve a rather different function, in the mind of the user, from that of more inherently gestural devices. Accordingly, we treat buttons (and analogous devices such as keyboards) as generators of events rather than functions. Naturally, it is possible to remember the timing of these events, and reproduce them later without explicit user intervention. However, because of the role of events in the larger scheme of control software for the ASP, this is more conveniently seen as the integration of button derived events into a score, rather than as the replay of a time varying function. Furthermore, as we have hinted above, the state transitions of physical Console Functions are typically mediated by buttons associated with the input device to which the Console Function corresponds. In this case at least, there is a logical necessity to treat buttons more simply than Console Functions, otherwise we get into an infinite regress.

7. Putting It All Together

The format we have set out for Console Functions allows for a single saved version and a single live version. Sometimes, users may want to save several versions. This can easily be done by creating a *detached* Console Function from the *GMF* of some physical one. Given the ability to detach UP’s under different names, one can build up more complicated facilities, such as saving some number of previous versions, instead of just 1, or checkpointing (on request), etc.

In designing software for handling performance input, we adopted the Console Function concept for several reasons. As stated above, we feel that the general notion of presenting gestures as time-varying functions is flexible and extensible, while at the same time, easy to grasp. We incorporated a notion of state, into the Console Function, as it were, because in the film sound application, it is clear that gesture objectiveion must include facilities for incrementally modifying and perfecting performances. Indeed, this is natural whenever one has the opportunity to perfect a performance before it goes in front of an audience. When this is not the case, the ability to edit and modify gestures (as Console Functions) does no harm, excepting a possible performance penalty.

We would argue, then, that our conception of Console Function is of considerable use in many (though not all) musical situations, and we hope to gain experience with it in dealing with gestures derived from sources other than the traditional knobs and sliders.

Acknowledgements

We wish to acknowledge the considerable contributions of Jim Lawson in the elaboration and testing of programs to implement the ideas presented here.

References


193