COMMANDER: FAST LIVE ACCESS TO MUSICAL COMMANDS

Tom Ritchford

Swirly Labs
76 North 7th St, Brooklyn, NY

ABSTRACT

Commander is an organizational system to allow a performer with a limited interface to quickly and intuitively execute musical commands of the most general nature. Commander organizes commands into nested cycles and allows fast, error-tolerant navigation without much memory work on the part of the performer. While this idea is applicable to any computer music performance system, a reference Max For Live implementation in Javascript is demonstrated as a MIDI effect that dynamically controls an Ableton Live session.

1. INTRODUCTION

A computer musician performing today has potential access to an unprecedentedly huge number of sounds and effects, but in practice must spend long periods hunched over the computer to access the desired musical material. Three goals apparently conflict: the desire to access the largest possible volume of material, the need to easily move from scene to scene when prearranged sets are needed, and the necessity that the performer be able to recover gracefully from errors. Solutions are also constrained by the limited number of mode or "program change" buttons on many musical instruments, and the limited ability of the performer to memorize long patch tables.

Commander solves many of these issues neatly by organizing named musical commands in nested, circular lists called *cycles*.

Commands are nearly always reached by advancing a cycle at some level – in the case of a pre-written song or set, this allows for "one button operation". Any given command can be accessed in multiple ways at the convenience of the performer, so it's possible to quickly reach a small set of very useful commands while still being able to reach the entire universe of commands with only a few more gestures.

An unlimited undo and simple navigation that makes it intuitive as to how to "get back to where you were" handle mistakes and encourages the performer to wander around and experiment.

Commander stores its data as editable text files in the JSON [6] format. In the reference implementation, you can also edit the Commander data directly from Ableton Live [3] while the system is running.

Commander data is by default stored in a single text file but it's perfectly possible to split it into multiple data files where a central file includes others. This makes it easy to create a set on the fly in a minute or two with just a short text file that refers to each song. In fact, the Commander "data" is in fact directly interpreted as Javascript code so the venturesome performer has the full power of Javascript and access to many utilities already written.

2. ABOUT THE REFERENCE IMPLEMENTATION AND THE DEVELOPMENT ENVIRONMENT

The reference Commander implementation is written in Javascript [1] residing in a js box in a Max For Live [4] patch and can be downloaded at the Commander homepage [7].

The Max For Live patch starts empty except for the js box – the Javascript program creates, links and deletes user interface elements in the patch as needed.

The code is broken into small files, each representing a single Javascript function, a Javascript class, or even just one method on a class. The files are combined using a small Makefile [5] and gcc [2].

Unit tests are compiled and executed in a separate Max patch that is not needed by the performer. For testing, the Max/Max For Live environment is simulated by mocks, so a test can examine the actual messages a program sends to the outside world.

This development environment is discussed at greater length in another article. [8]

3. DEFINITIONS

3.1. Commands

Let a *command* be some executable object or description that somehow affects the internal program state or external environment. The definition of a Command is intentionally abstract; in the reference implementation, for example, a command resolves to a Javascript function. When executed, a command either returns nothing, or it returns the *inverse command* – a command that undoes what just happened. Inverse commands are saved on a stack to allow later undoing.

3.2. Cycles

A *cycle* is either a *leaf*, containing only a command, or a *node*, a non-empty ordered *list of cycles* together with a *selected index* to the *selected cycle* in that list. Both leaf cycles and node cycles have an optional string *name*.

Given some descendent cycle recursively contained in an ancestor cycle, we define the *level* between them to be zero if the two cycles are the same, or, recursively, one plus the level between the descendant's parent cycle and the ancestor.

A *fixed level* cycle has a fixed level between all its leaves and itself. We'll assume that all cycles are fixed level unless otherwise noted, and talk freely about this fixed level as the *level* of a cycle.

The *selection* for a cycle is the sequence of cycles obtained by taking the selected cycle recursively until a leaf is reached. We talk about the leaf being at the *bottom* of the selection at level 0 and the original cycle as the *top* of the selection. Because of the fixed depth condition, and the condition that node cycles are always non-empty, the length of a cycle's selection is one greater than its level.

4. ADVANCING NODES AND SELECTIONS

Advancing is the fundamental operation in Commander.

Advancing a node simply increments the node cycle's selected index, using modular or circular arithmetic so the selected index always within list index boundaries.

Advancing a selection advances the cycle containing a specific level in a selection, thus changing all levels in the selection below that cycle, then executes the command that has now appeared on the bottom of the selection.

Advancing a selection with carry has the additional property that when the selected index in the cycle wraps around to 0 again, the cycle in the level above, if any, is advanced as well. Repeatedly advancing a selection with carry at level 0 will iterate through all the cycles

The definitions above logically extend to advancing a cycle or selection with an increment greater than one or less than zero.

Advancing a selection is the industrial-strength operation that makes Commander useful. While there are editing commands for cycles and selections, it's anticipated that for larger projects with complex scores the performer would prepare everything as text files and use only Advance commands in the final performance.

5. NAME TABLES AND DATA CONTEXTS

A *name table* is an associative array that maps string names to commands or cycles, preventing the performer from having to look up tables of cycles, program changes, controllers or other numerical data when creating an Commander data file. Name tables are also stored as JSON text files; this makes it easy to copy data like a program change table from a manual and edit it into the right format.

A name table is a kind of data definition, a generic facility for adding new symbols and functions to the context used to interpret data files. As mentioned above, our data files are stored as JSON files. Javascript, the language of our reference implementation, as well as other languages like Python allow you to evaluate this JSON data within a *data context*, a collection of *data definitions*, each of which is an assignment of a values to a named variable that can be later used in the Commander data file.

Data values need not be static data: values can be functions, and in the reference implementation the data values could even be fragments of Javascript, though it's recommended to separate the code from the data for many reasons (if nothing else, because it's hard to write code fragments back out to a data file if you're editing on the fly).

6. USER INTERFACE AND DISPLAY

Because of the stripped-down nature of Commander, only a small display is needed. Indeed, the program has been used to run shows where the performer is unable to see the display at all.

More typically, the interface just shows the selection with a few controls for each cycle.

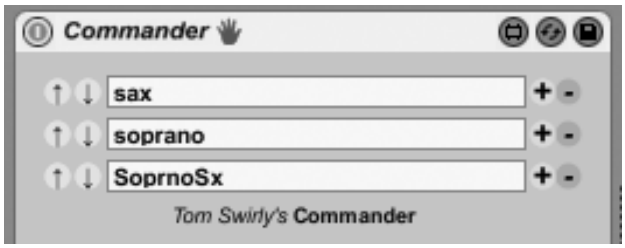

Figure 1. The user interface of Commander's reference implementation, as seen in Ableton Live.

In Figure 1, cycles in the selection are listed from top to bottom: a node cycle named "sax" contains a node cycle named "soprano" which contains a leaf cycle with a command named "SoprnoSx". The up and down arrows to the left of the cycle names Advance the cycle forward and back. These may be mapped to some external controls using the facilities of the hosting system - in the reference system, that facility is Ableton Live's "MIDI mappings" table, though there is special-purpose code to work around the issue that Live's MIDI mappings don't respond to program changes.

Figure 2 shows the same user interface after Advancing the top cycle. Note that all the cycles in the selection have

changed; if the bottom cycle had been Advanced, only it would have changed.

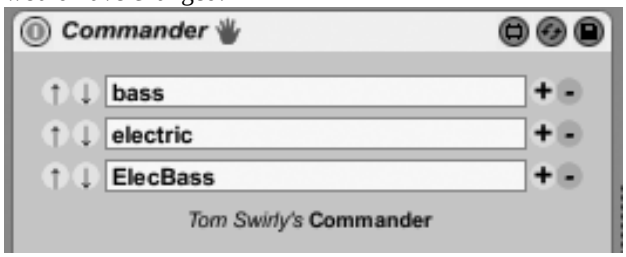

Figure 2. The user interface after Advancing the top cycle.

The + and – buttons to the right of the cycle names add and delete new cycles (a text box will appear to edit the command definitions when you add a new leaf). In practice it's unnecessarily cumbersome to build cycles or edit from the user interface and much faster to make changes to the JSON text file, so these buttons see little use.

7. EXAMPLES

A single performer has just three program change buttons but a large number of sound patches and so decides to organize them hierarchically into three levels.

Group	Type	Patch
string	violin	Bright Violin
		Pizz Violin
		Gypsy Violin
	cello	Bowed Cello
		Plucked Cello
bass	electric	Fender Bass
	synth	Moog Bass
		Arp Bass
percussion	drums	WX Drums

Table 1. Example sound patches.

For simplicity, Table I only has a handful of patches but the author's own patch cycles contain hundreds of programs in dozens of categories.

Let's call these four buttons 0 through 2 and assign them to Advance the cycles at level 0 through 2 (without carry).

Initially, the performer has selected Group "string", Type "violin", and Patch "Bright Violin". Pressing button 0 from this state will cycle through the three violin patches "Bright Violin", "Pizz Violin", "Gypsy Violin" repeatedly.

Pressing button 1 will cycle through the two types of strings, "violin" and "cello". Note that when we return to the cycle "violin", we also return to the most recent patch selected within that cycle – the selected cycle does not reset itself just because we leave and re-enter that cycle.

Finally, pressing button 2 Advances through the three cycles "string", "bass" and "percussion". Note that "percussion" contains just a single cycle "drums" which contains a single cycle, "WX Drums", so if your top-level cycle is "percussion" then neither button 0 nor button 1 will have any effect (they'll try to advance a cycle of size 1).

```
{
  'string': {
    'violin': {
      VL['Bright Violin'],
      VL['Pizz Violin'],
      VL['Gypsy Violin'],
    },

    'cello': {
      VL['Bowed Cello'],
      VL['Plucked Cello'],
    },
  },
  // …more entries here omitted.
}
```

Table 2. An example of a Commander data file.

Table 2 shows the JSON data file that represents the Commander session described in Example 7.1. The symbol "VL" is an example of a name table, in this case representing patches for a specific instrument. While the name table in this case only selects a patch for an external instrument, it's important to remember that a command can be any operation or list of operations at all: the reference implementation allows the performer to specify almost any change to the Ableton Live environment by using the facilities of Max For Live.

```
'VL': {
  'Moog Bass':     BankPC(1, 56),
  'Bright Violin': BankPC(1, 75),
  'Pizz Violin':   BankPC(2, 15),
  'Gypsy Violin':  BankPC(1, 74),

  // …more entries here omitted.
}
```

Table 3. An example of a Commander name table

Table 3 shows a section of a name table for a single instrument. It uses the name BankPC from the data

context, a function which sends out MIDI bank changes and program changes to a MIDI instrument.

```
{
  'patches':
    Read('patches/patches.data'),

  'songs': {
    'a-train':
      Read('songs/a-train.song'),

    'Help':
      Read('songs/help.song'),

  // …more entries here omitted.
  }
}
```

Table 4. A Commander data file that includes other files.

Table 4 shows how to read external files from a data file. Data files and cycle definitions from one project can be included entirely inside another project – for example, the file patches.data could be exactly the file referenced in Table 2. The same facility is used to load name tables or even new functions into Commander without having to change the original program.

8. REFERENCES

[1] https://developer.mozilla.org/en/JavaScript
[2] http://gcc.gnu.org/
[3] http://www.ableton.com
[4] http://www.ableton.com/maxforlive
[5] http://www.gnu.org/software/make/manual/make.html
[6] http://www.json.org/
[7] http://www.swirly.com/commander
[8] Ritchford, Tom, "Rapid Development of Large, Reliable Music Programs In Javascript and Max", unpublished essay, 2010