An Animation Extension to Common Music

Bret Battey
Center for Advanced Research Technologies in the Arts and Humanities
and the School of Music Computing Center,
University of Washington, Seattle
bbattey@u.washington.edu
http://www.BatHatMedia.com/

Abstract

The Animation Extension to Common Music (AECM) version 1 is a set of extensions to the Common Music (CM) infrastructure. These extensions allow musical event algorithms authored in CM to also generate scripts to control a computer animation environment. The current version of AECM works with Common Music 1.4 and generates MaxScript, the scripting language for 3-D Studio Max 2.5. While facilitating the use of algorithmic methods for generation of both audio and visual events, it can encourage reconceptualization of relationships between sound and image mediums. Examples are provided from the author's recent work *Writing on the Surface* for computer-realized sound and image.

1. Introduction

There has been a strong surge of interest in recent years in the relationship between image and sound and the composition of works that integrate the two mediums as equals. Activity in this arena has been encouraged in part by the fact that software packages for manipulating different time-based mediums often share similar timeline-based interface paradigms.

However, algorithmic approaches to texture generation are not easily supported by these time-line interfaces. Further, while there are strong traditions of procedural approaches to creation in both computer music and animation (less so in video), the tools that support such approaches are often focused on a single medium and support other mediums in only a cursory fashion.

The Animation Extension to Common Music (AECM) is designed to allow one to use a single programming environment to generate both sonic and visual controls or events. In particular, it extends Rick Taube's Common Music (CM) LISP-based algorithmic music tools [URL 1] to also generate computer animation scripts. AECM version 1 can create MaxScripts to control the 3-D Studio Max 2.5 animation software [URL 2]. AECM works with version 1.4 of CM. CM works on numerous platforms. 3-D Studio Max operates under Windows NT.

2. General Approach

MaxScript has constructor and modification functions for each of the vast array of object types in the 3-D Studio Max environment. In MaxScript, one creates a new "node" or object in the scene by calling a particular construction function. For example, the following line of MaxScript would create a sphere with the name "sph01" and radius of 2 units:

```
sphere name:"SPH01" RADIUS:2.0
```

Or a box:

```
      box name:"myBox" HEIGHT:3 WIDTH:5
```

AECM, on the other hand, does not provide a specific constructor function for every class in MaxScript. Instead, AECM provide a handful of generalized functions that allow the LISP programmer to generate appropriate scripts for any MaxScript object or modification. For example, rather than run a "sphere" function, the programmer utilizes the general `make-node` function and passes it the node type, desired name, and, optionally, one or more attribute pairs (an attribute name followed by a value):

```
      (make-node 'sphere 'sph01 'radius 2.0)
      (make-node 'box 'myBox 'height 3 'width (* amplitude 6))
```

Therefore, use of AECM requires understanding MaxScript; the programmer must know valid MaxScript node types, attributes, and values.

Once a node is created, one can alter attributes of that node. This activity is supported in AECM with a generalized `edit-node` function to which one passes a node name and pairs of node attributes and values.

```
      (edit-node 'myBox 'height 5.2 'width 8)
```

As per normal for animation tools, animating nodes and their attributes involves setting keyframes and attributes for those keyframes. In 3D Studio Max in particular, one assigns a "controller" of a desired type to an attribute and then keyframes the controller. For example, one might assign a "bezier_position" controller to the position of a box. Then one makes keyframes for that controller, thereby creating a bezier data curve that controls the position of the box over time.

In AECM the assignment of a controller is achieved with the `assign-controller` command, and keyframes are generated with the `add-key` function. Here is an example that sets the box to position x=3, y=4, z=30 at time 0 seconds:

```
      (make-node 'box 'myBox)
      (assign-controller 'myBox 'position 'bezier_position)
      (make-key 0.0 'myBox 'position 'value (point3 0 4 30)
            'outtangenttype bz-slow)
```

This code segment also demonstrates some of the utility functions provided in AECM. The `point3` function formats the x y z values in the point3 style required by MaxScript for a bezier_position key's position attribute. AECM provides a core set of such converters, though if one knows the needed MaxScript string one can simply insert that instead of using a converter. In this case, one could replace the `(point3 0 4 30)` with `"[3.0,4.0,5.0]"`.

The value `bz-slow` above is an AECM defined constant that contains the standard MaxScript string for requesting a slow-release bezier curve. Again, this is just a string substitution issue: it may be easier for the programmer to remember `bz-slow` than MaxScript's `#slow`, which would have to be expressed in LISP as `"\#slow"`. The choice is up to the user.

Another crucial component of MaxScript is the "modifier". Modifiers are special tools for altering nodes. For example, one might apply a "bend" modifier to an object and then animate the bend. In AECM, one uses the `add-modifier` command to apply a modifier of the desired type and name. A node has an array of modifiers, so if we want to change attributes of our modifier, we have to use MaxScript array references, as in the following:

```
(make-node 'box 'myBox)
(add-modifier 'myBox 'bend 'myBend)
(edit-node 'myBox "modifiers[#myBend].center"
      (point3 0 0 6))
(assign-controller 'myBox "modifiers[#myBend].angle"
      'bezier_float)
(make-key 0.0 'myBox "modifiers[#myBend].angle"
      'value 80 'outtangenttype bz-smooth)
```

This demonstrates again the AECM is not a substitute for understanding the workings of the MaxScript language.

Finally, if a function is not available in AECM that is capable of generating a needed MaxScript command, the programmer can resort to the "add-string" function to add a specific string to the MaxScript:

```
(add-string "animate-on (")
```

3. Output

All AECM commands are queued in the order in which they are executed in the LISP code. This means that some thought must be given to how one orders the code. However, as long as nodes are created before any edits that reference those nodes, things will work fine. In practice this is quite easy to achieve. Also, since the animation is enacted via keyframes and keyframes have a time attribute, it does not matter what order the keyframes are generated in.

Due to the architecture of CM, one can freely create sound events and animation events within the same algorithm. The extension of the output file you request determines which type of events get written to the file. To generate a file of MaxScript commands, one requests an output filename ending with the extension .ms.

4. Implementing Note-to-Object Correspondence

In most computer music synthesis approaches, a “score” provides instructions regarding when note events should be initiated, what their characteristics will be, and how long they will last. As the score reader moves forward in time, it initiates and ends events. Notes are conceived of as objects that are instantiated and discarded in time.

This paradigm does not translate directly into a computer animation environment, where object creation, modification, and animation are quite separate processes. That is, one first creates a primitive object. Then one modifies that object. In this fashion one creates all objects needed for a scene. Then one applies animation instructions to create time-based modifications to those objects. All objects in a scene exist all of the time.

As a result, AECM usage tends to take one of two forms. In the first case, one generates a script that creates all objects and modifications needed for a scene and then applies animation. In the second case, one creates a scene manually and uses AECM only to provide animation of the scene's objects.

In the former case, one can run into difficulties if one is trying to follow the obvious approach of creating one-for-one note-to-object correspondences. When one tries to create one object for every note, the execution of the resulting MaxScript will create all of the nodes. All "notes" will exist all of the time in the resulting scene. So one must use animation parameters to make an object invisible until the period in time where the note occurs, then make it disappear (perhaps) after an appropriate duration.

In this case it is very easy to overload the animation environment due to the sheer number of objects you might instantiate. To facilitate this kind of work, AECM provides the `get-available-object` function to track which objects have been created and which are "in use" at any particular point in time, allowing inactive objects to be modified and reallocated to appear with new musical events. In creating the animations for my own work *Writing on the Surface*, I was able to reduce the node count in some animations from the thousands (which brought rendering to a near standstill) to between 20 and 30.

5. Note-to-Object Versus Correspondence of Complexes

Obviously AECM can facilitate strong note-to-object correspondence, or even note-to-object counterpoint. But I find that, as a composer, encountering the computer animation paradigm in which all objects exist in the scene and are transformed in time encourages another kind of thinking. This thinking is more in keeping with the tendency of electroacoustic music to emphasize texture, trajectory, and gesture over discrete events and regular pitch and time lattices [1]. The separation of object creation from distribution in time can encourage a less atomized conceptualization of sound-event and visual-object relationship. Instead, one may be led to focus on the ways in which the trajectory of a whole audio gesture can have a rich correspondence to changes in a whole visual complex. In this case, one is more likely to create complex objects in the animation scene—objects capable of exhibiting a rich variety of behaviors—and trigger or shape those behaviors in ways that have strong morphological relationships to sonic events.

6. Examples from *Writing on the Surface*

I created most of the computer animation elements in my work *Writing on the Surface* for computer-realized sound and image with the assistance of AECM.

Most of the computer animation material occurs in the dense, percussive section of the work. The primary approach here was direct note-to-object correspondences. However, in one aspect of the visual composition I took something closer to the "visual complex" approach. In the mid-background of the visual composition, an angled ellipse is roughly circumscribed by the trajectories of many rapidly moving, obliquely lit spheres. This texture is derived from the same algorithms generating the high-pitched, dense percussive gestures in the audio tracks. The density of the spheres and the rapidity and range of throw in their motion have a correspondence to the density of audio activity. In this way I began to move away from note-object correspondence and towards parallel expressive gestures. (See Figure 1.)

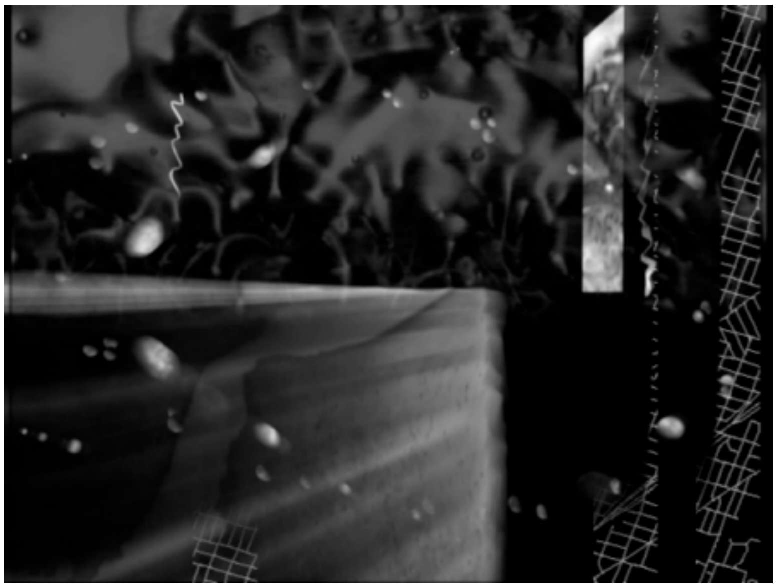

Figure 1. Many note-to-object correspondences here, but a textural-density correspondence between sound and image occurs in the ellipse trajectory of spheres.

A central audio and visual “image” in the work is a simple recurring ticking figure. In this case, I created an animation scene with a set of six cones with their tips meeting, lit obliquely from the side with a virtual spotlight containing a Quicktime movie of vibrating water. The cones rotate around their meeting point, seemingly impulsed forward by the clicks and gaining speed as the ticks gain speed and intensity. The behavior of the cone-complex is clearly triggered by the audio events, but seems to take on its own momentum as a result of that triggering. (See Figure 2.)

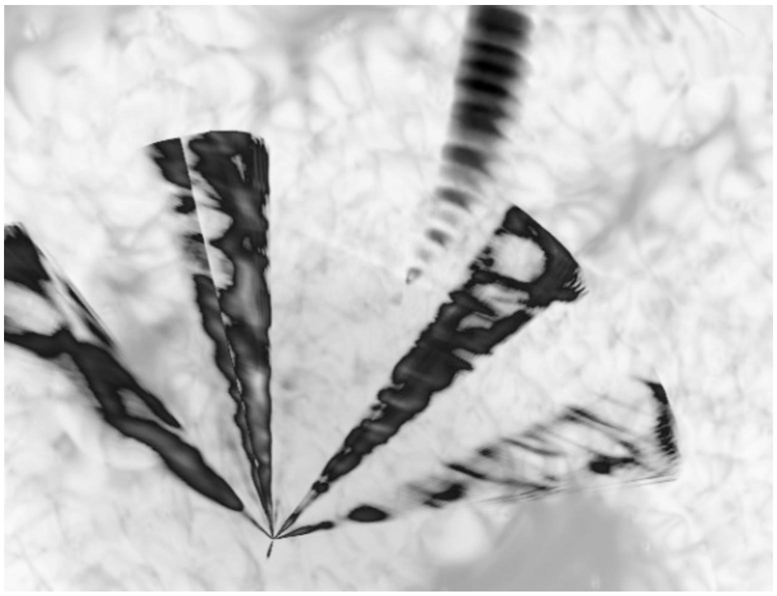

Figure 2. A twirling complex of obliquely-lit cones.

Both of these examples are initial forays into what should prove to be a rich arena for ongoing experimentation.

7. Future Steps

The highest priorities for AECM are converting it to work with Common Music version 2.0. Also under consideration is the possibility of moving away from generating MaxScript towards creating output that could be used with freeware or shareware animation and rendering tools.

8. Getting AECM

A beta version AECM is now available at my web site: http://www.BatHatMedia.com. At this time it has only been tested with Common Music 1.4 running under Allegro Common Lisp 5 on Red Hat Linux. I do not intend to provide cross-platform testing or versioning of AECM, but I am highly likely to move my Common Music and animation work fully to the Macintosh platform in the near future.

9. Acknowledgements

The creation of AECM would not have been possible without the resources of the Center for Advanced Research Technologies in the Arts and Humanities at the University of Washington.

10. References

[1] Wishart, Trevor. *On Sonic Art*. Harwood Academic Publishers, 1996.
[URL1] http://sourceforge.net/projects/commonmusic/
[URL2] http://www.discreet.com/

Revised February 26, 2001