SCRAMO: A Model for MIDI-Choreographed 3D Animation

Jui-Chieh Richard Hsu, Jui-Yuan Fred Hsu, Hung-Wen Chen
Department of Computer Science
Cornell University
juiyuan@cs.cornell.edu

1. Introduction

Digital music and computer animation have long been studied in the research community. MIDI-Choreographed 3D Animation explores both fields by examining the problem of mapping musical events to graphical animations. A MIDI Animation is a process in which objects in a computer-simulated world are animated under the direction of some digital music. For a particular animation sequence, the animator would create a 3D world, specify relationship between musical events and 3D world, then input arbitrary MIDI music to drive the animation. Our work is inspired by Wayne Lytle's animation "More Bells and Whistles" and paper [6], in which musical notes are mapped to animated instruments.

Our goal is to formulate an abstract Model for MIDI-Choreographed 3D Animation, which shall encompass the whole production sequence, from initial 3D object modeling to final audio-video output. The model shall be simple, yet it shall be powerful enough to allow for nearly all MIDI processings and 3D world animations.

The model shall identify animator-specific components of the animation process from generic data manipulations. The later can be handled automatically by library routines, alleviating much of the burden from animator. Our ultimate goal is to show that in deed the model can reduce all animation programming down to mouse clicking and simple script writing.

The rest of this paper is organized as follows. Section 2 gives an overview on MIDI and 3D animation. Section 3 describes ad hoc approaches to the MIDI animation problem, and points out difficulties with them. In section 4, we introduce our SCRAMO Model for MIDI animation and present an overall structure of the model. Individual components of the SCRAMO Model are described with much more details in section 5. Issues concerning model implementation are discussed in section 6. Finally, section 7 concludes this paper with future directions.

2. Terminology

MIDI (Musical Instrument Digital Interface). MIDI is a communication protocol specifying how musical events are transmitted from one musical instrument to another through electrical cable[1,2,3]. A **MIDI Message** consists of a status byte denoting a unique musical event, followed by optional data bytes (fig 2.1). A sequence of MIDI Messages becomes a **MIDI Stream**, and effectively describes a piece of music (fig 2.2).

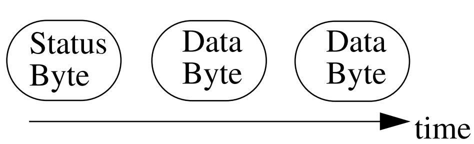

Fig 2.1 MIDI Message

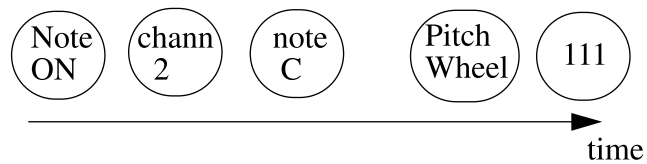

Fig 2.2 MIDI Stream

3D Scene. A Three-dimensional scene comprises of **3D Objects** arranged in a hierarchical fashion[4,5]. Each 3D object is the root node for the subtree beneath it. A child node's location and orientation in space are described by **Local Transformation Matrix T**, in terms of parent location and orientation (fig 2.3).

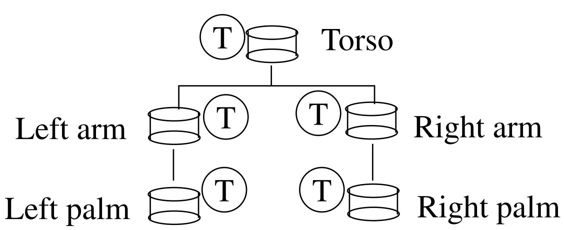

Fig 2.3 3D Scene of Objects

3D Animation. By modifying transformation matrices, a programmer can change location, orientation and shape of objects over time. Object ownership can be altered by changing hierarchical arrangement of nodes; the programmer can then describe node transformations with respect to new parents. It is also possible to modify object properties, such as surface reflection index, texture mapping, etc. A single instance of changing the scene is called a **Modification** (**Mod**). An **Action** is a sequential set of modifications.

Animator. A **programmer** who is responsible for writing the animation program.

3. General Approaches to MIDI-Choreographed Animation

To create a MIDI-choreographed animation, the animator first generates a MIDI stream using music software, and constructs a 3D world using some graphical library. 3D objects are saved in such a way that they are accessible to programs. The animator then writes a program to examine MIDI events from the MIDI stream, and to perform appropriate actions on the 3D scene. Finally, animation frames are synchronized with MIDI events; they are sent to audio and video devices respectively (fig 3.1). Thus the task of animation program is mainly concerned with event-to-action mapping and audio-video synchronization.

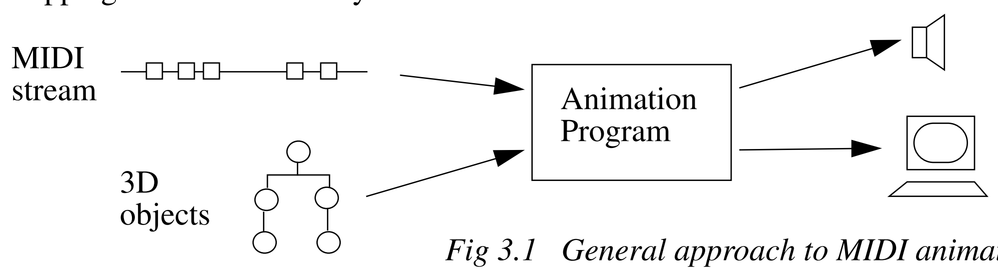

Fig 3.1 General approach to MIDI animation

A very simple mapping between musical events and animation actions can be specified for drumstick movement. We first search MIDI stream for a note-ON event, and modify transformation matrix T for the drumstick over a time interval (fig 3.2).

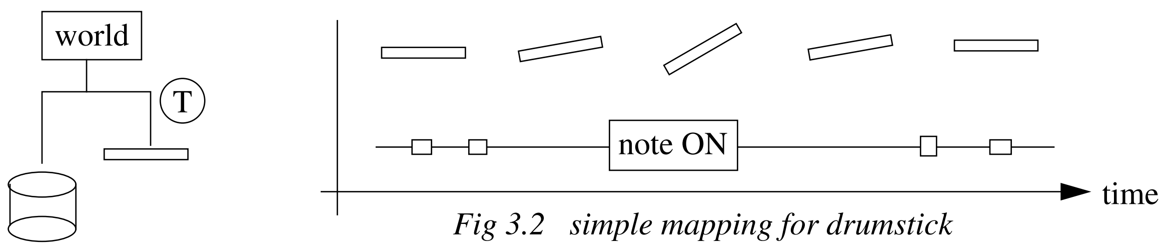

Fig 3.2 simple mapping for drumstick

Unfortunately, few animations are as simple as drumstick movement. For instance, when animating a cello player, we would like to see vibrato effects (fig 3.3). To perform vibrato, the finger no longer moves in terms of palm; it now moves in terms of fingerboard. A hierarchy change of objects is required (reverse kinematics). Moreover, we do not want to perform vibrato for every instance of note-ON; the effect is needed only if a note lasts longer than a quarter note. These impose a great deal of work on behalf of programmer, as the animation program must search for all applicable musical conditions, reconfigure the object tree, and recompute transformations.

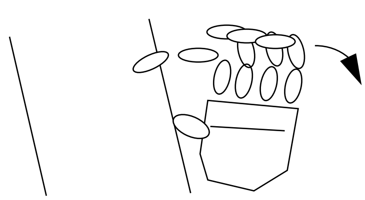

Fig 3.3 Index Finger Performing Vibrato

4. SCRAMO - A Formal Model for MIDI-Choreographed Animation

In attempt to abstract the process of MIDI animation, we propose a formal model which encompasses the whole production sequence, from initial 3D object modeling to final audio-video output. With the abstraction of our model, we are able to isolate animator-specific components of the animation process from generic data manipulations. The later can be handled automatically by library routines, alleviating much burden from animators. We will show in coming sections that in deed our model can reduce all animation programming down to mouse clicking and simple script writing.

4.1 Basic SCRAMO Model

Basic SCRAMO is a simplified model. It identifies essential data components of MIDI animation (fig 4.1.1). The whole animation is built on top of raw data: **MIDI Stream S** and **3D Objects O**. Animator specifies a set of **Musical Conditions C**, such as "when tempo doubles", "when note C is longer than a quarter note", "When we see a sequence of octave notes", etc. For objects, a set of **Modifications (Mods) M** is specified. Examples of Modifications are: "cut node X from its parent", "reparent node X to Y", "rotate node X by 20 degrees in terms of Y", etc. A set of **Actions A** is constructed from M, each action being a sequence of modifications. Ultimately, the animator supplies a **Mapping Relation R** which maps conditions to actions. Hence the name SCRAMO is coined.

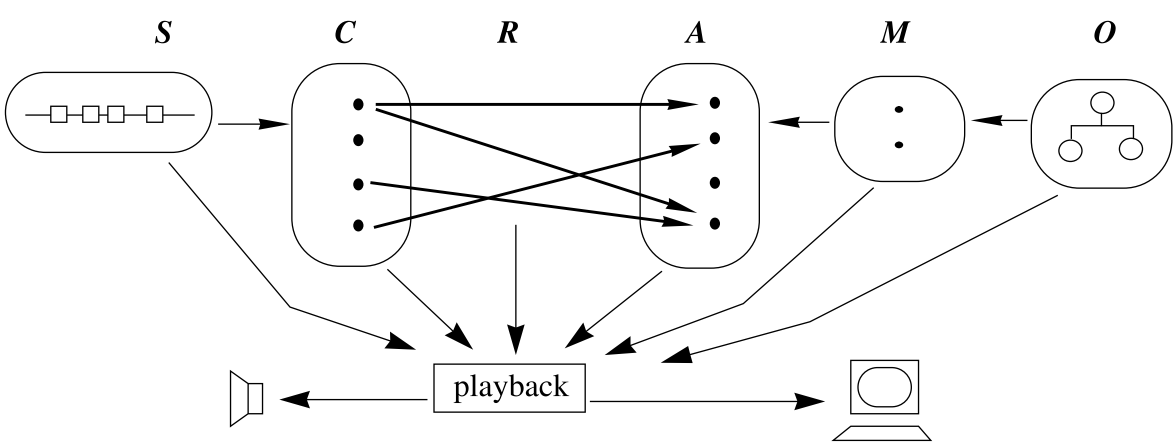

Fig 4.1.1 Basic SCRAMO Model

4.2 Full SCRAMO Model

In Basic SCRAMO model, conditions, mods and actions are defined loosely. There are virtually infinite number of conditions, mods and actions. Full SCRAMO Model defines a concise yet powerful set of **Mod Templates**, from which user modifications can be instantiated (fig 4.2.1). Actions reference mod instances to describe an atomic movement in the scene. For MIDI processing, a database of **Event Lists** is generated automatically from MIDI stream. A **Library of Operators** is provided to manipulate this database. Any condition can be specified by an expression of events and operators. Animator-interactions are restricted to top row.

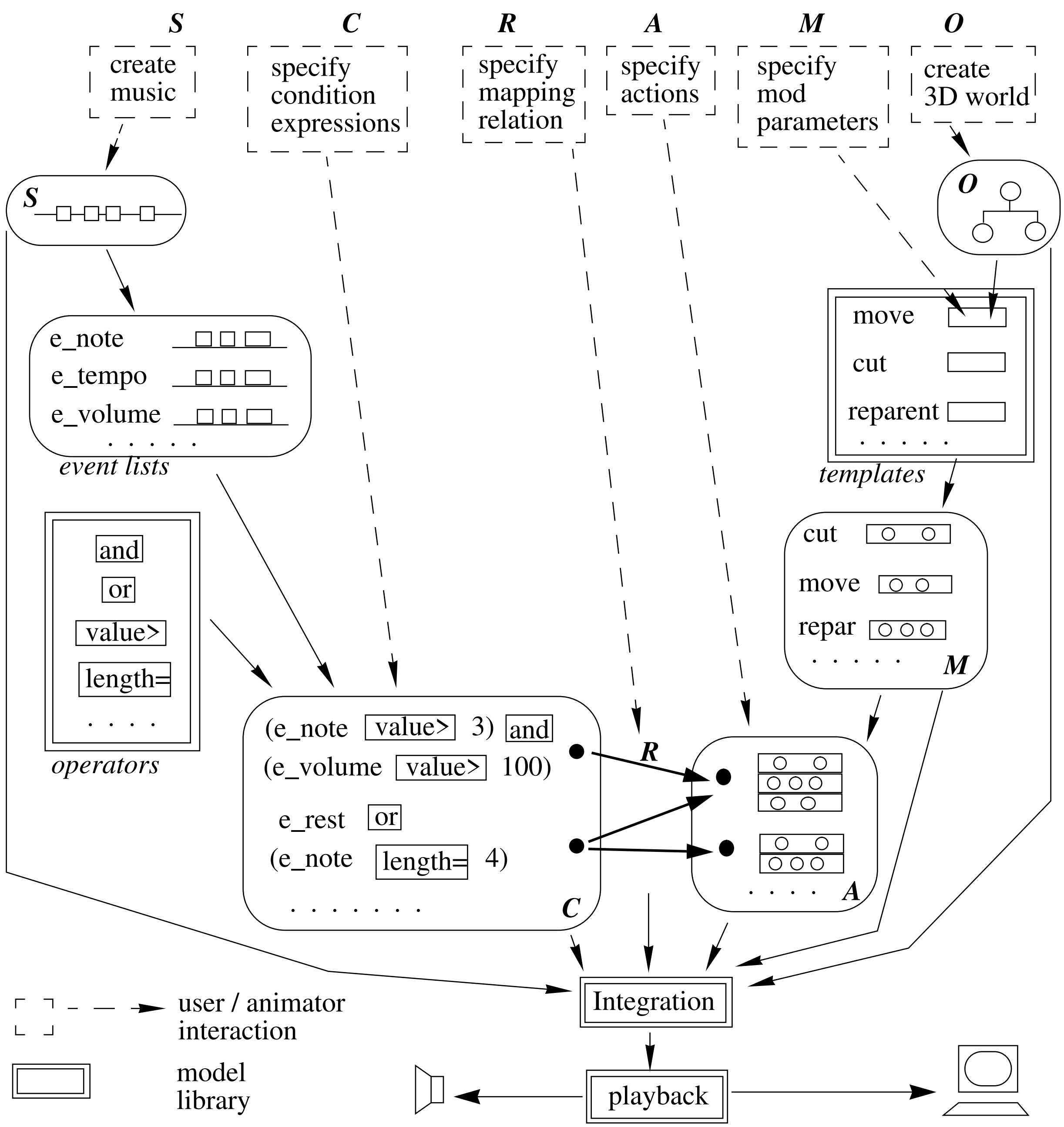

Fig 4.2.1 Full SCRAMO Model

5. Components of the SCRAMO Model

5.1 Modification Templates

Animation of a 3D scene can be done by modifying node transformations and properties. Since complicated actions usually involves burdensome and repetitious changes to the objects (section 3), we have classified modifications into a concise set of categories. Each category is represented by a **Modification Template (Mod Template)** that encapsulates data requirement and modification behavior. Different real instances of a template can be created by giving the template different data parameters (fig 5.1.1). Note that template-instance relationship can be implemented as class-object with Object-Oriented languages.

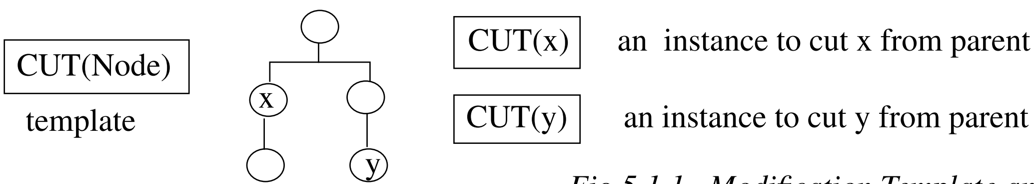

Fig 5.1.1 Modification Template and Instances

To "apply" a mod instance on the scene, we simply call a function setprogress(num,den), which will make all necessary changes on 3D objects so that the goal of the mod instance is satisfied. Parameters num and den denote the progress of desired change; a ratio num/den of 1 demands a fully accomplished modification, while a ratio of 1/n describes a partial modification from current scene to target scene. Note that most mod templates impose no requirement on the configuration of current scene; they only specify a target configuration. Thus mod instances can be executed at any time from almost any scene configuration. A set of Mod Templates for SCRAMO model is provided in fig 5.1.2. The set can be extended by adding more templates.

Ownership Arrangement

- CUT-from-parent(node)
- REPARENT(node,new-parent)
- FIX-on(node,target-node)

Constraints

- CONNECT(node1, joint1, node2, joint2)

Movement

- SET(node, final-transformation, in-terms-of-node2)
- ADD(node, incremental-transformation)

Fig 5.1.2 Set of Mod Templates

5.2 Action - A Sequence of Modifications

An Action is a set of Modifications arranged in order. Each modification is associated with a number indicating the desired number of **frames** the mod takes (fig 5.2.1). Mods in the Action are executed one by one sequentially. In preparation for future mapping between Conditions and Actions, we specify an **anchor** point for the action. Notice again that the action described in figure 5.2.1 can be executed at any scene configuration; the mods will figure out correct changes for the scene.

Action “Punch”

SET(fingers, fist-configuration-transf, in-terms-of-palm)	2 frames
ADD(palm, pull-back-transf)	2 frames
SET(palm, enemy-head-location-transf, in-terms-of-world)	1 frame
SET(palm, pull-back-transf, in-terms-of-body)	4 frames

anchor point

Fig 5.2.1 Example of Action illustrating a punch

5.3 MIDI Event Lists

The statement “when any note is on, perform punch-action” is really another way to state: “for the **subset E** of MIDI stream S, perform punch-action” (fig 5.3.1). We call any subset of MIDI stream S an **Event List**. In order to retrieve event list E from stream S, we examine the properties of MIDI messages and extract those that meet our criterion. Notice that once the event list is obtained, we no longer need musical properties for individual events; we simply apply punch-action to every event in the list.

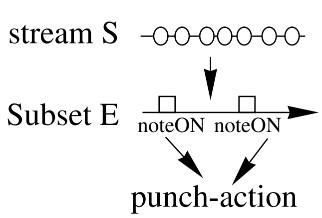

Fig 5.3.1 Subset E

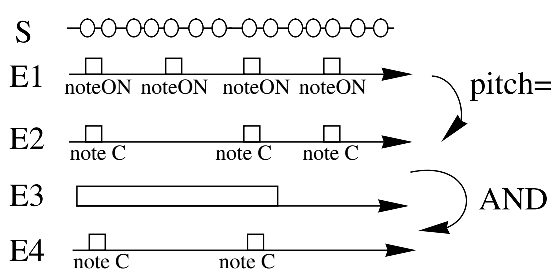

Fig 5.3.2 Event List Operations

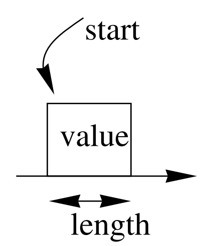

Fig 5.3.3 An Event

Retrieval of event list for complicated criteria is difficult. However, we observe that complex event lists can be constructed by manipulating simple event lists (fig 5.3.2). To create a list for “when note C is on during the first half of music,” we start with a event list E1=“when any note is on.” Provided that each event contains a number denoting its pitch, we can filter out undesirable events based on pitch comparison. Then we perform boolean AND between the list E2 and some time-event list E3, acquiring final list E4.

It is clear that we must generate a formatted database of simple event lists from the chaotic MIDI stream. We will define list operators in section 5.4 to manipulate simple lists. To allow for operator processing, events need be specified in a consistent way. Figure 5.3.3 illustrates the specifications of an event. The SCRAMO model generates 8 event lists: E-note, E-rest, E-volume, E-tempo, E-key, E-signature, E-channel, and E-vibrato. Each list interprets event values differently. For E-note, value=pitch.

5.4 Musical Conditions - Event List Operators

A **Musical Condition** is an expression of **Event Lists** and Event List **Operators**. Operators receive Event Lists and/or numbers, and always produce another Event List as the return value. A summary of operators are described in figure 5.3.4. The set can be extended by adding user-defined operators.

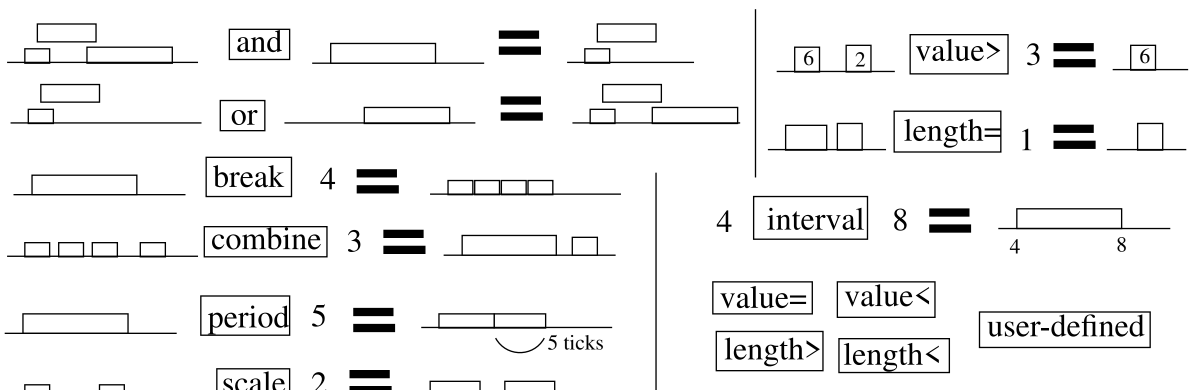

Fig 5.3.4 Event List Operators

5.5 Mapping Relation

Once condition expressions are evaluated, we can map resulting event lists to actions. **Mapping Relation R** takes a set of event lists as input domain, and map each list to a set of animation actions. Each event in an event list is mapped to the same set of actions.

The above can be done using software-equivalent of combinatorial circuit, but many times we really need a sequential circuit with memories (a state machine). Suppose a mapping link between "all notes C in the fifth octave" to some fingering position is desired for a cello player. Fifth-octave C can be performed on different strings with different finger positions, depending on the current configuration of bow and fingers. Therefore, each mapping link is implemented as a function, which retains its "state" between calls.

5.6 Integration of S.C.R.A.M.O.

In order to produce real animation, the Integration module takes in S.C.R.A.M.O. as raw data, and produces two repository arrays (fig 5.6.1). The Audio Repository stores MIDI messages for each time frame, while Animation Repository records modification instances for the 3D world. The two arrays are synchronized. Audio Repository is actually a copy of MIDI stream segmented into time frames. Animation Repository is filled by evaluating each condition expression, following the mapping relation to applicable actions, and transferring mod instances from actions to the array cells. After arrays are prepared, the Integration module starts a playback loop to send out MIDI messages promptly. Video rendering can be delayed or dropped if necessary, but animation mods must always be applied to 3D objects to preserve the integrity of 3D world.

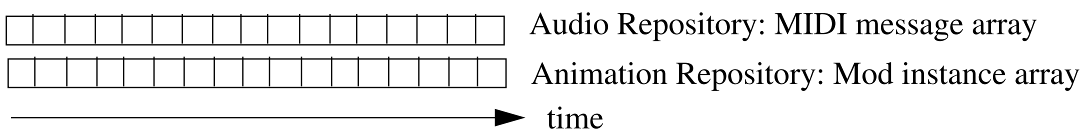

Fig 5.6.1 Audio Repository and Animation Repository

6. Implementation

6.1 Model Implementation

A full SCRAMO model implementation must provide **programming interfaces** to create MIDI Streams, 3D Objects, Condition expressions, Modification instances, Action set, and Mapping Relation. Model libraries must generate simple Event Lists from MIDI stream, provide Event List Operators, define Mod Templates, and implement Integration and Playback modules. An animator can then write simple programs to interact with the model (animator-interactions are blocks on the first row of figure 4.2.1).

Nevertheless, SCRAMO only outlines a model for MIDI-choreographed animation. It does not impose any restriction on the actual implementation. MIDI Streams can be read from file, fed from keyboard, or even created by program. 3D Objects can be stored and manipulated in a functional-oriented way as handled by PHIGS, or managed as encapsulated data and function set by Object-Oriented SGI Open Inventor. Operators and Condition statements need not necessarily be implemented to look like real expressions and to return an event list; they can be coded as query expressions into event database. Mod Templates are best accomplished using Object-Oriented Languages, but even Pascal allows for record creation for Mod instances.

6.2 Application Implementation

An application implementation of SCRAMO shields animators from the details of programming completely. On top of SCRAMO Model, the software should provide **user interface** for animator to specify all parameters for S.C.R.A.M.O. components. 3D Objects, Modifications, and Actions are specified by mouse movement and button clicking. MIDI Stream is fed by keyboard or specified by drawing music score on screen. Condition expressions can be described by either expression scripts or by mouse-editing. Mapping Relation should be specified graphically, but the state machine for each condition need be written as script, as variables are needed to keep record of current state.

6.3 MIDIAnimator™

MIDIAnimator is our application implementation of SCRAMO. The program is written in C++. Although we did not have enough time to implement Condition and Mapping script writing, animators can still write short C/C++ program to interact with MIDIAnimator. We use Open Inventor library as building blocks for 3D object handling. We have experimented with various public-domain MIDI parsing libraries, but ended up writing our own. Actions are specified by mouse movements on screen, and can be saved in files. MIDIAnimator finally synchronizes actions with MIDI messages. It renders 3D scene on SGI Indigo workstation and sends MIDI output to Roland 320 synthesizer.

6.4 MIDIAnimator™ at Work

We present a snapshot of an animation scene produced by MIDIAnimator in figure 6.4.1. For this simple animation, notes in musical scale are mapped to different drumsticks, while a canon in the center of scene acting as a metronome swings back and forth. A summary of animator-interactions is listed below:

1. *Create drum set and canon using MIDIAnimator object editing interface*
2. *Create Mod instances for drumsticks and canon using Mod Template SET()*
3. *Action A1 for drumstick = mod-hit-drum-surface + mod-pull-back*
4. *Action A2 for canon = mod-swing-back + mod-swing-forth*
5. *Condition C1 for drumstick = (E-note* $\boxed{\textit{value}}$*= 48)*
6. *Condition C2 for canon = (0* $\boxed{\textit{interval}}$ *end-of-time)* $\boxed{\textit{period}}$ *ticks-for-quater-note*
7. *Map C1 to A1, C2 to A2*

Fig 6.4.1 A Simple Animation with Drum Set and a Canon

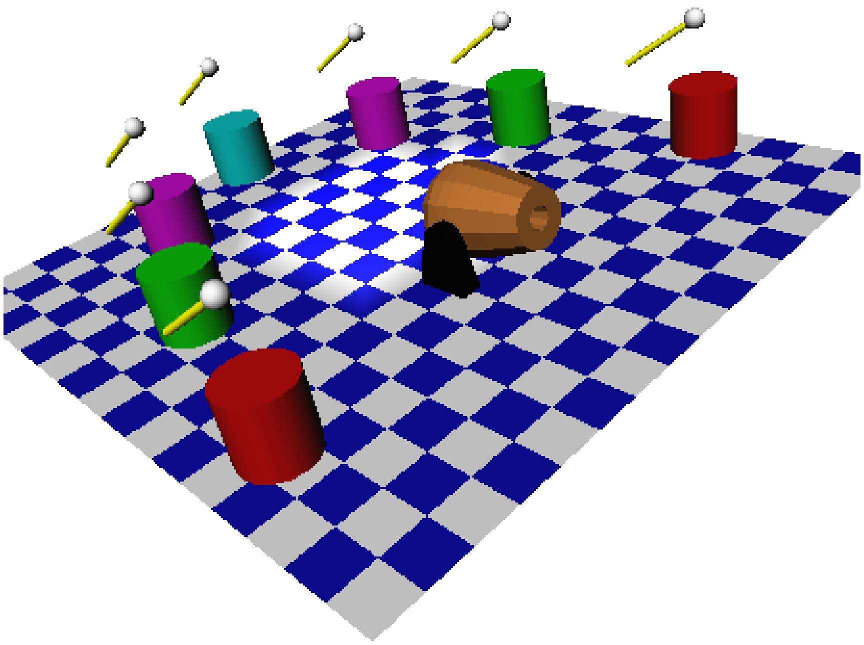

Fig 6.4.1 MIDIAnimator

7. Conclusion and Future Work

The SCRAMO model for MIDI-choreographed animation is described in this paper. It identifies essential processes and data components throughout the animation production. The model can be implemented as blueprint for stand-alone animation programs, as library resources for animation programmers, or as interface for general users. An application is developed according to the model, which provides user interface for most animator-interactions. With this application, we were able to build MIDI-choreographed animations as specified by our goal.

Future work will be directed toward the following areas:

- Enhance Mod Template set to include object “property” handling.
- Consider extending constraint modifications to include physically based animations
- Specify Condition expressions using script language (MIDIAnimator)
- Specify Mapping State Machine using script language, with support for variables

References

[1] Kurt. Gjerde, “PRIMER: A MIDI tutorial,” http://www.uib.no/People/midi_doc_remote.html, 1994.

[2] Eric Lipscomb, “Introduction To MIDI,” http://www.eeb.ele.tue.nl/midi/intro.html, 1994.

[3] Eric Lipscomb, “Summary of MIDI Status & Data Bytes,” http://www.eeb.ele.tue.nl/midi/midistat.html, 1994.

[4] Bruce Mielke, “Integrated Computer Graphics,” West Publishing Company, St. Paul, 1991 281-307.

[5] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, “Computer Graphics: Principles and Practice,” Addison-Wesley Publishing Company, Inc., New York, 1992 229-346.

[6] Wayne T. Lytle, “Driving Computer Graphics Animation From a Musical Score,”