Active Media

Tsichritzis, Dionysios; Gibbs, Simon; Dami, Laurent

Active Media

Dennis Tsichritzis, Simon Gibbs and Laurent Dami

Abstract

In this paper we explore an environment for "active media." The environment consists of a lower-level object-oriented framework intended for multimedia programmers and a higher-level facility intended for multimedia designers. We claim that such an environment will be both flexible and powerful for constructing complex multimedia applications. We first define multimedia objects and then explore composition techniques for these objects. Finally, we outline a facility for "scripting," that is, specifying the cooperation of such objects.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—user interfaces; H.1.2 [**Models and Principles**]: User/Machine Systems—human information processing; I.3.7 [**Computer Graphics**]: Miscellaneous

General Terms: Design, Human Factors

Additional Key Words and Phrases: multimedia, scripting, active objects, video/graphics integration

1. Introduction

Current technology enables the encoding, storage and retrieval of multimedia (sound, images, video) in a digital form which can be operated on. The operations on these "multimedia objects" are, however, considered independently. We usually treat these objects as data which are processed by functions programmed beforehand. Interaction, if present, is provided in an *ad hoc* fashion through some user interface. In this approach the media objects are passive. That is, they do not incorporate any threads of control or trigger their operations. In this paper we will incorporate dynamic behaviour in multimedia objects themselves. We will associate processes directly with these objects making them in a sense active objects. Each object interprets independently its own multimedia data and cooperates with the other objects by exchanging messages. This latter aspect provides some sort of "orchestration" so the total effect is harmonious. We refer to such object ensembles as *active media*; it is active in the sense that it produced by active objects and may respond to user interaction or changes in the environment.

There are many reasons for generalizing multimedia to what we call active media. First, the equipment used by the traditional "composers" of multimedia (video professionals, music editors, etc.) is relying more and more on digital technology and consequently becoming more and more programmable. For instance, sophisticated DVE ("digital video effect") units which support solid modelling and real-time texture mapping [10] surpass some of the functionality found in high-end graphics workstations. Until fairly recently, multimedia equipment could be viewed as interconnectable hardware "boxes" (recorders, mixers, monitors, etc.). Now, however, with software controllable components playing a greater role in both producing and transforming multimedia data, such a model breaks down. One result is that people like video artists may have to delegate some of their work to programmers when they would rather not. However, whether or not we will continue to see this separation of roles, we must recognize that software-related concepts, such as process and operation, are an integral part of multimedia and a general environment for multimedia must allow for the incorporation of software-based components with the traditional hardware-based components.

A second reason for considering active media is that complex user interfaces, such as virtual realities [12] can be viewed as multimedia applications. If a multimedia environment is to support such interfaces it must be flexible enough to handle novel interface devices (e.g., stereoscopic displays, head-position and orientation trackers). Furthermore, virtual realities can be significantly enhanced if they portray dynamic rather than static worlds. As we will show, such flexibility and dynamics can be provided by active media.

Third, we hope that active media, and the concepts we introduce, will provide a general model for describing and developing a wide range of multimedia applications. At present there seems to be a tendency to develop multimedia applications in an *ad hoc* one-of-a-kind fashion, so, we believe, a unifying conceptual model is needed.

Finally, experience shows that live media presentations, e.g., live television, theatrical performances, concerts and opera, carry an extra sense of excitement. Group participation and spectacular presentation is of course a major advantage of live performances. Both of these, however, can be simulated, to some extent, by sophisticated presentation techniques (large HDTV screens, stereo sound, theatre sitting). One aspect of live performances which is still missing from multimedia presentations is the excitement of unexpected individual behaviour of the actors participating in a live performance. This type of behaviour can not be expected if we treat multimedia objects as passive objects to be operated on; it can only be produced if we incorporate dynamic behaviour.

The environment we explore in this paper is based on an object-oriented framework. There are a number of advantages derived from object orientation. First, objects will be used to produce, consume, and process several different multimedia values, e.g., audio, video, music, animation. In this way hardware-oriented details of such values can be encapsulated and the difference between hardware-based and software-based components concealed. Second, object-oriented programming techniques, such as specialization, can be used to extend the framework; for example, to add a new form of multimedia data or a new device. In addition, organising the compositions and connections of such objects is a form of programming within an object-oriented language. Such programs provide unambiguous, executable, representations of complex multimedia objects. Finally, composite multimedia objects involving many independent multimedia objects can be specified by using high-level description techniques. In this way the specification of such objects by non-programmers becomes feasible.

In the second section of this paper we will discuss multimedia objects and their operations. In the third section we will discuss composition of such objects, exploring in particular their temporal relationships. In the fourth section we will discuss "scripting" as a way of specifying the cooperation of many independent multimedia objects.

2. Multimedia and Active Objects

The object-oriented framework has been described elsewhere [6], so here we provide just a summary. Our starting point is the use of data types to characterize multimedia information. For instance, a multimedia data type CDaudio could consist of all possible audio signals as encoded on a compact disc, i.e., CDaudio values would be sequences of the form sample_i where the bit representation used by the samples would be specified by the CD recording format. As with other data types, one could then define operations for CDaudio, for example operations dealing with compression, filtering, or concatenation.

This suggests the following definition for multimedia data values:

Definition: A *multimedia value*, v, of data type D, is a (finite) sequence d_i, where the encoding and interpretation of the d_i are governed by D. In particular D determines how the *presentation* of v (the physical realization of v, within some medium, over some time interval) can be obtained from the d_i. Presentation of v takes place at a rate r_D, the *data rate* of D. This rate indicates the number of sequence values presented per second.

The data rate of course varies from data type to data type, hence the subscript in r_D. For example, CD audio requires some 44 thousand samples per second, while smooth animation needs a minimum of about 8 frames per second. In some cases the rate may be merely an ideal, and presentation may take place at a different or varying rate, perhaps with loss of quality. Also there may be data types where the d_i contain explicit timing information. In such cases, although the interval from one d_i to the next will vary, there is likely to be a maximum data rate which can then be assigned to r_D.

The above definition associates each multimedia value with a lower-level sequence, there are, however, cases where the sequence may not be known *a priori* (i.e., before presentation) or may not be directly accessible by software:

- the multimedia value may exist only as an analog signal. For example, there are many computer-controllable video products, including VCRs, LaserDisc players, and cameras, with analog video inputs or outputs.
- presentation may involve synthesis. For example, a pure audio tone might be generated from a sine function rather than from a stored sequence of audio samples.
- we may want to associate a multimedia value with an on-going activity (for instance, a recording session). Here the sequence is not known until the activity has completed.

The first case can be accounted for by allowing analog signals as multimedia data types, with the understanding that the underlying d_i are unavailable within a program. Values of these types are software handles; they are used in controlling hardware devices which in turn manipulate the analog signal.

The second and third cases involve processes (the sine generator and the recording session) that produce multimedia values. These processes are likely to have their own internal state (e.g., the frequency of the sine generator) and their own operations (e.g., change the frequency, increase the recording volume). To allow programmers to conveniently handle such cases we need a construct which combines the notion of data, behavior *and* process.

An appropriate programming language construct appears to be the *active object* [4][13]. Active objects, like ordinary or *passive* objects, have state (instance variables) and behavior (methods). In addition, each active object is associated with a process which may be running even if no messages have been sent to the object. (There are many ways in which concurrency can be added to an object-oriented programming language in order to support active objects [9], here we assume a simple model where active objects are multi-threaded and synchronization is the responsibility of the programmer.)

Using active objects, we can then define multimedia objects as follows:

Definition: A *multimedia object* is an active object which produces and/or consumes multimedia values (of specified types) at their associated data rates.

There are a number of observations we can make from this definition. First, multimedia objects are subject to real-time constraints in that they have to produce or consume data at particular rates. Second, each multimedia object can be viewed as a collection of *ports*. A port has a (multimedia) data type and is used either for input or output. A multimedia object's ports (the number, their data type and direction) would be determined by the class of the object. Third, we can divide multimedia objects into three categories: *sources*, *sinks*, and *filters*. A source produces multimedia values, a sink consumes values, and a filter both produces and consumes.

It will be convenient to use a graphical notation for multimedia objects. We will represent such objects by circles (see Figure 1) with attached boxes for their ports (extruding for output ports and intruding for input ports).

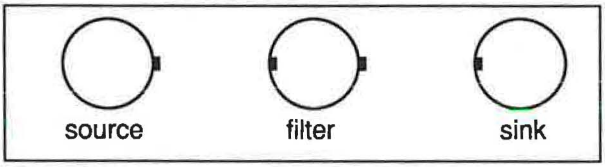


Figure 1 Multimedia Objects.

Multimedia objects belonging to classes such as those listed above are examples of what we will call *multimedia primitives*. In general a multimedia primitive is a multimedia object which cannot be decomposed into other multimedia objects. Is this section we look at the operations supported by these primitives. In the next section we will look at *composite multimedia*, i.e., objects where such a decomposition is possible.

2.1 Multimedia Hierarchies

Multimedia primitives make use of two hierarchies – a supertype/subtype hierarchy of data types and a class inheritance hierarchy. Figure 2 shows some possible multimedia data types. This figure is not intended to be complete but merely to give some indication of the richness of the hierarchy. The division into five high-level subtypes: Audio, Video, Image, Scene and Music is also somewhat arbitrary and incomplete. Other data types could be added if needed. Some possibilities are static text, graphics or images (all sequences of length 1); moving text (e.g., as in a film's credits); or data from very specialized acquisition devices (e.g., volume data produced by a CAT scanner). However, the five basic types give a good range and are common enough to be supported on many platforms.

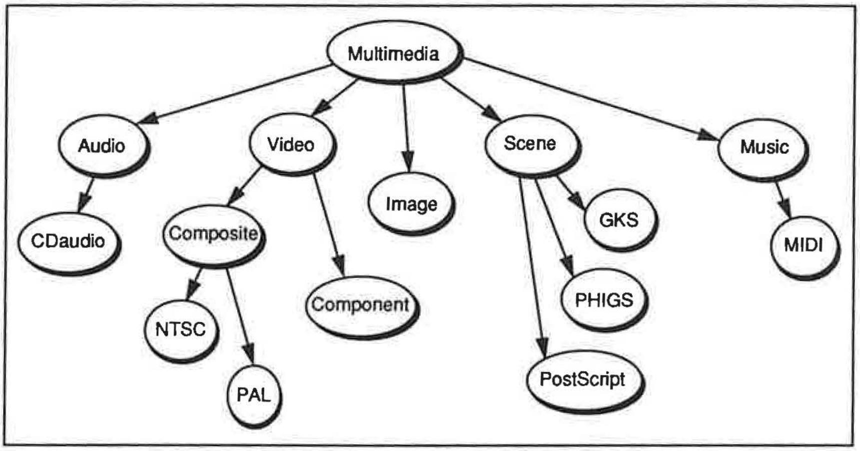


Figure 2 Some Multimedia Data Types.

Similarly the class hierarchy in Figure 3 is also incomplete. Since many of these classes corre-

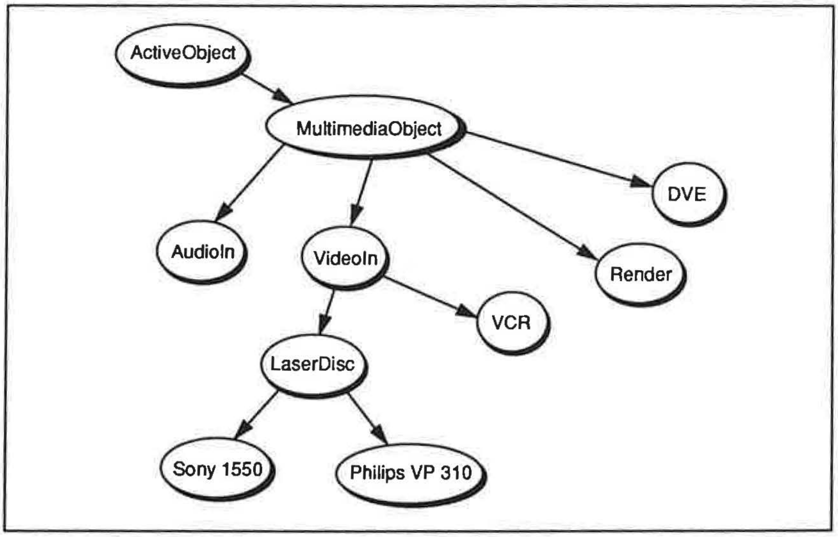


Figure 3 Some Multimedia Object Classes.

spond to hardware devices there may be different specializations for different models and manufacturers (illustrated here by the two subclasses of LaserDisc). Another source of variety is the range of filter objects. For example, there are many subclasses of DVE (corresponding to wipes, fades, etc.) and Render (corresponding to different rendering systems).

2.2 Operations on Multimedia Objects

All classes of multimedia objects inherit methods from ActiveObject and MultimediaObject. The methods for ActiveObject are:

Start()
Pause()
Resume()
Stop()

At any time an ActiveObject is in one of three states: IDLE, RUNNING or SUSPENDED. The sbove methods are used to change state.

Specializations of ActiveObject may define other methods. Instances of these classes will respond to messages regardless of whether they are RUNNING, IDLE or SUSPENDED – unless the programmer has introduced explicit state dependant code (for example, blocking if the state is RUNNING).

The class MultimediaObject introduces a more interesting set of methods. These methods make use of two temporal coordinate systems: *world time* and *object time*. The origin and units of world time are set by the application. The origin would normally be set to coincide with the beginning of multimedia activity. World time would run while the activity is in progress, and be stopped or resumed as the activity is stopped or resumed.

Object time is relative to a multimedia object. In particular, each object can specify the origin of object time with respect to world time and the units used for measuring object time. (Normally these units relate to the data rates of the object's ports.) Furthermore, each object can specify the *orientation* of object time, i.e., whether it flows forward (increases as world time increases) or backwards (decreases as world time increases).

The methods of MultimediaObject include:

WorldToObject(worldTime) – transform from world time to object time.
ObjectToWorld(objectTime) – transform from object time to world time.
Cue(worldTime) – position the object at a particular world time.
Sync(worldTime) – used for synchronization (see [6], also the eighth paper in this volume).
SyncInterval() – also used for synchronization.
Translate(worldTime) – translate the object in world time.
Scale(factor) – scale the duration of the object.
Invert() – flip the orientation of object time between "forward" and "reverse."

To illustrate these operations, suppose we have a LaserDisc object named source#1 which produces a (recorded) Video value on its output port. Convenient units for world and object times could be seconds and frame numbers. If we send the following messages to source#1:

```
Cue(30.0)
Scale(0.5)
Invert()
Start()
```

the effect would be to play the first 30 seconds of laser disc in reverse at twice the normal rate (i.e., in 15 seconds).

The last three methods of MultimediaObject: Translate, Scale and Invert, are called *temporal transformations*. The effect of these operations can be visualized using *timeline diagrams* of output ports. Figure 4 shows one such diagram for an output port which at time T_B starts to produce the sequence $S = d_1, d_2, ..., d_n$ of duration T_L. The effect of a temporal transformation can be visualized as displacement, stretch, or flip of the timeline.

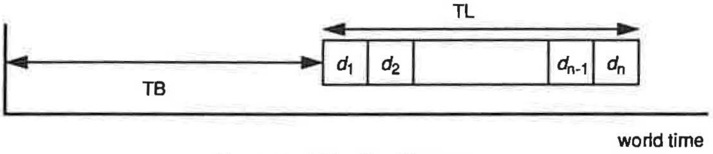


Figure 4 A Timeline Diagram.

3. Composite Multimedia

One reason for the complexity of multimedia information is that it often consists of a richly structured collection of lower-level multimedia components. There are several ways in which this structure may be organized. For instance, *multimedia documents* make use of a hierarchical component structure, while *hypermedia* involves a network structure. While these two forms of organization are certainly useful, they tend to neglect possible temporal relationships between components. A third method of organizing multimedia information, and one that focuses on these relationships, is what we will call *temporal composition.*

3.1 Temporal Composition

The motivation for temporal composition comes from the need to model situations where a number of multimedia components are simultaneously presented. Television and films are two obvious examples, each containing both audible and visual components.

Within our object-oriented framework, we will model composite multimedia through the use of special objects defined as follows:

Definition: A *composite* multimedia object is a multimedia object containing a collection of *component* multimedia objects and a specification of their temporal and configurational relationships.

The two groups of relationships specified by a composite multimedia object are used for different purposes. In particular:

- *temporal relationships* – indicate the synchronization and temporal sequencing of components.
- *configurational relationships* – indicate the connections between the input and output ports of components.

As an example, suppose we want to construct a multimedia composite, c_1, which, when presented, behaves as follows:

Starting at time t_0 an audio object $audio_1$, and a video object, $video_1$, are presented. At time t_1, a fade starts from $video_1$ to a second video object, $video_2$. The transition is completed at time t_2 and at time t_3 both $audio_1$ and $video_2$ are stopped.

Here $audio_1$, $video_1$ and $video_2$ are source objects. To complete the composite, three additional objects are needed: audioOut, an audio sink; videoOut, a video sink; and dve_1, a DVE filter for performing the fade from $video_1$ to $video_2$.

The temporal relationships of a composite object are easily depicted with a *composite timeline diagram.* Such a diagram contains one timeline for each output port within the composite. The diagram for the composite c_1 is shown in Figure 5.

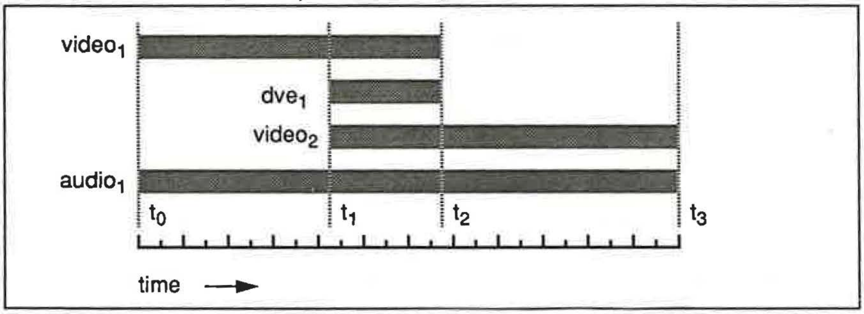


Figure 5 A Composite Timeline.

In constructing a composite timeline, the Translate transformation is used to adjust the positioning of the components; Scale and Invert may also be applied to alter presentation timing. In general, each component's "placement" within the composite is specified by its translation offset, scale and orientation. This is analogous to how complex graphics objects can be constructed by applying geometric transformations to groups of simpler objects. In graphics, the sequence of transformations that have been applied to an object is captured by the object's transformation matrix. Similarly, the sequence of temporal transformations used to place a component within a composite can be described by a 2x2 transformation matrix (if time is expressed in homogeneous coordinates).

Since composites are multimedia objects, it is possible to apply temporal transformations to composites as well as to their components. This allows the construction of *hierarchical composites.* As in graphics modelling, the transformation from object to world coordinates is then calculated by traversing the hierarchy from the node representing the object to the root or "world" node.

A final transformation, the *presentation transformation*, analogous to the viewing transformation in graphics, provides the mapping from world time to *presentation time* – the time of the presentation (i.e., the hour of the day, the minute of the hour, etc.). By adjusting this transformation it is possible to dynamically control the presentation, allowing implementation of user-level commands such as "fast forward" or "reverse."

The timeline representation shows the concurrency within a composite due to the superimposition of a number of multimedia *channels* (each horizontal bar in the timeline diagram), it also identifies *transition points*, i.e., times where sources start or stop. For composite c_1, there are four channels and four transition points: t_0, t_1, t_2 and t_3.

Transition points divide world time into a number of intervals. A composite's configurational relationships specify, for each such interval, the connections between input and output ports of its components. This information can be depicted with a *component network*, where nodes correspond to components of a multimedia composite and edges to port connections. The component network for c_1 during the interval $[t_1, t_2]$ is shown in Figure 6.

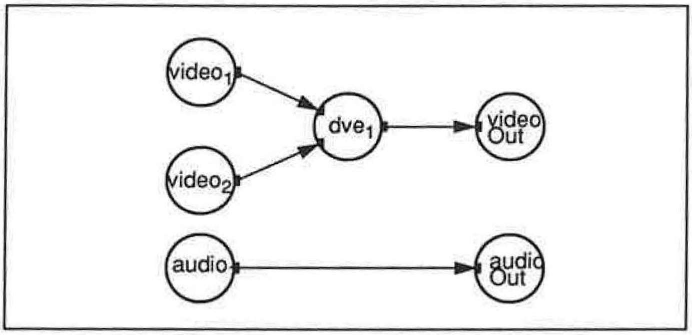


Figure 6 A Component Network (for the interval $[t_1, t_2]$ of Figure 5).

3.2 Example: Integration of 3d Graphics and Video

As a more elaborate example, consider combining interactive 3d graphics with video imagery. Suppose that presentation takes place on a stereo display and that the user is wearing an input device which measures head position and orientation (there are commercial user-interface devices which combine both the stereo display and the position/orientation sensor into a single helmet-like package). The problem is to generate 3d imagery which responds to changes in the user's visual perspective and combine this imagery with an appropriately transformed video signal. The video is provided by a source, such as a VCR or laser disc and overlaid on a "video surface" appearing in the 3d imagery. For simplicity we assume that the video surface is rectangular (in world coordinates) and of the same aspect ratio as the video signal. However, because of the perspective transformation, the surface may appear skewed. Also the surface may be fully or partially hidden by other surfaces.

The combined video and graphics can be represented by a composite object. This object is not interesting from a timeline perspective since all its components start and stop together. However, the component network is rather complex (see Figure 7) and contains fourteen objects.

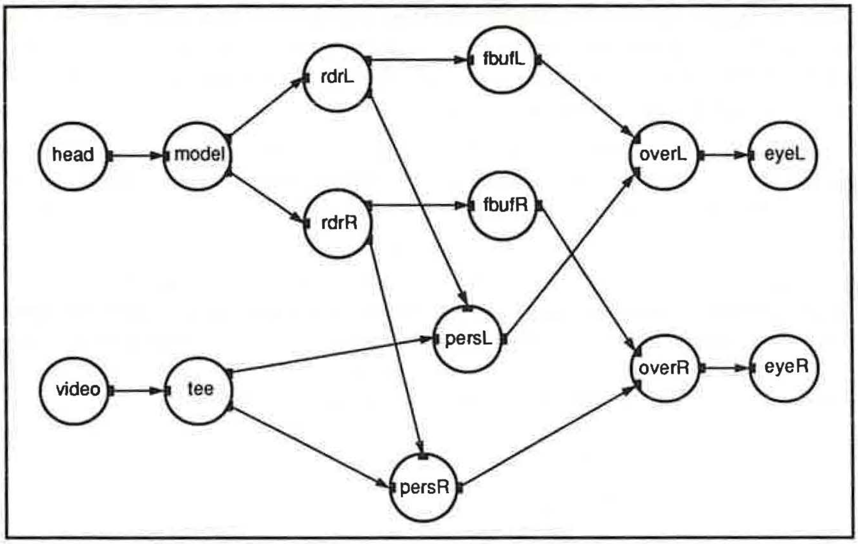


Figure 7 Stereoscopic 3d Graphics with Video Overlays.

They are:

head – produces a sequence of head position/orientation values.

model – maintains a 3d scene model, alters the viewing perspective according to deduced eye positions and sends scene descriptions for the left and right eye to its output ports.

rdrL, rdrR – two Render objects. Each accept a scene description, generate an image, and determine the display coordinates of the video surface.

fbufL, fbufR – two FrameBuffer objects.

video - the source of the video signal.

tee - duplicates its input on two output ports.

persL, persR – two DVE objects used to apply a perspective transformation to a video signal, the transformation is specified by the second input port.

overL, overR – two DVE objects. Each overlay two input two video signals to produce a third (some clipping may be done using a technique known as "chroma keying").

eyeL, eyeR – the final display devices.

Using current technology, most of these objects would be hardware-based, the exceptions being model, rdrL and rdrR. This does not mean, though, that our software structure is redundant. By encapsulating the hardware within software objects we gain much flexibility. For example, if we wanted to ignore the video, or ignore stereopsis, we could just form other, simpler, composites. We also gain flexibility from being able to substitute objects. For instance, one might replace the head object by an object which outputs recorded information – this could be useful for debugging or demos. Also, since the composite would then have no live sources we could apply temporal transformations and reverse or change the speed of presentation.

4. Scripting

In the previous section we discussed techniques to combine objects in order to obtain more elaborate behavior. Such composition techniques are very powerful but their proper application depends on two important constraints. First, the objects to be composed have to be well understood both individually and in partnership with other relevant objects. Second, composition requires programming, i.e, it is both tedious and error prone. In this section we will discuss a higher-level way to specify composites called "scripting" [8]. Scripting is based on a *scripting model* which defines the allowed ways that objects can be composed. In this manner many of the details of the composition do not have to be explicitly stated. In addition, scripting smooths over certain incompatibilities between objects and allows the composition of objects which have not been *a priori* designed to work together.

The idea of scripting is rather general. It has been tried in many environments, e.g., there are shell scripts, application scripts, and user interface scripts. In addition, scripting appears often in situations with dynamic, multi-agent activity. For instance, a coach explains a play to his players by using a script. He does not need to explain the details of the play or the rules of the game but only the essence of the combined actions of the players. In theater or film there is also a notion of a script. It explains in general terms the movements and the cooperation between the actors. In our environment we will use scripts and scripting as a technique to outline ways that multimedia objects can be put together. It is only natural since scripting is heavily used in artistic environments such as film, theater, or video production.

We now show how the notion of scripting can be applied to active media. We first provide a definition of "script" for our environment:

Definition: A *script* is an instance of a *script class*. Script classes are specializations of the class of composites.

Scripts differ from composites in that there are constraints on the types of components allowable within a script and, possibly, constraints on their configuration. These constraints are part of the specification of script classes.

A scripting language will be used to specify scripts. Interpretation of such a language relies on the scripting model. For active media, the scripting model contains:

1. multimedia hierarchies
   The scripting model knows about multimedia data types, multimedia object classes, and their subtype / subclass relationships. In particular, the scripting model contains the graphs shown in Figure 2 and Figure 3.
2. connection types
   We have not discussed component connections in detail, but in our prototype implementation of the framework there are a number of different types of connections. Examples

are connections corresponding to communication by message passing, buffering, or physical cable.

3. ports
   For each multimedia object class the scripting model contains port descriptions. A port description identifies whether the port is for input or output, the multimedia data type of the port, the connection types which can be attached to the port, and whether the port will accept multiple connections.

4. object interfaces
   Part of the interface of a multimedia object is only used within the framework whereas other parts are available for scripting (these correspond to the *FII* or *framework internal interface* and the *FEI* or *framework external interface* described in [3]). The scripting model identifies the FII and FEI for each multimedia object class. For instance, temporal transformations belong to the FEI for all multimedia object classes, while synchronization methods belong to the FII.

5. script membership constraints
   The scripting model contains the constraints on component types for the various script classes.

6. script configuration constraints
   Script classes may specify constraints on the configuration of components and connections within their instances. This information is part of the scripting model.

The scripting language we shall illustrate contains two main constructs: scripts themselves and *activities*. While at the framework level a script is a composite multimedia object, working at the scripting level, a script is specified by combining activities. An activity, in turn, is either a script or a multimedia object. There are three operators used to combine activities:

- a_1 >> a_2 : sequential execution. Activity a_2 will be scheduled after the completion of a_1.
- a_1 & a_2 : parallel execution. Activities a_1 and a_2 start together.
- n*a : repeated execution. Activity a is repeated n times.

We next give two examples of active media scripts. These examples are derived from separate scripting facilities implemented by our group [1][2][5]. Here we have re-expressed and integrated this earlier work by using a more general scripting language. Also, we want to point out that scripting can be either textual or visual. It is not the representations that matter but the flexibility and ease of use of the scripting facility.

4.1 Example: Musical Scripting

Typically, a musical script first declares a number of *note lists*, these are objects which generate sequences of notes. Our example is based on MIDI[1] and we will make use of an multimedia object class called Notes, instances of this class read a file and produce a MIDI sequence on their single output port.

1. MIDI, or *Musical Instrument Digital Interface*, is a standard for communicating with musical devices.

One or more note lists can be converted into a *melody* by temporal composition. Here we will use the script class Melody which aggregates Notes objects. Complex hierarchies of melodies can be developed by implementing functions at the framework level that return melodies, by applying temporal transformations, or by developing different sources of note lists.

Here is an example of a musical script:

```
activity Notes     frJ("frereJacques.midi");
activity Notes     dmV("dormezVous.midi");
activity Notes     sM("sonnezLesMatines.midi");
activity Notes     ddd("dingDingDong.midi");
activity MidiMix   mix(4);                         // a MIDI mixer with 4 input ports
activity MidiOut   mout;                           // converts MIDI to analog audio
connect mix.out to mout.in;

script Melody chanson = 2*frJ >> 2*dmV >> 2*sM >> 2*ddd;
script Melody ch1 = 2*chanson;
script Melody ch2 = 2*chanson;
script Melody ch3 = chanson;
script Melody ch4 = chanson;

ch3.Transpose(-12);                                // Melody transformations
ch4.Transpose(12);                                 // applied to scripts
ch4.Crescendo(12);                                 // ch3 and ch4

time entry2 = 2 * frJ.Duration();                  // calculation of the
time entry3 = entry1 + 2 * dmV.Duration();         // "entry" times for
time entry4 = entry2 + 2 * sM.Duration();          // ch2, ch3 and ch4

script Melody canon = {
     ch1
     &
     ch2.Translate(entry2)
     &
     ch3.Translate(entry3)
     &
     ch4.Translate(entry4);

     connect ch1.out to mix.in1;
     connect ch2.out to mix.in2;
     connect ch3.out to mix.in3;
     connect ch4.out to mix.in4;
}
```

This script takes four short sequences of notes which have been previously played at a keyboard and loaded into files. The basic sequences are combined into a whole melody called chanson, where each sequence is repeated twice and concatenated to the others. This is the famous french folk tune *Frère Jacques*.

Since this tune is a "catch," the next step is to take four instances of it and play them in round. This is done in the melody called canon, where chanson is referenced four times, with different delays. Some other variations are also added, like playing one of the versions in the superior or inferior octave, or performing a *crescendo*. These effects are achieved by applying Transpose and Crescendo – two methods which are defined and implemented within the framework for melodies and note lists, and that are declared within the scripting model as being available at the scripting level.

The timeline diagram and component network that would be constructed by the script interpreter are shown in Figure 8. (Here the local time coordinates of the script could be in number

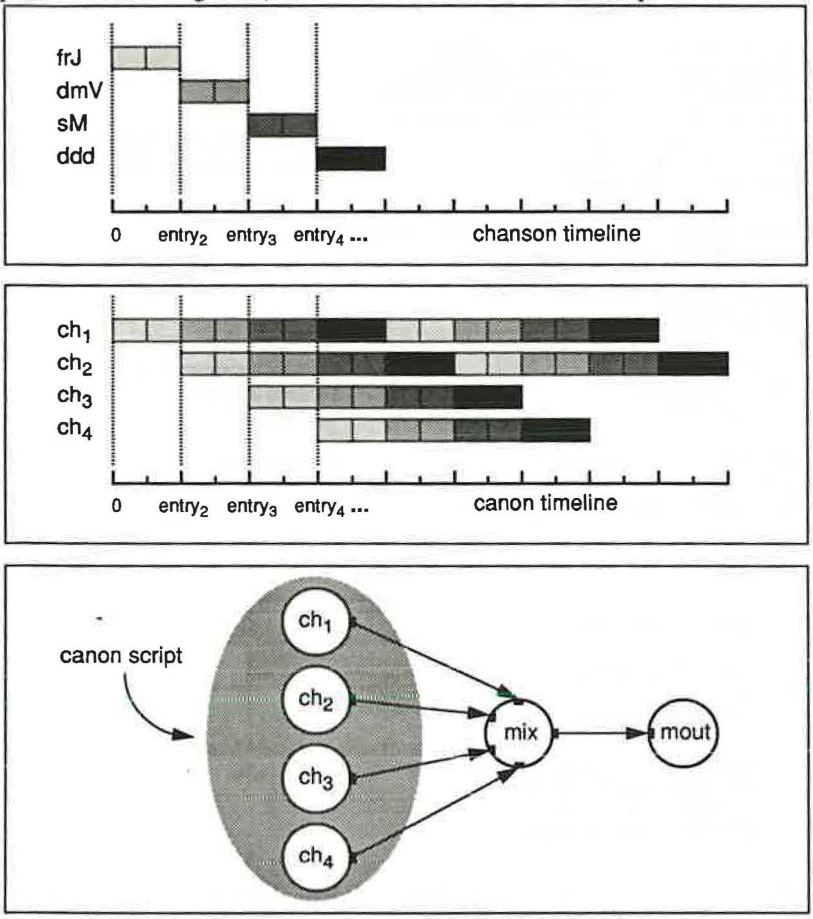


Figure 8 Musical Script: Timeline and Component Network Diagrams.

of beats. The presentation transformation would then fix the number of beats per second.)

One final observation has to do with the mix object. Note that there are four input ports, one for each MIDI sequence. The reason that the sequences are not merged into a single input port is that this object may perform operations which apply to specific inputs (e.g., raise the volume on channel #2) and so it must be able to separate its input sequences.

4.2 Example: Animation Scripting

Animation scripts rely on graphical objects and two multimedia object classes: Scene and Animator. A Scene instance contains a hierarchy of graphical objects, light source descriptions, and a viewing position (and direction). A graphical object contains a description of itself, in terms of surface primitives and surface attributes, plus a set of methods for modifying this description. For instance, the specification of a Bird graphical object class could look something like:

```
class Bird subclass of GraphicalObject {
      // surface description
      // and state information
methods:
      Flap();                    // move the wings
      Move(x, y, z);             // go somewhere
      ...
}
```

The Animator class is the key to animation scripting. Every Animator instance is connected to a Scene object and bound to a method of a graphical object within the scene. Each scene contains a frame counter; an Animator will attempt to invoke its bound method once per frame. Furthermore, if an Animator has to perform an activity within a particular duration, for instance, move a graphical object over a certain distance, it measures the duration by the local time of the scene. Thus, if real-time animation is desired, say at *n* frames per second, then the scene must increment it's frame counter every $1/n^{th}$ of a second *and* increment it's local time in real-time. (Of course this doesn't assure that the animation can be produced in real-time, merely that the "clocks" of the Animator objects are running at the correct rate.)

We shall now design a small animation using the Bird object class. It creates three birds that fly together, but with different speeds and behaviors.

```
GraphObjNode       flock;                        // an abstract graphical object
Bird               bird1 partof flock;           // three birds,
Bird               bird2 partof flock;           // members of a flock
Bird               bird3 partof flock;
activity Scene     model containing flock;       // a scene model

bird1.flapSpeed = 0.15;                          // number of flaps
bird2.flapSpeed = 0.8;                           // per unit time

// initial positioning
//
bird1.SetPos(0, 150, 0);         bird1.SetOrient(0, 0, PI*2/3);
bird2.SetPos(0, 150, -20);       bird2.SetOrient(0, 0, PI*6/5);
bird3.SetPos(0, 150, 30);
model.lookFrom.SetPos(-400, 0, 0);

// activities which will modify the scene model
//
activity Animator          flap1(bird1, Flap);
activity Animator          flap2(bird2, Flap);
activity Animator          flap3(bird3, Flap);
activity Animator          spin(flock, Rotate, from Orient(), to(0, 0, PI*6), duration 40);
activity Animator          move1(bird1, Move, from Pos(), to(0, 0, 20), duration 15);
activity Animator          move2(bird2, Move, from Pos(), to (0, 0, -20), duration 25);
activity Animator          backAway(model.lookFrom, Move, from Pos(), to (-800, 0, 0),
                                        duration 15);

script Animation flockFlight = {
      flap1 & flap2 & flap3
      &
      spin
      &
      ( move1 >> move2)
      &
      backAway.Translate(20);
      connect *.out to model.in;
}
```

The first lines are declarations that create the graphical object hierarchy contained in the scene. Within the hierarchy, each object's position and orientation is relative to its parent. For example the wings of a bird move in the space of the bird's body, and the bird in turn moves in the space of the flock node.

After the declarations, a collection of statements are used to bring the objects to appropriate initial positions. Next, a number of Animator objects are created. Finally the description of the motion is specified in the script flockFlight: the three birds start flapping wings; at the same time the first bird starts an upwards motion during 15 units of time, followed by a downwards motion during 25 units; 20 units after the beginning of the scene, the viewing position starts to move away from the birds.

The timeline diagram and component network generated by this script are shown in Figure 9. (We have not completed the example by including the rendering of the animation and its display, however this can be performed by attaching a pipeline of Render, FrameBuffer, and Monitor objects to model's output port.)

Animation scripts illustrate a notion that seems very common when dealing with multimedia. Note that Animator activities are declared in the following two ways:

```
activity Animator    a1(<graphical object>, <method>);
activity Animator    a2(<graphical object>, <method>, from <val>, to <val>, duration <val>);
```

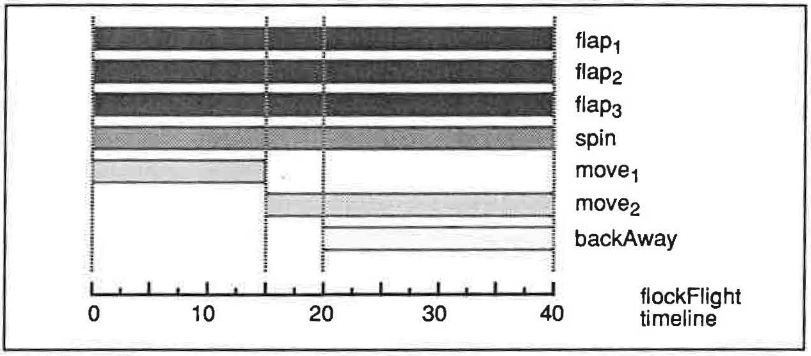
The first form is used if <method> takes no arguments. The interesting case, however, is when arguments are required, for then the arguments must be *interpolated* over a certain interval. To be specific, suppose we have an Animator, a, declared as follows:

```
activity Animator    a(somebird, Move, from (0, 0, 0), to (10, 0, 0), duration 10);
```

Furthermore, assume that the frame counter is being incremented twice per unit time. Then the method invocations interpolated by this animator are:

```
somebird.Move(0.5, 0, 0);
somebird.Move(1, 0, 0);
somebird.Move(1.5, 0, 0);
...
somebird.Move(10, 0, 0);
```

Another use of interpolation within multimedia occurs in keyframing. For instance, to produce a commercial with a tumbling and stretching logo, one would typically need a series of keyframes, each specifying a set of parameters which control transformations to be applied to the



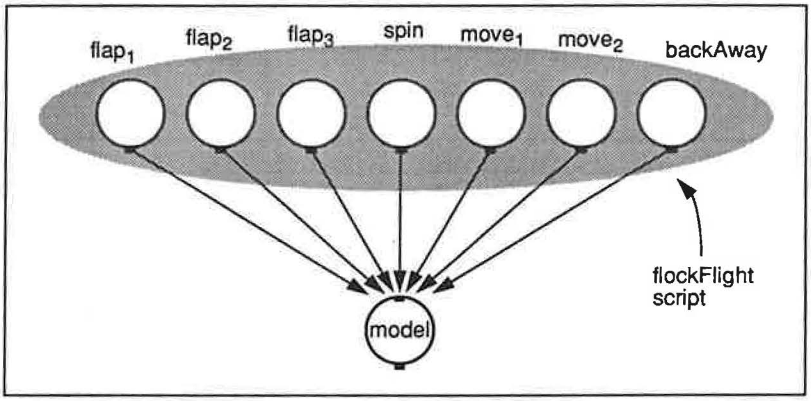


Figure 9 Animation Script: Timeline Diagram and Component Network.

logo. By interpolating these parameters between keyframes one obtains a smoothly transforming logo.

Interpolation is intimately related to temporal transformations on multimedia objects. In general, if a source object supports the Scale transform, it will make use of interpolation in some manner.

5. Conclusion

In this paper we have outlined an environment for active media and have described the composition of multimedia objects from the programming and scripting perspectives. Many of the examples we have discussed have been implemented. We are currently working on a more general active media scripting environment to allow us to experiment with the power and difficulties of such an approach. In parallel we are acquiring hardware and collecting software which can be incorporated within the framework.

We will apply our techniques to a specific domain which is both interesting and aesthetically pleasing. Our chosen application is "virtual museums," that is, the audiovisual rendering of artifacts in virtual settings [11] (see also the tenth paper in this volume). The active aspect of the objects allows the virtual museum to be dynamic and provides user or group interaction.

In addition to developing an active media environment, work is continuing to elaborate more scripting models and a more general facility for scripting [8]. In particular we would like to develop a active media design tool which uses a graphical representation of scripts rather than the textual language discussed in the previous section. In this way the users can build and understand scripts in a visual manner, this seems appropriate because of the strong visual-orientation of multimedia.

References

[1] Dami, L, Fiume, E., Nierstrasz, O. and Tsichritzis, D. Temporal Scripting using TEMPO. In *Active Object Environments*, (Ed. D. Tsichritzis) Centre Universitaire d'Informatique, Université de Genève, 1988.

[2] Dami, L. Musical Scripts. In *Active Object Environments*, (Ed. D. Tsichritzis) Centre Universitaire d'Informatique, Université de Genève, 1988.

[3] Deutsch, L.P. Design Reuse and Frameworks in the Smalltalk-80 System. In *Software Reusability, Vol. II*, (Eds. T.J. Biggerstaff and A.J. Perlis) ACM Press, 57-71, 1989.

[4] Ellis, C. and Gibbs, S. Active Objects: Realities and Possibilities. In *Object-Oriented Concepts, Applications, and Databases*, (Ed. F. Lochovsky and W. Kim) Addison-Wesley, 1988.

[5] Fiume, E., Tsichritzis, D., and Dami, L. A Temporal Scripting Language for Object-Oriented Animation. *Proc. Eurographics'87*, North-Holland, 1987.

[6] Gibbs, S. Composite Multimedia and Active Objects. To appear in *OOPSLA'91*.

[7] Gibbs, S., Dami, L., and Tsichritzis, D. An Object-Oriented Framework for Multimedia Composition and Synchronisation, *Eurographics Multimedia Workshop*, Stockholm, 1991.

[8] Nierstrasz, O., Dami, L., de Mey, V., Stadelmann, M., Tsichritzis, D., and Vitek, J. Visual Scripting: Towards Interactive Construction of Object-Oriented Applications. In *Object Management*, (Ed. D. Tsichritzis) Centre Universitaire d'Informatique, Université de Genève, 1990.

[9] Papathomas, M. Concurrency Issues in Object-Oriented Programming Languages. In *Object Oriented Development*, (Ed. D. Tsichritzis) Centre Universitaire d'Informatique, Université de Genève, 1989.

[10] Sony DME-9000. *Product Brochure.*

[11] Tsichritzis, D. and Gibbs S. Virtual Museums and Virtual Realities. To appear in the *Proc. of the International Conference on Hypermedia and Interactivity in Museums*, 1991.

[12] Virtual Environments and Interactivity: Windows to the Future. (SIGGRAPH panel session). *Computer Graphics 23*, 5 (Dec. 1989), 7-18.

[13] Wegner, P. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger 1*, 1 (Aug. 1990), 7-87.