Session VIII: Multimedia Abstractions II

Chair: Venkat Rangan, University of California at San Diego

The eighth session was devoted to new abstractions for programming multimedia applications, and tools for specifying their execution and presentation. Programming abstractions for multimedia have not received much attention before, and are becoming increasingly important as multimedia applications grow in both complexity and number.

In the first paper of this session, Simon Gibbs of the Universite de Geneve together with Christian Breiteneder, Laurant Dami, Vicki de Mey, and Dennis Tsichritzis presented "A Programming Environment for Multimedia Applications." The environment consists of two levels: A systems level environment for handling synchronization between media, and a user level environment for specifying media presentation.

The systems level environment encapsulates hardware dependencies, and enforces synchronization between media objects which can be sources, sinks, or filters, depending on whether they produce or consume media values, or both. The user level environment provides an object-oriented framework in which, application designers can specify in a scripting language the composition, presentation, and communication among media objects. Sequences of presentations constitute activities, and multiple activities can be executed sequentially, concurrently, or periodically.

The second paper was by Gerold Blakowski, Jens Hübel, and Ulrike Langrehr of the University of Karlsruhe ("Tools for Specifying and Executing Synchronized Multimedia Presentations"). Gerold presented tools for managing synchronized presentation of multimedia in distributed heterogeneous environments.

Users can define objects of type MediaInformation, Presentation, or Transport. A graphical synchronization editor helps users to specify both intra-object and inter-object synchronization using a context-free grammar based description language. From these synchronization specifications, the synchronizer constructs a presentation thread of all the media objects, using which it enforces the synchronization relationships. The synchronizer also takes actions such as pause/wait or acceleration/skip in order to speed up a lagging media presentation or slow down a leading media presentation.

The entire media synchronization management system including the graphical editor is being designed as part of the MODE project in the NESTOR authoring-learning environment being developed at the Universities of Karlsruhe and Kaiserslautern.

The last paper by Ralf Steinmetz and J. Christian Fritzsche of IBM European Networking Center and Johann-Wolfgang-Goethe University of Frankfurt ("Abstractions for Continuous-Media Programming"), presents an interesting new concept in which continuous media are treated as data types within programming languages.

Christian elaborated that earlier approaches commonly use multimedia libraries and toolkits for removing hardware and implementation dependencies of multimedia applications. However, neither libraries nor toolkits permit full integration of multimedia into programming environments; the encapsulation they provide frequently comes at the expense of performance, and they are not efficiently supported by the operating system. Simply adding abstract data types to programming languages does not suffice to adequately express synchronization, communication, and parallelism in multimedia applications.

The authors propose rather to use media as fundamental data types within high-level languages, thereby greatly enhancing the expressive power within multimedia applications. Using an object-oriented language, starting with media devices and their data units as fundamental objects, higher-level media abstractions are built using class hierarchies. Media objects can also have a lifetime, in which case they become active objects.

In summary, this session touched upon two significant aspects of multimedia systems: synchronization and programming abstraction. Interesting questions were raised as to whether synchronization mechanisms can assume the existence of globally synchronized clocks or not. Another interesting issue was whether media such as video and audio must be regarded as composed of frames and samples, or higher-level objects such as images and phonemes. These topics will continue to assume increasing importance as the use of computer systems to support digital continuous media becomes more and more pervasive.

A Programming Environment for Multimedia Applications

Simon Gibbs, Christian Breiteneder,
Laurent Dami, Vicki de Mey, Dennis Tsichritzis

Université de Genève[1]

Abstract

A programming environment for the development of multimedia applications is described. The environment is based on a two-level architecture: a systems-oriented *framework-level* concerned with hardware control and synchronization and a user-oriented *scripting-level* concerned with presentation specification. The two levels are outlined.

1. Introduction

It is perhaps self-evident that programming environments providing specific support for multimedia would facilitate the development of multimedia applications. Looking more closely, however, one can discern many reasons why such environments are needed:

First, multimedia involves concepts from audio recording, video production, animation, and music – concepts that are novel to many programmers. Additionally, multimedia operations often involve special hardware, leading to lack of portability and longer development times. Information about media properties and hardware dependencies should be incorporated within the programming environment.

Second, 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. 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. 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 third reason for considering a multimedia programming environment is that complex user interfaces, such as virtual realities can be viewed as multimedia applications. Yet such applications, because of novel interface devices (e.g., stereoscopic displays, head-position and orientation trackers) and the need for real-time performance, are difficult to construct with current programming tools.

Finally, we hope that multimedia programming environments, and their associated concepts, may lead to a general model for describing and developing a wide range of multimedia appli-

1. *Authors' address:* Centre Universitaire d'Informatique, 12 rue du Lac, CH-1207 Geneva, Switzerland.
Email: {simon, chris, dami, vicki, dt}@cui.unige.ch
Tel: +41 (22) 787.65.80
Fax: +41 (22) 735.39.05

cations. 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.

This paper describes a programming environment for developing multimedia applications. The requirements for the environment are that it:

- be based on a simple conceptual model of multimedia functionality, yet one which is general enough to capture the variety of multimedia, including sound, video, music, and animated sequences,
- be easy to use and not require expertise in multimedia technology, yet be open and extensible so that more experienced programmers are not constrained,
- encapsulate hardware dependencies,
- allow complex multimedia effects, for example the synchronization of an audio and video signal, or the juxtaposition of two video signals.

One approach for such an environment is a two-level architecture consisting of a system-oriented layer and a user-oriented layer. The first, called the "framework level," is concerned with hardware control and synchronization, the second, called the "scripting level," is concerned with presentation specification.

2. The Framework Level – Multimedia Objects

The framework level has been described elsewhere [5][6], here we provide just a summary[1]. The starting point is the use of data types to characterize media information:

Definition: A *media 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.

Media values are related to *media objects*, these are defined as follows:

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

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.

Each media object can be viewed as a collection of *ports*. A port has a (media) data type and is used either for input or output. Media objects are divided into three categories: *sources*, *sinks*, and *filters*. A source produces media values, a sink consumes values, and a filter both produces and consumes.

Informally, *multimedia values* are aggregates of media values, while *multimedia objects* are aggregates of media objects. How these aggregates are formed is discussed in section 2.2.

Media objects and media values make use of two inheritance hierarchies. For example, the class LaserDiscPlayer would be (ultimately) a subclass of MediaObject, similarly LaserDiscVideo would be (again ultimately) a subclass of MediaValue.

1. It should perhaps be pointed out that there has been a change in terminology from references [5] and [6]. In particular, what were previously called Multimedia, MultimediaObject and CompositeMultimediaObject are now MediaValue, MediaObject and MultimediaObject respectively.

2.1 Operations on Media Objects

All classes of media objects inherit methods from the class ActiveObject and the class MediaObject. A schematic and partial C++ specification of these classes is:

```
class ActiveObject {
public:
    bool            Start();
    bool            Stop();
    bool            Pause();
    bool            Resume();
};

class MediaObject {
public:
    //
    // temporal coordinates
    //
    objectTime              CurrentObjectTime();
    objectTime              WorldToObject(worldTime);
    worldTime               CurrentWorldTime();
    worldTime               ObjectToWorld(objectTime);
    objectInterval          WorldToObjectI(worldInterval);
    worldInterval           ObjectToWorldI(objectInterval);
    //
    // composition
    //
    void                    Translate(worldTime);
    void                    Scale(float);
    void                    Invert();
    MultimediaObject        Parent();
    //
    // synchronization
    //
    void                    Sync(worldTime);
    worldInterval           SyncInterval();
    worldInterval           SyncTolerance();
    SyncMode                SyncMode();
    void                    Cue(worldTime);
    void                    Jump(worldTime);
};
```

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

The class MediaObject makes 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 presentation 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 media 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).

2.2 Composition and Synchronization of Multimedia Objects

The framework level provides a technique for aggregating media objects. This technique, called *temporal composition*, is used to form *multimedia objects*.

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

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

The two groups of relationships specified by a 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.

A composite, c, maintains synchronization by attempting to assure

$$\text{Abs}(c.\text{CurrentWorldTime}() - c_i.\text{CurrentWorldTime}()) < c_i.\text{SyncTolerance}()$$

for each activated component c_i. However, because of the varying nature of components, multimedia objects must be flexible and support a variety of synchronization techniques. In the framework, each component has a *synchronization mode* attribute. Depending on the value of this attribute, which can be queried by the method SyncMode, the multimedia object adopts different approaches to synchronization. Presently there are four synchronization modes: NO_SYNC, DEMAND_SYNC, TEST_SYNC, and INTERRUPT_SYNC (see [6] for further details).

3. The Script Level – Scripts and Activities

In the previous section we discussed techniques to combine media 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 discuss a higher-level way to specify composites called "scripting" [7]. 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.

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

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

Scripts differ from multimedia objects 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 can be used to specify scripts. Interpretation of such a language relies on the scripting model. For multimedia, the scripting model contains:

1. multimedia hierarchies
   The scripting model knows about MediaValue and MediaObject classes and their respective subclass hierarchies.
2. connection types
   We have not discussed component connections in detail, but the framework level supports a number of different types of connections. Examples are connections corresponding to communication by message passing, buffering, or physical cable.
3. ports
   For each media object class the scripting model contains port descriptions. A port description identifies whether the port is for input or output, the media data type of the port, the connection types which can be attached to the port, and whether the port accepts multiple connections.
4. object interfaces
   Part of the interface of a media object is only used within the framework level 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 media object class. For instance, "temporal transformations" (e.g., the Scale and Translate methods) belong to the FEI for all media 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 contains two main constructs: scripts themselves and *activities*. A script is specified by combining activities; an activity, in turn, is either a script or a media object. (Consequently a script reduces, at the framework level, to 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.

Examples of scripting facilities are described in [1][2][4].

4. Conclusion

The above has outlined an approach for constructing an environment for programming multimedia applications. The environment contains two layers: a system-oriented layer consisting of an object-oriented class framework, and a user-oriented layer based on a scripting language. At the moment we are refining the environment's design by developing a demanding "driver" application [8].

References

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

[5] Gibbs, S. Composite Multimedia and Active Objects. *Proc. OOPSLA'91*, 97-112.

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

[7] 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.

[8] Tsichritzis, D. and Gibbs S. Virtual Museums and Virtual Realities. *Proc. of the International Conference on Hypermedia and Interactivity in Museums*, 17-25, 1991.