A Standard for Multimedia Middleware

D.J. Duke[1] and I. Herman[2]

[1]Department of Computer Science, The University of York, Heslington, York, YO10 5DD, UK
[2]Centrum voor Wiskunde en Informatica (CWI), 413 Kruislaan, 1098 SJ Amsterdam, The Netherlands

Abstract

Hardware, software, and coding standards for digital media have played a significant role in making multimedia presentation an intrinsic compponent of many systems. However, these standards are primarily concerned with the storage, encoding and transport of media content, and have not been intended to address the integration of multimedia data into more general programming environments for media presentation. PREMO (PResentation Environments for Multimedia Objects) is a project within the SC24 committee of the International Organisation for Standardization (ISO) aimed at developing an API (Application Programmer Interface) that integrates the processing and presentation of distributed multimedia with that of synthesised graphics. This report summarises the contents of the PREMO standard and explains how the integration of graphics into a general framework for media processing is achieved.

Keywords: distributed multimedia; standards; PREMO

1. Introduction

The use of multimedia is now so widespread that the term 'multimedia computing' has become almost a tautology. Few people today would conceive of purchasing or using a computer system that was *not* capable of displaying or processing multimedia data. Standards are now available for the encoding, transport and presentation of a rich variety of media data. Many of these, such as MHEG[11], MPEG[40], MIDI and VRML[27] are well known even amongst non-professional computer users. New standards, such as PNG and SMIL[59] are under development in response to the opportunities and needs created by the world wide web. This apparent wealth of media standards makes it all the more important to situate PREMO and understand its role:

- *PREMO is a presentation environment.* PREMO, like previous SC24 standards, aims at providing a standard "programming" environment in a very general sense. The aim is to offer a standardised, hence conceptually portable, development environment that helps to promote portable multimedia applications. PREMO concentrates on the application program interface to "presentation techniques"; this is what primarily differentiates it from other multimedia standardization projects. PREMO has also been developed at a time when object-oriented programming techniques have become of interest to the graphics community[43,58], and this is reflected in the standard by the explicit use of object-oriented concepts as the foundation of PREMO.
- *PREMO is aimed at a multimedia presentation.* Whereas earlier SC24 standards concentrated either on synthetic graphics or image processing systems, multimedia is considered here in a very general sense; high–level virtual reality environments[41], which mix real–time 3D rendering techniques with sound[6], video, or even tactile feedback, and their effects, are, for example, within the scope of PREMO.

In the remainder of this section, we will explore these two points in more detail, and in so doing establish the fundamental rationale for the technical content and approach of PREMO that is described in the remainder of this report.

1.1 What PREMO Is

Programming interfaces for graphics ("graphics packages") are now widely known and used. These include *de jure* standards developed for example within ISO, such as GKS and PHIGS[32,34], and industry-developed platforms such as GL[48] and Inventor[56] that have now become *de facto* standards themselves. The process is ongoing, with a new generation of graphics applications emerging based on the Java

technologies (e.g. Java3D[52]), and also in response to the needs and opportunities of web-based applications (e.g. VRML[27]). In contrast, programming interfaces for multimedia are rather less well known; while toolkits for multimedia applications have been developed, for example MET++[1] and MADE[31], standards for multimedia have concentrated largely on formats for the storage and transport of media, declarative models of media content (for example HyTime[47]). While the interface to presentation engines for such formats does provide a starting point for the applications programmer, the level of control over media processing that these affords is significantly lower than can be achieved in computer graphics. And significantly, none of the existing presentation models or engines integrates their media content with synthetic graphics.

The separation between synthetic graphics and other presentation media may reflect the different communities in which the fundamental developments took place (e.g. much of the early interest in multimedia was stimulated by applications in publishing and human-computer interaction, whereas graphics originally had stronger links with engineering and scientific applications). Irrespective of these differences in origin, two technological trends have meant that there is now a growing need to integrate these two threads of activity. At one end of the cost-performance spectrum, virtual environments and visualisation are emerging as mature technologies with needs that encompass both synthetic graphics and other media, e.g. 3D audio, acoustic and haptic rendering. At the other end of the cost spectrum, the availability of powerful, low-cost personal computing platforms has made it feasible to develop multimedia applications for mass markets, and for users of such machines to experiment with multimedia. An issue that spans this spectrum of applications is how application programs can access, construct, and control multimedia and graphics presentation. This is the context in which PREMO has been designed.

PREMO as middleware

The term “middleware” has come to the fore in recent years. It refers to a software layer, between the fundamental services of an operating system and more specific application development environments. PREMO provides a level of middleware which supports the implementation of a range of processing models for multimedia presentation. As an form of middleware, PREMO does not define stand-alone services in the way, for example, that a PHIGS renderer does. Instead, it provides an environment where various, vendor–specific components can cooperate. The middleware nature of PREMO has implications for how the software objects defined by the standard are described. On the one hand, these must not be too detailed, otherwise it would restrict the range of possible implementations, but on the other hand these objects must provide a non-trivial set of services. This strive for balance has fundamentally shaped the standard.

Why is middleware important? Consider, for example, the task of implementing a distributed multimedia application such as a multi-platform video–conferencing system. Due to the variety of available media formats, resource requirements, means of distribution control, etc., a significant portion of such an application is dedicated to issues like configurability, adaptability, access to remote resources, distribution, etc. A similar level of adaptiveness is also required when using media in combination, for example synthetic graphics, video, and computer animation. No one applications package addresses such a variety of needs, and without middleware such as PREMO, much of this infrastructure has to be developed from scratch, or adapted from a similar application. And the costs involved in modifying software to meet new demands are well known.

In addition to enabling interoperation, the existence of a middleware level such as PREMO can also assist in system evolution. The variety of graphics formats, available primitives, animation algorithms, etc., continues to expand, and portable applications increasingly have to adapt to an evolving environment. PREMO assists in this process by factoring out at least some of the technological constraints into components that can be interchanged and replaced, and by providing a flexible and extensible architecture in which new software components can be defined for use by existing applications.

Multimedia presentation is not the only concern that is open to support by middleware. Another, well known, example is architectural support for distributed object–oriented applications, as is provided by the CORBA[49] specification of the Object Management Group (OMG).[†] Although PREMO itself is not related to the various OMG specifications, PREMO should be viewed as a multimedia–oriented extension of the basic object services and architecture provided by systems like CORBA or, as another example, Java’s RMI[28,57] services. Seen in this way, PREMO fills the gap between the application-independent set of facilities offered by CORBA, and a distributed multimedia application. Indeed, the relationship between PREMO and a distributed object–oriented architecture is so close, it would be ill-advised to attempt an implementation of PREMO without the use of such services. More information on how the PREMO specification builds on the concept of distributed multimedia *without* committing to a particular model will be found in [14] and [29].

[†]In fact, a liaison existed between OMG and the relevant ISO group, during the development of PREMO, which clearly influenced the design of the standard.

PREMO as a reference model

As PREMO describes an implementation environment (a prototype is currently under preparation in Java[24]), the specification encompasses a range of concepts needed in multimedia systems development. By providing a broad, application independent model of media processing, the specification itself also serves as a reference model for distributed multimedia. This is significant, as in practice, "portability of programmers" is almost as important as the "portability of programs". Although only the latter role of information processing standards is usually publicised by organizations like ISO, the need for "programmer portability" in this area was also considered to be a major goal for PREMO. Having a common, well understood set of principles and techniques as a reference point greatly helps in understanding both the specificities and the commonalities of various multimedia programming environments. To achieve this goal, the PREMO specification deliberately sets out a number of details which are sometimes hidden in other systems. As a reference model, PREMO is not only significant in a didactical sense; a unifying set of concepts may play an important role in classifying, relating and organising the growing range of software toolkits that are available to the potential developer of a multimedia system. Without such concepts, this technological cornucopia is in danger of becoming an anarchic ensemble of incompatible and/or incomparable artifacts.

1.2 What PREMO *Isn't*

The characteristics that define what PREMO is – middleware and reference model – also reflect what PREMO is not. In particular, PREMO is intended to build on and utilise existing media standards, not to replace them. Given that there are standards in place for media formats and processing, these are two concerns that PREMO does not address.

PREMO is *not* a Media Format

The PREMO specification does not describe any new format for the representation and storage of media data. Instead, the standard makes it quite clear that the data processed by PREMO-based applications is expected to be stored in existing formats; ALAW, JPEG, MHEG, MIDI, MPEG, SMIL, VRML, to name a few. What PREMO does provide are mechanisms by which new PREMO objects can be defined for new formats, and by which existing objects can coordinate the formats that they use to exchange and process media data.

PREMO is *not* a Media Engine

The object types defined in the PREMO standard are not sufficient in themselves to realise a working multimedia application. To do this would have required the standard to commit to particular kinds of media processors and renderers, with specific interfaces. All that this would achieve would be to add yet another type of media engine into the growing collection of such devices. Instead, PREMO provides a number of object types that can act as "wrappers" around existing engines, and allow these to be used within a processing network involving other devices that may be based on quite different media formats or models. Rather than thinking of PREMO as a media engine, a somewhat better analogy is to view PREMO as a software architecture for multimedia applications; the objects defined by PREMO represent the basic constructs, the building blocks, for multimedia applications. Even this analogy is not quite the whole story though. Although parts of the PREMO specification provide building blocks that are "shaped" for supporting a particular architectural model of an application, these in turn rely on a set of lower-level PREMO objects, and users of PREMO is free to build on these, or modify the higher-level components, in order to instantiate whatever model of multimedia architecture that is most appropriate for their needs.

Just as PREMO is not a media engine, it is not a complete environment, either. It does not, for example, provide a framework for quality of service management. This may seem strange, since quality of service is a particularly fundamental problem with multimedia applications. However, quality of service management is currently bound tightly up with network management issues; as of yet, there is no emerging consensus on what application mechanisms are needed to implement quality of service, and indeed, it seems probable that, like the concept of a network, 'quality of service' actually spans a whole range of levels of concern, from well known physical properties such as bandwidth and latency of raw transmissions, up to questions that impinge on the eventual presentation of the data, for example synchronization constraints between lip motion in video frames and the corresponding speech in an audio stream. What PREMO does provide here is a basic set of hooks and facilities which a quality of service management protocol is free to utilise for monitoring and realising its requirements.

PREMO is *not* a user-oriented Specification

In addition to the technical problems of building a media application, multimedia systems designers need to address the question of how well a particular media system (both in terms of technology, and media content) meets the demands of its users. Like the issue of quality of service, usability involves a spectrum of concerns, from low level issues of signal quality, through questions about the cognitive resources and processes needed to interact with an application, through to questions of how a particular system is situated in the work context and environment of its users. These human factors must obviously be addressed by media systems designers by making appropriate use of the technologies at their disposal. PREMO is one such technology – the specification itself does not describe how it should be used to realise user requirements.

1.3 Structure of the Report

In this introduction we have set out the need for a standard to address distributed multimedia, and the rationale for designing the standard to be extensible from the outset. The remainder of this report is intended to provide an overview of PREMO. Section 2.introduces the concept of a PREMO component, around which the standard is structured as a means of promoting extensibility. Thereafter, the four 'components', or parts, that make up the official PREMO standard are summarised. Sections 3 to 7 then each take one of these components as its focus, and describes its design, main functional provisions, and role within the standard.

2. The Structure of PREMO

The concepts of modularity, data abstraction and component-based design are now well established within software engineering, where structures such as classes, modules and packages are used to manage the complexity of systems development by allowing the decomposition of a design into a set of parts which can be developed independently or incrementally, before being composed to form the desired system. The object-oriented basis of PREMO allows one level of structuring. However, this is relatively fine-grained, and in practice multimedia applications require *families* of objects that can be assembled to implement particular functionalities. Today, this concept is becoming widely adopted in the form of design patterns[22] and software architectures. These were, however, less well known when development of PREMO began, and consequently a somewhat simpler approach was adopted to structure the standard.

PREMO is defined as a collection of *components*, each of which provides one or more *profiles*. A component defines a collection of entities, such as object and non-object types. Object types provide services (in the form of operations that can be invoked by clients), or can have a more passive role, for example as data encapsulators. As not all of the types defined within a component are necessarily needed in a given context, PREMO components define one or more *profiles*, each consisting of a cluster of entities. A component can build on (extend) the profiles of other components, in the same way that a class in object-oriented programming can be defined as an extension to existing classes. The components defined in the PREMO standard are general purpose; they provide a progressively richer, more structured model of multimedia processing. It was the intention of the designers that functionality to address specific technologies, such as 3D audio, or virtual reality, or specific application domains, for example medical simulation or battlefield models, could be realised by the development of new components that extend some or all of the profiles defined in the standard. The four components of the PREMO Standard are as follows:

1) *Fundamentals*. This specifies the object model used by PREMO, and the requirements that a PREMO system places of its environment. Although the PREMO object model is similar to the core model of the OMG, it contains particular features needed to address the requirements of distributed systems.

2) *Foundation*. Object and data types that are generic to multimedia applications are defined in this component, including facilities for event management, synchronization, and time.

3) *Multimedia Systems Services*. Multimedia systems typically integrate a variety of logical and physical devices, for example input and output with devices such as video editors, cameras, speakers, and processing with devices such as data encoders/decoders and media synthesizers (e.g. a graphics renderer). This component of PREMO defines the infrastructure needed to set up and maintain a network of heterogeneous processing elements for media data. These facilities include mechanisms by which media processors can advertise their properties and be configured to match the needs of a network, and can then be interconnected and controlled. MSS was originally defined by the Interactive Multimedia Association[33] and subsequently adopted by SC24 and refined into a PREMO component.

4) *Modelling, Rendering and Interaction*. The MSS component defines concepts of media streams and processing resources that are independent of media content. In the MRI component, these facilities are used to define generic objects for modelling and rendering data, and basic facilities for supporting interaction. To support interoperability, the component defines a hierarchy of abstract primitives for structuring multimedia presentations. These are not sufficient in themselves to build a working presentation, but provide the abstract supertypes from which a set of concrete primitives could be derived.

3. The PREMO Object Model

Although with the emergence of UML[21] there is now some level of consensus on a set of concepts for object oriented modelling, at the implementation level there still remain a number of different approaches, as represented by the range of programming languages that are claimed to support object oriented techniques. These differences vary from the fundamental, such as whether a system is class based, or object-based (using prototypes[44] to define the structure of objects), to finer details, such as the various levels of visibility or accessibility that can be assigned to the components of an object.

Within a development project using an object oriented target language, the choice of object model is effectively made once the target language is chosen. Indeed, the precise details of the available object model may be one criteria by which the language is chosen. In the case of PREMO, however, the situation is rather more complicated. Like the standards that it follows (GKS and PHIGS)[4], PREMO is intended to be independent of any particular programming language. Thus, just as one can obtain a C binding or a FORTRAN binding for GKS, it should be possible to obtain a C++[53] or Ada'95[5] binding for PREMO. The need to provide this flexibility raises a number of difficult technical questions, not the least being whether it should be possible to bind PREMO to a language with no explicit support for object-oriented programming (e.g. FORTRAN). For now, the main point is that if PREMO is to be language independent and described in an object oriented framework, it requires the definition of some object model that to define the concepts from which the remainder of the standard will be constructed.

One of the fundamental issues that had to be decided at an early stage in the project was whether to adopt a "classical" object oriented approach, in which objects are instances of classes that can be arranged in a hierarchy through inheritance[9,50], or opt for a more radical approach based for example on the use of prototypes and delegation. The former is typical of the models that underlie object oriented design methods, and has been in widespread use in the form of languages such as Simula, SmallTalk, and C++. Prototype based approaches have, in contrast, been largely the concern of the research community; there has already been discussion on the value of such approaches in graphics and multimedia[10]. In particular, the use of delegation, and the notion of "trait" objects used for example in the SELF system[54] are attractive from the viewpoint of building highly adaptable and extensible systems. However, technical issues aside, the fact that prototype models are strongly bound to experimental systems, and are not in widespread use, represented a serious barrier to their use within PREMO. The result is that the PREMO object model is based from the outset on a fundamental distinction between objects and classes, which in PREMO are called "object types". The remainder of this section describes other high-level design decisions that affected the content of this component.

3.1 Overview

A PREMO system consists of a collection of objects, each with a local (internal) state, and an interface consisting of a set of operations. Each object is an instance of an object type, which defines the structure of its instances. An object type can be defined as an extension to one or more other object types through inheritance; note that this allows for multiple inheritance. An important property of the model is that objects are never accessed directly. Instead, a PREMO client requests a facility called an "object factory" to generate an object satisfying specific criteria, and if it is able to comply, the factory will return a handle to the new object called an object reference. All subsequent activities involving the object is then done via the reference, for example invoking an operation on the object, or passing the object as a parameter to another operation. This separation of objects (i.e. physical storage) from their references is needed to support the aim of distribution, as an object reference can be used to encode both local address information and the location of a particular object across a network.

3.2 From Language Bindings to Environment Bindings

Although the choice of a class, rather than object-based model is relatively straight forward, a number of further options are rather less clear cut. In particular, the aim of making the standard language independent introduces a tension in the design, between introducing features that offer descriptive or computational power but are specific to a restricted set of languages, or using a simple, less powerful model to describe the standard in the expectation that it will be easier to map the model onto the facilities of a given implementation language. Features that are problematic range from the mundane, for example how (or even whether) objects are copied, through to complex problems such as the management of remote (distributed) objects.

One approach that PREMO employs to prevent overcommitment to a particular object model is to introduce the notion of an environment binding. Previous standards in computer graphics have also been developed using a language independent description, and have been mapped onto a specific implementation language through a language binding, that associates the abstract data types and operations defined in the text of the standard with concrete data types and operation signatures within the target language. Such a binding is still needed for PREMO. However, while some concepts in the standard will be mapped onto language-specific features (for example, object types and operations), other aspects of the model, for example how objects are to be copied, or how remote objects are accessed, are left as facilities to be provided by the *environment* of a PREMO implementation. These facilities may be realised through language constructs, but more generally they may be provided by library packages, or even via the use of other standards. Thus, access to distributed objects within a C++ implementation of PREMO could be realised through a custom-built mechanism, or through a separate standard such as CORBA. In the case of a Java implementation, these two options again exist, but in addition it is possible to use the Java RMI package. By viewing features such as object copy and remote access as requirements on the environment, rather than requirements on the object model, the object model itself is simplified and is consequently easier to map against the provisions of a specific implementation model.

3.3 Object References

It is widely accepted that a fundamental component of object orientation is that each object in a system has an identity that is independent of that object's state. Therefore, two objects that have the same state can nether the less be distinguished. At a very practical level, this corresponds to the use of pointers to reference objects within an implementation. These pointers, or object references, may be implicit or explicit. In the case of SmallTalk or Java, for example, it is not possible to access an object other than through an object reference - this is enforced in the definition of the languages, which provide no constructs for referring to an object other than through pointers. C++ and Ada'95 have a different model. Objects in these languages are defined as generalised records, and a pointer to an object is a well defined type that is quite distinct from the type of the object itself.

As PREMO objects can be distributed, various mechanisms for accessing objects may be used within even a single system. For example, local objects might be referenced via pointers, while remote objects are referenced by some form of extended URL. To avoid confusion or implementation bias, the standard introduces the concept of an object reference as an explicit part of the object model, with the intention that this be bound to whatever means are used within the target language and/or environment to access or refer to specific objects. The approach taken in PREMO combines elements of the explicit and implicit approach. In line with the former, the model defines both the concept of an object, and an object reference. However, the distinction is there to simplify the use of multiple implementation strategies — it is not possible to refer to, or use, an object directly. Instead all access to an object, for example to invoke an operation, must be via an object reference.

3.4 Active Objects

Concurrency is by definition an integral aspect of multimedia presentation, and will certainly be a property of the type of distributed application which PREMO is intended to support. Fundamental to such a model is the idea that several threads of control, or processes, can be active within a system at one time, and that such processes interact through communication events. Here again there is a tension between adopting a simple model based on a particular set of facilities, or a more general model that is harder to use within the standard but is hopefully easier to implement.

On the one hand, there is a natural and appealing parallel between the idea of a process and that of an object. A process is an entity which encapsulates a thread of control and that interacts with its environment through events; an object is an entity that encapsulates state and interacts with its environment through operations. Languages such as Eiffel[46] and Java[24] have built on this view by treating processes (or threads) as particular types of object; in Java for example, an object will be active if it implements the Runnable interface. In contrast, other languages have maintained a separation between these concepts. In Ada'95 for example, processes are realised through a sophisticated task model, quite separate from the notion of task, while in C++ there is as yet, unfortunately, no standard model for dealing with processes.

The PREMO object model assumes that all objects are conceptually active; as we will discuss in section 4.1, the standard does however, for efficiency reasons, define certain types of objects to have trivial activity. What the standard does *not* do is to mandate any particular mechanism through which object activity should be realised. What is required is that each object has the capability to have an internal thread of activity. In parallel with this internal activity, an object may receive requests for an operation to be invoked; these requests arrive at operation receptors. At any time an object can select which requests it is willing to service. The PREMO object model does not completely specify the execution order for operations, for example pending requests may be serviced sequentially or concurrently.

3.5 Operation Dispatching

The delays inherent in remote object access and operation invocation mean that asynchronous operations are a fundamental tool in the development of distributed systems. Synchronous operation calls, in which the caller is suspended until the called operation terminates, are also required. To support multimedia applications, the design of PREMO also allows for a third kind of operation, sampled. A sampled operation is similar to an asynchronous one, in that once the operation has been invoked the caller is able to continue its processing while the request is held in a queue. The difference is that the queue of requests for a sampled operation is effectively a one-place buffer, with any request for the operation overwriting any pending request.

Each PREMO operation is defined as using one of these *operation request modes*. The existence of these modes is one of the more significant differences between the PREMO object model, and that found in most programming languages, or indeed the model defined by the OMG.

3.6 Attributes

One of the positive aspects of object orientation is the emphasis on data hiding and encapsulation — clients of an object should only use the operations in the interface of an object, and should not have access to the internal state. Instead, if access to a variable is required, it should be realised through operations that retrieve and/or set the value of the variable. A number of such state variables appear within PREMO object types, and rather than define explicit operations for manipulating these variables, the standard introduces the concept of an *attribute*. The definition of an attribute looks like that of a variable, however an attribute of an object type is understood as being a shorthand for a pair of operations in the interface of that object type which

set and get the value of an (internal) state variable. An attribute can be declared as read–only, or write–only, meaning that the corresponding 'set' or 'get' operation is not available.

3.7 Non-object Data Types

SmallTalk was for some time presented as the prototypical object oriented programming system, and many of the ideas it pioneered were adopted in subsequent languages and systems. One of its strengths was its simple ontology; everything in the system is presented as an object, even "atomic" data such as numbers and characters. While this view produces a remarkably uniform model, it does have a number of consequences. First, there are a number of general raised by such an approach, including how one interprets the "identity" of numbers, how one relates binary operations on data "objects" to the conventional mathematical view of numbers. Second, there is the issue of efficiency: treating data values as objects implies that operations such as addition are handled by the same run-time dispatch mechanism as other operation calls. Data processing in computer graphics and multimedia often involves a considerable amount of numerical processing with large data sets (geometric structures, digital image formats, etc.) and here the need to use a general dispatch model is clearly an efficiency concern. Finally, while PREMO is intended to be language independent, the most likely targets for a language binding were seen as the family of object-oriented languages, including C++, Ada'95 and Java, in which object-oriented structures have been added to a language in which primitive data are treated as values. For these reasons, PREMO has adopted a model that distinguishes between non-object data types, such as integers and characters, and object types.

4. The Foundation Component

The implementation of most multimedia systems involves a number of fundamental concerns: control and management of progression through media content, synchronisation between activities, time, and coordination. Existing standards provide specific facilities for some of these tasks, while for others an implementor may need to utilise a general library (for example, for synchronisation) or develop ad-hoc solutions. Without mandating any specific approach to these general concerns, the PREMO Foundation component provides a set of general-purpose object and data types that can be used by a developer to implement the functionality mentioned above. A developer can either use these facilities "raw", to create a customised architecture, or they can be used via the higher level object types and services provided by Parts 3 and 4 of PREMO which are described later in this report. As the Foundation component is essentially a toolkit, the remainder of this section describes its main provisions in terms of the principle media system requirements that are supported.

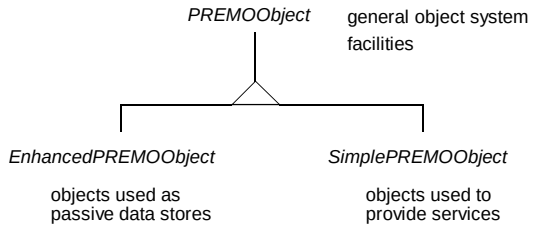

Figure 1: *Two kinds of object type in the PREMO hierarchy*

4.1 Structures, Services, and Types

The requirement that, conceptually, all PREMO objects are active means that in principle all access to an object must allow for the possibility that the object will have its own thread of control. Depending on the implementation platform, this assumption may impose a high overhead on the cost of accessing components of objects; such access will for example have to pass through the operation receptor and request handling infrastructure. For some aspects of media processing, these overheads are unavoidable; they are needed to support the provision of distributed services across a media network. However, in a typical media application, not all objects will necessarily be used as "active" entities that provide services. One use of objects is as data encapsulators, similar to the use of records (structures) in languages such as Ada and C. There is clearly a trade-off here, between the elegance and simplicity of a homogeneous object model on the one hand, and the practical problems involved in storing and processing large multimedia datasets on the other. For example, a visualisation application may need to operate on a volume data set containing in the order of 10^6 vectors. If each vector is represented as an object, the overhead in processing this dataset will become significant.

PREMO has adopted an approach that retains a fundamentally simple object model while allowing implementors to avoid the overhead of the full operation request system where it is not required. The approach is based on the top-level organisation of the PREMO object type hierarchy shown in Figure 1. All object types in PREMO are subtypes of `PREMOObject`, in which fundamental object behaviour, such as the ability of each object to return information about its type, is defined. Below this the hierarchy bifurcates. `SimplePREMOObject` serves as a supertype for those object types that represent data encapsulators. Such object types are referred to as *structures*. `EnhancedPREMOObject` is the abstract supertype for those object types that provide services, and which therefore incur the overhead of the operation dispatch mechanism. This separation is further formalised through the profiles that are defined in each component to identify those object and non-object

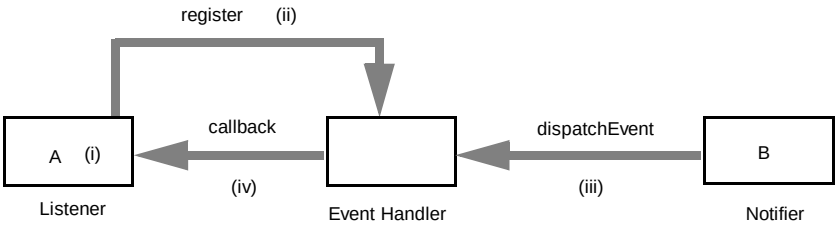

Figure 2: *Overview of Event Management*

types that should be made available to clients of the component. Each profile consists of lists of object types, either under the category "provides type", or "provides service". Only an object type that inherits from `EnhancedPREMOObject` is allowed to appear in the "provides service" clause, and it is only objects of these types that a client can expect to interact with through operation dispatching.

4.2 Inter-Object Communication

Although ultimately all interaction between objects within a PREMO system takes place via operation requests, this is not a particularly useful way of representing communication and cooperation within a distributed system. In the case of multimedia, two models are now well known:

- Stream based models, in which information related to processing is sent on communication channels or media streams between objects; these may be the same streams that are used to carry media data.
- Event based models, in which there is conceptually a separate mechanism by which specific operations in the interface of a collection of objects can be invoked in response to a specific situation in one object.

PREMO does define media streams that in principle can be used to support communication between objects; these are described in section 5.1. However, streams are a comparatively "heavyweight" facility, intended primarily to manage the transport of media data. Consequently, the foundation component defines a collection of object types that provide an event management facility specifically for inter-object communication.

The event mechanism is based on callbacks and event handlers. Callbacks are now widely used in the graphics and user interface management communities, having been popularised through systems such as the X library, GL, and more recently the Java AWT[20]. Essentially, a callback is just an operation in the interface of an object that will be invoked by some other entity within a system in response to an event. A typical low level example is an operation in a user interface object that a run time system will invoke to notify the object of a mouse-button being pressed or released. Callbacks often take parameters that carry information about the event that has taken place. Since the event management facilities in PREMO are used to address a range of concerns, it was sensible to introduce a systematic approach for carrying event information. To this end, an `Event` object types is defined to carry such information, specifically the name of the event, a reference to the source of the event, and additional data specific to the event.

Figure 2 provides an overview of the approach. Objects that are interested in a particular event, (object A in the figure) must (i) be of a type that inherits from the `Callback` object type, which provides a general `callback` operation, and (ii) must register their interest with an instance of the `EventHandler` object type. When an object (B in the figure) wants to notify the system that an event has occurred, it invokes the dispatchEvent operation on an event handler (iii), and all objects that have registered with that handler to be notified of the event will have their callback operation invoked (iv). Chains of event handlers can be set, as the `EventHandler` object type itself inherits from `Callback`, and defines its `callback` operation to have the same effect as `dispatchEvent`. Thus, object A in the figure could be an event handler that subsequently distributes the event received by the callback to further objects.

In the case of a basic event handler, objects are only required to register with the handler if they should to be notified of a particular event; any object in the system can signal to the handler that such an event has occurred. A specialised form of event handler, called an `ANDSynchronizationPoint`, provides a richer service. Objects not only register to be notified of an event, they also register as *notifiers* for a particular kind of event. When appropriate, a notifier signals the event handler as usual, however, the event handler postpones the notification of objects interested in the event until *all* objects that have registered as notifiers have signalled the event to the handler. This object type has a role in the general synchronization facilities of PREMO, which are discussed next.

4.3 Synchronization

Like event handling, synchronization requirements in PREMO span a range of levels. At the level of data streams, fine-grained synchronization may be used to implement quality of service requirements, for example maintaining an adequate alignment between related audio and visual content. At a higher level, a multimedia presentation will typically

consist of a collection of components, some of which may be presented in parallel. In addition to any fine level of synchronization between such strands, synchronization between key milestones (such as the start/end of component strands) may be required. Beyond direct control of media presentation, synchronization may also be needed within the control structure that manages the overall media system.

Synchronization models and mechanisms have been widely reported in the multimedia literature, see for example [7, 12, 25] and [42]. Synchronization in PREMO is supported at two levels - in terms of *events*, and in terms of *time*. Event based synchronization has obvious application in dealing with the processing of structured presentations composed of more primitive media streams, however it also has a role in synchronizing the presentation of the data within a stream, where significant milestones are defined by the content of the stream, rather than its absolute position. An example of this is the synchronization of ultrasound or other medical scan data, where milestones defined by physiological events need to be aligned. Such an example is described in more detail in [45]. Time based synchronization is better known, and involves ensuring that multiple activities reach particular milestones at times specified relative to each activity.

The event and time-based approaches are both supported by a common framework, the `Synchronizable` object type, which PREMO uses as the basis for representing, monitoring and controlling the transmission and processing of media data. Although the interface to this object type is large, it is based around three main ideas:

1) An internal progression space, which acts as a coordinate system for defining the concept of location within some media stream or content. `Synchronizable` objects do not themselves carry media data, but instead are inherited by object types which are involved in the transport and processing of such data. Conceptually, the progression space represents the temporal extent of some media representation, and progress through the progression space is made during processing of that media.
2) Progression is controlled by a finite state machine; this is actually achieved by having `Synchronizable` inherit from another object type, called a `Controller`, which is also defined in this component. Controllers are essentially finite state machines that can raise events on entry to, exit from, and transitions between states; their details are not of concern here. It suffices to say that a `Synchronizable` object can be in one of four states: stopped, playing, paused, and waiting. Conceptually, when an object is in the playing state, progress is being made through its progression space. Transitions between the states occur as a result of operation invocation, and also through interaction with reference points, which are discussed below. A number of attributes define the parameters that affect how progress is made, for example, the direction of progression.
3) *Reference points* can be placed along the progression space, either individually, or repeated with a given period. Each reference point consists of an event, a reference to an event handler, and a special boolean 'wait' flag. When a reference point is encountered during progression, the event is sent to event handler specified. The wait flag indicates whether progression should be suspended at this point, and if has the value *true*, the `Synchronizable` object is placed into the 'waiting' state, where it will remain until the `resume` operation in its interface is invoked.

Reference points and the 'wait' flag are intended to be used in conjunction with other PREMO facilities to implement synchronization schemes. For example, by combining reference points with the `ANDSynchronization` object type described in section 4.2, processing of one part of a presentation can be suspended once a particular milestone has been reached until all other `Synchronizable` objects that involved in implementing the presentation have reached related milestones. An example of such a scheme is shown in Figure 3.

4.4 Time

Media such as sound, video and animation is fundamentally grounded in time, and to describe and control the presentation of such media it is necessary to have some means of representing and measuring time. The question of how time should be represented (for example, as a continuum, or discretized) has been the subject of much philosophical debate, and is a non-trivial concern in areas such as real-time systems modelling and verification. PREMO adopts a pragmatic approach, in which all representations of time are based on 'ticks' produced by some clock. The granularity of a 'tick' is not fixed by the standard, but rather depends on the particular clock used.

PREMO introduces object types to represent abstract clocks, a subtype of clocks representing 'real time' system clocks, and a resetable timer. All clocks are derived from the abstract object type `Clock`, and specify a 'tick unit', which is the unit (for example, seconds) represented by each tick, and a measure of the accuracy of the clock. An actual measure of time is obtained by invoking the `inquireTick` operation in the interface - however, it is up to subtypes of `Clock` to attach a meaning to the number of ticks that are returned. Thus an object of type `SysClock` returns the number of ticks (to its level of accuracy) since the start of the defined PREMO era. The object type `Timer` defines a start/stop timer by extending the interface of `Clock` with operations for stopping, starting, and pausing the progression of time. For objects of this type, the number of ticks returned by `inquireTick` are the number of ticks that have elapsed, while the object has been in its running state, since it was started (i.e. ignoring time spent in the pause state).

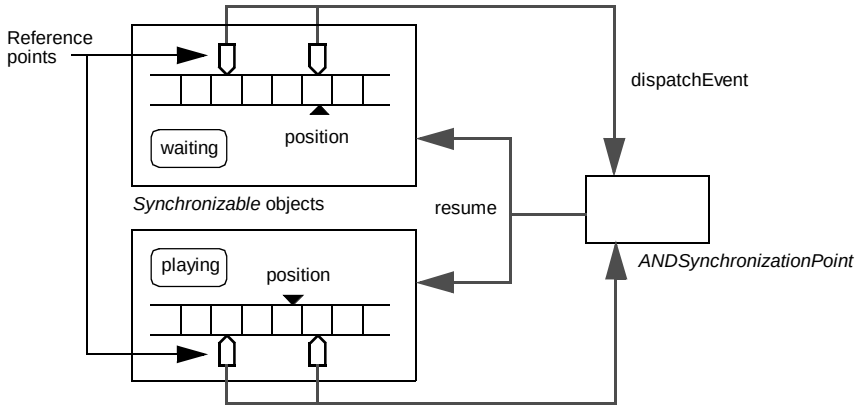

Figure 3: *Example of a Synchronization Scheme*

The link between time, and the event-based synchronization model described in section 4.3, is defined by the object type `TimeSynchronizable`, which couples the behaviour of a `Synchronizable` object with that of a `Timer` object, thus making it possible to measure and control the `speed` of progression through the internal span of a synchronizable object. The interface of `TimeSynchronizable` allows reference points to be placed against positions on the progression space specified in terms of time, for example, placing a reference point 30 seconds from the start of a video sequence. Obviously, the actual point in the video content at which this reference point will be reached will depend on the speed at which progression is being made through the video. Two subtypes of `TimeSynchronizable` are identified in the standard. A `TimeSlave` object is one for which the rate of progression can be 'slaved' to the rate of progression of some other time-synchronizable object. A `TimeLine` object can be used to set reference points against milestones in real time.

4.5 Property Management

In the PREMO object model described in section 3., the attributes and operations of a type are defined statically, when the object type itself is defined. Once an instance of a type is created, the interface of the object is fixed. This "static" approach to object structure has clear benefits, not the least being support for compile-time checking that can reduce the likelihood of programmer error. However, as we mentioned earlier in the report, more dynamic object models are also available, and their potential use in graphics and multimedia has been noted[10]. Features such as delegation, or on a more modest level, the ability to alter the interface of an object at run time (as adopted in Python[55] for example) would play a useful role in the implementation of constraint management[19] for example. However, the experience of the MADE project[31] was that implementing such features within a class-based, 'static' object models was a significant problem.

PREMO introduces the concept of object *properties* as a compromise between a purely static model and the facilities offered by dynamic models. A property is a pair, consisting of a key (i.e. a string) and a sequence of values. Each value in the sequence can come from any PREMO non-object data type, and as these include object references, an object property is essentially a dynamically typed variable. The `EnhancedPREMOObject` type introduces operations to define, delete, and inquire values associated with a given property key. Properties can be used to implement various naming mechanisms, store information on the location of the object in a network, create annotations on object instances, and underpin a framework for inter-object negotiation. In support of this, the standard stipulates that objects of certain types will have a property with a given key, and possibly particular values. However clients of any object whose type inherits from `EnhancedPREMOObject` can attach new properties at any time. Properties may also be declared as 'retrieve only'.

The basic facilities provided by `EnhancedPREMOObject` are developed by two further object types, `PropertyInquiry` and `PropertyConstraint`. In the first of these types, each property key can be associated with a corresponding 'native property value', which describes the range of values (capabilities) that the corresponding property can take on. This can be viewed as a form of dynamic typing. The `PropertyConstraint` type extends this model by ensuring that a value added to a property lies in the corresponding native property value, if this exists. This object type also introduces a number of 'meta' properties, for example, the key '`dynamicPropertyListK`' is associated with a list of values representing the keys of certain properties. The operations `bind` and `unbind` allow keys

to be added to and removed from the values of `dynamicPropertyListK`. Only while a property's key appears under this property can the corresponding value be changed.

4.6 Object *Factories*

One specific use of properties is in the creation of objects. In section 3.2 we noted that PREMO relies on its environment to provide certain fundamental services, and the creation of objects is one such service. In most object-oriented programming languages, creation is a comparatively simple mechanism, handled either by a language construct (e.g. the '`new`' operator of Java[20]) or through some meta-object system, in which classes are themselves objects and can respond to message requesting object creation, as in SmallTalk[23]. This situation is complicated in PREMO by the use of properties to describe features of objects. For example, a PREMO system may define a GIF decoder as an object type that has a property, say "`GIFversionK`" which can be set to either the value '87a' or '89a' representing the two versions of the specification that are in widespread use. Alternatively a system may offer two types of GIF decoder object, one for each version of the standard, in which the property "`GIFversionK`" is fixed. There is thus interaction between the structure of the type hierarchy, and the use of property keys.

In fact, from the viewpoint of a PREMO client, the specific type of an object will often be uninteresting. What is important is (i) that the object is a member of a subtype of a given type, and/or (ii), that the properties of an object satisfy a given constraint. In the example above, what the client may really want is a device that can decode JPEG v87, and the client is not concerned whether this device is an instance of an object type specifically for this version, or is an instance of a more general object type that can be configured to the given requirement.

In order to hide these issues, and provide a uniform interface for object creation, the foundation component of PREMO introduces the concept of an *object factory*. A factory is itself an instance of the `GenericFactory` object type that provides a single operation, `createObject`. This operation accepts an object type, and a set of constraints in the form of a sequence of key / permitted value pairs, and (if possible) returns a reference to an object that is an instance of the given type or a subtype, and whose properties satisfy the constraint.

Factories are themselves objects, and a PREMO system provides a factory finder object that is able to locate a factory capable of producing an object that will meet given constraints.

5. The Multimedia Systems Services Component

Multimedia systems typically integrate a variety of logical and physical devices. For example input and output might involve devices such as video cameras, microphones, and a sophisticated speaker system. Processing in turn may involve logical devices such as data encoders/decoders, media synthesizers (e.g. a graphics renderer), and a video mixer. The data produced an consumed by these devices takes a variety of forms, for example a discretised audio signal, a sequence of video frames, or a discrete graphics model. In turn, these forms can be encoded in a variety of formats (ALAW and ULAW for audio, for example). Finally, different protocols may be available to communicate such data, depending on the source and destination hardware, and on the available network infrastructure.

As explained in the introduction, PREMO does not aim to define new standards for the encoding or transport of media data. Rather, it seeks to provide a set of facilities that abstract away from the details of low level system services, instead providing an application developer with a uniform high level view of media processing. To this end, the multimedia systems services (MSS) component of PREMO defines the infrastructure for creating and maintaining a network of heterogeneous processing elements for media data. This includes object types for describing generic resources, devices, and facilities for organising a collection of such components into higher level units with a single interface. MSS encompasses mechanisms by which media processors can advertise their properties for network construction, can be interconnected and controlled, and can be configured dynamically to match the needs of a network while in operation.

MSS was originally defined by the Interactive Multimedia Association[33], a large consortium of industrial vendors and developers. IMA were aware of the work within SC24 on the development of PREMO, and donated the MSS framework to the Committee. It was subsequently adopted by SC24 as the basis of a distinct PREMO component. During the development of the standard, several of the main provisions of MSS were refined and integrated with facilities from the Foundation component.

5.1 The Paradigm of Media Networks

In order to abstract away from the details of specific media types, media processing elements are viewed as "black boxes" that can be interconnected through a high-level interface to construct a network of such elements appropriate for a given application. At this level, a PREMO application using MSS resembles a dataflow network, where the nodes correspond to media processors, and the data streams carry media content. The adoption of a dataflow-oriented view of media system architecture is not peculiar to PREMO. It has ap-

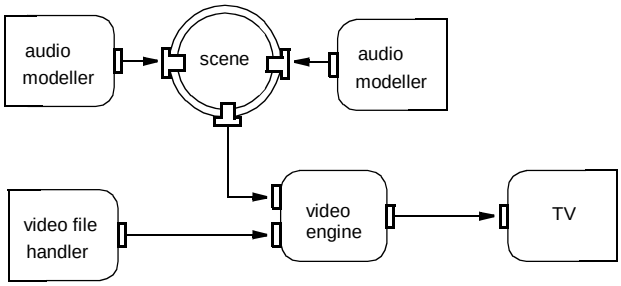

Figure 4: *Simple Multimedia Network*

peared in published approaches to multimedia systems (for example, [26]), and is also increasingly used in "plug and play" applications environments, for example for visualisation[51].

Figure 4 contains an example of a small network. It represents a video engine combining input from a local file (for example, in MPEG) with audio clips stored as media primitives within a remote database (scene). The audio primitives in the scene are constructed by a number of audio modellers (MIDI devices, or waveform editors, for example). The combined audio/video output is presented on a TV device.

The devices in the figure are all instances or subtypes of specialised object types defined in the fourth component of PREMO, and which is discussed in section 6. What makes the construction and operation of such a network possible are that all of the object types involved extend the virtual device and resource concepts defined in Part 3. This allows the devices to be connected together, and subsequently to exchange media data along the streams shown. In the remainder of this section we describe the principle concepts and types that the MSS component provides for the creation of such networks.

5.2 Virtual Resources

A high level view of a media network is of a collection of resources that cooperate in the task of creating and/or processing media. These resources encompass physical devices (such as cameras or mixing suites), software processes such as graphics renderers and audio filters, as well as supporting infrastructure such as connections and software for managing collections of lower-level resources. What is fundamental to this view is, first, that a resource is something that has to be acquired for a task, and second, that many of what we consider to be resources are inherently configurable. For example, an audio mixer may involve both hardware and software elements, access to which must be acquired before the mixer can be installed in a processing network. In fact, a number of mixers might potentially be available, differing in characteristics such as the number of channels that they can accept, the kind of audio formats that can be processed, and the type of filters that can be applied.

The property description and management facilities described in section 4.5 form the basis for realising this model. The characteristics of a particular resource are described by properties; some of these can be set by a client of the resource, often to one of a set of possible values defined as the native property values for the given key. Other properties, representing immutable aspects of a particular resource (for example the number of input channels to the audio mixer) are read only, but still play an important role in establishing a media network.

The fundamental operation of a PREMO resource is defined by the `VirtualResource` object type. Each resource (or more generally, each subtype of `VirtualResource`) defines a set of property keys and values that are relevant to the description and control of the resource. In addition, each resource encapsulates a number of *configuration objects*. These objects store data about the resource to which they are associated, and this information is used by other objects, for example in providing communication services or quality of service management. The MSS component defines three types of configuration object explicitly; each inherits from `PropertyConstraint`:

- `Format` objects represent the details of a media format, for example the organisation of a bitstream;
- `MultimediaStreamProtocol` objects provides information about how media data is conveyed between processing nodes; and
- `QoSDescriptor` objects capture quality of service characteristics, such as the level of guaranteed service, and bounds on delay and jitter.

It must be emphasised that the PREMO standard does not describe all details of these object types; for example the specifics of particular media stream formats. The purpose of these object types is to provide placeholders and hooks that can be specialised or used as required within a particular implementation environment. What the `VirtualResource` object type does provide are operations for

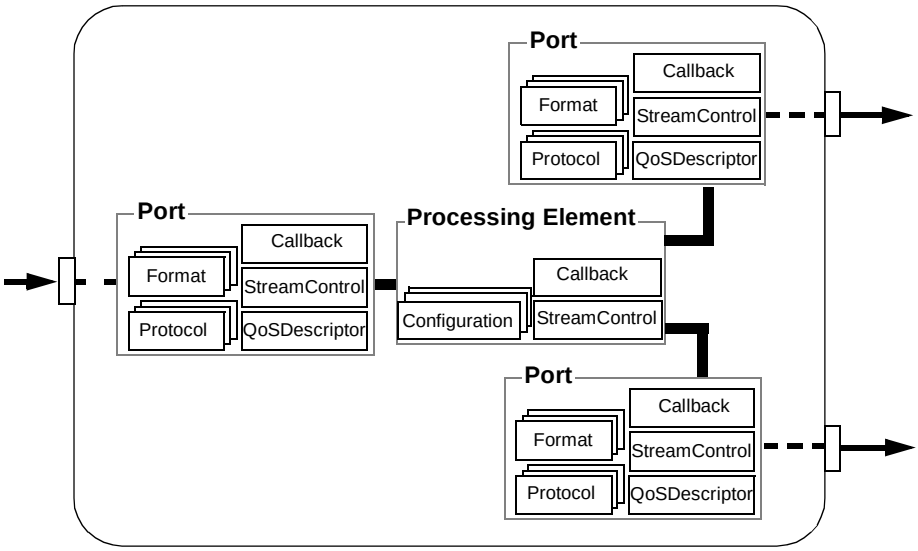

Figure 5: *The Structure of a VirtualDevice Object*

accessing particular configuration objects using semantic names (strings), acquiring the physical and software resources managed by the object, and validating whether the configuration requirements expressed by the combination of properties and configuration objects can be satisfied. Each resource is also associated with a stream control object, the purpose of which is described next.

5.3 Stream Control

Virtual resources are involved in the production and transport of media data. Control and monitoring of media streams is provided in PREMO by the `StreamControl` object type. Different kinds of resource will have different views on media streams, ranging from a low-level signal oriented view, through levels that abstract signals into packets, and packets into media samples or chunks. This range of views is accommodated by basing stream control on the `TimeSynchronizable` object type discussed in section 4.3; by inheriting from this type, stream control can be defined with respect to the coordinate system of the progression space, or (relative) time. To facilitate fine control over progress, the `StreamControl` object type refines the state machine inherited from `Synchronizable` by introducing states that allow media content to be drained (discarded) or buffered and subsequently released. These facilities, along with the ability to place reference points along the progression space connected to the event handling system, are intended, for example, for use as part of an overall quality of service management strategy. A further object type, `SyncStreamControl`, allows progression through its stream to be synchronized (slaved) explicitly with the progression of some other object that is derived from the `Synchronizable` type.

Virtual resource objects have an associated `StreamControl` object that allows, where applicable, monitoring and control of the end-to-end processing carried out by that resource. Stream control objects are also a feature of an important kind of resource, the virtual device.

5.4 Virtual Devices

The “nodes” in the dataflow network shown in Figure 4 are defined to be so called `VirtualDevice` objects that form the basic building block for interaction and processing capabilities within PREMO. The anatomy of a virtual device is shown in Figure 5.

The principle features that the `VirtualDevice` object type adds to a resource is the presence of “openings”, called ports, which act as input or output gateways for the virtual device, and the concept of a “processing element”. Ports are the means by which data can be passed from one device to another. A port is not itself an object, rather, it an identifier or handle that is used to reference a particular opening, and through the interface of a virtual device, access and control information about that opening. Like a resource (and hence a device itself), each port is associated with a collection of configuration objects that characterise the flow of data through the port. More specifically, each port has associated `QoSDescriptor`, `Format`, and `MultimediaStreamProtocol` objects. The client can set the properties of these objects, and can refer to them when configuring a

network. These configuration objects are combined into a port configuration object, which also contains a reference to an event handler dedicated to that port, and a `SyncStreamControl` object that controls and monitors the transfer of media data via the port. Just as with `VirtualResource`, an operation is provided by `VirtualDevice` to validate the requirements captured by the configuration of each port.

The "processing element", shown within the virtual device in Figure 5, is a conceptual, rather than concrete, component. That is, there is no object type for a processing element, nor does the `VirtualDevice` introduce variables or operations to implement it. The only part of a virtual device that directly relates to processing is the end-to-end stream control and configuration objects inherited from `VirtualResource`. One of the tasks to be addressed in implementing the `VirtualDevice` type is to decide how the transfer of media data within the device is to be effected. By *not* being prescriptive about this aspect, the PREMO designers have sought to better accommodate the wide range of existing media processing software that might be "wrapped" within a subtype of `VirtualDevice` for use in a PREMO–based network.

5.5 Virtual Connections

The lines in Figure 5 entering and leaving device ports represent the flow of media along streams. PREMO itself does not define a "Stream" object type, since much of the detail here depends both on the underlying network technology, and the context of the connection (i.e. whether two devices are on the same host, local network, etc.). Streams however are established and maintained by objects derived from another subtype of `VirtualResource`, the `VirtualConnection` type. As a resource, a virtual connection object contains a stream control object that represents the end-to-end flow of media data along the stream controlled by the connection. A subtype of VirtualConnection supports multicasting, with operations to attach and detach a device/port combination to and from the connection. All connections are unidirectional.

If the underlying devices are located on the same hardware, a connection may be realised by directly linking the input and output ports of the associated devices. More generally, the devices will be on distinct, possibly remote, machines and using different local facilities for inter–object communication. In such cases a virtual connection may need to create a *virtual connection adapter*, that provides appropriate interfaces to the end-parties while managing any recoding or translation of raw data required. Connection adapters exist only as concepts within the PREMO standard; they do not correspond to any particular object type, and in fact their implementation will in general require a collection of objects to manage the transfer between the different protocols.

5.6 Higher-Levels of Organization: Groups and Logical Devices

Even the simplest non–trivial media network, involving two devices with a single connection between them, involves a significant number of objects: the devices themselves, the connection, the connection adapter (if needed), event handlers for the ports and devices, and possibly supporting objects to, for example, monitor quality of service. For a realistic application, the number of objects is significantly greater, and the problem of tracking which particular groups of objects are relevant to any given part of the network becomes significant.

To prevent organizational anarchy, it is often convenient for clients to interact with a single object that represents each "significant component" of a network. PREMO provides a `Group` object type to support management of a collection of devices and connections. `Group` objects are resource objects which control a number of other virtual resources (in particular devices and connections), and their respective network. By default, the constituent devices remain hidden to the external client; instead, groups provide a single entry point for stream control, as well as other services. By using the basic group interface, the client does not have to know about the interfaces of these constituent devices. As `Group` inherits from `VirtualResource`, each group is itself a resource, and consequently, the configuration of group components can be validated, and the components themselves acquired, via the one group interface, rather than individually. As a group is itself a resource, a group can itself be a member of a further group.

Although groups can be organized into hierarchies, it is important to remember that a group is not a device; it has no ports of its own. Instead, a client using a number of groups is responsible for ensuring that, where necessary, components of distinct groups are connected. A specialised form of group, called `LogicalDevice`, combines the central resource management capabilities of a group with the processing model of a virtual device. Resources are added to and managed by a logical device in the same was as a group, but the client of a logical device can also dynamically define ports on the interface of the device. When defined, each port on the logical device is associated explicitly with a port on a device that it manages. A logical device thus acquires input and output ports, and can be built into a network in the same way as other devices.

5.7 Working in Unison

At this point we have described the main features of the multimedia systems services component. Given the importance of this component to the aim underlying PREMO, of supporting the development of distributed multimedia systems, it is useful to summarise the roles played by the various object types described here. We do so by outlining the steps involved in setting up a network using MSS.

1) Assuming that the client has a suitable factory, it first uses the factory to create the various objects (devices, connections, and other resources) that make up the network. Part of the specification for the objects given to the factory may involve constraints on properties of the objects, for example a device is able to receive data using a particular format.

2) The connections are defined by sending each connection object a `Connect` request, specifying the source and destination device/port combinations.

3) The client may create a group object, and then add all of the resources to the group by sending the request `addResourceGraph` to the group. At this point the structure of the network has been fixed, but no actual resources (e.g. bandwidth) have been allocated to it.

4) Using the `acquireResource` request of the group object, the client attempts to allocate the resources needed for each of the objects in the network. Inability of the underlying system to meet this request will result in an exception which the client can detect. In this situation it may modify its requirements by changing properties of any of the objects within the group, for example by settling for a less reliable connection.

5) Once the resources have been allocated, the client can start the transport of media data through the network by accessing the `StreamControl` object of the group.

6. The Modelling, Rendering, and Interaction Component

A feature of the MSS component is that its provisions are independent of the data processed by the devices and resources within a network. Thus the same approach can be used for setting up a video editing system as for setting up a virtual reality modelling and rendering environment. The fourth component of PREMO describes general facilities for the modelling and presentation of, and interaction with, multidimensional data that utilises multiple media in an integrated way. That is, the data may be composed of entities that can be rendered using graphics, sound, video or other media, and which may be interrelated through both spatial coordinates and time.

The MRI component is interesting for two reasons. It is the point within the PREMO standard where the actual structure and content of media data becomes significant. It is also the point at which 'traditional' computer graphics, i.e. modelling and rendering of synthetic scenes, is integrated into the broader concerns of multimedia. This integration within an object-oriented framework highlights a significant design issue regarding the implementation of graphics (and for that matter, other media) processing, which we discuss in the first of four sub-sections.

The actual description of the MRI component ranges over three concerns, which are each covered by a separate heading thereafter. Section 6.2 concerns is the design of a hierarchy of modelling primitives for characterising multimedia presentation. Section 6.3 deals with the collection of devices that extend the `VirtualDevice` type of the MSS component to allow modelling, rendering and interaction to take place within a media network of the kind described in section 5.1. Section 6.4 focuses on a particular device, the `Coordinator`, that plays a key role in mapping presentation requirements of media streams against the devices that are available for processing media.

6.1 Object-oriented Rendering

A fundamental question that must be addressed within any object oriented graphics or multimedia system concerns the allocation of fundamental behaviour, such as transformations and rendering, to object types defining media content within an API. Two quite distinct approaches emerge. The first is to attach behaviour to the object types that are affected by that behaviour. For example, geometric objects and other kinds of presentable media data can be defined with a 'render' method, with the interpretation that such an object can be requested to produce a rendering of itself. Such an approach can be extended to collections of presentable objects, and fits well with the concept of an object as a container for data along with the operations that manipulate that data. The second approach is to define objects whose principle purpose is to act as information processors, and which receive the data that they operate on as parameters to operation requests or through some other communication mechanism. In this case, a 'renderer' object would receive presentable objects as input through some interface, and produce a rendering of those objects via some output mechanism. From the discussion in section 5.4 it may already be clear that PREMO has adopted for the second option. Separating operations (in the form of devices) from the data that they manipulate may appear to violate a central tenant of object–oriented design. However, it has two important benefits for PREMO.

1) First, a direct and desired consequence of a distributed model is that one model or data set may be rendered by several processes working in parallel at various locations. It is difficult to see how this can be realised efficiently in an architecture in which each media object renders or processes itself. Either such objects must be able to support multiple concurrent threads internally, or any object that is to be rendered must first be copied. In contrast, treating renderers as a form of processing device means that multiple renderers can be created (relatively) easily to operate on a given database of objects representing media data (see for example Figure 4). Such a database can either be shared by several renderers, or there may be several copies of the data. Strategies for managing the distribution, update,

and access control of data within such a system are well known, and thus this approach is rather more practical and flexible than the alternative.

2) It supports an approach to application development based on interconnecting a number of processing devices — irrespective of whether those devices are operating on continuous media such as video, or a series of discrete data sets within a rendering pipeline. Once such a network has been defined, it can be used for a variety of data sets or models, and can be readily modified. In contrast, in an architecture where (for example) graphics objects render themselves, the control of processing and flow of data is encoded within specific operations, making it difficult to develop an application that can be modified or extended without wholesale reprogramming of those operations.

By opting for a model in which media data is essentially passive, while media processors are active objects that provide services, PREMO aims to provide a uniform, integrated treatment of both digital media and synthetic graphics.

6.2 Primitives

The potential domains of application for a system such as PREMO are diverse. When considering the design of a component for modelling and rendering, this raises the difficult problem of identifying an appropriate set of 'media primitives' — or indeed, whether to include any model of primitives at all. Two directions initially appear feasible when considering how primitives for modelling and rendering could be supported in a system like PREMO. First, it would be possible to take an existing set of primitives from an established system, for example the node set provided by Open Inventor[56], and adopt these to the needs of PREMO, possibly through some further extensions. The problem here is in finding a set of primitives suitable for the range of applications addressed by PREMO, and then deciding on what, if any, extensions to include. The second approach is to derive some minimal framework of elementary primitives from which those used in practice can be derived by composition.

Although an interesting research problem, both this and the first approach are biased towards a model in which PREMO devices for modelling and rendering would effectively be implementing a new standard for graphics primitives. It is simply unrealistic today, given the investment in graphics and media technologies, to expect industries to adopt a new standard for media data. Instead, the philosophy underlying PREMO is to view the standard as a framework for supporting the integration of different modelling and rendering technologies, with their own models of media data, within a heterogeneous distributed system. This has already been reflected in the discussion on virtual devices, where we noted that the virtual device specification does not mandate any specific strategy for implementing the processing element, thus allowing existing media processors to be accommodated.

In this context, the role of primitives is rather different from their role in a detailed standard such as PHIGS[32]. PREMO clearly cannot attempt to describe a closed set of primitives for modelling and rendering. Instead, it defines a general, extensible framework that provides a common basis for deriving primitive sets appropriate to specific applications or renderer technologies. Graphics modellers, for example, may use specific representations such as constructive solid geometry, NURBS surfaces, particle systems etc. Audio modellers may use primitives that represented captured waveforms, or raw MIDI data for synthesis. The aim of the primitive hierarchy defined in this part is to provide a minimal common vocabulary of structures that can be extended as needed, and which can be used within the property and negotiation mechanisms of PREMO as a basis for devices involved in modelling and rendering to identify their capabilities for use in a network. The seven categories of primitive defined in PREMO are:

1) `Captured` primitives. These allow the import and export of data encoded in some format defined externally to PREMO, for example MPEG[40].

2) `Form` primitives. Here the appearance of the primitive is constructed by some renderer or more general media engine. These include geometric primitives (polylines, curves etc.), as well as audio primitives for speech and music, etc.

3) `Wrapper` primitives allow an arbitrary PREMO value to be carried as a primitive, for example for use in returning the measure of an input device.

4) `Modifier` primitives alter the presentation of forms, for example visual primitives encompass shading, colour, texture and material properties that affect (for example) the appearance of geometric primitives.

5) `Reference` primitives enable the sharing and reuse of clusters of primitives via names that can be defined within structures.

6) Forms and modifiers are combined within `Structured` primitives. An `Aggregate` is a subtype of `Structured` which contains a set of primitives, where some members of the set may be interpreted in application dependent ways; it is thus up to an application subtyping from `Aggregate` to impose a specific interpretation on such combinations. Of particular importance, given that PREMO is concerned with multimedia presentation, is the `TimeComposite` primitive and its subtypes which allow a time-based presentation to be defined by composing simpler fragments. Subtypes of `TimeComposite` provide for sequential and parallel composition, as well as choice between alternative presentations as determined by the

behaviour of a state machine; these primitives are similar to provisions found in the HyperODA standard[2]. Additional control over timing is achieved via temporal modifiers, and subtypes of `TimeComposite` define events that can be used within the PREMO event handling system to monitor the progress of presentation.

7) `Tracer` primitives carry an event. This event can be detected at the port of a device configured to use `MRI_Format`, and will be dispatched to the event handler associated with the port. This facility is used for coarse-level synchronization.

6.3 Modelling and Rendering Devices

The MRI component derives a number of object types from the `VirtualDevice` type of the MSS component, as described in section 5.4. As in MSS, these do not represent concrete devices. They instead define the interface that a device must offer in able to provide certain kinds of service within a PREMO system, and in the case of Part 4, with primitives derived from the hierarchy described above. The device network shown in Figure 4 incorporates a number of devices, the types of which would inherit from MRI object types. The MRI component defines a subtype of `VirtualDevice` for use as the base type for deriving devices for modelling, rendering and interaction. The so-called `MRI_Device` object type is required to support a format at its input and/or output ports that allows MRI primitives to be transmitted and received. Such a device is also required to define properties setting out which primitives it can accept, and some measure of the efficiency with which it can process primitives. In the standard, the following specialisations of `MRI_Device` are defined:

1) `Modeller` and `Renderer` guarantee to provide an output or (respectively) input port that accepts `MRI_Format` streams for carrying primitives. The devices also contain properties that characterise their ability to process primitives.

2) A `MediaEngine` is a device that can act both as a `Modeller` and a `Renderer`, i.e. a device that can transform one or more streams of primitives into new streams.

3) The `Scene` object type defines a database that can be used to store primitives produced and/or accessed by other devices within a network. It is assumed, for example, that multiple devices may have concurrent read access to specific primitives, but the exact form of concurrency control is not specified. The interface of the device allows requests for access to be granted or denied depending on the policies adopted.

4) Two devices are introduced to support interaction. The `InputDevice` object type (a mouse would be a concrete example) supports interaction in either sampled, request or event mode through the stream and event handling facilities defined in other parts of PREMO, while the `Router` object type allows streams of data to be directed based on an underlying state machine.

When accessing primitives stored in a scene, or coordinating the processing of multiple media streams, it is necessary to be able to determine when a particular stream has been fully processed (or received, in the case of database access). This task is supported by the `Tracer` primitive, which carries a reference to an `Event`. Whenever such a primitive is encountered at the port of a device that is a subtype of `MRI_Device`, the event carried by the tracer will be dispatched to an event handler associated with the port. In this way, other objects that need to be aware of the progress of media processing can register interest in such events and be updated of processing activity.

6.4 Coordination

By using the primitives derived from the hierarchy described in section 6.2, an essentially declarative description of a multimedia presentation can be defined. Typically however, at some point this presentation will need to be processed or presented, and during this activity the internal structure of the presentation, for example as a collection of media data to be presented in parallel, becomes important. If a media network contains a device that can process such structures directly, the problem is solved. However, it is also possible that the presentation of a structured media primitive will require the services of multiple devices, whose activities must then be coordinated to reflect both coarse synchronization constraints, as well as quality of service requirements, inherent in the declarative model.

The MRI component defines a subtype of `MRI_Device` called a `Coordinator`. Such a device encapsulates a number of other media devices (derived from `VirtualDevice`), each of which provides the coordinator with one input port. The coordinator itself has one input port, and as it receives primitives in `MRI_Format`, the coordinator is responsible for decomposing any structured presentation into components that can be processed by the devices that it encapsulates. In the example, the coordinator may receive presentations that involve synthetic graphics, video, and audio components. The audio component of the presentation is delegated to the logical device responsible for audio rendering, while the graphics and video are managed by the second logical device. The coordinator is also responsible for ensuring that its components maintain any synchronization constraints captured by the overall presentation. It may achieve this by monitoring the overall end-to-end progression of its encapsulated devices, and placing synchronization constraints on those progression spaces, or by using more specific mechanisms available within PREMO or a given implementation.

7. Closing Remarks

This report has presented an overview of the PREMO standard. In the process, we have set out some of the design constraints that have determined the shape of the standard, and have discussed some of the alternatives that were considered. The description of PREMO object types and their behaviour has been, by necessity, incomplete and informal. In the process of developing the standard, the formal description techniques LOTOS[8,39] and Object-Z[16,17] were used to develop more precise models of particular aspects of PREMO, specifically the object model[13] and the synchronization facilities[18].

Further information about the development of the PREMO standard can be found in the papers [14, 15, 29, 30], and [31]; it must be remembered however, that the content of the standard evolved, and the material in some of these papers reflects some design choices that were subsequently modified. The definitive description of PREMO is, of course, the official ISO/IEC Standard, published in four parts [35, 36, 37, and 38]. The authors are currently producing a book on PREMO, which is expected to be published late in 1998.

8. References

1. P. Ackermann, Developing Object-Oriented Multimedia Software, Dpunkt, 1996
2. W. Appelt and A. Scheller. HyperODA: Going Beyond Traditional Document Structures. *Computer Standards & Interfaces*, 17(1):13–21, 1995.
3. F. Arbab, I. Herman, and G.J. Reynolds. An Object Model for Multimedia Programming. *Computer Graphics Forum*, 12(3):C101–C113, 1993.
4. D.B. Arnold and D.A. Duce. *ISO Standards for Computer Graphics: The First Generation*. Butterworth, 1990.
5. J. Barnes. *Programming in Ada'95*. Addison–Wesley, 1996.
6. D.R. Begault. *3D Sound for Virtual Reality and Multimedia*. Academic Press, 1994.
7. G. Blakowski and R. Steinmetz. A Media Synchronization Survey: Reference Model, Specification, and Case Studies. *IEEE Journal on Selected Areas in Communications*, 14(1):5–35, 1996.
8. T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1986.
9. G. Booch. *Object–Oriented Analysis and Design with Applications (Second edition)*. Prentice–Hall, 1997.
10. D. Brookshire Conner and A. van Dam. Sharing Between Graphical Objects Using Delegation. In C. Laffra, E.H. Blake, V. de Mey, and X. Pintado, editors, *Object–Oriented Programming for Graphics*, Springer–Verlag, Focus on Computer Graphics Series, 1995.
11. F. Colaïtis and F. Bertrand. The MHEG Standard: Principles and Examples of Applications. In W. Herzner and F. Kappe, editors, *Multimedia/Hypermedia in Open Distributed Environments*, Springer–Verlag, 1994.
12. R.B. Dannenberg, T. Neuendorffer, J. Newcomer, D. Rubine, and D. Anderson. Tactus: Toolkit-level Support for Synchronized Interactive Multimedia. *Multimedia Systems Journal*, 1(2):77–86, 1993.
13. D.A. Duce, D.J. Duke, P.J.W. ten Hagen, I. Herman, and G.J. Reynolds. Formal Methods in the Development of PREMO. *Computer Standards & Interfaces*, 17:491–509, 1995.
14. D.J. Duke and I. Herman. Programming Paradigms in an Object-Oriented Multimedia Standard. In P. Slusallek and F. Arbab, editors, *Proc. of the Eurographics Workshop on Programming Paradigms in Computer Graphics*, Eurographics Publications Series, 1997.
15. D.J. Duke, I. Herman, T. Rist, and M. Wilson. Relating the primitive hierarchy of the PREMO standard to the Standard Reference Model for Intelligent Multimedia Presentation Systems. *Computer Standards & Interfaces*, 20, 1998.
16. R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language, Version 1. Technical Report, The University of Queensland, No. 91-1, 1991.
17. R. Duke, G. Rose, and G. Smith. Object–Z: A Specification Language Advocated for the Description of Standards. *Computer Standards & Interfaces*, 17(5-6):511–534, 1995.
18. G. Faconti and M. Massink. Investigating the Behaviour of PREMO Sychronization Objects. In *Proceedings of the 4th Eurographics Workshop on Design, Specification and Verification of Interactive Systems*, Springer–Verlag, 1997.
19. B.N. Freeman–Benson and A. Borning. Integrating constraints with an object–oriented language. In I. Lehrmann Madsen, editor, *Proceedings of the ECOOP'92 European Conference on Object–Oriented Programming*, Springer Verlag, Lecture Notes in Computer Science 615, 1992.
20. D. Flanagan. *Java in a Nutshell (Second edition)*. O'Reilly, 1997.

21. M. Fowler, K. Scott, UML Distilled: Applying the Standard Object Modeling Language, Addison Wesley, 1997

22. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*, Addison-Wesley, 1995

23. A. Goldberg and D. Robson. *Smalltalk–80: The Language*. Addison–Wesley, 1989.

24. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison–Wesley, 1996.

25. S.J. Gibbs, L. Dami, and D.C. Tsichritzis. An Object Oriented Framework for Multimedia Composition and Synchronisation. In L. Kjelldahl, editor, *Multimedia (Systems, Interaction and Applications)*, Springer–Verlag, 1992.

26. S.J. Gibbs and D.C. Tsichritzis. *Multimedia Programming*. Addison–Wesley, 1995.

27. J. Hartman and J. Wernecke. *The VRML 2.0 Handbook*. Addison–Wesley, 1996.

28. P. Heller, S. Roberts, P. Seymour and T. McGinn. *Java 1.1 Developer's Handbook*. Sybex, 1997.

29. I. Herman, G.J. Reynolds, and J. Van Loo. PREMO: An emerging standard for multimedia. Part I: Overview and Framework. In: *IEEE MultiMedia*, 3(3):83–89, 1996.

30. I. Herman, N. Correia, D.A. Duce, D.J. Duke, G.J. Reynolds, and J. Van Loo. A Standard Model for Multimedia Synchronization: PREMO Synchronization Objects. *Multimedia Systems*, 6, 1997.

31. I. Herman, G.J. Reynolds, and J. Davy. MADE: A Multimedia Application development environment. In L.A. Belady, editor, *Proc. of the IEEE International Conference on Multimedia Computing and Systems, Boston*, IEEE CS Press, 1994.

32. T.L.J. Howard, W.T. Hewitt, R.J. Hubbold, and K.M. Wyrwas. *A Practical Introduction to PHIGS and PHIGS PLUS*. Addison Wesley, 1991.

33. IMA, *Multimedia System Services*, Interactive Multimedia Association, September 1994, ftp://ima.org/pub/mss/.

34. Information Technology — Computer Graphics — Programmer's Hierarchical Interactive Graphics System (PHIGS). International Organisation for Standardization, ISO/IEC IS 9592 1997.

35. Information Technology — Computer Graphics and Image Processing — Presentation Environments for Multimedia Objects (PREMO), Part 1: Fundamentals of PREMO. International Organization for Standardization, ISO/IEC 14478–1, 1998.

36. Information Technology — Computer Graphics and Image Processing — Presentation Environments for Multimedia Objects (PREMO), Part 2: Foundation Component. International Organization for Standardization, ISO/IEC 14478–2, 1998.

37. Information Technology— Computer Graphics and Image Processing — Presentation Environments for Multimedia Objects (PREMO), Part 3: Multimedia Systems Services. International Organization for Standardization, ISO/IEC 14478–3, 1998.

38. Information Technology — Computer Graphics and Image Processing — Presentation Environments for Multimedia Objects (PREMO), Part 4:Modelling, Rendering, and Interaction Component. International Organization for Standardization, ISO/IEC 14478–4, 1998.

39. Information Processing Systems — Open Systems Interconnections — LOTOS (Formal Description Technique based on the temporal ordering of observational behaviour). International Standardization Organization, ISO/IS 8807, 1989.

40. Information Technology — Coding of Moving Pictures and Associated Audio for Digital Storage up to about 1.5 Mbit/s (MPEG). International Organisation for Standardization, ISO/IEC 10744, 1992.

41. R.S. Kalawsky. *The Science of Virtual Reality and Virtual Environments*. Addison–Wesley, 1994.

42. J. F. Koegel Buford, editor. *Multimedia Systems*. Addison–Wesley, 1994.

43. C. Laffra, E.H. Blake, V. de Mey, and X. Pintado, editors. *Object–Oriented Programming for Graphics*. Springer–Verlag, Focus on Computer Graphics Series, 1995.

44. H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proc. OOPSLA'86*, ACM Press,1986.

45. A. Lie and N. Correia. Cineloop Synchronization in the MADE Environment., in: *Proceedings of the IS&T/SPIE Symposium on Electronic Imaging, Conference on Multimedia Computing and Networking*, San Jose, 1995.

46. B. Meyer. *Eiffel: The Language*. Prentice–Hall, 1990.

47. R. Newcomb, N.A. Kipp, and V.T. Newcomb. The "HyTime" — Hypermedia/Time–based Document Structuring Language. *Communication of the ACM*, 34(11):67–83, 1991.

48. OpenGL Architecture Board, OpenGL Reference Manual, 2nd edition, Addison Wesley, 1997

49. R. Otte, P. Patrick, and M. Roy. *Understanding CORBA, The Common Object Request Broker Architecture*. Prentice–Hall, 1996.

50. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object–Oriented Modeling and Design*. Prentice–Hall, 1991.

51. W. Schroeder, K. Martin, and B. Lorensen, The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics, 2nd edition, Prentice Hall, 1998

52. H. Sowizral, K. Rushforth, and M. Deering, The Java 3D API Specification, Addison Wesley, 1998.

53. B. Stroustrup. *The C++ Programming Language*. Addison–Wesley, 1990.

54. D. Unger, C. CHambers, B.-W. Chang, and U. Holzle, Organising Programs Without Classes, *Lisp and Symbolic Computation*, 4(3), 22-22, Kluwer, 1991

55. A. Watters, G. van Rossum, J.C. Ahlstrom. *Internet Programming with Python*. M&T Books, 1996.

56. J. Wernecke, *The Inventor Mentor*, Addison Wesley, 1994.

57. A. Wollrath, J. Waldo, and R. Riggs. Java-Centric Distributed Computing. *IEEE Micro*, 17(3):44–53.

58. P. Wißkirchen. *Object–Oriented Graphics*. Springer–Verlag, 1990.

59. W3C SMIL Draft Specification. Public document, available at http://www.w3.org/TR/WD-smil, 1998.