Plugging Graphics into Distributed Multimedia

D.J. Duke
Department of Computer Science, The University of York, Heslington, York, YO1 5DD, UK
email: duke@minster.york.ac.uk

I. Herman
Centrum voor Wiskunde en Informatica (CWI), Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
email: Ivan.Herman@cwi.nl

Abstract

Advances in hardware, software, and coding standards for digital media have now made the delivery of multimedia information a standard component of many systems. Unfortunately, the pace of this technological development, coupled with strong commercial competition between leading vendors, has meant that little consensus has emerged over the design of programming interfaces to allow the creation, manipulation and presentation of such data. 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) for distributed multimedia. This work goes beyond previous SC24 standards such as GKS and PHIGS in combining both synthesised graphics with general digital media. This paper describes 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 origins of PREMO date back to 1989, when discussion within SC24 began on a proposed new standard for Computer Graphics. The distinctive feature of the proposed project would be the use of object-orientation as the fundamental design framework. Such an approach seemed timely, as object-oriented design and programming was rapidly becoming established, and work on a number of experimental object oriented APIs for computer graphics indicated that there was considerable virtue in such an approach. Indeed, it was around this time that Silicon Graphics Inc. were developing the Open Inventor toolkit. The ‘PREGO’ project, as it became known, evolved into PREMO as the result of two further requirements. First, any new standard in the area of graphics should encompass other media, such as video, audio (both captured and synthetic) [6], and in principle be extensible to new modalities such as haptic output and speech or gestural input. Second, the standard should allow the construction of distributed systems, where parts of a system involved in the generation, processing and presentation of media data could be distributed across geographically remote sites, interacting through a network. These three issues: object orientation, multimedia data, and distribution, became the key design constraints on the PREMO project [7,8,10], which was initiated within ISO/IEC JTC 1 in 1992. PREMO will become an International Standard early in 1998 [13].

One of the problems faced by the designers of PREMO integrating the generation and processing of synthetic graphics, which is traditionally viewed as a pipeline from modelling primitives through to raster output, with continuous media such as video and audio. More specifically, it should be possible to integrate different kinds of media within the description of a presentation, and yet utilise media-specific processing elements and devices in the realisation of that description. One part of the solution came to the PREMO working group in the form of the Multimedia Systems Services framework, a proposal for media processing developed in initially by the IMA (Interactive Multimedia Association), a consortium of systems vendors [11]. Significant refinements were made to the original IMA material, in particular to integrate the concepts and facilities of the MSS with the underlying framework and basic components of PREMO. While MSS provides architectural support for viewing graphics processing in similar terms to other media processing structures and tasks, it does not directly address the data that is used to describe the presentation. This becomes an issue within the Modelling, Rendering and Interaction (MRI) component of PREMO, which defines a hierarchy of abstract primitives for representing media content, and extends the provisions of the MSS component with devices that utilise this notion of primitive. However, primitives in PREMO are quite unlike those in other APIs such as GKS, PHIGS, or OpenGL [2]. PREMO is not intended as a rendering engine, but as a framework for integrating media processing and rendering, performing a service for digital media somewhat like what a coordination language like Linda [4] or Manifold [1] does for concurrent processing. The main objective of this paper is to describe just how PREMO does integrate support for graphics into its general model of distributed multimedia; a second objective is to provide an overview of the Standard. Section 2 describes the main provisions of the first three parts of the PREMO standard, namely the object model, the foundations, and the MSS. Section 3 then describes how modelling and rendering is built on top of this infrastructure. Some of the issues relevant to the implementation of PREMO are described briefly in Section 4, which concludes the paper.

2 An Overview of PREMO

The PREMO Standard consists of four parts, also called components. PREMO was designed from the outset to be extensible - it was envisaged that the needs of specific domains or application areas would be met by creating new components. Typically, a component will define a number of 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. Not all of the types defined within a component are necessarily needed in a given context, and PREMO components define one or more *profiles* which consist of a cluster of type definitions. A component can build on (extend) the profiles of other components. 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 that of the OMG (Object Management Group), there are some novel features in the model. In particular, operations are defined to operate in one of three modes: synchronous, asynchronous, and sampled.

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 [11] and subsequently adopted by SC24 and refined into a PREMO component.

4. *Modelling, Rendering and Interaction*. The MSS component defines concepts of 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.

The major features of PREMO can be summarised as follows.

- *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 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, which mix real–time 3D rendering techniques with sound, video, or even tactile feedback, and their effects, are, for example, within the scope of PREMO.
- *PREMO is a framework.* This means that the PREMO specification does not provide all the possible object types for making a graphics or multimedia system or application. Instead, *PREMO* provides a general programming framework, a sort of middleware, where various organisations or applications may plug in their own specialised objects with specific behaviour. The goal is to define those object types which are at the basis of any multimedia development environment, thereby ensuring interoperability.

The remainder of this section describes the first three components of PREMO components in greater detail. Further discussion of the MRI component, which is fundamental to integrating graphics into the general media facilities of PREMO, is reserved for Section 4.

Fundamentals of PREMO (Part 1)

The PREMO object model follows widespread practice in the sense that it views objects as entities that consist of a state and a collection of operations on that state. More radical approaches to object-oriented systems, for example based on delegation models and prototypes [3], were considered insufficiently mature for the PREMO standard. Instead, each object in PREMO is an instance of an object type (also called a *class* in the literature), that defines the structure of the state and behaviour of each operation. Each object has an identity that is independent of the object's state, and the environment of an object interacts with it by means of an object reference. Thus to invoke an operation on an object, it is necessary to have a reference to that object. There is no concept in PREMO of having 'direct access' to an object; this is an important principle, since objects in PREMO may be distributed. However, the object model of PREMO differs from other standards such as the OMG reference model in two ways. First, any object in PREMO may be active, with its own thread of control, and second, each operation is defined to operate in one of three possible request modes, described below.

PREMO makes a distinction between object types and non-object types, which represent unstructured values such as integers, characters and booleans, and simple structured values such as sequences. Object references form a non-object data type. An operation consists of a name, and a signature which consists of a sequence of input types and a sequence of output types. Operations can only accept or yield values of non-object data types, but as the non-object types include object references it is possible to pass a reference to an object as a parameter to an operation. In particular, the first type in the signature of an operation is a reference to the object type on which the operation is defined.

An object type *S* in PREMO can be defined to be a subtype of another object type *T*, meaning that a reference to an object of type S can be used in place of a reference to an object of type *T* in any context. Multiple subtyping is allowed, i.e. an object type can be a subtype of more than one other object type. The standard defines the meaning of subtyping with respect to the signatures of the operations defined by the types. An object type can be defined as an extension of another object types by inheritance; the 'new' object type acquires the state and operations of the object type that it extends (its *supertype*). Multiple inheritance is permitted, but the Standard does not define the meaning of an object type for which more than one parent provides an implementation of an operation with a given signature. If object type *S* inherits from object type *T*, then *S* is automatically a subtype of *T*.

An operation can be defined as *protected,* meaning that the operation cannot be invoked by entities external to instances of the object type, but such an operation can be modified through inheritance. An operation, when invoked with incorrect data or in an unexpected situation, can raise an exception, and provision is made for data to be associated with the exception to indicate the cause. How an exception is handled once raised is dependent on the language binding. An operation is invoked through an operation request, which depending on the definition of the operation, is serviced in one of three modes.

- A *synchronous* operation request causes the caller of the request to be suspended until the request has been serviced and a result returned.
- The caller of an *asynchronous* request can continue its own thread of control as soon as the call is made; at some point the requested operation will be invoked, but no result is returned.
- A *sampled* request is similar to an asynchronous request, except that any pending request for a given operation (i.e. a call that has not been serviced) is overwritten by a new request; conceptually, each operation has a 1-place buffer for storing pending requests.

The Foundation Component (Part 2)

The role of Part 2 is that of a general purpose toolkit, providing a number of processing facilities that are needed across a range of multimedia applications, and indeed which have multiple uses within a single application. Figure 1 gives an overview of the types defined in this part. All object types within PREMO are derived ultimately from a common supertype called *PREMOObject,* which defines a number of fundamental services, such as the ability to enquire the type of an object, and the position of that type within the hierarchy of types defined by inheritance. Below this type the hierarchy bifurcates. The object type *SimplePREMOObject* serves as a common supertype for a group of object types called *structures*. These are object types that primarily exist to encapsulate data in the form of attributes; the operations of these objects are of less interest. In contrast, *EnhancedPREMOObject* is a common supertype for those object types that can provide services over a network, and which therefore can be distributed. In sup-

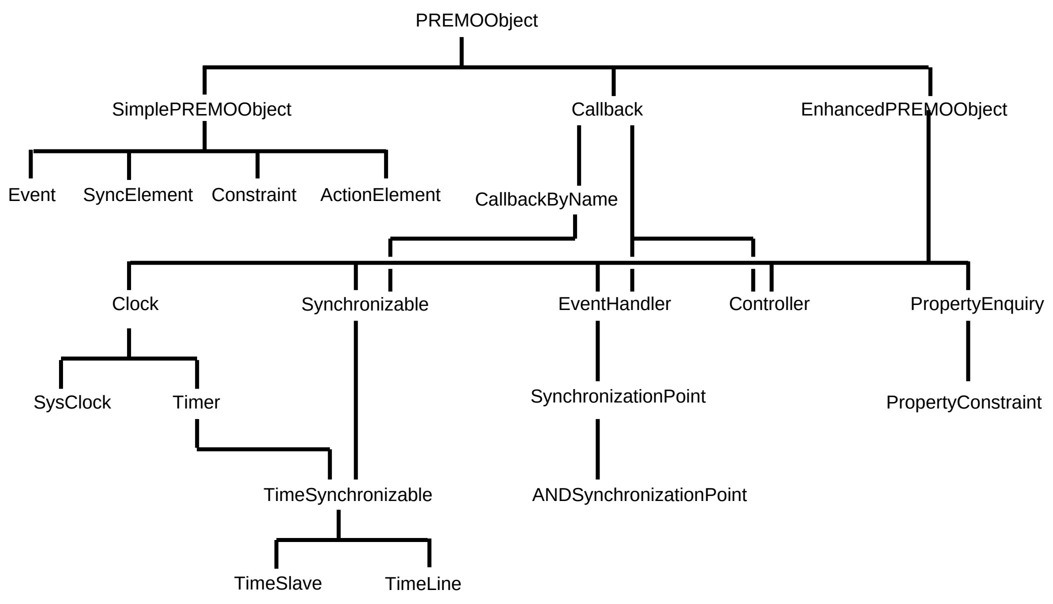

Fig. 1. The Object Types of the Foundation Component

port of this role, objects of this type are associated with properties, and the type defines a set of operations for accessing, creating and modifying these properties. A property is a pair consisting of a key, and a sequence of values. The key is represented by a string, and properties, unlike the internal state of an object, can be accessed freely by other objects within a system.

PREMO objects can communicate via an event mechanism 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. Events are defined as simple object types to carry information, specifically the name of the event, the source of the event, and any additional data. Figure 2 illustrates the approach. Objects that are interested in a particular event, such as A in the figure must (1) be of a type that is a subtype of the *Callback* object type, which provides a *callback* operation, and must register their interest with an instance of the *EventHandler* object type (2). When an object B wants to notify its environment that an event has occurred, it invokes the *dispatchEvent* operation on an event handler (3), and all objects that have registered with that handler to be notified of the event will have their callback operation invoked (4).

An important use of event management is to realise synchronization requirements within an application. Synchronization in PREMO is based on the use of events and event handlers to achieve complex synchronization patterns. Events related to synchronization are generated by a type of PREMO object called *Synchronizable* objects. These are autonomous objects that have an internal progression space, on which reference points can be attached. Conceptually, the progression space represents the temporal extent of some media representation, and progress through the progression space will be made during processing of that media. The mode - stopped, playing, paused etc. - of a synchronizable object is controlled by a number of operations, and a number of attributes collectively define the parameters that affect how progress is made, for example, the direction of progression. When a reference point is encountered during progression, an event is sent to a specified event handler. The reference point also contains a flag which indicates whether progression should be suspended; by placing a synchronizable object into a so-called 'waiting' state, the processing of one part of a presentation can be delayed to enforce synchronization constraints with another part.

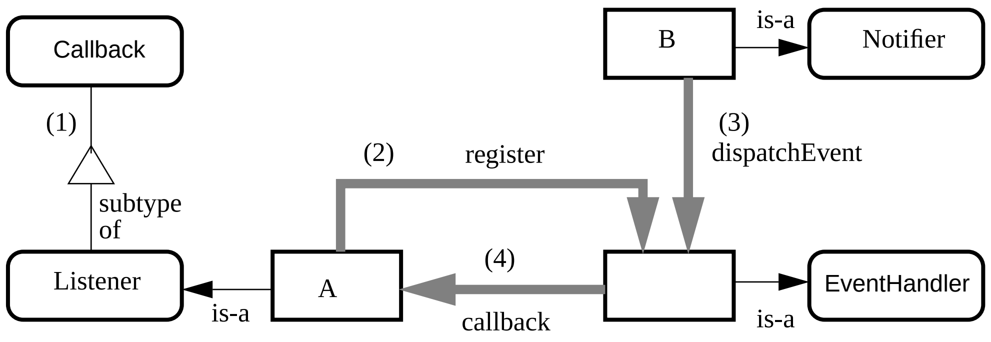

Fig. 2. Event Handling in PREMO

PREMO introduces object types to represent arbitrary clocks, a subtype of clocks representing ‘real time’ system clocks, and a resetable timer. An important subtype of the *Synchronizable* object type is *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 span of a synchronizable object. 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. PREMO also provides an extended form of event handler to support synchronization. The *ANDSynchronizationPoint* object type requires that an object that will dispatch an event to a handler register this with the handler; only after all objects that have registered to dispatch an event have raised that event will the callbacks be invoked.

The basic facilities provided by *EnhancedPREMOObject* to associate values with named properties is developed by two further object types, *PropertyInquiry* and *PropertyConstraint*. In the first of these types, each property key is associated with a corresponding capability key, which describes the range of values that the corresponding property can take on. The *PropertyConstraint* type extends this approach through two operations that attempt assign values to properties based on a set of property values passed as parameters to these operations. Properties play an important role in allowing PREMO application to be configured from a collection of possible devices. Mechanisms to support this configuration form part of the Multimedia Systems Services component, discussed next.

Multimedia Systems Services (Part 3)

Multimedia Systems Services was the name given by the Interactive Multimedia Association to a model for building distributed multimedia applications. The original model incorporated ideas such as property management, which were subsumed into Part 2 of PREMO, and a collection of object types for managing networks of media devices that was developed into the object types and definitions of PREMO Part 3.

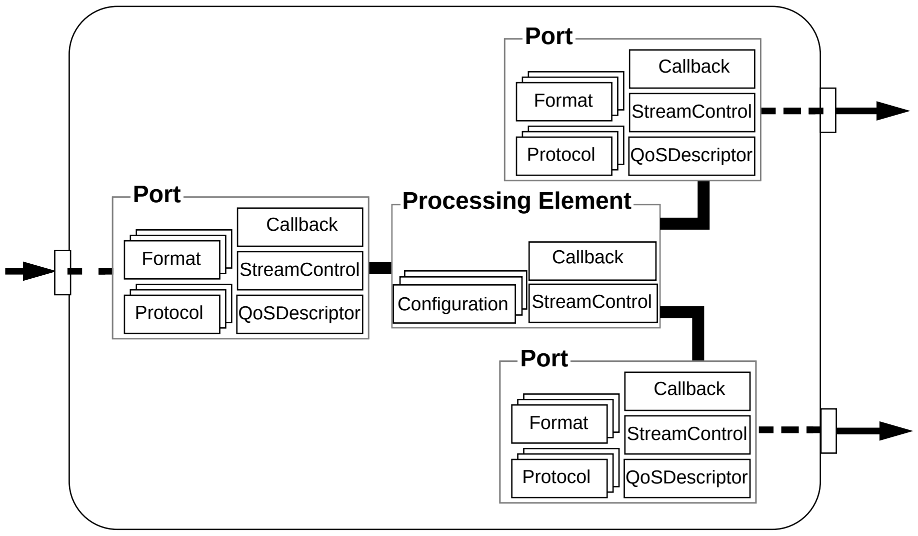

Fig. 3. Structure of a Virtual Device

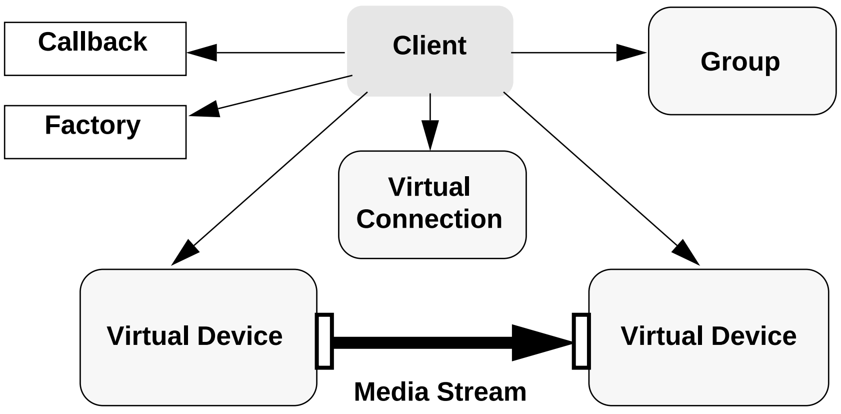

Fig. 4. Media Stream support within MSS

The conceptual model of Part 3 (and by implication, of a distributed PREMO system) is one of a collection of devices, each of which is an autonomous processing unit. The actual processing component of a device is not specified within PREMO, nor is the means by which the processor communicates with its environment. What Part 3 of PREMO does define is the interface of the so-called *VirtualDevice* object type, shown in Figure 3. This consists of a collection of input and output ports, with which are associated various objects that characterise the nature of the communication that takes place via that port. The objects are instances of the types *StreamControl*, *Format*, *Protocol*, and *QoSDescriptor*. Ports provide the basis for one of three ways in which devices can communicate and cooperate. As shown in Figure 4, ports can be connected by, conceptually, a media stream, which is managed by a *VirtualConnection* object. There is actually no ‘stream’ object type in the standard, instead the control of data flow from one device to another is the role of the *StreamControl* objects attached to the ports involved. How the communication is actually realised will depend on the kind of environment in which the PREMO system is running. Connections can be unicast or multicast.

VirtualConnection and *VirtualDevice* are both subtypes of *VirtualResource*, which extends the *PropertyInquiry* type of Part 2 with mechanisms that allow resources to be acquired and released. Resource acquisition utilises the property system to specify requirements on a resource. Indeed, the use of properties within all of the major object types defined in Part 3 represents a second mechanism for cooperation between objects in a multimedia network. Device properties, and specific structures such as the *QoSDescriptor*, are intended to allow networks of devices to be configured and controlled through a negotiation mechanism based for example on constraint management. The range of technologies that might be used for controlling a network is however large and diverse, and rather than mandate a limited range of solutions, the specification of PREMO simply provides appropriate hooks that could be used by a suitable management infrastructure.

Each *VirtualDevice* also contains references to a number of event handlers, and in this way the event mechanism of Part 2 is available as the third means for communication and cooperation within a network. Two further subtypes of *VirtualResource* are the *Group* and *LogicalDevice* object types. The former allows a number of VirtualDevices to be treated as a single network resource, while the second of these types allows a group of devices to be combined into a single, higher-level device. These have a further role in the context of modelling and rendering, as will be explained shortly.

3 The Modelling and Rendering Component

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 description of the MRI Component consists of three parts. The first concerns the hierarchy of modelling primitives for characterising multimedia presentation. The second 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 network of arbitrary media processing elements. The third concerns 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.

Primitives in PREMO

The potential domains of application for a system such as PREMO are vast. Two directions initially appear feasible when considering how primitives for modelling and rendering could be supported in a system like this. First, it would be possible to take an existing set of primitives from an established system, for example the nodes provided by OpenInventor[15], and adopt these to the needs of PREMO, possibly through some further extensions. The problem here is in finding a set of primitives acceptable to all parties, and 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. Instead, the approach taken in PREMO is to view the standard as a framework for supporting the integration of different modelling and rendering technologies within a heterogeneous distributed system. In this context, the role of primitives is rather different. PREMO cannot and does not 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. Modellers, for example, may use specific representations such as constructive solid geometry, NURBS surfaces, particle systems etc. Such techniques may require an enriched set of basic primitives. 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 set primitives described in PREMO is shown in Figure 5. The top of this hierarchy consists of a single abstract object type called *Primitive*, from which is derived seven key subtypes, several of which support further sub-hierarchies.

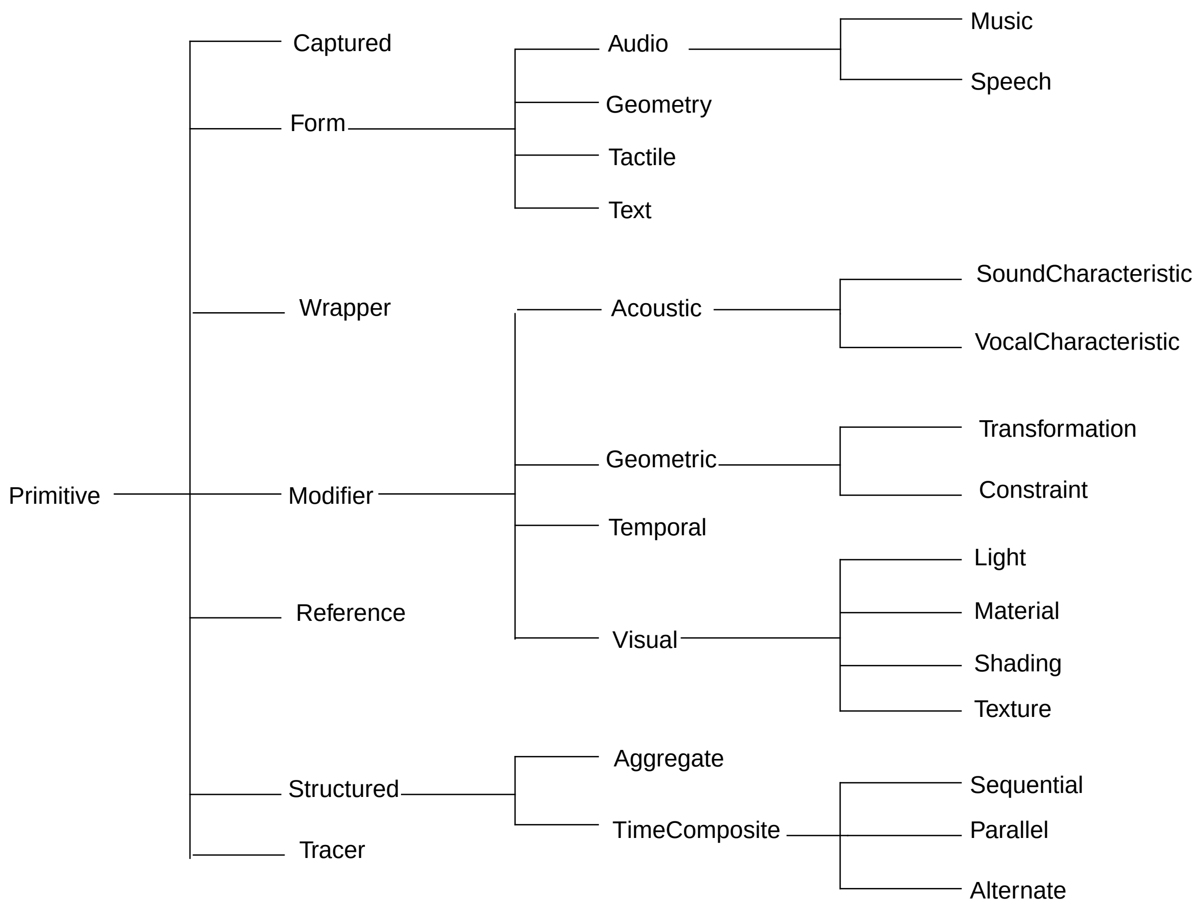

Fig. 5. The PREMO Primitive Hierarchy

1. *Captured* primitives allow the import of data encoded in some format defined externally to PREMO, for example MPEG[12].
2. *Form* primitives are those where the appearance of the primitive is constructed by some renderer or more general media engine. These include geometric primitives (polylines, curves etc.), and audio primitives for speech and music.
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 of primitive hierarchies by 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. 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, and are used for synchronization as described later in this section.

Devices for Modelling, Rendering and Interaction

The MRI Component derives a number of object types from the VirtualDevice type of Part 3. As in Part 3, these do not represent concrete devices. They instead define the interface that a device must offer in able to work within a PREMO system, and in the case of Part 4, with primitives derived from the hierarchy described above. Figure 6 shows how a system might be configured using MRI devices, though as MRI devices are abstract types, in practice these would be concrete subtypes provided by an application or an additional PREMO component.

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 that allows MRI primitives to be transmitted and received via the ports of the device. 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 figure, the following specialisations of *MRI_Device* are used:

1. *Modellers* and *Renderers* 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.
5. Collections of MRI devices that implement specific functionalities (e.g. provide a form of workstation) can be organised into higher-level components via the *LogicalDevice* object type defined by MSS. Three such devices are shown in Figure 6, identified by the letters A, B and C.

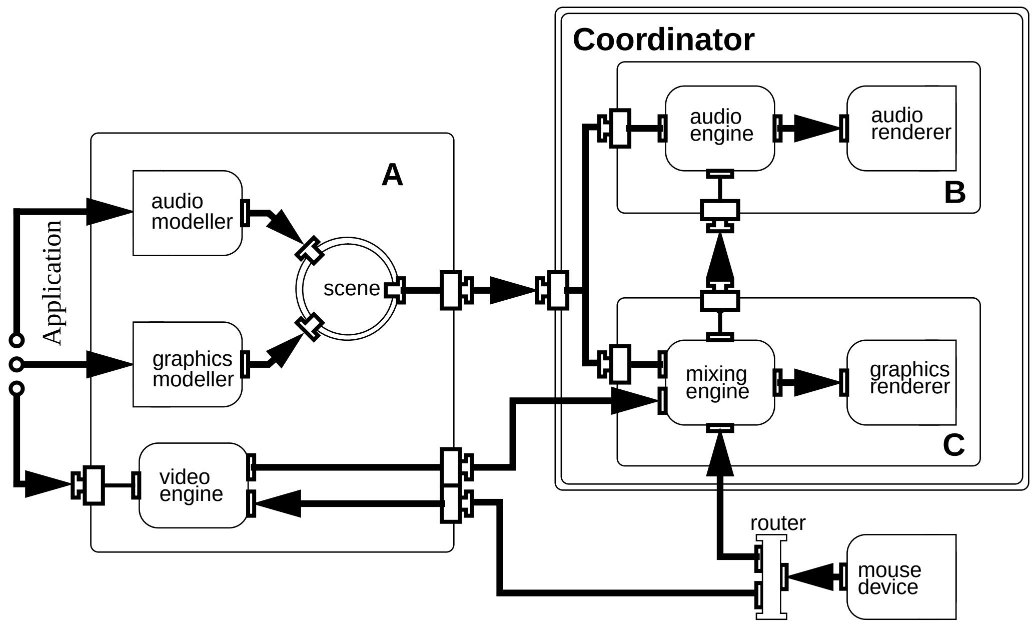

Fig. 6. A Configuration of MRI Devices

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.

Coordination

The MRI Component also 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 does this by monitoring the overall end-to-end progression of its encapsulated devices, and placing synchronization constraints on those progression spaces. Note that these encapsulated devices may receive input from other components of the system; the coordinator is only responsible for realising the presentation of media data received via the designated ports.

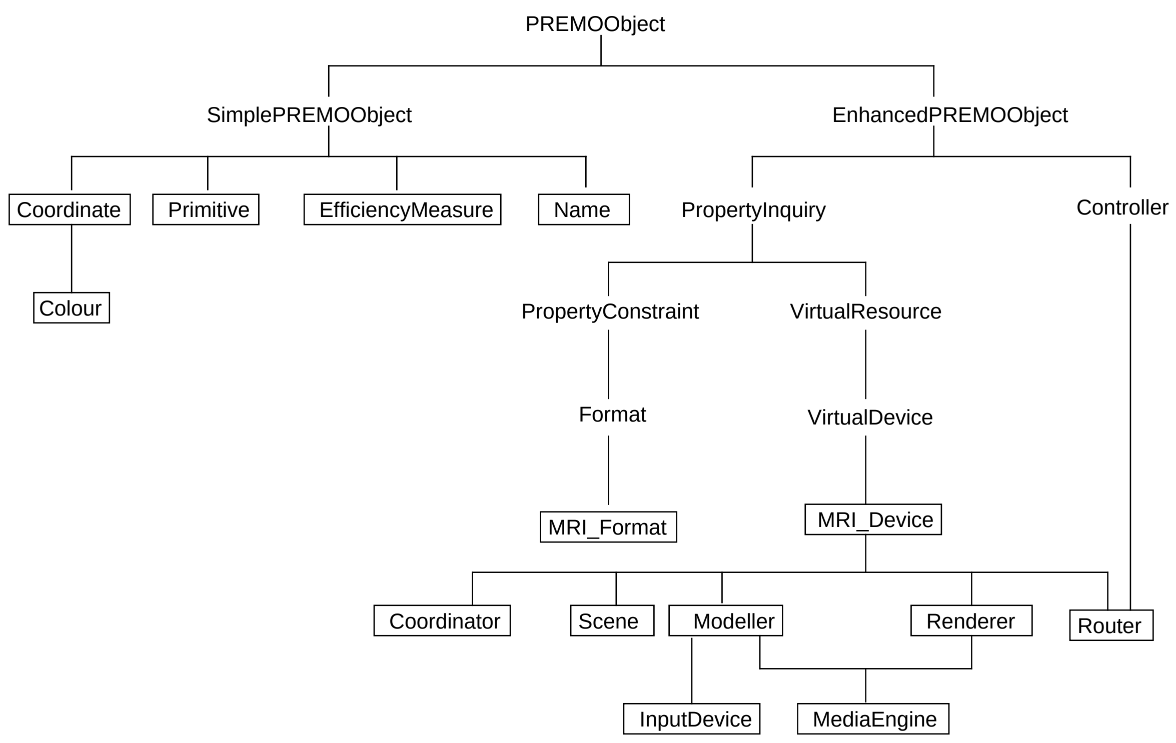

Fig. 7. The PREMO MRI Object Types

Further object types are defined within the MRI Component to support these mechanisms, for example a representation of coordinates within an arbitrary space, colour, and a means of coordinating synchronization between various processing devices. Figure 7 shows the object types defined for modelling, rendering and interaction within PREMO.

4 Conclusion

The authors are currently developing a reference implementation of PREMO within the Java language. The task of implementing PREMO is obviously non-trivial, but there are a number of issues that arise not from the technical requirements of the standard, but from the fact that PREMO comes defined with its own object model and assumptions about its environment. Each object oriented programming language is also equipped with its own object model, and such a model has a variety of features and constraints, which may or may not be compatible with those of the standard. SC24 appointed a Special Rapporteur to examine the problems that were likely to be encountered in mapping PREMO onto an implementation in an object oriented language. Three of the current 'state of the art' languages, specifically C++, Java, and Ada'95 were examined, and aspects of the binding of PREMO to each of these that would require careful thought were identified. Issues considered included:

- support for inheritance within the language, for example whether single or multiple inheritance is permitted;
- support for (or the difficulty of defining) objects with their own thread of control; and
- mechanisms by which the different operation request modes (synchronous, asynchronous and sampled) could be implemented.

In the case of PREMO, there is also a further level of complexity, as PREMO objects are potentially distributable. It was decided that, in addition to the language binding, a standard like PREMO would also need to define an environment binding that defines how the requirements that the standard places on its environment, for example the ability to invoke remote methods, could be realised through the use of specialised services such as the CORBA technology of OMG [14], or the RMI package of the Java library. Some of the problems that might have been encountered during the implementation of PREMO were identified early on in the project through the use of formal description techniques [5]; the use of these methods in the development of PREMO has been the focus of an number of other papers.

Object orientation is now widely used as a design and implementation technology, and it is very likely that any future work items within SC24 will incorporate this technology. Consequently, the results of the PREMO-related work have implications beyond the immediate life of the standard.

Acknowledgement

The PREMO Standard is the result of work carried out by the members of ISO/IEC JTC 1/SC24 WG6 over a number of years, and the authors are pleased to acknowledge the contributions that all members of the committee have made to the successful conclusion of the project. Particular mention is due to Jim Van Loo of Sun Microsystems Inc. whose effort was instrumental in the development of the MSS component. Thanks are also due to Scott Marshall (CWI), and to the ERCIM Computer Graphics Network (European Commission Human Capital and Mobility Project, contract number CHRX–CT93–0085) that supported development of the standard through a number of key workshops and working meetings. The helpful comments of the anonymous reviewers are gratefully acknowledged.

References

[1] F. Arbab: “The IWIM model for coordination of concurrent activities”, in: *Coordination Languages and Models*, Springer Verlag, Lecture Notes in Computer Science, vol. 1061, 1996.

[2] D.B. Arnold and D.A. Duce, ISO Standards for Computer Graphics: The First Generation, Butterworths, London, 1990.

[3] S.D. Brookshire Conner and A. van Dam, “Sharing between graphical objects using delegation”, in: Object–Oriented Programming for Graphics, C. Laffra, E.H. Blake, V. de May, X. Pintado (Eds), Focus on Computer Graphics Series, Springer Verlag, 1995.

[4] N. Carriero and D. Gelernter: “Linda in Context”. In: *Communication of the ACM*, **32**,1989.

[5] D.A. Duce, D.J. Duke, P.J.W. ten Hagen, I. Herman, and G.J. Reynolds: “Formal Methods in the Development of PREMO”. In *Computer Standards & Interfaces*, **17**, pp. 491-509, 1995.

[6] J. Gibbs and D.C. Tsichritzis, *Multimedia Programming*, Addison-Wesley, ACM Press series, 1995.

[7] I. Herman, G.S. Carson, J. Davy, P.J.W. ten Hagen, D.A. Duce, W.T. Hewitt, K. Kansy, B.J. Lurvey, R. Puk, G.J. Reynolds, and H. Stenzel, “Premo: an ISO Standard for a Presentation Environment for Multimedia Objects”, in: *Proceedings of the Second ACM International Conference on Multimedia (MM’94)*, San Francisco, D. Ferrari, editor, ACM Press, 1994.

[8] 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”, In *Multimedia Systems*, to appear in 1998.

[9] I. Herman, G.J. Reynolds, and J. Davy: “MADE: A Multimedia Application development environment”. In *Proc. of the IEEE International Conference on Multimedia Computing and Systems, Boston*, L.A. Belady, S.M. Stevens, and R. Steinmetz (Eds.), IEEE CS Press 1994.

[10] I. Herman, G.J. Reynolds, and J. Van Loo: “PREMO: An emerging standard for multimedia. Part I: Overview and Framework”, In *IEEE MultiMedia*, **3**, pp. 83-89, 1996.

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

[12] International Organization for Standardization, Information processing systems — 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.

[13] International Organization for Standardization, Information processing systems — Computer graphics — Presentation environment for multimedia objects (PREMO), ISO/IEC 14478, April 1998.

[14] R. Otte, P. Patrick, M. Roy, *Understanding CORBA*, Prentice Hall, 1996.

[15] J. Wernecke, *The Inventor Mentor*, Addison Wesley, 1994.