INTERNATIONAL STANDARD

ISO/IEC 14478-4

First edition
1998-12-15

Information technology — Computer graphics and image processing — Presentation Environment for Multimedia Objects (PREMO) —

Part 4:
Modelling, rendering and interaction component

Technologies de l'information — Infographie et traitement d'images — Environnement de présentation d'objets multimédia (PREMO) —

Partie 4: Composant pour la modélisation, le rendu et l'interaction



Reference number
ISO/IEC 14478-4:1998(E)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Contents

Foreword ..... **vii**

Introduction ..... **viii**

1 Scope **9**

2 Normative references **9**

3 Definitions **10**

3.1 PREMO Part 1 definitions 10

3.2 PREMO Part 2 definitions 10

3.3 PREMO Part 3 definitions 10

3.4 Additional Definitions 10

4 Symbols and abbreviations **12**

5 Conformance **12**

6 Overview of the Modelling, Rendering and Interaction Component. **12**

6.1 Introduction 12

6.2 Overview 12

6.3 Devices for Modelling, Rendering, and Interaction 16

6.4 Primitives and Coordinates 17

6.4.1 Introduction 17

© ISO/IEC 1998

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

6.4.2 Coordinates . . . 17
6.4.3 The Primitive Hierarchy in PREMO . . . 17
6.4.3.1 Overview . . . 17
6.4.3.2 Captured Primitives . . . 18
6.4.3.3 Form Primitives . . . 18
6.4.3.4 Modifier Primitives . . . 18
6.4.3.5 Reference Primitives . . . 19
6.4.3.6 Structured Primitives . . . 19
6.4.3.7 Tracer Primitives . . . 19
6.4.3.8 Wrapper Primitives . . . 19
6.4.4 Primitives and MRI Devices . . . 19
6.5 Scene . . . 20
6.6 Interaction . . . 21
6.7 Coordinators . . . 21
6.8 Dependencies on other Parts . . . 23
6.9 Subtyping Diagram . . . 23
7 Coordinates . . . 23
7.1 General Coordinates . . . 23
7.2 Colour . . . 24
7.3 TimeLocation . . . 25
8 Primitives . . . 25
8.1 Introduction . . . 25
8.2 Captured Primitives . . . 27
8.3 Form Primitives . . . 27
8.3.1 Introduction . . . 27
8.3.2 Audio Primitives . . . 27
8.3.3 Geometric Primitives . . . 28
8.3.4 Tactile Primitives . . . 28
8.3.5 Text Primitives . . . 28
8.4 Modifier Primitives . . . 28
8.4.1 Introduction . . . 28
8.4.2 Acoustic Modifiers . . . 29
8.4.3 Structural Modifiers . . . 29
8.4.4 TimeFrame Modifiers . . . 29
8.4.5 Visual Modifiers . . . 30
8.5 Reference Primitives . . . 30
8.5.1 References . . . 30
8.5.2 The Name Object Type . . . 30
8.6 Structured Primitives . . . 31
8.6.1 Introduction . . . 31
8.6.2 Aggregate . . . 31
8.6.3 TimeComposite . . . 32
8.7 Tracer Primitives . . . 35
8.8 Wrapper Primitives . . . 36
9 Modelling, Rendering and Interaction Device . . . 36
9.1 Introduction . . . 36

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.2 MRI_Format ... 36
9.3 Efficiency ... 36
9.4 Behaviour ... 37

10 Modeller ... 37

11 Renderer ... 38

12 MediaEngine ... 38

13 Scene ... 39

14 Interaction ... 42
14.1 Introduction ... 42
14.2 Input Device ... 42
14.3 Router ... 43

15 Coordinator ... 43

16 Functional Specification ... 45
16.1 Introduction ... 45
16.2 Non-object data types ... 45
16.3 Exceptions ... 46
16.4 Objects for coordinate spaces ... 47
16.4.1 *Coordinate* object ... 47
16.4.2 *Colour* object ... 48
16.4.3 *TimeLocation* object ... 48
16.5 *Name* object ... 49
16.6 Objects for media primitives ... 49
16.6.1 *Primitive* object ... 49
16.6.2 *Captured* object ... 50
16.6.3 Objects describing primitives with spatial and/or temporal form ... 50
16.6.3.1 *Form* object ... 50
16.6.3.2 Objects describing form primitives for audio media data ... 50
16.6.3.3 Objects describing form primitives for geometric media data ... 51
16.6.4 Objects describing primitives for the modification of media data ... 52
16.6.4.1 *Modifier* object ... 52
16.6.4.2 Objects describing modifier primitives for audio media data ... 52
16.6.4.3 Objects describing modifier primitives for structural aspects of media data ... 53
16.6.4.4 *TimeFrame* object ... 54
16.6.4.5 Objects describing modifier primitives for visual aspects of media data ... 54
16.6.5 *Reference* object ... 55
16.6.6 Objects for organising primitives into structures ... 56
16.6.6.1 *Structured* object ... 56
16.6.6.2 *Aggregate* object ... 56
16.6.6.3 Objects for organising media data within time ... 57
16.6.7 *Tracer* object ... 58
16.6.8 *Wrapper* object ... 59
16.7 Objects for describing properties of devices ... 59
16.7.1 *MRI_Format* object ... 59
16.7.2 *EfficiencyMeasure* object ... 60
16.8 Processing devices for media data ... 60
16.8.1 *MRI_Device* object ... 60

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16.8.2 *Modeller* object 60
16.8.3 *Renderer* object 61
16.8.4 *MediaEngine* object 61
16.9 *Scene* object 63
16.10 Objects for supporting interaction 65
16.10.1 *InputDevice* object 65
16.10.2 *Router* object 66
16.11 *Coordinator* object 67

17 Component Specification 69

A Overview of PREMO Modelling, Rendering and Interaction Object Types . 70

B Diagrammatic Conventions 73
B.1 Introduction 73
B.2 General Graphical Signatures 73
B.3 Conventions for Devices and Communication 74

C Relationship between Part 4 and the CGRM 75
C.1 Introduction 75
C.2 Architectural Links 75
C.3 Processing Links 76
C.4 Input and Output Primitives 76
C.5 Storage 76

D A typical example scenario of MRI usage 77

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, government and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee: ISO/IEC JTC1. Draft International Standards adopted by the joint technical committees are circulated to the national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

ISO/IEC 14478-4 was prepared by Joint Technical Committee ISO/IEC JTC1, *Information technology*, Subcommittee SC24, *Computer graphics and image processing*.

ISO/IEC 14478 consists of the following parts under the general title *Information technology — Computer graphics and image processing — Presentation Environment for Multimedia Objects (PREMO):*

- *Part 1: Fundamentals of PREMO*
- *Part 2: Foundation component*
- *Part 3: Multimedia systems services*
- *Part 4: Modelling, rendering, and interaction component*

Annex A forms an integral part of this part of ISO/IEC 14478. Annexes B to D are for information only.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Introduction

The Modelling, Rendering and Interaction component of PREMO describes 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 objective of this component is to provide developers and users of modelling and rendering applications with a framework for supporting the definition and use of interoperable devices within a distributed setting. It achieves this by:

a) providing an extensible framework of primitives for use in modelling, rendering and interaction which encompass multiple media, and which can be organized into larger structures and embedded into scenes.

b) extending the resource and device hierarchies of the PREMO Part 3 (Multimedia Systems Services) Component to allow modelling, rendering and interaction to be uniformally integrated into a network of objects for managing the production and utilization of multimedia data.

c) utilizing the property and capability management services of PREMO Part 3 to characterize the behaviour of modelling, rendering and interaction devices, allowing an application to be configured from such devices such that constraints on performance and functionality are satisfied.

d) building on the object model and foundation objects of PREMO Part 1 and Part 2 to allow subsequent components to realize and extend specific modelling, rendering and interaction functionality.

This component follows PREMO Part3 in describing the external interface of object types and other entities involved in modelling, presentation and interaction, but not the internal structures needed to implement these. That is, it is not the purpose of this component to provide a set of building blocks that can be assembled into a modeller or a renderer. Rather, the component provides facilities to enable devices, built with various applications or performance trade-offs in mind, to interoperate in a heterogenous presentation environment.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Information technology — Computer graphics and image processing — Presentation Environment for Multimedia Objects (PREMO) — Part 4: Modelling, Rendering, and Interaction Component

1 Scope

This part of ISO/IEC 14478 describes a set of object types and non-object types to provide the construction of, presentation of, and the interaction with Multimedia information. The multimedia information can be graphics, video, audio, or other types of presentable media. This information can be enhanced by time aspects. Throughout this document this part of ISO/IEC 14478 will also be referred to as "Modelling, Rendering and Interaction", and abbreviated as MRI.

The Modelling, Rendering and Interaction Component constitutes a framework of 'Middleware', system software components lying between the generic operating system and computing environment, and a specific application operating as a client of the services and type definitions provided by this component. It provides a framework over the foundation objects and multimedia systems services defined in other Parts of the international standard for the development of a distributed and heterogeneous network of devices for creating multimedia models, rendering these models, and interacting with this process.

The Modelling, Rendering and Interaction Component encompasses the following characteristics:

a) provision of a hierarchy of multimedia primitives as an abstract framework for describing the capabilities of modelling and rendering devices, and for enabling their interoperation;

b) within the primitive hierarchy, specific provision for describing the temporal structure of multimedia data through the stepwise construction of structured primitives from component data;

c) provision of abstract types for modellers, renderers and other supporting devices, enabling the integration of such devices or any future subtypes representing real software or hardware, into a processing network of such devices;

d) provision of an object type to map synchronization requirements expressed within multimedia primitives into control of the stream and synchronization mechanisms provided by ISO/IEC 14478-2 and ISO/IEC 14478-3.

The Modelling, Rendering and Interaction Component relies on the object types and services defined in PREMO Part 2: Foundation Components (ISO/IEC 14478-2), and PREMO Part 3: Multimedia Systems Services (ISO/IEC 14478-3).

2 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 14478. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO/IEC 14478 are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 11072:1992, *Information technology — Computer graphics — Computer Graphics Reference Model (CGRM).*

ISO/IEC 7942-1:1994, *Information technology — Computer graphics and image processing — Graphical Kernal System (GKS) — Part 1: Functional description.*

ISO/IEC 9592:1997, *Information technology — Computer graphics and image processing — Programmer's Hierarchical Interactive Graphics System (PHIGS).*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

ISO/IEC 14478–1:1998, *Information technology — Computer graphics and image processing — Presentation Environment for Multimedia Objects (PREMO) — Part 1: Fundamentals of PREMO*

ISO/IEC 14478–2:1998, *Information technology — Computer graphics and image processing — Presentation Environment for Multimedia Objects (PREMO) — Part 2: Foundation component*

ISO/IEC 14478–3:1998, *Information technology — Computer graphics and image processing — Presentation Environment for Multimedia Objects (PREMO) — Part 3: Multimedia systems services*

3 Definitions

3.1 PREMO Part 1 definitions

This part of ISO/IEC 14478 makes use of all terms defined in ISO/IEC 14478-1 (Fundamentals of PREMO).

3.2 PREMO Part 2 definitions

This part of ISO/IEC 14478 makes use of all terms defined in ISO/IEC 14478-2 (Foundation component).

3.3 PREMO Part 3 definitions

This part of ISO/IEC 14478 makes use of all terms defined in ISO/IEC 14478-3 (Multimedia systems services component).

3.4 Additional Definitions

For the purposes of this part of ISO/IEC 14478, the following definitions apply.

3.2.1 modeller: a virtual device that produces primitives on at least one output port.

3.2.2 renderer: a device that accepts primitives on at least one of its input ports.

3.2.3 media engine: a virtual device that accepts primitives from at least one of its input ports, and produces primitives on at least one of its output ports.

3.2.4 presentation: a collection of primitives that can be perceived by the operator.

3.2.5 coordinate: a primitive used to define a location in an nD space.

3.2.6 primitive: a structure describing information to be rendered, or information received through interaction.

3.2.6.1 form primitive: a primitive whose presentation has to be constructed by a renderer from an explicit description in terms of aspects or properties that characterize a class of perceivable representations.

3.2.6.1.1 geometric primitive: a form primitive used to define a shape or extent within nD space.

3.2.6.2 captured primitive: a primitive for which some or all of the perceivable aspects of the primitive have been encoded in some format defined externally to the PREMO standard.

3.2.6.3 structured primitive: a primitive that contains a collection of other primitives.

3.2.6.4 modifier primitive: a primitive that describes a change to the presentation of another primitive.

3.2.6.4.1 acoustic modifier: a modifier that changes properties of the sound generated by other primitives.

3.2.6.4.2 structural modifier: a modifier that affects the spatial and/or temporal aspects of another primitive.

3.2.6.4.3 visual modifier: a modifier that affects the (non structural) visual appearance of a primitive.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

3.2.6.5 wrapper primitive: a primitive that carries a value drawn from some PREMO non-object data type.

3.2.6.6 tracer primitive: a primitive that carries an event for use in monitoring and coordinating the transfer of media data across a network.

3.2.7 input device: a device used to obtain data from the operator.

3.2.8 graphics: the construction, manipulation, analysis and presentation of pictorial representations.

3.2.9 scene: a device for storing and controlling access to a collection of primitive structures.

3.2.10 primitive structure: a collection of primitives organized into a structure that represents some or all of the data that describes a multimedia presentation.

3.2.11 coordinator: a MRI device that can manipulate the streams connecting its components.

3.2.12 router: a device for controlling the flow of data between streams connected to its ports.

3.2.13 temporal extent: the duration in time allocated or used for the presentation of some primitive.

3.2.14 operator: the external object that sends or receives information through a virtual device interface.

3.2.15 application: the external object or client that uses a PREMO system by creating objects, invoking operations on objects, and using types defined by PREMO. Applications are not modelled in the PREMO system, but their interactions with a PREMO system are modelled.

The following alphabetical list gives the sub-clause of each definition.

acoustic modifier	3.2.6.4.1
application	3.2.15
captured primitive	3.2.6.2
coordinate	3.2.5
coordinator	3.2.11
form primitive	3.2.6.1
geometric primitive	3.2.6.1.1
graphics	3.2.8
input device	3.2.7
media engine	3.2.3
modeller	3.2.1
modifier primitive	3.2.6.4
operator	3.2.14
presentation	3.2.4
primitive	3.2.6
primitive structure	3.2.10
renderer	3.2.2
router	3.2.12
scene	3.2.9
structural modifier	3.2.6.4.2
structured primitive	3.2.6.3
temporal extent	3.2.13
tracer primitive	3.2.6.6
visual modifier	3.2.6.4.3
wrapper primitive	3.2.6.5

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4 Symbols and abbreviations

CGRM:	Computer Graphics Reference Model.
CSG:	Constructive Solid Geometry
GKS:	Graphical Kernel System.
IEC:	International Electrotechnical Commission.
ISO:	International Organization for Standardization.
MPEG:	Moving Picture Experts Group.
MRI:	Modelling, Rendering and Interaction
MSS:	Multimedia Systems Services
PHIGS:	Programmers Hierarchical Interactive Graphics System.
PREMO:	Presentation Environment for Multimedia Objects.
VRML:	Virtual Reality Modeling Language.
nD:	Multi-dimensional.
2D:	Two-dimensional.
3D:	Three-dimensional.

5 Conformance

A conforming implementation of the PREMO Modelling, Rendering and Interaction Component shall comply with the general conformance rules defined in clause 5 of ISO/IEC 14478-1 and the component specification in clause 16.

6 Overview of the Modelling, Rendering and Interaction Component.

6.1 Introduction

This clause presents an overview of the modelling, rendering, and interaction component (Part 4) of PREMO. It summarises the concepts defined in the document, and explains how these concepts contribute to the goals set out in the Introduction. More detailed descriptions of the concepts used in the overview are given in subsequent clauses. This part of ISO/IEC 14478 also makes extensive use of facilities provided by PREMO parts 1-3, in particular the device and stream concepts introduced in ISO/IEC 14478-3. A summary of these dependencies is included in this clause.

6.2 Overview

The model underlying this part of ISO/IEC 14478 is that a multimedia system consists of modellers, renderers, and other devices (some media specific) linked together via streams that carry data of a particular format. These concepts of stream and device are those defined in ISO/IEC 14478-3. A device consists of a processing facility, together with a number of ports through which it can accept input and produce output, using a format defined by the port. Figure 1 shows a high-level view of a (simplified) example system in which a graphical user interface is used to control parameters of an audio-visual presentation system. Rectangles on the sides of devices represent ports, and the thick lines between such ports represent media flow via streams. Thin lines represent other forms of interaction, for example operation invocation. See also Annex B for a table defining the symbols used in the figure. The system in Figure 1 consists of:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

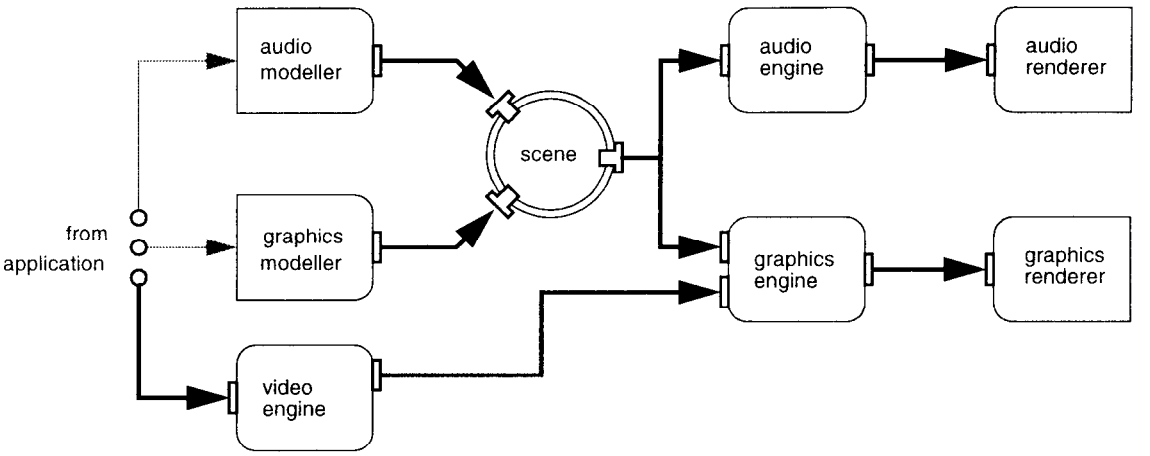

Figure 1 — An audio/visual system

— two modellers, one for audio data and one for graphical data, that might be used to construct and edit primitives via an interface to the application specific to the needs of each modeller;

— a video engine, accepting a stream of video data from the application and constructing a stream of information in some video format;

— a scene, which encapsulates primitives produced by the two modellers and which mediates access to the collection of primitives by both the modellers and the associated renderers;

— an audio engine, that takes primitives from the scene and converts into a data format that can be used to drive an audio-specific device;

— a graphics engine (acting as a mixer or composition tool) uses the video output from the video renderer and primitives from the scene to construct a further stream of primitives, integrating the two sets of source data within some appropriate visual representation. The primitives produced by the engine may be some subset suitable for input to a specific renderer;

— two renderers, one for audio and one for graphics, that convert a stream of primitives into a form that can be processed in the context beyond the MRI network (in this case, presented to the end user of the application).

The figure shows just one way in which such a system might be implemented. A different implementation may collapse the three engines into a single device, if it has access to a media engine that can take both graphical and audio primitives as input and that can generate the corresponding output streams. Another implementation may decompose the graphics engine into, for example, a number of components that manage specific functionalities such as viewing or clipping. Finally, a high-performance implementation might collapse all components into a single device.

Key components of this part are derived from the object types defined in ISO/IEC 14478-3, in particular the property inquiry and constraint facilities. These types and facilities may be used by the factory mechanism described in ISO/IEC 14478-2 to produce objects that meet certain requirements, and by the negotiation and QoS mechanisms for establishing and maintaining a network of objects that satisfies specific properties. As a result, many of the object types defined in this part have an associated list of properties for use in creation and negotiation. For example, modellers and renderers are derived (indirectly) from the *VirtualDevice* type of ISO/IEC 14478-3, and thus inherit the Property Inquiry services. Each renderer and modeller has a collection of properties that characterise its capabilities in terms of its inputs, outputs and quality of service. An application using the facilities of this part can request creation of a renderer from a renderer factory by invoking the '*create_object*' operation using the type 'renderer' as the name of the object type and passing a structure containing the required capabilities. Alternatively, an application that is aware that the rendering interface it requires is defined by a specific subtype of the renderer can request the factory to produce a renderer of that specific type.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

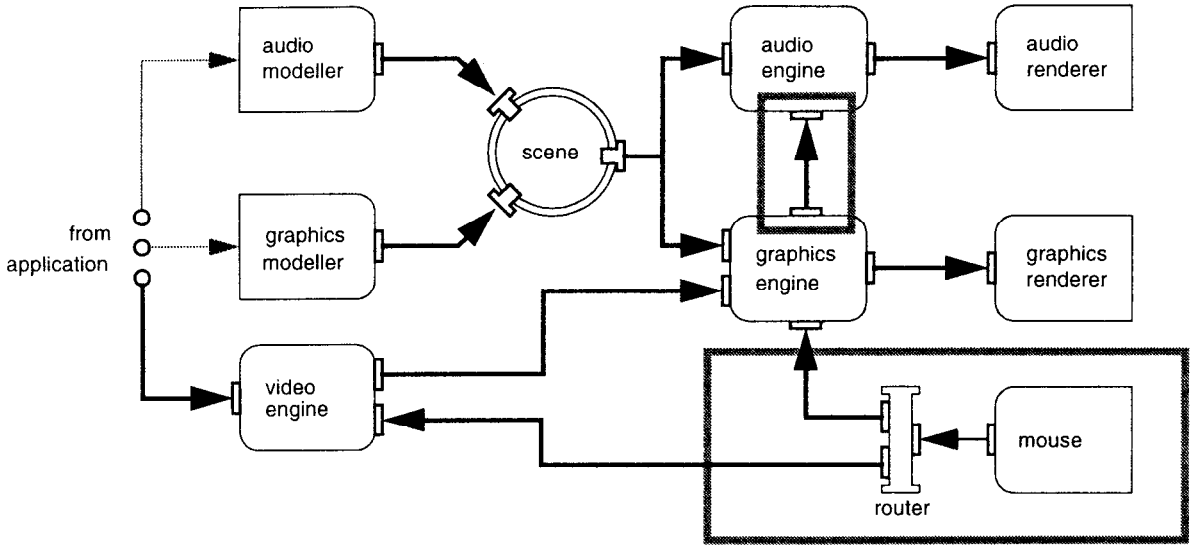

Figure 2 — MRI network including interaction handling

For simplicity, Figure 1 illustrates devices used in presentation only. The same approach of using specialised virtual devices connected by data streams is employed for handling input and interaction. Figure 2 extends the example with a simple framework for input handling. This introduces two new devices, and a new information flow between two renderers, shown in the outlined regions of the figure.

— The mouse is an example of an *InputDevice* that can provide primitives for processing elsewhere in the network, either via a stream (as shown) or through a procedural interface or callback mechanism.

— A *Router* can be incorporated to allow a data stream to be sent to specific devices depending on some internal state. The *Router* device achieves this by also subtyping from the *Controller* object defined in ISO/IEC 14478-2.

— Although *Engine* objects primarily operate on streams of primitives intended for presentation, as the example shows, an engine may also have ports that are used to receive (and in the case of the graphics engine, to transmit) primitives used to carry data about input.

Figure 1 and Figure 2 focus on the main streams and virtual devices involved in a simple MRI network. In addition to these, ISO/IEC 14478-3 provides object types for establishing and controlling a collection of streams and devices. These types are called *VirtualConnection* and *Group*, respectively.

An instance of *VirtualConnection* is an object that represents an abstract view of media transport between devices, allows control over aspects of the connection, and is responsible for negotiating the connection in terms of formats and quality of service considerations etc. Several kinds of virtual connections are possible, depending on whether the devices have compatible ports, and on whether a particular connection is local or networked. General examples of these are given in Annex D of ISO/IEC 14478-3. If the ports of two devices are not compatible, or the devices are in different parts of a distributed environment, *connection adaptors* will be employed. These adapters are an implementation concept, not visible to the application, and are not defined as an object type in the profile of ISO/IEC 14478-3.

Independent from issues of distribution and the existence of virtual connections, the *Group* object type provides applications with the ability to manage a collection of objects that are instances of *VirtualResource* or its subtypes (e.g., devices and connections) as a single resource. As *Group* is subtyped from *VirtualResource,* this arrangement can be hierarchical. Groups provides facilities to acquire the resources needed to establish a number of connections, to monitor the end-to-end quality of service, and to provide an application with a single access point for monitoring and controlling the flow of data across the resources that make up the group.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

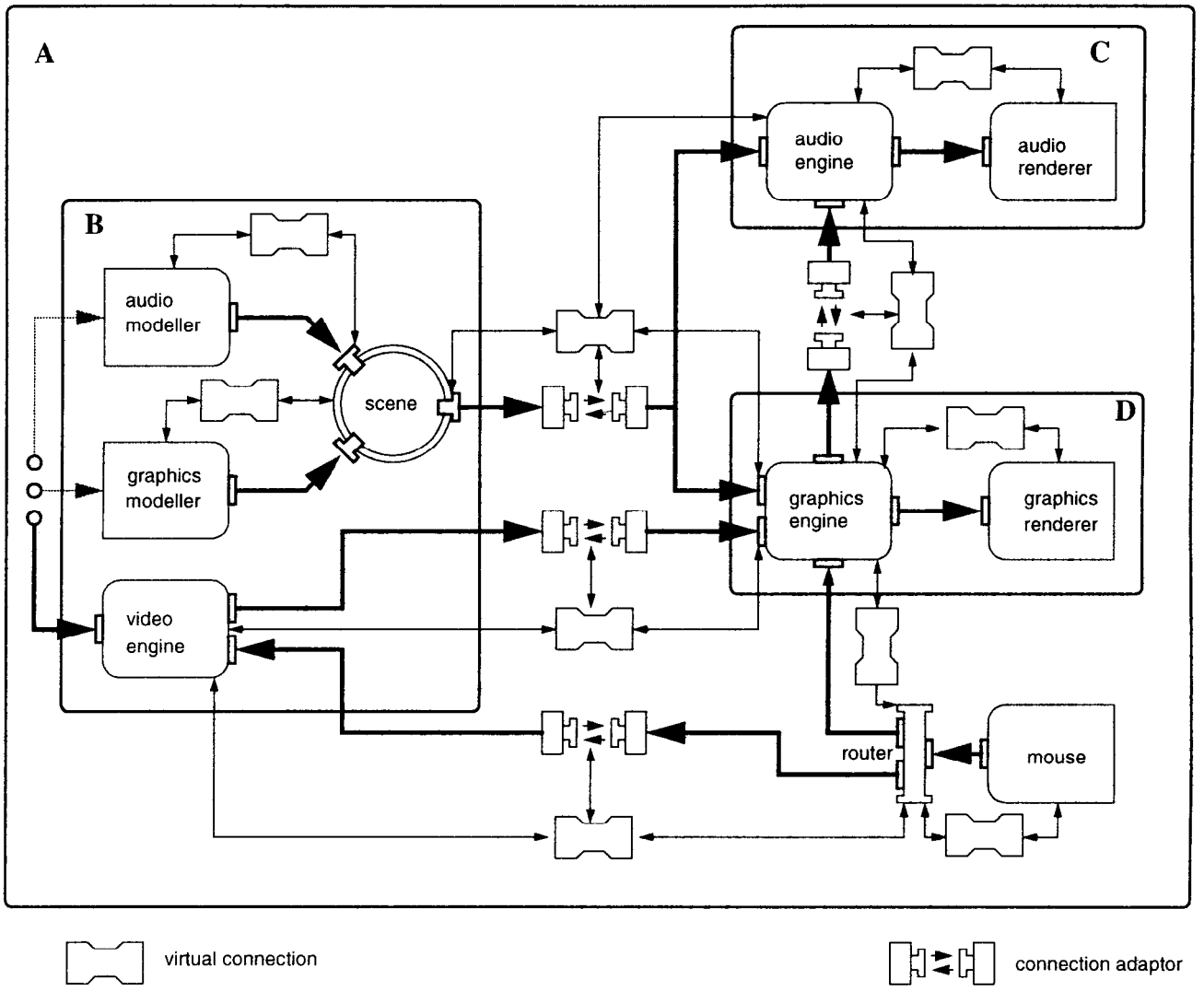

Figure 3 — Groups and connections in audio/visual system

Figure 3 shows one arrangement of groups could be used to implement the audio-visual rendering example. It also illustrates, for completeness, where connection adaptors may be required. In considering distribution, it has been assumed that the following sets of components are each located at separate nodes of a distributed system:

a) the modellers, the video engine, and the scene;

b) the audio engine and renderer;

c) the graphics engine, renderer, mouse, and router.

With respect to groups, there are a number of possible arrangements for the network. One such arrangement, consisting of four groups (labelled A-D) has been shown in the figure. In more detail,

— group A is the outer(most), and contains the other three groups, the mouse and router, and the connections and adaptors used to link devices in Groups B, C and D;

— group B contains the modellers, the video engine, and the scene, plus the virtual connections needed to link devices within the group;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

— group C contains the audio engine and renderer, and the virtual connection between them;

— group D contains the graphics engine and renderer, and the virtual connection between them.

Although these groups in the example are loosely based around the devices located at each node there is no assumption or requirement in PREMO that there should be such a correspondence. Groups are a logical concept, and in general are independent of the system organisation. Although users of this part of ISO/IEC 14478 may need to be aware of groups and connections and may wish to make use of them explicitly when constructing a MRI network, ISO/IEC 14478-3 also provides the *LogicalDevice* object types that can be used to hide much of this detail from an application. Furthermore, for applications that distribute presentation data (e.g. primitives, or other media-specific formats) to multiple renderers or devices, the use of *Coordinator* devices may also be necessary. The *Coordinator* object type is defined in this part of PREMO, and is a subtype of *MRI_Device*. Its role is described in 6.7.

The purpose of this overview is to illustrate some of the key connections between the object types defined in this part and those introduced by ISO/IEC 14478-1 to ISO/IEC 14478-3. In practice, the object types mentioned in the discussion will only be a subset of those utilized by an application. For example, in the audio-visual scenario, mouse input is passed along a stream to the graphics renderer, which may be providing immediate visual feedback on the changes made to certain control parameters (perhaps by adjusting the drawing of interface elements showing the volume, speed etc. of the audio and video playback). Other mechanisms, in particular event handlers and reference points, may also be present to control or mediate interaction between the components of the system, but have not been mentioned in this overview.

6.3 Devices for Modelling, Rendering, and Interaction

This part of PREMO introduces a hierarchy of devices for processing streams of media data constructed using objects based on the primitive hierarchy described elsewhere in this part (see 6.4). The root of this local hierarchy is the *MRI_Device* object type that extends the *VirtualDevice* object type defined in Part 3. This means that the devices defined in this part as subtypes of *MRI_Device* can be directly integrated into a network of more general devices that may include media-specific input and output devices as well as more abstract processing nodes. An example of such a network has been developed in 6.2. As virtual devices, *MRI_Device* and its subtypes contain a number of ports that allow either input or output of data in a particular format. The *Format* object type is introduced in ISO/IEC 14478-3, and a small subtype hierarchy for certain common media formats is presented as an informative annex (Annex C) of that part. The Modelling, Rendering, and Interaction component defines a subtype of the *Format* object type, called *MRI_Format*, that characterises a data stream that carries the primitives described in 6.4. Subsequent PREMO components, or applications, may specialize this format object type to define the input and output format of a MRI device that can utilise a richer collection of primitives. Figure 4 illustrates the relationship between the Format and *MRI_Format* object types, the examples of *Format* object types from Annex C of ISO/IEC 14478-3, and any future extensions for processing specialised streams of MRI primitives.

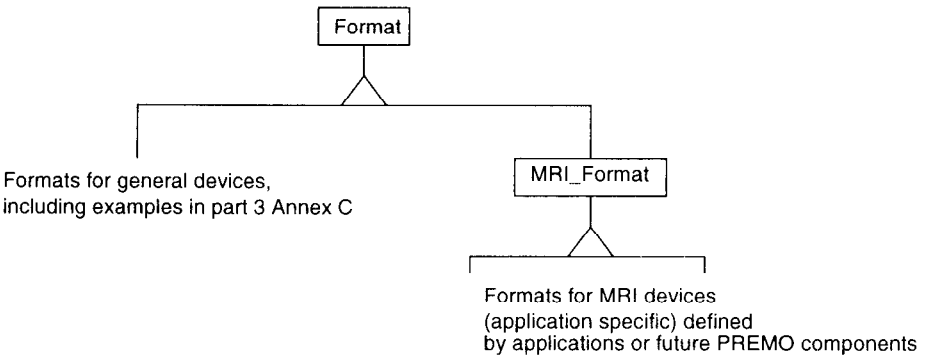

Figure 4 — Format hierarchy

This part defines a number of subtypes of MRI_Device as abstract object types for particular kinds of processing functionality. Three of these generalize the traditional computer graphics concerns, of modelling and rendering, to multimedia data processors. These devices differ in the kind of interface that they are required to support for input and output.

— A *Modeller* is a device that provides at least one output port that can support *MRI_Format*. In contrast, the input interface of a modeller is typically application-specific. A CSG modeller might offer operations that allow the construction and

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

manipulation of a CSG solid, while a modeller for music scores defined using audio primitives might have a direct manipulation interface through which an end user interactively constructs the score, which the modeller can then pass as a stream of primitives to other MRI devices for processing.

— A *Renderer* is a device that provides at least one input port that can use *MRI_Format* to accept primitives. The output interface of a renderer is typically application specific. Renderers include devices that output primitives directly to end users as a presentation (images, sound etc.) as well as devices that drive further software or hardware components of a system through some interface, the nature of which is not of concern to this part. This does not preclude a specific renderer from utilising PREMO functionality for its output mechanisms.

— A *MediaEngine* is both a *Modeller* and a *Renderer*. A *MediaEngine* is a device that offer at least one input port and at least one output port that are capable of operating with *MRI_Format*, allowing the engine to accept, process and produce streams of primitives. Media engines, like other MRI devices, may have additional ports that allow the device to accept or produce data in other formats.

NOTE — If a media format such as MPEG is to be taken directly as input by a renderer, as for example in the audio/visual system shown in Figure 1, then the raw input stream may need to be converted into a stream of primitives in which the MPEG data is carried within the primitive structure described shortly. This will allow additional information to be attached, such as a geometric data to characterise the position of the video data within a graphical scene. This transformation may either be a capability of a renderer, or may be realized by a specific device that acts as a translator or pre-processor. Such translators could be introduced by subsequent components, if any, as an object type derived from *MediaEngine*. A translator would have exactly two ports, one accepting a stream of an arbitrary format, the other generating a stream of primitives, with the semantics that the translator simply 'wraps' the media stream into an appropriate primitive object.

6.4 Primitives and Coordinates

6.4.1 Introduction

The MSS component, which provides the base object types from which MRI devices are derived, contains no statement on the content and structure of the data carried on streams and operated on by virtual devices. The contribution of the MRI component is to add just such information about the data used to realize multimedia presentation and interaction. This data, which is described by a hierarchy of PREMO object types, are called *primitives*. The organization of primitive types into categories or hierarchies varies widely in practice. This part of ISO/IEC 14478 defines primitive object types where it is intended that client applications, or any subsequent components that address the needs of specific domain areas or presentation technologies, extend this hierarchy in a way most appropriate to their specific needs. By providing a common set of basic primitive types, this part of PREMO is able to utilize the property and negotiation frameworks defined in ISO/IEC 14478-2 and ISO/IEC 14478-3 for establishing a network of MRI devices, and ensures that there is common vocabulary for basic MRI device capabilities.

6.4.2 Coordinates

Primitives within the PREMO hierarchy that address specific media such as graphics and audio are defined as abstract object types to ensure that they can adequately be specialized to the needs of specific application areas or implementations. As such, primitive object types do not define the structure of primitives in terms of coordinates within the various spaces (cartesian or other geometric spaces, properties of sound, etc.) through which primitives are specified in practice. However, to support inter-operability between implementations of the primitive hierarchy, this part of ISO/IEC 14478 defines a generic object type for coordinates within spaces of arbitrary dimensionality, subject to the constraint that the components representing each dimension of a coordinate are each drawn from the one PREMO type.

A specialization of this notion of coordinate, to the representation of colours within a colour model, is also provided by this part. A mapping from four colour models into the dimensions of the colour object type is provided.

6.4.3 The Primitive Hierarchy in PREMO

6.4.3.1 Overview

Primitives are structures that primarily carry information that is to be rendered to the user, or information that characterises input. PREMO does not attempt to describe a closed set of primitives for modelling and rendering. Instead, the approach of this part is to provide a general, extensible framework that provides a uniform basis for deriving primitive sets appropriate to specific appli-

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

cations or renderer technologies. In general, modellers may use specific techniques such as Constructive Solid Geometry or NURBS surfaces for a particular range of applications. 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, either by applications using PREMO, or by subsequent PREMO Components providing modellers and renderers to utilise a specialized set of primitives. For example, a component for Constructive Solid Geometry may enrich the primitive hierarchy with object types to represent a CSG tree, for example in the form of object types for basic geometric solids together with Intersection and Union types.

More specifically, the output generated by modellers and renderers, and the input to renderers, can consist of structures that are derived from a type called *primitive*. Examples of the information that might be carried by a primitive include:

— a polyline (in the sense of GKS and PHIGS);

— a Bézier curve;

— a triangular mesh;

— MPEG data;

— an audio waveform envelope located at a spatial coordinate;

— a location on the display (obtained by a PICK device).

At the top level, the primitive hierarchy consists of seven categories.

6.4.3.2 Captured Primitives

A captured primitive contains a reference to a source of raw data encoded in some standard format such as JPEG, MPEG, MIDI, or VRML. This data may happen to be recorded (for **example**, an image stored as a JPEG file, or a video stored as an MPEG file), or live, for example fed from a microphone device.

6.4.3.3 Form Primitives

This category represents primitives whose presentation has to be generated in some way, for example lines, curves, sounds, or textual information. This type is further classified into types representing specific kinds of information. The text below lists the different kinds of form primitive and provides some examples of their use - note that the examples do not necessarily correspond to subtypes defined in this part of ISO/IEC 14478. Rather, they illustrate how the hierarchy might be utilized in any subsequent components or applications of PREMO. Full details of the primitive types in PREMO are given in clause 8.

— Audio: information for generating sounds;

— Geometric: lines, planes, surfaces, curves, point data sets etc.;

— Tactile: a description of haptic properties;

— Text: character strings to be represented in some way.

6.4.3.4 Modifier Primitives

A modifier primitive carries information that affects the presentation of other primitives. Examples include visual effects (colour and texture), transformations on the coordinates that define the structure of media data, and audio effects.

— Acoustic: sound effects, and characteristics of voices;

— Structural; transformations on coordinates, such as scaling and rotation in geometric space, operations on time; constraints for use, for example, in geometric and/or temporal clipping are also subtypes of Structural modifier;

— TimeFrame: changing the clock against which progression of time is measured;

— Visual: properties of light, texture information etc.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.4.3.5 Reference Primitives

A reference primitive allows some part of a primitive structure to be reused, by providing a means of referring to that part of the structure in terms of strings carried by a *Name* object. *Name* objects are introduced into a presentation by structured primitives.

6.4.3.6 Structured Primitives

A 'structured primitive' is one that contains a collection of other primitives (which may including further structures), and a name by which that structure can be referenced. By naming structured primitives, it becomes possible to refer to part of a primitive structure from elsewhere in that structure, thereby facilitating reuse. Such names can also be used in application-dependent ways to provide feedback, for example on structures selected during interaction. PREMO identifies two kinds of structured primitive, each with a specific semantics:

— *Aggregate*: An aggregate is a combination of primitives in which information carried by some primitives is intended to modify the presentation of others in the aggregate. For example, a coloured line might be defined by aggregating a polyline (geometric) primitive with a visual colour modifier. Similarly, a sound effect might be applied to a recording referenced through a 'captured' primitive, while a polygonal mesh might be defined via some geometric primitive aggregated with specific colour, linestyle and linewidth modifiers.

— *TimeComposite*: Multimedia presentations exist in time, and may involve the coordinated presentation of different streams of data. A *TimeComposite* primitive defines the way in which the presentation of its components are to be related in time. PREMO defines three kinds of primitive that organise the components of *TimeComposite* to achieve specific effects.

- Sequential; one primitive follows another.
- Parallel; the temporal extents of the primitives overlap.
- Alternative; there is a choice between components, depending on the current state of a controller object.

NOTE — Any of the primitives that appears as a component of a structured primitive may itself be structured, i.e., may be a *TimeComposite* or *Aggregate* primitive. Consequently it is possible to construct complex multimedia presentations from simpler elements by organising them into a hierarchy that approximates the required pattern of coordination. Fine control over synchronization can then be effected through other mechanisms, such as the use of synchronization points.

6.4.3.7 Tracer Primitives

A tracer primitive carries an event object. When such a primitive is encountered at the input or output port of a device derived from the *MRI_Device* object type, the event handler attached to that port will be notified of the event carried by the primitive. The use of tracers in supporting coordination between devices is explained in clause 13.

6.4.3.8 Wrapper Primitives

This class of primitive allows arbitrary PREMO values to be carried as primitives, for example for use in handling interaction.

6.4.4 Primitives and MRI Devices

The semantics of the primitive hierarchy with respect to devices for media processing are defined in terms of the diagram in Figure 5. The two parts of the figure show part of a primitive hierarchy, and a renderer accepting a stream of primitives via a port with a format that is defined to accept certain kinds of primitive. A sample input and output stream is also shown, with the 'effect' of rendering indicated (for illustration) by the application of a pattern to the primitives. The renderer in Figure 5(a) accepts two types of geometric primitive, P1 and P2, that are defined as subtypes of *Geometric* in the corresponding type hierarchy. If the type hierarchy is subsequently extended as shown in Figure 5(b), by deriving new geometric primitives P3, P2.1 and P2.2, and the format attached to the renderer's input port is modified to allow the port to accept primitives of type P3, then the following behaviour will result.

a) primitives of type P1 and P2 will be rendered as in part (a) of Figure 5.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

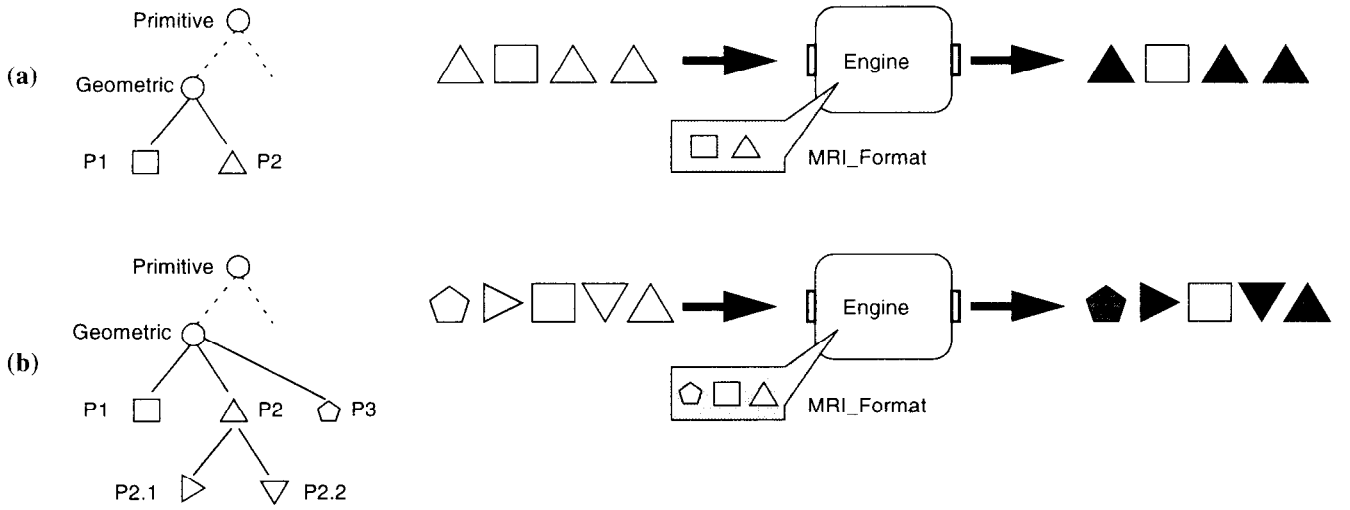

Figure 5 — Media devices and the primitive hierarchy

b) primitives of type P3 (pentagon) will be rendered in a way defined by the renderer, as these have been included explicitly in the list of primitive types that the renderer can accept via that port.

c) primitives of type P2.1 and P2.2 (drawn as rotated triangles) will be rendered as if they were primitives of type P2. These primitives are subtypes of a primitive type (P2, shown as a triangle) that the renderer can accept through the input port, but as the specific details of these subtypes are not explicitly known to the renderer, it will treat instances of P2.1 and P2.2 as instances of P2.

If a primitive *p* is presented to a *Renderer* or *MediaEngine* via a port, and the type of *p* is not a subtype of any of the primitive types that are defined for that port, then the device will raise an exception, *PrimitiveNotRecognised*. Apart from this exception, the behaviour of a MRI device on receipt of an unrecognized primitive is application dependent.

6.5 Scene

A simple PREMO application may realise a rendering network by connecting directly output ports of one device (for example a modeller) to the input ports of other devices, such as renderers. However, this simple strategy is not always appropriate, particularly in the kind of high-end application areas for which PREMO is intended. For example, there may be more than one modeller contributing primitives for presentation, or there may be multiple renderers at different locations that want to render a subset of the primitive structure. To support these needs, this part defines an object type called *Scene*, instances of which can be placed into a modeller/renderer network to act as a storage container and access mediator for a shared primitive structure. As a scene is also a kind of *VirtualDevice* it can be inter-connected freely with other devices in the modelling and rendering network.

The Scene object type provides operations that allow applications to construct, access, and delete a number of *primitive structures*. These structures are built by organising smaller primitives (or primitive structures) into larger units through the various structuring mechanisms provided by this part of ISO/IEC 14478 or by new mechanisms introduced by subsequent components. A primitive structure represents information that will form some or all of a multimedia presentation. Access to primitive structures within a scene is effected through the use of a *name*–valued attribute associated with the *Structured* primitive object type and its subtypes.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.6 Interaction

Interaction refers to the ability of system components to accept and respond to input. In terms of the CGRM (see Annex C), this is information that originates as input from the operator to the Realization Environment and is then sent towards the Construction Environment. The input can be processed at various points along the pipeline, for example to provide feedback, or can be passed to further components of the modelling/rendering network. Input data is realized as instances of object types in the *Primitive* hierarchy described in 6.4. That is, PREMO does not impose a separation between primitives used for output and those used for input. As this part does not impose any specific structure on devices for modelling, rendering and interaction, there is also no explicit provision of mechanisms for the control of interaction or feedback. Exactly how feedback can or should be handled will depend on the configuration of specific devices and the combination of modalities that are employed by a given application.

Interaction in PREMO is effected through the use of objects that are instances of *InputDevice*, an object type derived from *Modeller*. As an input device is also indirectly an *MRI_Device* and a *VirtualDevice*, it contains, in addition to a procedural interface, a reference to an event handler and ports for stream data. Through these various mechanisms, PREMO supports the construction of three fundamental approaches to input handling: event, sampled and request modes.

a) *Event mode*. A client that is interested in receiving input from the device registers itself with the event handler attached to the processing element of the device. This will generate an event when new data are available, and will pass the data to registered objects via events.

b) *Sampled mode*. A client is connected to the device by a stream which is initially muted. When the client wishes to sample input, the stream is put into play mode until the next item of data arrives; it is then returned to mute.

c) *Request mode*. A client invokes the operation request in the interface of the input device. This operation is synchronous, and therefore control is transferred to the device until the operation returns. The type of the datum returned by the operation is a subtype of *Primitive*.

That is, in 'event mode' the event handler is used to return input data, in 'sampled mode' the client utilises the output stream, and in 'request mode' the input data is returned as a result parameter of the *request* operation.

Management of event mode interaction is handled by the use of *EventHandler* object types and subtypes defined in ISO/IEC 14478-2, and no further mechanism is defined in this part. For sampled mode interaction, this part introduces the concept of a *Router* object to assist applications in coordinating the distribution of input data. A router is a virtual device that is also subtyped from *Controller*, and thus combines a collection of input and output ports with a finite state machine. Each state of a router is characterized by the connections from input ports to output ports. This assignment of data flow to states is dynamic, and can be changed by the application or client using operations in the interface of the *Router* object type.

6.7 Coordinators

The primitive hierarchy of PREMO, in particular the *TimeComposite* object type and its immediate subtypes, forms the basis of a declarative model of multimedia presentation content. That is, a collection of primitives, organised into a hierarchy using various kinds of *Structured* primitive, represents a static description of a presentation. For this presentation to be realised, i.e. displayed to a user or otherwise processed, its component primitives must at some point be operated on by a suitable device, i.e. a device that can accept this kind of primitive on its input port. Also, if the presentation consists of parallel components (in the form of a *Parallel TimeComposite* primitive, for example), then either a device has to be found that can manage parallel presentation, or two or more devices have to be used in parallel within a processing network. In the latter case, some mechanism is required to enable synchronization between these devices. These two requirements — the allocation of primitives to suitable media devices, and the scheduling and synchronization of parallel media content — are significant technical problems in the design of a multimedia system. While ISO/IEC 14478-1 to ISO/IEC 14478-3 do not mandate any one solution, this part of ISO/IEC 14478 defines an object type, called *Coordinator*, that can provide the behaviour described above.

The *Coordinator* object type is a subtype of *MRI_Device*. Each instance of this type has exactly one input port on which it can receive a stream of primitives in *MRI_Format*. Its interface contains operations that allow a specified device to be added to or removed from the set of devices that the coordinator can use in processing its own input stream. These facilities are similar to those provided by the *Group* object type defined in ISO/IEC 14478-3, but are restricted in comparison with those of the *Group* object type. Also, when a device is added to a coordinator, it is necessary to specify the input port of the device to which the coordinator is to direct appropriate primitives. For these reasons, *Coordinator* is not a subtype of *Group*.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

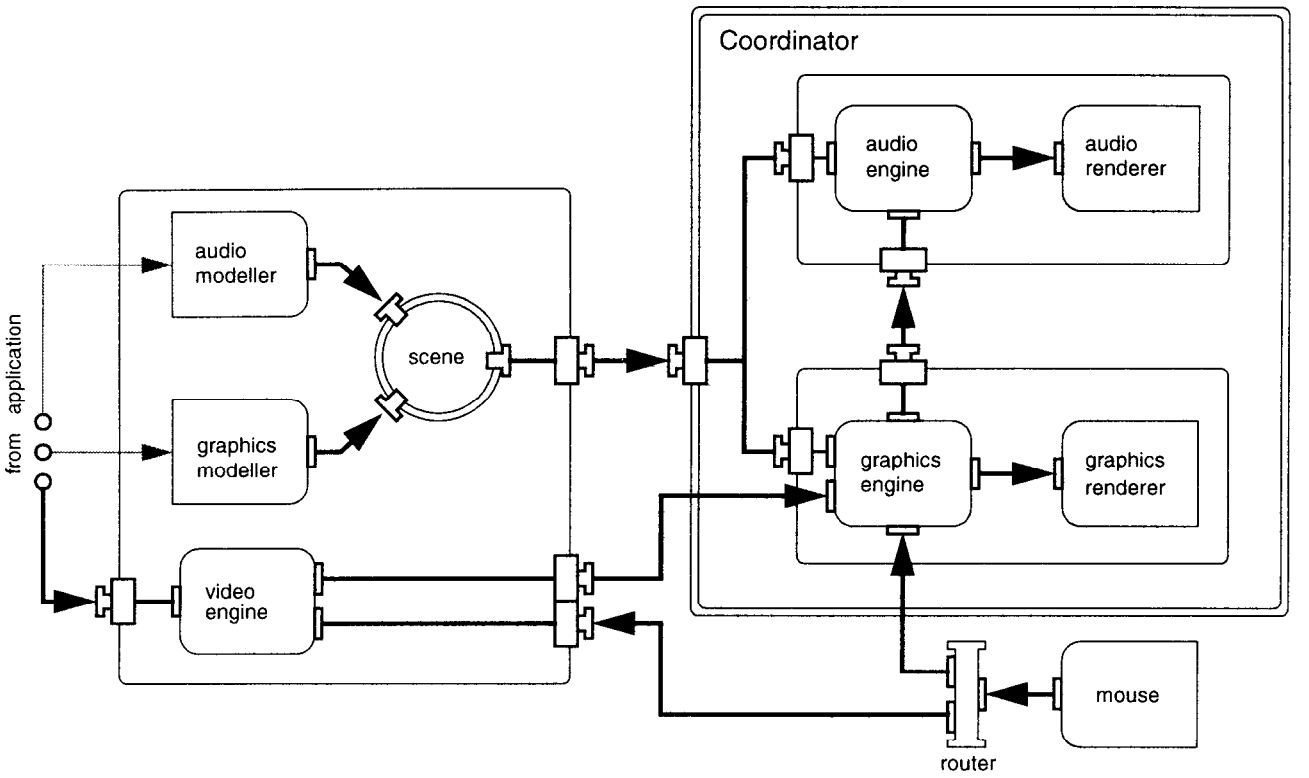

Figure 6 — Audio/visual system using logical devices and coordinator

The devices that can be added to or removed from a coordinator are required to be subtypes of *Renderer*. However, functionality that is not directly available within a single *Renderer* can still be utilised by a coordinator provided it is encapsulated in a type that inherits from *Renderer*. For example, an application that wishes to make a network of devices available to a coordinator could define a new type that inherits from *Renderer* and the *LogicalDevice* object type defined in ISO/IEC 14478-3. The operations inherited from *LogicalDevice* would allow the network to be established and viewed as a single device, while inheriting from *Renderer* ensures that objects of this new type can interoperate with a coordinator. Figure 6 shows how, by using such an approach, the system first illustrated in Figure 1 might be realised using three devices derived from *LogicalDevice* (labelled A, B, and C), and an instance of *Coordinator*. The coordinator encapsulate the two devices concerned with rendering, which will assume are also subtypes of *Renderer*. The presentations accessed from the scene object arrive at the coordinator, where they are decomposed into components suitable for the audio engine and renderer, and for the graphics engine and renderer. Note that the graphics engine can obtain data directly from another device, by–passing the input port of the coordinator.

As a coordinator has access to the devices to which it distributes media primitives, it has the potential to manage synchronization between these devices by manipulating the *StreamControl* object associated with the processing element of each device. However, other approaches are possible, either as alternatives to, or in parallel with, this scheme. For example, a coordinator could use an *ANDSynchronizationPoint* object, and reference points on the input streams to its component devices, to enforce synchronization constraints. The could also utilise *Tracer* primitives to monitor the progress of media data through the various processing elements. Given the range of possible synchronization mechanisms, ISO/IEC 14478 does not mandate that a coordinator implement any specific scheme. Instead, it is expected that particular synchronization strategies will be realized by subtypes of the *Coordinator* object type defined by specific applications or any future PREMO component that extends this part of ISO/IEC 14478. Further details on the role of the *Coordinator* object type in media synchronization can be found in clause 15.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.8 Dependencies on other Parts

This part of ISO/IEC 14478 builds extensively on the services and object types defined in other Parts of the standard. At a definitional level, it utilises the object model and notational conventions described in part 1. The creation and operation of an MRI system is also expected to utilise the Property Management services and Factory mechanisms set out in part 2. However, there are also some specific dependencies, where object and non-object data types defined in ISO/IEC 14478-2 and ISO/IEC 14478-3 have been used directly in defining the types and services of this part. These specific dependencies are listed below.

— The *Controller* object type defined in ISO/IEC 14478-2 is used as a supertype for the *Router* object type. Each *Alternate* primitive also contains a reference to a *Controller* instance.

— A *TimeComposite* primitive contains a reference to an instance of the *EventHandler* object type defined in ISO/IEC 14478-2.

— A *TimeFrame* primitive contains a reference to an instance of the *Clock* object type defined in ISO/IEC 14478-2.

— Instances of the *Event* structure defined in ISO/IEC 14478-2 are created by *TimeComposite* objects during processing.

— The *Time* non-object data type defined in ISO/IEC 14478-2 is used in *Temporal* and *TimeComposite* primitives.

— The *Port* non-object data type defined in ISO/IEC 14478-3 is used in the definition of *Captured* primitives, and in the definitions of the *Scene* and *Router* object types.

— The ISO/IEC 14478-3 *Format* object type is the superclass for the *MRI_Format* type.

— The *VirtualDevice* object type defined in ISO/IEC 14478-3 is the supertype of the *MRI_Device* object type defined in this part.

— The *Primitive, Coordinate, EfficiencyMeasure* and *Name* object types defined in this part are immediate subtypes of the *SimplePREMOObject* object type defined in ISO/IEC 14478-2.

6.9 Subtyping Diagram

Figure 7 on page 24 gives an overview of the object type hierarchies introduced in this part, and in particular their relationship to the object type hierarchy defined in ISO/IEC 14478-1 to ISO/IEC 14478-3. Object types introduced in this part are shown with their names enclosed in boxes. The hierarchy of object types subtyped from *Primitive* has been elided from this diagram, but can be found in Figure 8 on page 26. Also, full object-type diagrams can be found in Annex A. Note that the *Router* and *MediaEngine* object types are defined through multiple inheritance.

7 Coordinates

7.1 General Coordinates

PREMO is concerned with audio and video, as well as with synthetic graphics, tactile or other media. These data may be located within specialised coordinate spaces (e.g. frequency, pressure, etc.) and thus it is necessary for this part of PREMO to include a general, extensible representation of coordinate. To this end, a generic *Coordinate* object type is defined with the following interface:

— a dimensionality (read only), i.e. the number of dimensions;

— a *getRange* operation which, for a given dimension, returns the range of values allowed in that dimension; and

— *setComponent* and *getComponent* operations which access and update the *i*th component of the coordinate, where $1 \leq i \leq$ *dimensionality*.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

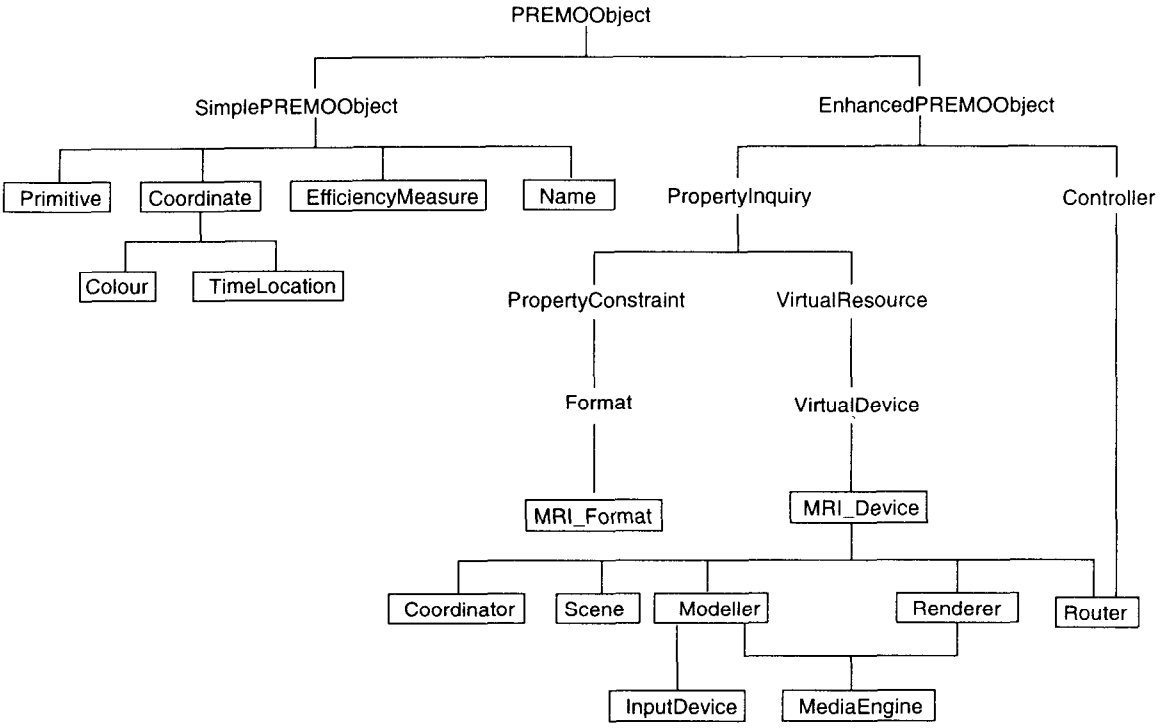

Figure 7 — Subtyping diagram

The coordinate object type is generic to allow the types from which components of coordinates are drawn to vary. However, for any given coordinate object, all components of that coordinate will be from the actual type used to instantiate the generic parameter. That is, each component of a coordinate must belong to the one type.

NOTE — To support efficient implementation of coordinates, the effect of accessing (via *setComponent* and *getComponent*) a component of a coordinate outside the dimensionality of that coordinate is not defined. Similarly, this part of ISO/IEC 14478 does not require that the value stored in the component of a coordinate be checked against the specified range for that component. Such checks can be implemented by subtyping the Coordinate object type and redefining the operations, for example to raise an exception when components or values outside of the specified range are supplied as parameters.

7.2 Colour

This part of PREMO defines an object type, *Colour*, as a subtype of *Coordinate* specialized to represent colours within a range of colour models in use. In this object type, the dimensions of the generic coordinate are mapped onto the attributes that locate a colour within a particular colour model. A string-valued attribute *colourModel* (read only) is defined in the *Colour* object type to name the colour model in which that colour is defined. The use of a given colour model imposes an interpretation on the dimensionality of a colour object and on each of the components of that object. To support such interpretations, a number of integer constants are defined to identify the component of a colour object that is used to store particular characteristics of a colour:

Constant	Value	Description
ColourRGBR	1	Red component of RGB
ColourRGBG	2	Green component of RGB
ColourRGBB	3	Blue component of RGB

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Constant	Value	Description
ColourHSVH	1	Hue component of HSV
ColourHSVS	2	Saturation component of HSV
ColourHSVV	3	Value component of HSV
ColourHLSH	1	Hue component of HLS
ColourHLSL	2	Lightness component of HLS
ColourHLSS	3	Saturation component of HLS
ColourCIEL	1	L component of CIELUV
ColourCIEU	2	U component of CIELUV
ColourCIEV	3	V component of CIELUV

The combination of constants that should be used in setting and getting the components of a colour object depend on the value of the colourModel attribute. The following combinations are defined by this part of PREMO:

colourModel	Description	Constants Used
RGB	Red-Green-Blue	ColourRGBR, ColourRGBG, ColourRGBB
CIELUV	1976 CIE Uniform Colour Space	ColourCIEL, ColourCIEU, ColourCIEV
HSV	Hue, Saturation and Value	ColourHSVH, ColourHSVS, ColourHSVV
HLS	Hue, Lightness and Saturation	ColourHLSH, ColourHLSL, ColourHLSS

The colour models mentioned in this sub-clause are those described in the ISO/IEC PHIGS standard.

7.3 TimeLocation

Time in PREMO is treated as a one-dimensional coordinate space. A *TimeLocation* object represents a point in this space. That is, an instance of this type is a coordinate with a dimensionality of one. That is, *TimeLocation* is defined as a subtype of the *Coordinate* object type. Values in the coordinate space for time are drawn from a particular PREMO non-object data type called *Time* (see 16.2). The value recorded within a *TimeLocation* object represents a number of ticks as measured with respect to some clock. The fact that time in PREMO is measured relative to a clock is important, as in a distributed setting different clocks may be available, offering varying degrees of accuracy and with different concepts of the current time. Consequently, the full interpretation of a given *TimeLocation* object depends on the context in which it occurs. For example, *TimeLocation* objects used in the context of a *Structured* primitive (see 8.6) may be related to a specific clock through the use of a *Temporal* modifier primitive (see 8.4.4).

8 Primitives

8.1 Introduction

Primitives are the means by which an application defines the structure and appearance of the data that is to be rendered for presentation. In some contexts, for example some graphics standards, the term 'output primitive' is used to distinguish between the information that determines the presentation, and information returned by input devices (input primitives). In PREMO, the one kind of entity is used for both roles. It is not uncommon for graphics systems to reserve the term (output) primitive for data that determines the structure of presented information, and to use the term 'attribute' or 'property' for data that affects the appearance of the presented data, for example colour or line thickness. In PREMO, the concept of primitive encompasses the description of both structure and appearance.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

PREMO is concerned with the presentation of multimedia information, and in allowing different renderers to interoperate within a potentially distributed system. For this reason, this part of ISO/IEC 14478 does not attempt to define the structure of primitives to the same level of detail as found for example in graphics standards such as GKS and PHIGS. Instead, this part of ISO/IEC 14478 defines a broad 'primitive hierarchy' that is intended to characterize the types of information that components of a PREMO system might produce or consume. A PREMO application may define a network of devices, including modellers and renderers, that utilize fundamentally different representations and methods. Consequently, it is not possible or desirable to define a canonical hierarchy. The primitive hierarchy described in this Clause is instead intended to provide a minimal framework through which MRI devices operating within a PREMO application can identify their properties and capabilities, and which applications can use as a starting point for defining, through inheritance, a set of primitives suited to that domain.

Primitives are structures, that is, the object type *Primitive* inherits from *SimplePREMOObject*. At the top level PREMO distinguishes between seven kinds of primitive; as shown in Figure 8 these are *Captured*, *Form*, *Tracer*, *Modifier*, *Reference*, *Structured* and *Wrapper*. A detailed account of each of the types shown in the hierarchy is given later within this clause.

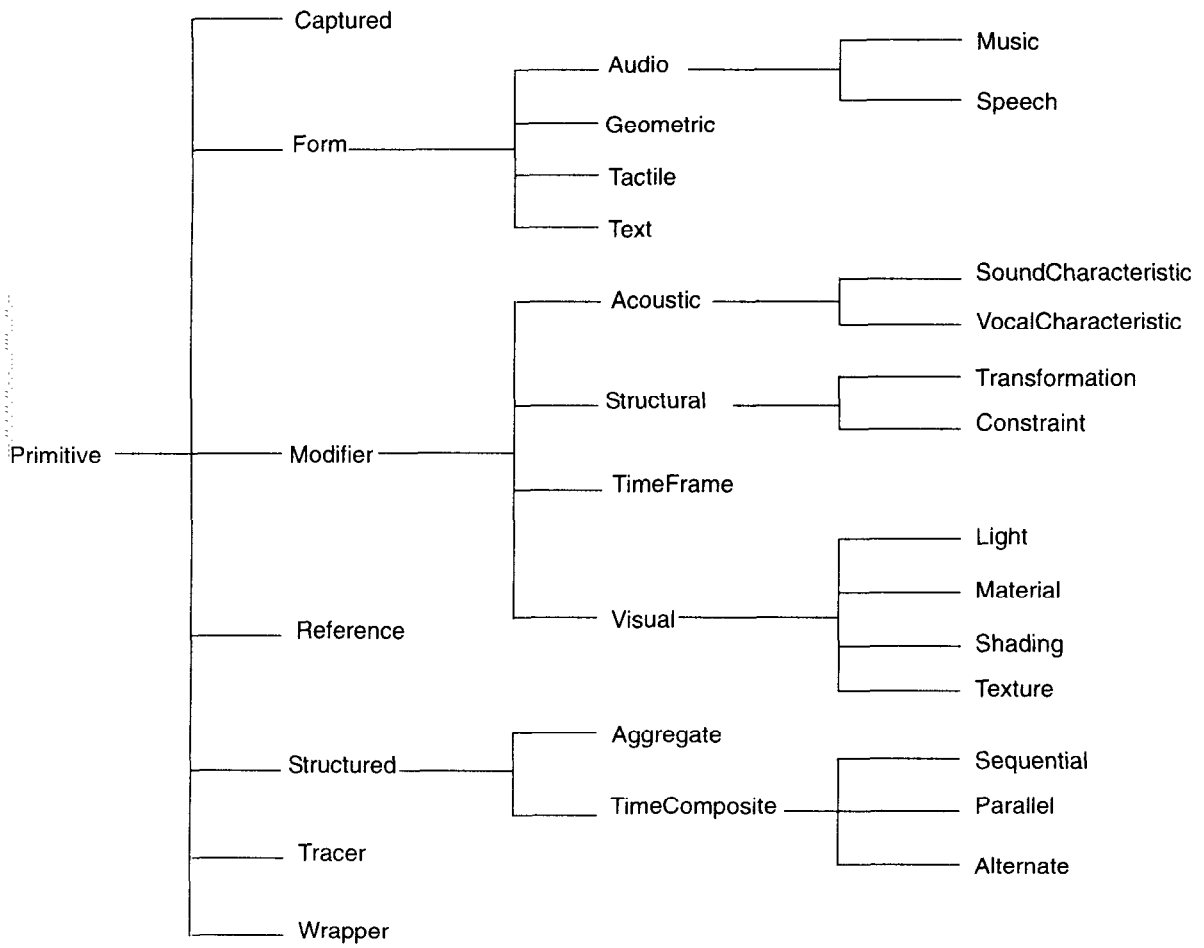

Figure 8 — The PREMO primitive hierarchy

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.2 Captured Primitives

Captured primitives are those for which some or all of the perceivable aspects of the primitive have been encoded in some format defined externally to the PREMO world. Rather than being synthesized, the data for presentation will be obtained from some other component of the system. In the context of this part, access to such data is supported via the port and virtual device mechanism inherited from ISO/IEC 14478-3. Thus, a captured primitive consists of a reference to a virtual device, and a reference to a port of that device from which the data can be obtained. As virtual devices include both data files and input devices such as microphones and cameras, the captured data may be either recorded or live. The format of the captured data can be determined by enquiring the port format, and could encompass:

— Recorded sound, for example in AIFF format.

— A digitized image, for example in JPEG or GIF format.

— A video feed as an MPEG stream.

— A metafile in some defined format, for example VRML.

8.3 Form Primitives

8.3.1 Introduction

A *Form* is a primitive for which the presentation is determined (constructed) by a renderer based on the information contained in the primitive, in contrast with captured primitives where the structure of the presented information has already been encoded in some transfer format. Forms describe structures in visual, audio, haptic or temporal space using the abstractions that characterise the space. For example, geometric primitives are described in terms of spatial coordinates. Additional kinds of form primitives may be added in future to include other categories such as olfactory and taste.

8.3.2 Audio Primitives

Audio primitives are those needed to represent sound characteristics. By using *Aggregate* and *TimeComposite* objects, more sophisticated sound characteristics can be described, for example by combining a number of audio primitives and acoustic effects (see 8.4.2) into a score. Audio information is located in several parts of the primitive hierarchy shown in Figure 8; a condensed version highlighting audio information is shown in Figure 9, with the primitives of concern in this clause shown in bold.

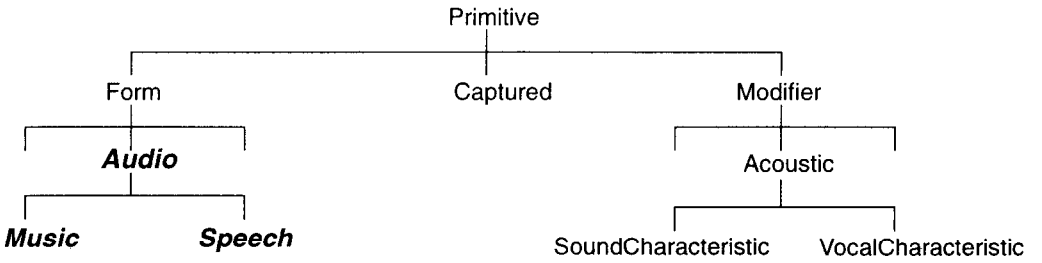

Figure 9 — Audio primitives within the PREMO hierarchy

Synthesized sound is described using some abstract representation that operates in terms of the constituents of the sound. This varies depending on whether it is speech or music being described. In the case of music, a typical example of a representation used in synthesis is MIDI. A music primitive contains information about the kind of instrument to be used in realising the sound, plus the data that represents the encoding of the music. Speech on the other hand consists of some textual representation of the words to be uttered, and a means of describing the characteristics of the voice that should be used to render the text. The latter is discussed in 8.4.2. The issue of which language should be used to render the text is unspecified at the level of this part of ISO/IEC 14478.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

8.3.3 Geometric Primitives

Geometric primitives describe locations and structures in multidimensional space, usually (but not necessarily) within a cartesian 2D or 3D system.

NOTE — Although this part of PREMO does not provide object types for representing specific geometric structures, it is expected that any subsequent components or applications will create subtypes of *Geometric* to represent concrete primitives from which a renderer may generate a presentation, the nature of which will depend on the semantics assigned to that subtype by the renderer. For example, both GKS and PHIGS contain Polyline and PolyMarker primitives that are defined by sequences of coordinates. A polyline is rendered as a sequence of line segments, starting from the first coordinate, and passing through each point in the primitive in turn. A polymarker primitive is usually rendered by placing some symbol at each coordinate within the primitive. Both of these would be defined by subtyping from the *Geometric* primitive. More complex geometric primitives, such as polygonal meshes, may also be defined by starting from a model of geometric structure consisting of a sequence of coordinates, and then imposing additional structure (and a different interpretation) on this sequence through the use of inheritance. In this context it should be noted that geometric primitives may be generic with respect to the coordinate system in which their locations are defined. In particular, applications that require additional information to be associated with locations, for example a normal, could be handled by deriving a extended form of *Coordinate* and using that as the basis for a geometric structure.

NOTE — Within this part of PREMO, primitives do not contain operations. Thus, although it would be in keeping with an object-oriented approach for geometric primitives to offer services such as 'compute bounding box' and 'apply clipping', the model adopted in this part assumes that these operations will be provided by some other object such as a renderer, that operates on primitives. One rationale for this is that a typical graphics application will generate large numbers of primitives. If primitives were required to contain operations, then their use in a distributed environment, where multiple processors may be operating on shared data sets, becomes problematic. However, any subsequent component, or application using PREMO, for example a package for object-oriented graphics, may of course choose to locate such functionality in the definition of its primitives.

8.3.4 Tactile Primitives

Tactile primitives describe parameters of touch-based interactions, for instance, temperature, thermal conductivity and hardness.

8.3.5 Text Primitives

A *Text* primitive contains a character string that is to be rendered on some display. This part of ISO/IEC 14478 makes no statement about properties of the text such as font, size, style or direction. Subsequent components can realize these in at least two ways. One is by extending the text primitive using subtyping to represent these properties as attributes of the primitive. The second is to define a subtype of the *Modifier* primitives described in 8.4, and then to combine these with the raw text primitive using the aggregation mechanism.

8.4 Modifier Primitives

8.4.1 Introduction

The primitives contained in this category have no perceivable representation by themselves. Instead they are used to modify or transform the presentation of other primitives by combining them through the aggregation mechanism. The modifiers have been grouped to reflect the kind of effect they produce, and the kind of primitives to which they can be applied. PREMO does not describe the order in which modifiers are applied, and whether or not they are accumulative or override previous modifications. For example, in the two hierarchies shown in Figure 10, the standard makes no statement about whether Mod–A should be carried out before or after Mod–B, and in the case of the second hierarchy, whether in fact Mod–B over-rides the effect of Mod–A with

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

respect to P. The reason for this non-commitment is that applications or any subsequent components of PREMO may realize graphical rendering through existing systems and standards, within which the order and scope of modifications within the rendering pipeline or scene structures varies widely.

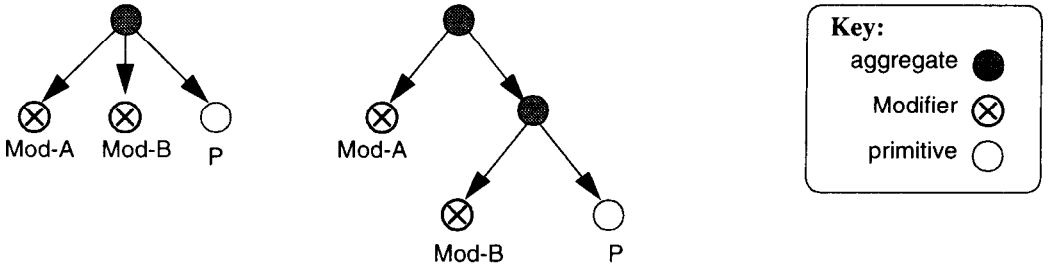

Figure 10 — Scope of modifiers

8.4.2 Acoustic Modifiers

Acoustic modifiers are not themselves sounds, but rather are modifiers that alter the presentation of captured or synthesised sounds. Two kinds of acoustic modifier are represented by abstract subtypes of *Acoustic*.

— A *SoundCharacteristic* is a modifier that is defined in terms of the physical characteristics of a sound, for example its amplitude, volume or properties of its waveform.

— A *VocalCharacteristic* is a modifier that applies to synthetic speech, and affects the way in which the constituents of a given speech object are realized. Examples of possible vocal characteristics include sex, age, intonation and dialect.

8.4.3 Structural Modifiers

Structural modifiers affect the interpretation of coordinate values representing, for example geometric structure or time, within some collection of primitives. When used in conjunction with geometric primitives, structural modifiers encompass both the 'standard' operations such as translation, scaling, rotation and shearing, and the definition of geometry that serves to constrain the appearance of other geometric primitives, for example clip regions. These two kinds of specialized structural modifier are identified explicitly through the provision of the following subtypes.

— *Transformation* objects, which include, but are not limited to, the common affine and projective transformations.

— *Constraint* objects, such as clipping, shielding, culling, level of detail or the definition of stencils.

It should however be noted that concepts of transformation and constraint can also be applied to non-geometric coordinates such as time, or colour.

NOTE — The structural modifier primitive and the given subtypes are abstract object types. The family of modifications relevant to an application will depend on the dimensionality of the primitives used. Different approaches to representing a structural modifier may also be used, for example an explicit matrix representation, or an implicit approach in which the modifier primitive stores parameters such as translation distance along each axis, scale factors etc. and it is up to the renderer to use this information to construct an internal matrix or carry out the operations in some other way.

8.4.4 TimeFrame Modifiers

All temporal primitives in PREMO are defined relative to some clock; this is important, as in a distributed setting different clocks may be available, offering varying degrees of accuracy and with different concepts of the current time. This part defines a single concrete modifier object type, *TimeFrame*, that contains a reference to a clock. The aggregation of a *TimeFrame* instance with other primitives allows a media processor to utilise the clock referenced by that instance when calculating or otherwise working with time units contained within the other primitives. Note that modifiers that affect other aspects of temporal structure, for example, transformations on time, are organised as *Structural* modifiers. The *TimeFrame* modifier is treated separately as it is independent of coordinate space.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.4.5 Visual Modifiers

Visual modifiers affect the appearance of geometric primitives by providing attributes such as reflection coefficients or material properties. The combination of visual modifier and geometric forms can either be effected by the use of aggregates, or by the use of multiple inheritance. Four specific kinds of visual modifier are identified in this part, as shown in Figure 11. These are all abstract object types.

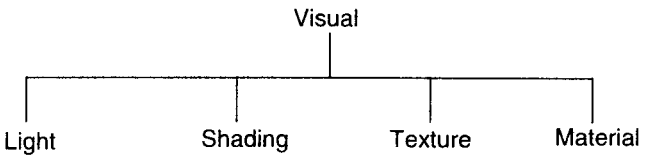

Figure 11 — The visual modifier hierarchy

a) *Light* is an abstract supertype for properties of light. PREMO makes no commitment to any specific lighting model, and it is up to any subsequent components or applications to extend this type in an appropriate manner.

b) *Shading* is an abstract supertype for object types that convey information about the shading model or parameters that should be used to render some or all of a primitive structure.

c) *Texture* is an abstract supertype for object types that define texture properties, for example a reference to another object that provides a texture map or bump map.

d) *Material* is an abstract supertype for object types that define properties such as translucency and transparency.

8.5 Reference Primitives

8.5.1 References

A *Reference* primitive introduces a link to a structured primitive defined in some other part of a primitive hierarchy. It contains a single attribute, *label,* which is a reference to a *Name* object that is intended to be matched against a name object either from some other part of the same structure or a second separate structure. The effect of matching on the subsequent processing of a primitive hierarchy is application dependent.

8.5.2 The *Name* Object Type

A *Name* object contains a single attribute which holds a sequence of strings, and a single operation, *equal*, that takes a name object as input and returns a boolean value that indicates whether the name passed as parameter should be considered equal to the name on which the operation is invoked. This part of ISO/IEC 14478 does not mandate rules for determining fully when two *Name* objects are equal, or should match when used as labels in *Structured* and *Reference* primitives. This is for the application or a subsequent component to determine. However, it is required that if the *equal* operation of a given name object is invoked with a reference to that object as input parameter, the result must be true. That is, any name object is always equal to itself. By default, the *equal* operator will implement set equality between the strings. That is, the sequence is considered to be a set of strings. For example, the string sequences <"alpha", "beta", "gamma"> and <"beta", "alpha", "beta", "gamma"> should be considered equal.

NOTE — For example, a GKS component may use the default, interpreting sequences as sets and ignoring duplicates and order when comparing name objects. However, extensions to the primitive hierarchy may also impose additional structure on the strings in the sequence, for example by treating them as paths as used in many filesystems.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

8.6 Structured Primitives

8.6.1 Introduction

Form, *Captured* and *Modifier* primitives can be viewed as atomic units of information that determine or affect the presentation of a multimedia system. Multimedia systems, however, need to define and manipulate collections of primitives, both to represent large-scale or application-specific structures, and to coordinate the presentation of primitives within time. These two roles are somewhat different, and are reflected in PREMO by two object types that encapsulate a collection of primitives. This collection (called the *components* of the structure) may itself include structured primitives, allowing the construction of hierarchical structures. At this level no constraint is placed on the primitives that can be components of a structured primitive. However, subsequent components, if any, may choose to define subtypes where membership of the structure is limited to objects of a specific type. One variation on this approach would be to define a generic subtype of structured primitive in which the type of the objects that make up the structure can be constrained.

Each *Structured* primitive has an attribute, *label*, that refers to a *Name* object (see 8.5.2). This allows the structure to be referenced from elsewhere within a collection of primitives, and is also used in operations of the *Scene* object type that are described in clause 13.

NOTE — Such names can be used to refer to some part of a primitive structure, either by an application or by the PREMO system. A particular use of names within this part is associated with the Scene object defined in clause 13. PREMO also provides a *reference* primitive to allow reuse of substructures within a primitive structure; see 8.5. The name attached to a structured primitive might also be used for returning pick-identification during interaction, or for structure editing, for example in a PHIGS renderer.

The object types in this sub-clause are defined at a greater level of detail than the other kinds of primitive objects introduced in this part. This difference reflects the role of PREMO as a framework for integrating the processing of digital media. It is not the role of ISO/IEC 14478 to prescribe the structure of data associated with specific media, but instead to provide a model in which media primitives can be organized and coordinated.

8.6.2 Aggregate

Aggregates allow a number of primitives to be combined into a structure without imposing any interpretation on the meaning of the structure. Their use in combining other primitives with modifiers has already been discussed (see 8.4), but their role is rather more general. They provide a facility for building larger-scale primitives, such as polygonal meshes, from smaller structures, and also allow an application to group semantically-related primitives into single units that can be named using its *label* attribute. In this respect they support the implementation of facilities such as segments in GKS and structures in PHIGS.

A property of aggregate primitives is that there is no distinction between their components; they are just a means of combining or grouping a number of primitives into a larger structure. This will not be sufficient in general, as there are different ways of interpreting an aggregate of primitives. For example:

a) The combination of a 3D point (a geometric primitive) and a MIDI file (a captured primitive) might be rendered by displaying the point on some display and playing the contents of the file. Alternatively, such a primitive may indicate that 3D rendering is to be used for the sound, with the point giving the source of the sound within some scene.

b) A polygon (a geometric primitive) and a further aggregate containing geometric features may define a structure that is intended for rendering on a display, or the polygon may define a region that is to be used for clipping the geometric presentation of the primitives in the second aggregate.

NOTE — It is expected that applications or any subsequent PREMO components will extend the primitive hierarchy by inheriting the aggregate object type into new types that associate a particular role or meaning to the constituents of the aggregate. Using the examples above as a basis, one application might define an extended hierarchy containing subtypes of aggregate for 3D sound and clipped polygon. While the former could also be realised by multiply inheriting from coordinate primitive and capture primitive, the latter contains two geometric primitives and therefore must be realized by aggregation.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

8.6.3 TimeComposite

Time and temporal extent are fundamental to multimedia presentation and in general a multimedia system will contain a number of primitives that need to be synchronized in time. Although time could arguably be treated in a way similar to that used for spatial coordinates, most multimedia implementations will typically treat time in a specialized way. For example, the timing of primitives may be adjusted dynamically to satisfy quality of service requirements, and synchronization requirements will be realized by placing synchronization points within streams or other time-synchronizable components of a system. Thus there is a degree of specialized functionality associated with time that is not reflected in other coordinate spaces, and the object types that manipulate temporal aspects of presentation must therefore have a standard and efficient means of accessing this information. This role is partially filled by the *TimeComposite* primitive.

The *TimeComposite* object type is a subtype of the *Structured* primitive object type, and therefore contains a sequence of component primitives. The meaning assigned to these components (that is, how their presentations are coordinated) depends on the kind of composition; PREMO recognises sequential, parallel and alternative *TimeComposites* whose meaning is defined below. *TimeComposite* objects do however have shared characteristics, in particular, each contains a reference to an event handler that will be notified when significant points in the structure of the *TimeComposite* are reached during presentation or other forms of processing. The points at which events are generated are defined by attributes of the *TimeComposite* object type and its subtypes.

Associated with each instance of a *TimeComposite* object (sub)type are the following attributes:

a) min and *max:* these two attributes are time values (i.e. numbers of ticks) that define a duration in which the content of the object, introduced through the *Structured* supertype, should be located. The clock against which this interval is measured is not specified by the primitive; it may be given as a *TimeFrame* modifier within a larger structure containing this primitive, or it may depend on the context in which the primitive is processed. This interval may be infinite, in which case an implementation is free to use as much time as necessary to process the components of the *TimeComposite*. It may however be bounded, in which case the devices involved may need to adopt a strategy for optimizing the presentation of components, possibly by making use of the overlap features in the case of synchronous *TimeComposite* objects, or more generally by degrading the quality of service provided. In the case that the required processing cannot be slotted into the specified duration, a device may take some application dependent action, for example raising an exception or simply terminating processing of the primitive abruptly.

a) *monitor*: a reference to the event handler that will be informed of progress as the *TimeComposite* is processed. As noted above, all *TimeComposite* objects generate events at the start and completion of the presentation of their components. Details of the events generated by *TimeComposite* objects and particular subtypes are included under each sub-clause.

b) *startTime* offset: this is a delay between the time that processing of the *TimeComposite* primitive commences, and the time at which processing of the first component commences. The time at which processing commences depends on the device that is performing the processing, and it is the responsibility of such a device to note the time at which it starts to process a primitive and to apply effects such as *startTime* as required, relative to the time that processing started. An event is generated once the *startTime* offset has elapsed. This event has the following structure:

eventName	=	"compStart"
eventData	=	< ("TimeComposite", *Ref TimeComposite*) >
eventSource	=	*Ref VirtualDevice*

The TimeComposite referenced by the *eventData* field is the *TimeComposite* being rendered, while the device referenced by the *eventSource* field is the device that is operating on the *TimeComposite*.

c) *endTime* offset: this is a delay between the time that processing of the final component of the *TimeComposite* is completed and the time at which processing of the *TimeComposite* itself should be completed. An event is generated on completion of the presentation of the components, and before the *endTime* delay begins. The structure of the event is similar to that in point (b), except that the *eventName* field takes the value "compEnd".

Both the start and end time offsets are specified as durations, meaning that they are stretchable.

Figure 12 shows the relationship between these attributes.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

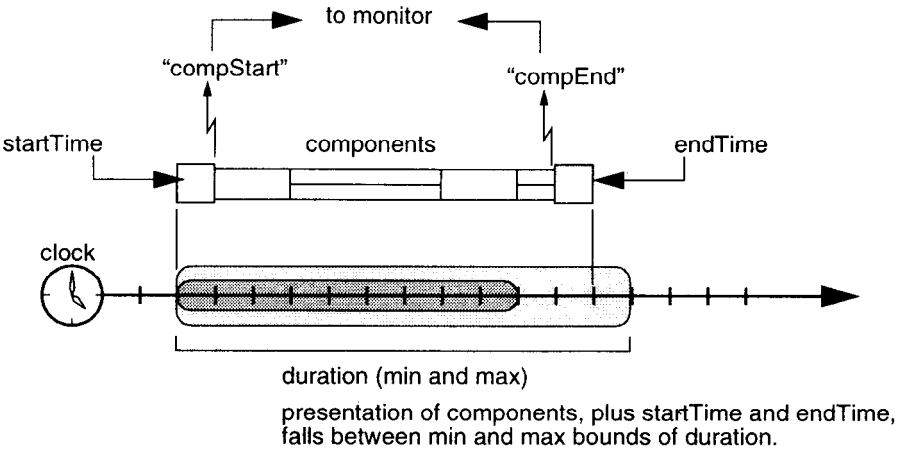

Figure 12 — Attributes of a TimeComposite primitive

The three kinds of TimeComposite primitive are as follows:

Sequential TimeComposite

Each primitive listed as a component is presented in the order in which it appears in the component sequence. Associated with this primitive object type are three attributes:

— *startDelta*: This is the offset between the time that the system begins processing a component, and the time at which the presentation of that component should begin. At the end of *startDelta* an event is sent to the event handler referenced by the inherited *monitor* attribute. The structure of the event is as follows.

eventName	=	"seqStart"
eventData	=	< ("sequential", *Ref Sequential*), ("position", integer) >
eventSource	=	*Ref VirtualDevice*

As before, the event data includes a reference to the *TimeComposite* (in this case a sequential *TimeComposite*), but here it also includes an integer giving the position in the component sequence of the primitive that is about to be processed. The event source is the renderer that is doing the processing.

— *endDelta*: This is the offset between the time that the presentation of a component primitive concludes, and the time at which processing of the next component should begin. An event is generated on conclusion of the component presentation, before the onset of the delay. The information carried in the event is the same as for *startDelta*, except that the event name takes the value "seqEnd".

— *overlap*: this attribute takes on one of three values, defined by the following non-object data type:

OverlapType ::= *never* | *left* | *right*

1) When the value is *never*, the presentation of one component must finish before processing the next component can begin.

2) When the value is *left*, the presentation of the previous component can be shortened if needed to fit the presentation of the current component. This will be achieved by reducing the *endDelta* time of the previous component, i.e. shortening the actual delay imposed for that primitive. This part of PREMO makes no statement about the behaviour that results if there is insufficient time available in the delay to ensure presentation of the current component.

NOTE — In the event of insufficient *endDelta* delay in the previous component to enable full presentation of the current, some possible behaviours include, but are not limited to: (i) the renderer may modify the presentation of the current component in an application specific way, or (ii) it may send a specific event to the *monitor*, or (iii) the renderer may raise an exception.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

3) When the value is *right*, the presentation of the next component can be shortened, if needed, to fully renderer the current component, by reducing the time allocated for the *startDelta* delay of that component. The caveats and comments of the previous point (2) also apply here.

The value of all temporal attributes is relative to the clock referenced in the inherited TimeComposite object type. Figure 13 shows the relationship between these attribute settings. Partial information is shown for two events, all of which are directed at the event handler named as the monitor for TimeComposite.

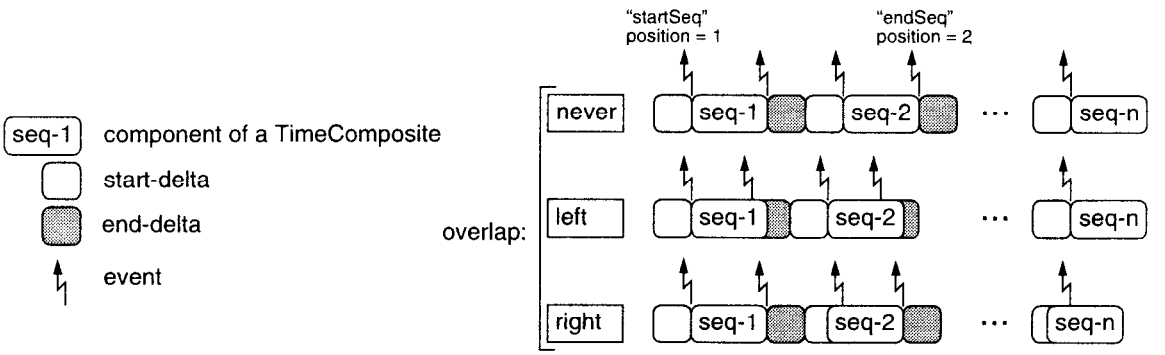

Figure 13 — Attributes of a Sequential TimeComposite primitive

Parallel TimeComposite

The presentations of all components of the *TimeComposite* occur concurrently, but are modified depending on the values of the following two attributes:

— *startSync* is a boolean value that indicates whether or not the presentation of all components of the *TimeComposite* must start at the same time.

— *endSync* is a boolean value that indicates whether or not the presentation of all components of the *TimeComposite* must end at the same time.

Together, these attributes define a space of four possible 'ideal' states for the coordination of parallel *TimeComposite* objects. A representative sample of these states is shown in Figure 14.

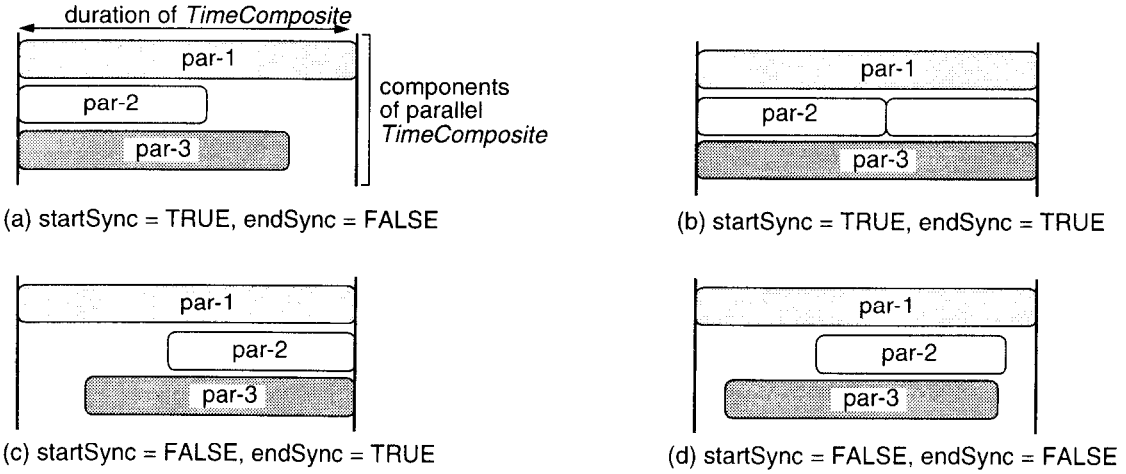

Figure 14 — Attributes of a Parallel TimeComposite primitive

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

There are four points to note.

— First, where the presentation of one component is shorter than that of another, the 'effect' of non-presentation is undefined. For example, if a graphics image is to be presented for a particular duration in parallel with a soundtrack, and the temporal extent of the soundtrack is longer than the given duration, then PREMO makes no statement about the appearance of the display between the times that the graphics duration and soundtrack end. Figure 14(b) shows two possibilities that an implementation might use - the primitive 'par–2' has been 'padded out' perhaps with a blank display in the case of a graphical presentation, while 'par–3' has been stretched, possibly by slowing the playback rate if it happens to be a video stream.

— Second, the temporal extent of a parallel *TimeComposite* primitive will be the smallest duration necessary to present each of its components subject to the constraints imposed by the attributes above. This means that the presentation of the longest component of a *TimeComposite* will begin as soon as the presentation of the *TimeComposite* itself begins.

— Third, if either *startSync* or *endSync* is *FALSE*, it means that the start, respectively end, of the presentations of the components do not need to be synchronized. It may however happen that they are synchronized, depending on the way that the primitive is handled by a specific renderer.

— Fourth, if *endSync* is *TRUE*, processing devices should attempt to ensure that the presentations conclude at the same time. It may be that satisfaction of this constraint cannot be guaranteed, in which case a value of *TRUE* for this flag is a signal that devices should make a 'best effort' to achieve this synchronization.

NOTE — If a component of a *TimeComposite* is another *TimeComposite*, then complex effects and coordination patterns can be achieved by manipulating attributes of the subelement such as its *startTime*, *endTime* and *duration*.

Alternate TimeComposite

One of the component primitives listed in the *TimeComposite* is chosen for processing and presentation. The choice of component is determined by the state of a controller object referenced by an attribute of the *TimeComposite* called *selector*. This choice is carried out at the time taken as the start of processing for the primitive itself (see the comment concerning the startTime offset attribute of the *TimeComposite* object type). The mapping between controller states, identified by strings, and component primitives (indexed by their position in the *components* sequence), is given by the attribute called *options*. This consists of a sequence of (string, integer) pairs, where a pair *(s, i)* means that *components(i)* should be rendered when the state of the controller object is *s*. One component of the *TimeComposite* may be linked to more than one state of the controller. The effect of having more than one component linked to a given controller state is undefined. The relationship between these attributes in summarised in Figure 15.

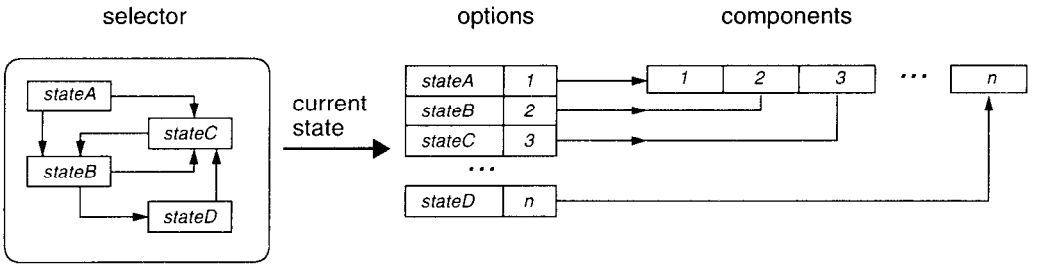

Figure 15 — Attributes of an Alternate TimeComposite primitive

NOTE — The PREMO *TimeComposite* hierarchy is analogous to the requirements set out in the HyperODA standard.

8.7 Tracer Primitives

A *Tracer* is a primitive that encapsulates an event as an attribute of the primitive. The *eventName* attribute of the event is set to "TracerEvent", and the *eventSource* attribute is set to reference the *Tracer* primitive object in which the event is contained. *Tracer* primitives are defined to allow modelling, rendering and interaction devices in a media network to determine the progress of primitives through such a network. This role of the *Tracer* primitive relies on the behaviour of the *MRI_Device* object type defined in clause 9.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

8.8 Wrapper Primitives

A *Wrapper* primitive encapsulates a non-object value within a primitive. The value carried by a *Wrapper* object is required to be of the non-object type *Value* defined in ISO/IEC 14478-2.

NOTE — As the non-object type *Value* includes object references, a wrapper primitive can also encapsulate arbitrary objects. One use of a wrapper primitive is to carry information from an input device from a logical class such as VALUATOR or PICK within the media streams that are processed by MRI devices.

9 Modelling, Rendering and Interaction Device

9.1 Introduction

An object type called *MRI_Device* is defined as an abstract supertype for elements of a media network that produce, consume or otherwise process data constructed from the *Primitive* type hierarchy described in clause 7. *MRI_Device* inherits from the *VirtualDevice* type defined in ISO/IEC 14478-3, and therefore can inter-operate with other devices and resources in a distributed network. However, the behaviour of *MRI_Device* specialises that of *VirtualDevice* to enable coordination of processing activities between objects that are instances of *MRI_Device* or its subtypes. Also associated with the *MRI_Device* type are two further object types, *MRI_Format* and *EfficiencyMeasure*.

9.2 MRI_Format

Each *MRI_Device* has at least one port that accepts or produces a stream of *Primitive* objects. The format object type associated with any such port is called *MRI_Format* (Modelling, Rendering and Interaction format). This format object subtype is associated with a number of properties and capabilities that are useful in characterising the behaviour of an *MRI_Device*. These are:

a) *DimensionsK*. This describes the coordinate space that any geometric primitives that use this port are expected to be defined in. An associated capability, *DimensionsCK*, gives the possible coordinate spaces that the data transferred via this port can be defined in.

b) *PrimitivesK*. This property gives the list of the types of primitives that the modeller or renderer is able to use via this port. For an input port, this will be the list of primitive types that the device can process; for output ports, it will be the list that the device can generate.

NOTE — Specialised formats may be created in subsequent components, if any, to deal with specific properties of audio, graphics etc. For example, a graphics renderer may associate a number of colours with its ports, whereas an audio renderer may have a number of channels that can be controlled concurrently via the port.

9.3 Efficiency

MRI devices that can process similar kinds of primitive may differ in the efficiency with which they can process those primitives. It is difficult, however, to give a simple measure for efficiency, since it may depend on the type and complexity of the input or output that the device handles. However, there are cases where it may be possible to compare devices, possibly as part of the process of creating and configuring a network through the negotiation framework set out in ISO/IEC 14478-3. To this end, an object type called *EfficiencyMeasure* is defined. It provides a single operation called '*compare*'. This operation accepts a reference to an object of type *MRI_Device* as input, and returns a result which is a value of the non-object data type *ComparisonRes*, defined by:

ComparisonRes ::= *worseThan* | *equivalentTo* | *betterThan* | *notComparable*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

The return result indicates whether, with respect to come criterion, the performance of the receiver is worse than, equivalent to, or better than that of the device given as input to the operation. *EfficiencyMeasure* objects are associated with subtypes of *MRI_Device* through the property list of the device. The property or properties of an object to which an *EfficiencyMeasure* is attached depends on the particular subtype of *MRI_Device* of which the object is an instance. The value *notComparable* will be returned if the *EfficiencyMeasure* is unable to relate the efficiency of the two objects. In particular this can arise if one object does not implement the functionality of the other object.

9.4 Behaviour

The facilities and mechanisms for stream control described in ISO/IEC 14478-3 are independent of the content of a stream or the purpose for which the stream is used. In contrast, this part is concerned with the use of streams for transporting specific kinds of media data between processing elements. One requirement that this imposes is the need for one processing node to determine when a particular datum has been received by another node. This requirement is fulfilled through a two-part mechanism.

— The *Tracer* primitive described in 8.7 can be placed by a client into a stream, either as a single object or as a component of a *Structured* primitive (see 8.6). This primitive carries an event object.

— When a *Tracer* primitive is encountered at a port configured for processing data in *MRI_Format*, the event carried by the tracer primitive is dispatched to the event handler attached to that port. As defined in 8.7, the *eventSource* attribute of the event is set to the be the *Tracer* primitive carrying the event. The *eventData* attribute will contain two pairs. One pair consists of the string "TracerDevice" and a reference to the device containing the port from which the event was dispatched. By definition, this device will be a *MRI_Device*. The second pair consists of the string "TracerPortId" and the identifier of the specific port at which the *Tracer* was detected.

This mechanism allows one client to ensure that media data has been delivered in full to a remote client. The sender registers itself with the event handler of the port to which it is sending data, and then transmits a *Tracer* primitive on the stream used for that data. Streams are guaranteed to preserve the order of data, and therefore the client will only be notified once all of the data sent on the stream before the tracer has been received.

NOTE — As *Tracer* primitives are objects, a reference to such a primitive may be contained within a *Wrapper* primitive. When such a *Wrapper* primitive enters or leaves a *MRI_Device* via a port of that device, no event is generated by the wrapped *Tracer*. That is, the contents of *Wrapper* primitives are not inspected at the ports of an *MRI_Device*.

10 Modeller

A *Modeller* is a *MRI_Device* that has an application-specific interface for obtaining input from outside of a PREMO application but which has at least one output port through which primitives can be passed, via a stream, to other *MRI_Device* objects. This does not preclude a modeller from also accepting PREMO primitives as input from the application, though a modeller which also processes PREMO primitives is properly a renderer, as described in clause 11. That is, a *Modeller* is a device that can acquire input through an interface that is not specified by ISO/IEC 14478, but which must provide at least one port in its output interface that allows the *Modeller* to interoperate with other devices using the stream and event mechanisms described in Parts 2 and 3 of this standard. The relationship between modellers and other devices is illustrated in Figure 16.

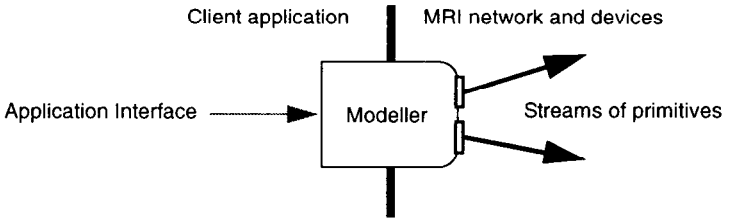

Figure 16 — A Modeller device

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The output of a *Modeller* is obtained via an output port that uses the format *MRI_Format*. The data available at this port will be objects of type *Primitive* or some subtype of this. The actual types of primitives that a *Modeller* can generate are defined by the *PrimitivesOutK* property. The capability of a *Modeller* to generate different output primitives is described in the capability *PrimitivesOutCK*.

The *EfficiencyOutK* property of a modeller provides a sequence containing the names of the primitive types that the *Modeller* can produce as output, and for each such primitive, an object of type *EfficiencyMeasure* for use in comparing the performance of that *Modeller* against another for that kind of primitive. See 9.3 for a description of this mechanism.

11 Renderer

A *Renderer* is a subtype of *MRI_Device* that provides the dual functionality of a *Modeller*. It provides at least one input port that can accept a stream of primitives using MRI_Format. However, the means by which a renderer passes output (if any) to a PREMO client is not specified within the PREMO standard.

NOTE — A display device for example might accept a stream of primitives and output graphical primitives as images on its screen.

The input to a *Renderer* is handled via at least one input port that uses the format *MRI_Format*. The data passed to this port will be objects of type *Primitive* or some subtype of this. The actual types of primitives that a *Renderer* can process are defined by the *PrimitivesInK* property. The ability of a *Renderer* to accept different input primitives is described in the capability *PrimitivesOutCK*.

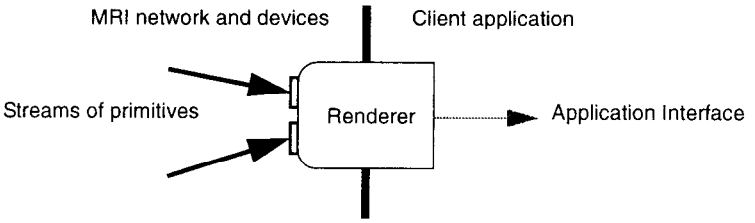

Figure 17 — A Renderer device

The *EfficiencyInK* property of a *Renderer* provides a sequence containing the names of the primitive types that the *Renderer* can accept as input, and for each such primitive, an object of type *EfficiencyMeasure* for use in comparing the performance of that *Renderer* against another for that kind of primitive. See 9.3 for a description of this mechanism.

12 MediaEngine

A *MediaEngine* is a combination of a *Modeller* and a *Renderer*, and thus also a subtype (indirectly) of *MRI_Device*. The interface of a *MediaEngine* contains at least one input port and one output port that can be connected to streams of primitives carried in *MRI_Format* or a subtype of it. Examples of such derived formats may include CGM, PHIGS Metafile and VRML. A *MediaEngine* maps primitives received via its input ports into primitives or media specific data format that can:

— be passed for further processing by other *MediaEngine*;

— be stored, or returned to a *Scene*, for use for example as a texture;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

— be presented to the operator via some *Renderer*.

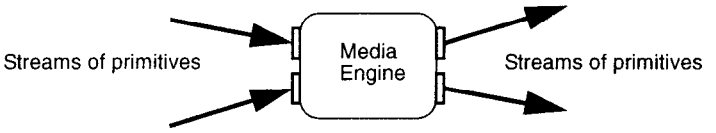

Figure 18 — A MediaEngine device

The criteria for selecting a particular *MediaEngine* will include the kinds of primitive that it can accept as input and produce as output, and also how well the engine maps particular types of input into outputs. As an engine is a subtype of both *Modeller* and *Renderer*, its ability to consume input and produce output over particular ports is represented by the *EfficiencyMeasure* objects associated with the inherited *EfficiencyInK* and *EfficiencyOutK* properties. In addition to these, a *MediaEngine* includes a property called *TransmutationK* that places a measure on its ability to generate specific kinds of output from given kinds of input. The value of this property is a sequence of entries, each consisting of a pair of primitives (i,o) associated with an *EfficiencyMeasure* object. Conceptually, this property is a matrix that indicates how well a given *MediaEngine* is able to map input in the form of primitives of type i into output of the form of primitives of type o. If the *MediaEngine* is unable to map instances of one type into another then the corresponding pair will be omitted from the list.

NOTE — Other properties may be introduced by subsequent components, in association with media-specific renderers.

13 Scene

Modellers, *Renderers* and *MediaEngines* may be connected directly via streams using the facilities associated with virtual devices and defined in ISO/IEC 14478-3. However, there are also situations where the input to and output from one of these subtypes of *MRI_Device* needs to pass through an intermediate object. One rationale for this is that a PREMO system is potentially distributed. Consequently it may be used in applications where multiple modellers are working on a common presentation, and where multiple renderers, either at different locations or for different media, need access to the primitives produced by the modellers. Some means of mediating the activities of multiple readers and writers is required, and the object type introduced to provide these functionalities is called a *Scene*. The name chosen for this type reflects the fact that it maintains a collection of primitives that are to be presented.

As the scene is a virtual device, it has a number of ports to which other devices can be attached. However, the way that information is transferred from input ports to the primitive store, and from the primitive store to output ports, is controlled by a specialised interface. A common feature of the operations provided in this interface is the use of a *Name* object, also used to label *Structured* primitives (see 8.6), to indicate the part(s) of the primitive structure affected by each operation. The operations are:

— create a new primitive structure in the scene with a given name as its label;

— delete a primitive structure with a given label from the scene;

— attach the stream of data from an input port to the node in the scene with a given name;

— attach an output port to a node in the scene with a given name;

— transfer (copy) the structure attached to a given output port onto the stream, followed by a tracer primitive;

— detach a port (input or output) from the scene;

— inquire the existence of a named primitive;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

An example will illustrate the relationship between these operations and the structures maintained within a scene object. Consider the system shown in Figure 19, consisting of a modeller, two media engines and a renderer, all sharing data via a scene. Assume that the scene is initially empty.

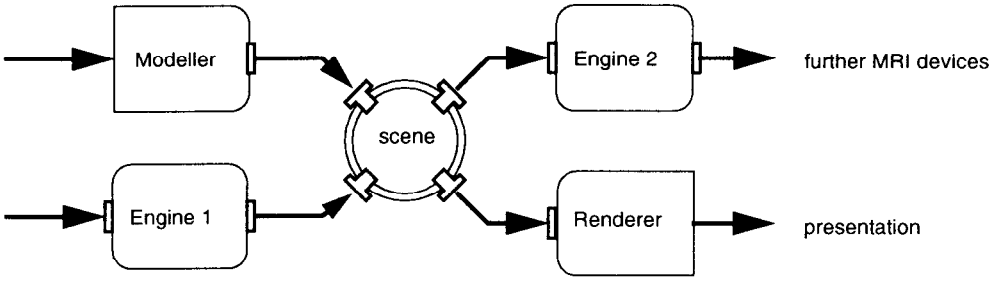

Figure 19 — A Scene within a media processing network

— Before it can receive any primitives for storage, the application has to make a call to create a new primitive, of type *Structured* or a subtype of *Structured*, identified by a given name. Figure 20 (a) shows the content of a scene after a structure labelled by the name 'Nm-A' has been created. In this example, the structure consists of a single *Aggregate* primitive, containing the string 'Nm-A' as the value of the *tag* attribute of the name object referenced through the *label* attribute of the aggregate.

— Input from a modeller can then be added to the structured primitive by attaching the port to the node labelled by a given name. For example, Figure 20 (b) shows the situation after the port for Modeller 1 has been attached to the node named 'Nm-A' and a collection of primitives (labelled 'P' in the diagram) has arrived on the stream. This collection of primitives may contain structures built from further *Aggregate* and *TimeComposite* objects.

— In general the scene maintains a collection of primitive structures - the application may for example create a new structure identified by 'Nm-B' which then exists alongside the existing structure, as in Figure 20 (c).

— Either of the devices attached to the scene for writing may contribute primitives to either structure. Thus, Figure 20 (d) shows the original structure being extended below the 'Nm-A' node by another sub-structure, here called Q, which might be contributed by 'Engine 1'.

— Any of the primitive structures contributed by modellers may themselves contain primitives derived from the *Structured* object type, and these can be used to allow one modeller to extend part of the structure created by another. To do this, the port used by the modeller has first to be *detached* from the part of the structure it is currently connected to, and then re-attached to the new node. For example, the port used by the modeller was initially attached to the structure identified by 'Nm-A'. If the output from the modeller is to be directed to a substructure, perhaps within P, identified by 'Nm–C', the port used by the modeller would first be detached from 'Nm–A' and then attached to 'Nm–C'. If the modeller then generates a primitive sub–structure called 'R' the result will be a scene containing the primitives in Figure 20 (e).

The transfer of primitives from one device to another via streams is conceptually an asynchronous operation, and one which, in a distributed context, may take a significant amount of time. It is therefore necessary to ensure, before the detach operation is invoked, that the structures sent to a scene via a stream have arrived. This task is supported by this part of PREMO through the tracer primitive and its interaction with the ports of an *MRI_Device* and its subtypes which include *Scene* (see 9.4). The device or process controlling the transfer of data can register itself with the event handler of the scene input port to be notified of tracer primitive arrival. Once the collection of primitives has been put into the stream, a tracer primitive is placed into the stream. As a stream guarantees order of delivery, the event handler will be notified of the tracer primitive once the primitives sent by device have been received by the scene. At this point the *detach* operation can be invoked safely.

Once defined in a scene object, a primitive structure may be accessed by a device *D* through the following protocol, which again uses the tracer primitive facility to ensure synchronization. In the paragraphs labelled a) to f) that follow, the term 'client' is used to refer to the entity that is initiating and controlling the interaction between *D* and the scene; this may be *D* itself, or some other object or control thread that has access to both:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

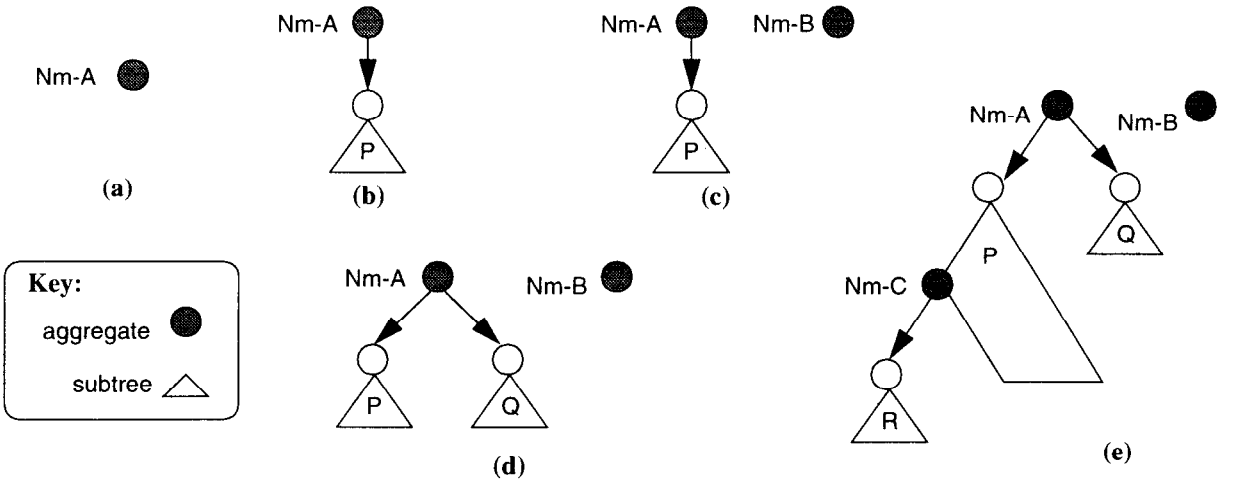

Figure 20 — Growth of a scene structure

a) The device *D* is connected by the client to an output port of the scene object.

b) The *attachRead* operation is invoked on the scene object by the client, with the port P as one argument and the name associated with the root node of the structure to be accessed by *D* as the second argument.

c) The client arranges to be notified of tracer primitive arrival on the input port of *D* being used to accept the data stream from the scene.

d) The client invokes the *transfer* operation on the scene object, with the port on the scene to which *D* is connected as parameter. The scene then begins to send the primitive structure to which the port is attached over the stream. When the structure has been sent, the scene places a tracer primitive onto the stream.

e) Once the client has been notified by the input port of *D* that the tracer primitive has been received, it invokes the *detach* operation on the scene object, giving the scene output port as parameter.

f) If a primitive structure is no longer needed, it can be removed from the scene by invoking the delete operation, and passing the name associated with the root node to be deleted. Invoking this operation on the scene containing the primitive structures shown in Figure 20 (e) using the name 'Nm-A' would result in a scene containing a single primitive structure consisting of the aggregate primitive labelled 'Nm-B'.

In general, a PREMO system may consist of a number of objects, derived from *MRI_Device,* that are sending primitives into a scene concurrently, and/or reading from the scene concurrently. One responsibility of the scene object is to provide concurrency control to prevent interference. In this respect a scene object is similar to a conventional database server, and it is assumed that the environment of a PREMO system will supply a suitable mechanism for controlling concurrent access, for example in the form of multi-granularity locking.

The scene object type provides an operation for enquiring whether a specific named structure exists in the scene for read or write attachment. However, the existence of a named primitive inside a scene does not necessarily mean that the associated structure is available for immediate reading or writing. In particular, the part of a structure of interest to a reader may, at the time of an *attachRead* request, still be under construction by another device. Resolution of such race conditions may involve a number of strategies whose definition is beyond the scope of this part of ISO/IEC 14478. Notification of generic access failure is supported by allowing the *attachRead* and *attachWrite* operations to raise an exception called '*AccessFailure*'.

NOTE — Information on the precise cause of an *AccessFailure* exception is dependent on the concurrency policy used by a particular PREMO system or application. Such information could be associated with the exception.

NOTE — Applications are not required to utilise the exception mechanism described above. Other strategies, for example a waiting call with timeout, may be utilised.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Four further exceptions may be raised (where relevant) by the operations of the scene object type. These are:

— *BadPort*: a parameter refers to a port that does not exist.

— *NoStructure*: a *Name* given as input does not occur as the *label* of any *Structured* primitive in the scene.

— *MultiplyDefined*: more than one *Structured* primitive in the scene matches a given *Name* parameter. To provide a client with some level of control over duplication of names, a scene object will raise an event when the device receives a primitive that has a name object equal to some other name object already defined within the scene. The event is raised on the event handler attached to the port on which the primitive was received. Testing for equality is carried out using the *equal* operation of the object most recently received. The name of the event raised is "AmbiguousName", the event source is set to the scene object, and the event data carried the data <("primitive", *Ref Primitive*)>, i.e. a reference to the primitive that carried the duplicate name.

— *NotAttached*: the *transfer* operation is invoked on a port which is not attached for reading to a structure in the scene.

14 Interaction

14.1 Introduction

The device and stream model of PREMO provides a flexible mechanism for passing data between nodes in a network. How a particular node handles or responds to that data is beyond the scope of this part. Instead, this section defines general mechanisms that support a range of interaction-handling styles and techniques.

14.2 Input Device

Input in a multimedia system is usually handled in one of two styles, sampled or event driven. A third style, called request mode, is also provided though its use may be more restricted. In the context of PREMO, input involves the transfer of data from the environment of a PREMO system into the system. The interface through which data is collected from the environment is not of concern in this part of the PREMO standard; it may be through a software interface to some external program, or input may be gathered by a hardware device. Independent of the method used to gather data, this part of PREMO defines the concept of an *InputDevice* object type to provide a standard interface for accessing input within the modes of operation described above. As an input device transfers data from an external context into a network of MRI devices, its behaviour is that of a *Modeller*, and thus *InputDevice* inherits from *Modeller*. Support for the three modes for input handling is described below.

— In event mode, the arrival of data causes an event to be generated and dispatched to the handler associated with the device. The list of possible events is maintained as a property with key *EventNamesK*. The data associated with the event will be a reference to an object of an appropriate subtype of *Primitive* that holds the input data.

— In sampled mode, all arriving data is placed on an output port. Clients interested in sampling this data can connect to the port. Muting can be used to discard input values. The format used by the port is *MRI_Format*, and the data placed on the stream connected to the port will be objects of appropriate subtypes of *Primitive*.

— Finally, in request mode, a client obtains an input datum by invoking a specific operation, *request*, in the interface of the input device. The operation is synchronous, and the client is suspended until the device is able to deliver a result.

This part does not define a specialized notion of input primitive, in contrast to Standards such as PHIGS and GKS. Instead, input data is stored and transmitted as objects of some subtype of Primitive.

NOTE — Geometric entities, for example locations, might be represented as subtypes of *Coordinate*. Media-independent input, for example the measure of a Valuator or Pick device, can be carried using the Wrapper primitive; see 8.8 for details of this.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NOTE — An application or any subsequent PREMO component could incorporate a notion of input primitive as follows:

1 Create a new subtype of *Primitive* called *InputPrimitive*
2 Subtype from *InputPrimitive* to obtain each kind of input primitive, possibly using multiple subtyping. For example, a *TextInput* primitive could be obtained by multiply subtyping from *InputPrimitive* and *Text*. Similarly, Valuator input could make use of a coordinate value in a one-dimensional coordinate space. For other kinds of input primitive, for example the *Locator* class of PHIGS, additional attributes may be needed to carry renderer-specific information such as viewport.

14.3 Router

The input device concept provides an interface between the external world and the concepts of event and stream that underlie this Component of PREMO, but they provide no means of structuring or controlling input. To provide a simple control mechanism for managing the distribution of stream input (and indeed other) data, PREMO defines the concept of a *Router*. This is built by combining a *Controller* object (defined in part 2) with a *virtual resource* (as defined in part 3). The states of the controller object determine the flow of data from input ports to output ports. In each state, each output port of the router can be connected to at most one input port of the router. Initially, there are no connections between ports in any state. Control over connections is provided by the following operations:

— A connection from Port A to Port B in state S is requested by invoking the operation *addConnection* that takes an input port (A), an output port (B), and a state (S).

— The connection to from any port to Port B in state S can be discarded by invoking the operation *dropConnection* that takes the output port (B) and state (S) as parameters. Note that the input port (if any) to which B is connected is not needed.

— The list of all connections that are defined in a given state can be inquired using the *inquireConnections* operation. This returns a sequence of port pairs, where a pair (A,B) in the sequence means that the input port A is connected to the output port B in that state.

The following exceptions may be raised (where appropriate) by the operations of the Router object type. A list of which exceptions each operation can raise specifically can be found in the functional specification in 16.10.2.

— *BadPort*: a parameter refers to a port that does not exist.

— *BadState*: a parameter refers to a controller state that does not exist

— *AlreadyConnected*: the (input) port passed as a parameter is already connected to some output port.

15 Coordinator

In order to render or process a presentation constructed from multiple types of media data, it will at some point be necessary to use media–specific devices. One possibility is that an application has access to a device that can itself carry out the processing of this multimedia data. More generally however, the devices available to an application will each be able to manage some subset of the media data, and it then becomes necessary to ensure that the original media data is decomposed and distributed to these specific processors, and that the actions of these processors are synchronized if required.

This part of ISO/IEC 14478 provides facilities for the description of multimedia presentations through the hierarchy of *Structured* primitives. The object types provided in ISO/IEC 14478-2 and ISO/IEC 14478-3 provide an application with middleware to support the distribution and synchronization of such presentations. This part of PREMO provides one more level of support, by defining an object type, *Coordinator*, that will manage the distribution of media primitives to specific processing devices, and which provides a foundation for realizing the synchronization constraints required for the overall presentation.

A *Coordinator* is a subtype of *MRI_Device* that provides a single input port, which accepts data in *MRI_Format*. It is also able to encapsulate a number of other media devices, each of which provides the coordinator with one input port. These devices must be instances of an object type that is a subtype of *Renderer*. As a coordinator receives primitives, it is responsible for decomposing any structured presentation into components that can be processed by the devices that it encapsulates. Figure 21 summarises the use of a coordinator within the AV system example from clause 6. In the example, the coordinator may receive presentations that

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

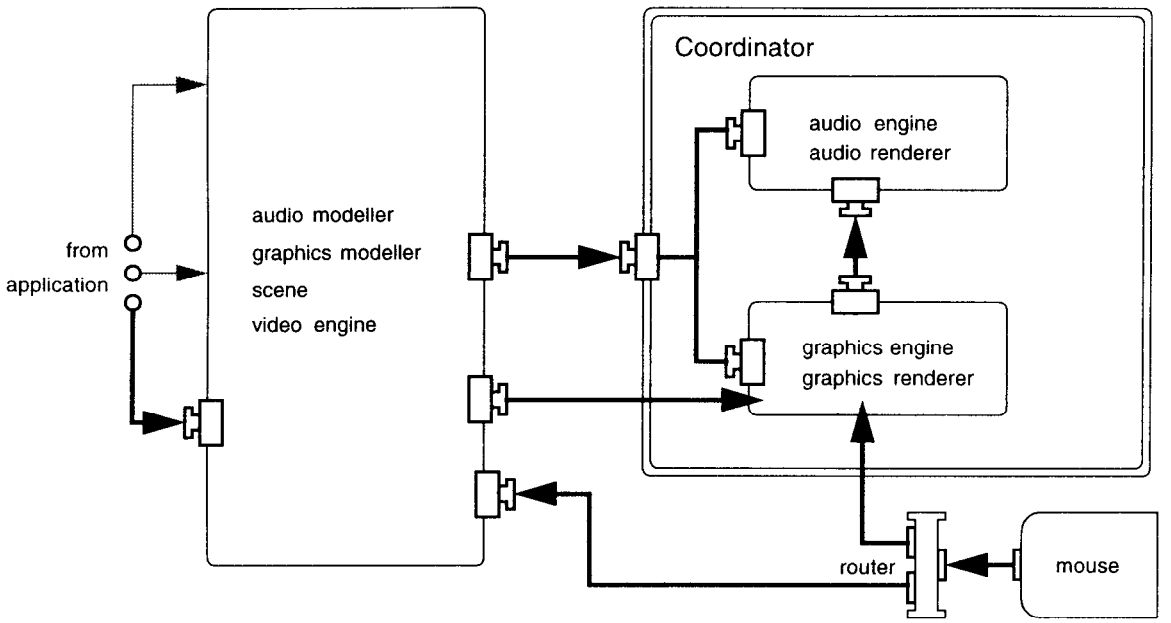

Figure 21 — The role of the Coordinator

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 coarse-grained synchronization constraints captured by the overall presentation. Note that these encapsulated devices may receive input from other components of the system; the coordinator is only responsible for processing media data received via its own input port. The distribution of primitives to the devices available to a coordinator is determined by the abilities of those devices to accept particular kinds of primitive as input. If a coordinator receives a presentation which it is not able to distribute in this way, because the presentation contains primitives which are supported by none of the available devices, the coordinator will raise the exception *PrimitiveNotRecognized*.

Processing devices are added to and removed from a coordinator using operations in the interface of the coordinator. These are:

— *addDevice*. This accepts a reference to a *Renderer*, and a port identifier, as input parameters. The port identifier must be that of an input port on the specified device that accepts *MRI_Format*; if this condition is not satisfied, the exception *BadPort* will be raised. If the device is already known to the coordinator, the only effect will be that the given port identifier will replace whatever port identifier is currently registered for that device. That is, *addDevice* also allows a client to change the port that the coordinator should use when forwarding data to a device.

— *removeDevice*. This takes as input a reference to a renderer, and removes the device from the collection of devices to which the coordinator can distribute media data. If the specified device is not within the collection of devices available to the renderer, there is no change to the state.

A third operation, *inquireDevices*, allows a client to find the set of devices that a coordinator can use, and for each such device, the port to which primitives will be sent by the coordinator.

The parallel streams of primitives that result from the distribution of a single multimedia presentation to the devices managed by a coordinator will need to be synchronized. In particular, the relative arrangement of media content within the 'slots' defined by sequential, parallel, and alternate primitives must be respected. Coarse–grained synchronization may also need to reflect structural or temporal modifiers within the media streams. At a finer level of granularity, synchronization may also be needed to maintain processing of the media streams within quality–of–service requirements. This too may be managed in whole or in part by a coordinator. ISO/IEC 14478-4 provides a variety of mechanisms, any of which might be used as, or within, an implementation

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

of synchronization constraints. This variety of techniques reflects the range of problem domains in which multimedia systems are found. Consequently, this part of PREMO does not require that a coordinator use a specific approach to managing the synchronization of media streams to the devices at its disposal.

NOTE — One possible approach to implementing the distribution and synchronization of media data within a coordinator is described here. The purpose of doing so is to further explain the role of coordinator objects; an implementation of *Coordinator* need not follow the outline given here. For simplicity, it is assumed that the devices available to a coordinator are not capable of managing parallel, sequential or alternate time composite primitives themselves. Consequently, when a coordinator receives a presentation (in the form of a primitive structure) on its input port, it must carry out an allocation of the primitives within the presentation to the devices that it has available. This allocation can be thought of as defining a set of media tracks. There is one track per device, and arranged along each track are the various primitives that the corresponding device is to process. A track itself represents time, so another way of stating the distribution problem is that the coordinator must create a schedule indicating which device will be processing a component primitive of the overall presentation at a specific time. Thus the allocation of media to devices becomes an instance of the general scheduling problem.

Once the presentation of media data has been scheduled, the coordinator can place the primitives onto media streams that are linked to the input ports of the rendering devices. To ensure that these media streams remain synchronized, the coordinator may adopt some or all of the following strategies. First, it can explicitly control the end-to-end processing of constituent devices by using the state-machine interface of the *StreamControl* objects associated with the input port and processing element of the device. The coordinator can monitor the progress of processing activities by inserting *Tracer* primitives into the media streams, and arranging to be notified when these streams are encountered at the input port of each device. A second strategy for synchronization is for a coordinator to place synchronization elements at reference points along each stream corresponding to significant features (e.g. the beginning or end of a component primitive) along any of the streams. By connecting these synchronization elements to an *ANDSynchronizationPoint* object, the coordinator can ensure that all media streams have reached a specific point before progression continues. Third, a finer level of control can be obtained by using periodic synchronization elements along the progression space of the streams.

16 Functional Specification

16.1 Introduction

This clause provides the detailed functional specification of the object types that define the PREMO Modelling, Presentation and Interaction Component. The notation used in this clause follows the rules detailed in Annex A of ISO/IEC 14478-1.

16.2 Non-object data types

This sub-clause defines the non-object data types that are introduced for the first time within this part.

A mapping from one type of primitive into another.

$$PrimMap == ObjectType \times ObjectType$$

Possible overlap between the presentations of components of sequential *TimeComposite* primitives.

$$OverlapType ::= never \mid left \mid right$$

The following data type describes the possible states of a primitive contained in a scene object.

$$SceneObjectState ::= NotPresent \mid Locked \mid Available \mid MultiplyDefined$$

The *ComparisonRes* data type defines values for use in comparing the efficiency of modellers and renderers.

$$ComparisonRes ::= worseThan \mid equivalentTo \mid betterThan \mid notComparable$$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

The following constants are defined for use with the *Colour* object type.

$ColourRGBG : \mathbf{N} \mid ColourRGBG = 2$

$ColourRGBB : \mathbf{N} \mid ColourRGBB = 3$

$ColourHSVH : \mathbf{N} \mid ColourHSVH = 1$

$ColourHSVS : \mathbf{N} \mid ColourHSVS = 2$

$ColourHSVV : \mathbf{N} \mid ColourHSVV = 3$

$ColourHLSH : \mathbf{N} \mid ColourHLSH = 1$

$ColourHLSL : \mathbf{N} \mid ColourHLSL = 2$

$ColourHLSS : \mathbf{N} \mid ColourHLSS = 3$

$ColourCIEL : \mathbf{N} \mid ColourCIEL = 1$

$ColourCIEU : \mathbf{N} \mid ColourCIEU = 2$

16.3 Exceptions

This sub-clause lists the exceptions raised by operations defined in PREMO MRI types and which are not defined as part of the exceptions defined for ISO/IEC 14478-2 (see clause 9.3 of ISO/IEC 14478-2) and for ISO/IEC 14478-3 (see clause 10.3 of ISO/IEC 14478-3).

AccessFailure == *Exception*

AlreadyConnected == *Exception*

AlreadyExists == *Exception*

NoStructure == *Exception*

BadPort == *Exception*

BadState == *Exception*

InvalidPort == *Exception*

InvalidResource == *Exception*

MultiplyDefined == *Exception*

NotAttached == *Exception*

PrimitiveNotRecognized == *Exception*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

16.4 Objects for coordinate spaces

16.4.1 *Coordinate* object

Coordinate[T]

SimplePREMOObject

dimensionality: **N** [Retrieve Only]

This generic object type describes coordinates of arbitrary dimensionality, subject to the restriction that each component of the coordinate belongs to a common type that instantiates the generic parameter *T*. The *dimensionality* is defined as a read-only attribute.

getRange

$index_{in}$: **N**
min_{out}: *T*
max_{out}: *T*

The minimum and maximum range of the dimension given by $index_{in}$ is returned. The effect of inquiring the range for dimensions outside of a coordinate's defined dimensionality is undefined.

Exceptions raised: None.

setComponent

$index_{in}$: **N**
$value_{in}$: *T*

The value of the coordinate for the dimension given by $index_{in}$ is set to $value_{in}$. If the value of $index_{in}$ is outside of the dimensionality of the coordinate, or if $value_{in}$ is outside the range of values for that dimension, the effect is undefined.

Exceptions raised: None.

getComponent

$index_{in}$: **N**
$value_{out}$: *T*

The value of the coordinate for the dimension given by $index_{in}$ is returned as $value_{in}$. If the value of $index_{in}$ is outside of the dimensionality of the coordinate the result is undefined.

Exceptions raised: None.

Coordinate

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

16.4.2 *Colour* object

Colour[T]

Coordinate[T]

colourModel: String [Retrieve Only]

This object type is a subtype of coordinate for use in representing colours within a given colour model. The name of the model in which the colour is defined is represented by the read-only attribute *colourModel*. Permitted values for this attribute include: "RGB", "CIELUV", "HSV" and "HLS". A dimensionality of 3 is associated with each of these colour models, and the interpretation of each component of a colour object under such a model is given by the tables in 7.2.

Colour

16.4.3 *TimeLocation* object

TimeLocation

Coordinate[Time]

A *TimeLocation* object is a specialization of *Coordinate* in which dimensionality is fixed at 1, and the type of values recorded for the single component is *Time*. This specifies an absolute moment in time, in terms of a number of ticks. For any instance of this object type, the clock against which these ticks are measured is determined from the context in which the instance appears.

TimeLocation

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

16.5 *Name* object

Name

SimplePREMOObject

tag: seq *String*

A *Name* is defined by a sequence of strings, and supports an operation to determine whether or not its sequence of strings is equal to that of another *Name* object. The semantics of equality are not prescribed by this part of ISO/IEC 14478, but the default behaviour is to treat the sequences as sets of strings and test for equality between sets.

equal

$otherName_{in}$: *Ref Name*
$result_{out}$: *Boolean*

Determine whether the name referenced by the parameter '*otherName*' is equal to the receiver.

Exceptions raised: None.

Name

16.6 Objects for media primitives

16.6.1 *Primitive* object

$Primitive_{abstract}$

SimplePREMOObject

This abstract object type is the common abstract supertype for the hierarchy of object types that describe the primitives defined for modelling, rendering and interaction within PREMO.

$Primitive_{abstract}$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

16.6.2 *Captured* object

Captured

Primitive

srcDevice: Ref VirtualDevice
srcPort: Port

A *Captured* primitive is one for which its presentation is determined by formatted data obtained via the port of some virtual device. The *Captured* object type contains a reference to the device on which the port is located, and the identity of the port on the device from which the data are to be obtained. The format of the data can be determined by inspecting the format object attached to the port.

Captured

16.6.3 Objects describing primitives with spatial and/or temporal form

16.6.3.1 *Form* object

$Form_{abstract}$

Primitive

This is a common abstract supertype for primitives whose presentation has to be constructed by a renderer based on some abstract description of the properties of the primitive.

$Form_{abstract}$

16.6.3.2 Objects describing form primitives for audio media data

P.6.3.2.1 *Audio* object

$Audio_{abstract}$

Form

This is the common abstract supertype for primitives that describe audio information that is to be synthesized.

$Audio_{abstract}$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16.6.3.2.2 *Music* object

Music

Audio

instrument: **N**
score: seq **N**

A musical primitive contains a reference to a musical instrument, here given by a natural number, and the score which is described simply as a sequence on numbers which encodes, in some way, the notes and other information about the sounds that are to be rendered. PREMO does not mandate any naming system for assigning numbers to specific sounds, instruments or notes, as none have yet been standardized. A common example of a system in current use however is MIDI.

Music

16.6.3.2.3 *Speech* object

Speech

Audio

voice: Ref VocalCharacteristics
text: seq Character

This is a primitive containing a sequence of characters (text) that is to be presented as synthesized speech. The text is to be articulated using the vocal characteristics captured by the attribute *voice*. The text may contain both words and other information needed by a specific system for emphasis etc. The object type for describing vocal characteristics is defined in 16.6.4.2.3.

Speech

16.6.3.3 Objects describing form primitives for geometric media data

16.6.3.3.1 *Geometric* objects

Geometric$_{abstract}$

Form

This is the common abstract supertype for primitives that describe or define geometric structures.

Geometric$_{abstract}$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

16.6.3.3.2 *Tactile* object

Tactile$_{abstract}$

Form

This is an abstract supertype for primitives representing aspects of tactile (haptic) feedback. As current systems for supporting this are still at an experimental stage, no further provision for this modality is available in this part of the standard.

Tactile$_{abstract}$

16.6.3.3.3 *Text* object

Text

Form

characters: String

This object type contains a sequence of characters to be rendered as text. Information about the size, font or other aspects that affect the appearance of the text will depend on the context in which the primitive appears. See 8.3.5 for further discussion on this.

Text

16.6.4 Objects describing primitives for the modification of media data

16.6.4.1 *Modifier* object

Modifier$_{abstract}$

Primitive

This is the abstract supertype for objects representing modifications. A modifier is a primitive that has not presentation itself, but rather, which acts to modify the presentation of other primitives. Modifiers are grouped according to the kind of primitive on which they act.

Modifier$_{abstract}$

16.6.4.2 Objects describing modifier primitives for audio media data

16.6.4.2.1 *Acoustic* object

Acoustic$_{abstract}$

Modifier

This is the abstract supertype for modifiers that act on audio primitives.

Acoustic$_{abstract}$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

16.6.4.2.2 *SoundCharacteristic* object

SoundCharacteristic$_{abstract}$

Acoustic

This object type provides a hook for any subsequent components or applications using sound renderers to define means by which the presentation of a sound may be modified, independent of whether the sound represents speech or music. Such modifications should thus be defined over external properties of the sound, such as volume, waveform, etc.

SoundCharacteristic$_{abstract}$

16.6.4.2.3 *VocalCharacteristic* object

VocalCharacteristics$_{abstract}$

Acoustic

This object type provides a hook for subsequent components, if any, or applications using sound renderers to attach models for describing the characteristics of voices that can be used to control the presentation offered by such renderers. At the time of writing, there is no standard or commonly used framework for this functionality.

VocalCharacteristics$_{abstract}$

16.6.4.3 Objects describing modifier primitives for structural aspects of media data

16.6.4.3.1 *Structural* object

Structural$_{abstract}$

Modifier

This is the abstract supertype for modifiers that affect the interpretation of coordinates associated with some collection of primitives. The affected coordinates can encompass, e.g., geometric structure or temporal properties.

Structural$_{abstract}$

16.6.4.3.2 *Transformation* object

Transformation$_{abstract}$

Structural

This is the abstract supertype for modifiers that transform coordinates into new coordinates.

Transformation$_{abstract}$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

16.6.4.3.3 *Constraint* object

Constraint$_{abstract}$

Structural

This is the abstract supertype for constraints that are used to modify the appearance of primitives by affecting or restricting their coordinate values, for example by clipping or stencilling.

Constraint$_{abstract}$

16.6.4.4 *TimeFrame* object

TimeFrame

Modifier

timeBase: Ref Clock

A *TimeFrame* modifier carries a reference to a *Clock* object. Media processors could use the referenced clock as the basis for working with time coordinates contained within the primitives that make up a presentation.

TimeFrame

16.6.4.5 Objects describing modifier primitives for visual aspects of media data

16.6.4.5.1 *Visual* objects

Visual$_{abstract}$

Modifier

This is the abstract supertype for modifiers that act on geometric primitives to modify non-geometric or temporal aspects of their presentation.

Visual$_{abstract}$

16.6.4.5.2 *Light* object

Light$_{abstract}$

Visual

This is an abstract supertype for primitives that represent properties of light. This part does not mandate the use of any particular lighting model.

Light$_{abstract}$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

16.6.4.5.3 *Material* object

Material$_{abstract}$

Visual

This is an abstract supertype for primitives that describe properties of materials constructed from geometric primitives.

Material$_{abstract}$

16.6.4.5.4 *Shading* object

Shading$_{abstract}$

Visual

This is an abstract supertype for primitives that describe the shading of structures constructed from geometric primitives.

Shading$_{abstract}$

16.6.4.5.5 *Texture* object

Texture$_{abstract}$

Visual

This is an abstract supertype for primitives that describe the texture of structures constructed from geometric primitives.

Texture$_{abstract}$

16.6.5 *Reference* object

Reference

Primitive

label: Ref Name

A *Reference* primitive contains a *label* attribute that references a *Name* object. At the time that the primitive is processed, it is assumed that an equal name will have been introduced elsewhere in the same primitive structure. The effect of *reference*-ing a name that has not been introduced is not defined.

Reference

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

16.6.6 Objects for organising primitives into structures

16.6.6.1 *Structured* object

Structured$_{abstract}$

Primitive

components: seq Ref Primitive$_{abstract}$
label: Ref Name

This is the abstract supertype for primitives that group together other primitives into larger structures. The semantics of such groupings are determined by concrete subtypes. All structured primitives however contain an attribute, *components*, that defines a sequence of references to primitives. This part places no interpretation on the order of the component primitives, though subsequent components, if any, or applications that extend structured subtypes are at liberty to do so.

Each structured primitive also has a *label* attribute that references a name that can be used to identify or refer to the primitive.

Structured$_{abstract}$

16.6.6.2 *Aggregate* object

Aggregate

Structured

An aggregate is a concrete form of structured primitive intended for combining collections of synthesized primitives into larger structures or for combining *Form* and *Modifier* primitives so that the effect of modifiers can be scoped. However, PREMO defines no specific scoping rules, as discussed in 8.6.

Aggregate

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

16.6.6.3 Objects for organising media data within time

16.6.6.3.1 *TimeComposite* object

TimeComposite$_{abstract}$

Structured

min, max: Time
startTime, endTime: Time
monitor: Ref EventHandler

This is the abstract supertype for primitives used to describe the temporal organization and synchronization of a structured presentation. The two attributes of type *Time*, *min* and *max*, define the size of the temporal context (duration) in which the media content inherited from the *Structured* supertype is to be presented.

The *monitor* attribute is a reference to an event handler that will be notified of progress in presenting the contents of the *TimeComposite*. The *startTime* and *endTime* attributes define the delay between the start/end of presentation of the TimeComposite, and the start/end of presentation of the first/last component. Note that all times are measured relative to the clock inherited through *Duration*. The event handler will be signalled between the end of *startTime* and the onset of processing of the first component, and between the end of processing of the last component and the beginning of *endTime*.

TimeComposite$_{abstract}$

16.6.6.3.2 *Sequential* object

Sequential

TimeComposite

startDelta, endDelta: Time
overlap: OverlapType

A sequential TimeComposite primitive describes a presentation in which each primitive in the component sequence is rendered in the order in which it appears in the sequence. The presentation of each component is preceded by a delay of *startDelta* ticks, and is followed by a delay of *endDelta* ticks. An event is generated between the end of each *startDelta* and the beginning of the presentation of the next component, and between the end of the presentation of each component and the beginning of the subsequent *endDelta*.

In order to fit the presentation of the sequence into the duration available for the TimeComposite, a renderer may need to compress or truncate the presentations of some or all components. The attribute *overlap* describes the way in which this can be performed. It takes on three possible values: *never*, *left* and *right*. The meaning of each of these values is explained in 8.6.3.

Sequential

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

16.6.6.3.3 *Parallel* object

Parallel

TimeComposite

startSync, endSync: Boolean

This object type describes a TimeComposite whose components should be presented in parallel. In addition to the components, it specifies two attributes, *startSync* and *endSync*. When *startSync* is true, the presentations of each component shall begin synchronously; when it is false, the presentations may begin asynchronously. If *endSync* is true, a media device should attempt to ensure that the presentations end at the same time.

Parallel

16.6.6.3.4 *Alternate* object

Alternate

TimeComposite

selector: Ref Controller
options: seq $(String \times \mathbf{Z})$

An alternate TimeComposite is one where one of the components will be chosen for presentation, depending on the state of a controller object referenced by the attribute *selector*. The link between controller states and component primitives is defined by the *options* attribute. The consequences of presenting an alternate primitive when either the state of the controller corresponds to none of the listed options, or when there is more than one option for the specified state value, is not determined by this part.

Alternate

16.6.7 Tracer object

Tracer

Primitive

trace: Event

A Tracer primitive carries an event. Whenever a tracer object is received or transmitted via the port of a MRI device (see 16.8) the event carried by the tracer will be dispatched to the event handler associated with that port. The *eventName* attribute of the event is set to "TracerEvent". The *eventSource* attribute is set to reference the *Tracer* primitive object in which the event is contained.

Tracer

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

16.6.8 Wrapper object

Wrapper

Primitive

content: Value

A *Wrapper* primitive is used to enclose a value from the union type *Value* defined in ISO/IEC 14478-2 into an object that can be used in contexts where a *Primitive* would be expected.

Wrapper

16.7 Objects for describing properties of devices

16.7.1 *MRI_Format* object

MRI_Format

Format

This is the format to be used by ports of MRI devices for input and output of primitives, i.e. for ports associated with a stream of data consisting of references to objects of type $Primitive_{abstract}$ or its subtypes.

MRI_Format

Properties defined:

Key	Type of Value	R.O or R/W	Description
DimensionsK	seq *(ObjectType* × **N** *)*	R/W	Dimensions of the coordinate space in which primitives of a specific type will be defined.
PrimitivesK	seq ObjectType	R.O.	The kinds of primitive that can be accepted or generated via this port.

Capabilities defined:

Key	Type of Value	Values
DimensionsCK	seq *(ObjectType* × **N** × **N** *)*	<primitive, minimum value, maximum value>
PrimitivesCK	seq ObjectType	The primitives that can be produced by a modeller of this type.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

16.7.2 *EfficiencyMeasure* object

EfficiencyMeasure

SimplePREMOObject

This object type provides an interface that supports comparison with another object of this type, for use for example in comparing efficiency of modellers and renderers with respect to some criterion.

compare

$alternative_{in}$: *Ref MRI_Device*
$result_{out}$: *ComparisonRes*

The level of efficiency represented by the receiver is compared with that of the input $alternative_{in}$. Provided that the receiver and argument are comparable, a result is returned indicating whether the receiver is worse than, equivalent to, or better than the argument. The value *notComparable* is returned otherwise.

EfficiencyMeasure

16.8 Processing devices for media data

16.8.1 *MRI_Device* object

MRI_Device

VirtualDevice

This is a device that contains at least one port that can either accept data using *MRI_Format* or can produce data using that format. Any port set to use *MRI_Format* will monitor the primitives that pass through the port. Whenever a *Tracer* primitive is observed, the event carried by the tracer will be dispatched to the event handler associated with the port.

MRI_Device

16.8.2 *Modeller* object

Modeller

MRI_Device

A *Modeller* is a *MRI_Device* that contains at least one output port that accepts *MRI_Format*.

Modeller

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Properties defined:

Key	Type of Value	R.O or R/W	Description
EfficiencyOutK	seq (ObjectType × EfficiencyMeasure)	R.O.	A measure of the efficiency with which the modeller can produce primitives of a particular type.

Capabilities defined:

None.

16.8.3 *Renderer* object

Renderer

MRI_Device

A *Renderer* is a *MRI_Device* that contains at least one output port that accepts *MRI_Format*.

Renderer

Properties defined:

Key	Type of Value	R.O or R/W	Description
EfficiencyInK	seq (ObjectType × EfficiencyMeasure)	R.O.	A measure of the efficiency with which the renderer can accept and process primitives of a particular type.

Capabilities defined:

None.

16.8.4 *MediaEngine* object

MediaEngine

Modeller
Renderer

A *MediaEngine* is both a *Modeller* and a *Renderer*, that is, it has both at least one input port and one output port capable of using *MRI_Format*.

MediaEngine

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Properties defined:

Key	Type of Value	R.O or R/W	Description
TransmutationK	seq (PrimMap × EfficiencyMeasure)	R.O.	A measure of the efficiency with which the engine can transform one kind of primitive into another

Capabilities defined:

None.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

16.9 *Scene* object

Scene

VirtualDevice

A scene object is a container for holding a collection of primitive structures and for mediating access to those structures by a number of modellers and renderers. The clients of a Scene object gain access to part of a primitive structure by requesting the object to attach one of its ports, either for reading or writing, to part of a primitive structure labelled by a specific name (via an Identification primitive)

create

$structname_{in}$: *Name*
$structureType_{in}$: *ObjectType*
exceptions: {*AlreadyExists, InvalidType*}

Create a primitive structure of the type specified by $structureType_{in}$ named $structname_{in}$. The type specified by $structureType_{in}$ must be a subtype of *Structured.*

Exceptions raised:

AlreadyExists	A structure of that name already exists in the scene.
InvalidType	The type $structureType_{in}$ is not a valid subtype of *Structured.*

attachRead

$structname_{in}$: *Name*
$portid_{in}$: *Port*
exceptions: {*NoStructure, MultiplyDefined, BadPort, AccessFailure*}

Attach the port named $portid_{in}$ to the primitive structure identified by the name $structname_{in}$ for subsequent reading, i.e. the primitive hierarchy rooted at $structname_{in}$ will be placed on the stream associated with the given port identifier. While primitives are being read from a stream, any *Name* object received is checked against the set of name objects already defined within the scene. If a new *Name* object contains a name that is equal to that of an existing name, an event is dispatched to the event handler associated with port $portid_{in}$.

Exceptions raised:

NoStructure	No structured primitive in the scene has the specified name.
MultiplyDefined	More than one structured primitive in the scene matches the name.
BadPort	The output port given by $portid_{in}$ does not exist.
AccessFailure	The structured primitive labelled by the given name cannot be accessed for reading.

inquireStatus

$structname_{in}$: *Name*
$result_{out}$: *SceneObjectState*

Inquire the status of a structured primitive in the scene labelled by $structname_{in}$. The result is a value of type *SceneObjectState* as defined in 16.2.

Exceptions raised: None.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

attachWrite

structname$_{in}$: *Name*
portid$_{in}$: *Port*
exceptions: {*NoStructure, MultiplyDefined, BadPort, AccessFailure*}

Attach the port named *portid*$_{in}$ to the primitive structure identified by the name *structname*$_{in}$ for subsequent writing, i.e. the primitive hierarchy rooted at *structname*$_{in}$ will be extended with new data arriving on the stream associated with the given port identifier.

Exceptions raised:

NoStructure	No structured primitive in the scene has the specified name.
MultiplyDefined	More than one structured primitive in the scene matches the name.
BadPort	The input port given by *portid*$_{in}$ does not exist.
AccessFailure	The structured primitive labelled by the given name cannot be accessed for writing.

transfer$_a$

portid$_{in}$: *Port*
exceptions: {*BadPort, NotAttached*}

Request the scene to begin transfer of the contents of the structure to which the input port *portid*$_{in}$ is attached. Note that this operation is asynchronous. Once the primitives from the structure have been transferred, a tracer primitive will be sent via the same port.

Exceptions raised:

BadPort	The input port given by *portid*$_{in}$ does not exist.
NotAttached	The port is not attached for reading to a structure in the database.

detach

portid$_{in}$: *Port*
exceptions: {*BadPort*}

Detach the port named *portid*$_{in}$ from the primitive structure that it is currently connected to for reading or writing.

Exceptions raised:

BadPort	The input port given by *portid*$_{in}$ does not exist.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

delete

structname$_{in}$: *Name*
exceptions: {*NoStructure, Locked*}

Remove *all* primitive structures from the scene that have as their root a Structured primitive with a label attribute that matches *structname*$_{in}$.

Exceptions raised:

NoStructure	No structured primitive in the scene has the specified name.
Locked	The structured primitive labelled by the given name is currently locked for writing by another device.

Scene

Properties defined:

None.

Capabilities defined:

None.

16.10 Objects for supporting interaction

16.10.1 *InputDevice* object

InputDevice

Modeller

An input device is a form of modeller that, in addition to the stream and event interface provided through its output ports, defines an operation through which a primitive can be requested.

request

input$_{out}$: *Ref Primitive*

This operation allows a client to obtain input in 'request mode'. The result of the request operation is an instance of the *Primitive* object type or a subtype, determined by the nature of the input device.

Exceptions raised: None.

InputDevice

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Properties defined:

Key	Type of Value	R.O or R/W	Description
InputEventNamesK	seq EventName	R.O.	Names of events that will be raised when particular input data arrives while in event mode.

Capabilities defined:

None.

16.10.2 *Router* object

Router

MRI_Device
Controller

A router object is a *MRI_Device* and *Controller* that copies data on its input ports to a set of output ports determined by the current state of its controller component.

addConnection

$state_{in}$: String
$inputPort_{in}$: Ref Port
$outputPort_{in}$: Ref Port
exceptions: {BadPort, BadState, AlreadyConnected}

Update the router so that when the control state is *$state_{in}$* there is a connection from the port *$inputPort_{in}$* to the port *$outputPort_{in}$*, i.e. any data received on the stream associated with *$inputPort_{in}$* is copied to the stream associated with *$outputPort_{in}$*.

Exceptions raised:

BadPort	One of the ports provided as input argument does not exist.
BadState	The state *$state_{in}$* given as input argument does not exist.
AlreadyConnected	Some input port is already connected to *$outputPort_{in}$* in *$state_{in}$*.

dropConnection

$state_{in}$: String
$outputPort_{in}$: Ref Port
exceptions: {BadPort, BadState, AlreadyConnected}

Remove all connections between input ports and the port *$outputPort_{in}$* for the state *$state_{in}$*. This means that when the router is in state *$state_{in}$*, there are no connections that will result in data being copied onto the stream connected to port *$outputPort_{in}$*.

Exceptions raised:

BadPort	The port provided as input argument does not exist.
BadState	The state *$state_{in}$* given as input argument does not exist.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

inquireConnections

$state_{in}$: *String*
$links_{out}$: seq $(Port \times Port)$
exceptions: {*BadState*}

Determine the set of links that exist from input ports to output ports in a designated state. The result is returned through the output parameter $links_{out}$ as a sequence of pairs of states. No specific order is implied.

Exceptions raised:

BadState	The state $state_{in}$ given as input argument does not exist.

Router

Properties defined:

None.

Capabilities defined:

None.

16.11 *Coordinator* object

Coordinator

MRI_Device

A *Coordinator* is a subtype of MRI_Device. Each coordinator device encapsulates a collection of devices that it utilises to process primitive structures received on its input port. When a primitive structure is received, a coordinator is required to distribute components of the structure to the devices so that (i) each primitive within he structure is sent to a device that is able to process it, and (ii) synchronization requirements within the original presentation are met. Although PREMO provides a number of facilities to support synchronization (e.g. synchronization elements, *ANDSynchronizationPoint*, and *Tracer* primitives), no specific approach is mandated for the implementation of synchronization requirements within the *Coordinator* object type.

addDevice

$device_{in}$: *Renderer*
$port_{in}$: *Port*
exceptions: {*BadPort*}

Add the renderer device $device_{in}$ to the collection of devices that the coordinator can use to process a primitive structure it receives. The parameter $port_{in}$ gives the port to which the coordinator should direct primitives. The *MRI_Format* attached to this port will determine the primitives that this device can accept. If the device is already available to the coordinator, the port used by the coordinator will be updated to the port specified by the parameter.

Exceptions raised:

BadPort	The port $port_{in}$ is not an input port of the renderer $device_{in}$, and/or does not accept data in *MRI_Format*.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

dropDevice

$device_{in}$: *Renderer*

The device specified by $device_{in}$ is removed from the collection of devices that the coordinator can use for processing media data. If the device is not one that the coordinator has in its collection, the operation has no effect.

inquireDevice

$devices_{out}$: seq (*Renderer* × *Port*)

The operation returns the set of renderers that the coordinator has in its collection, and for each renderer, the port of the renderer to which the stream used by the coordinator is attached.

Coordinator

Properties defined:

None.

Capabilities defined:

None.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

17 Component Specification

MRIComponent

Basic

provides service

MRI_Device, Modeller, MediaEngine, Renderer,
Scene,
Coordinator,
InputDevice, Router

provides type

MRI_Format,
Coordinate, Colour, TimeLocation,
Primitive, Form, Geometric, Text, Tactile,
Audio, Music, Speech, Reference, Name,
Modifier, Acoustic, SoundCharacteristic, VocalCharacteristic,
Structural, Transformation, Constraint,
Temporal, TimeFrame,
Visual, Light, Shading, Material, Texture,
Structured, Aggregate, TimeComposite, Sequential, Parallel, Alternate

requires service

Component MSSComponent Profile Basic

requires type

Component MSSComponent Profile Basic

MRIComponent

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex A
(normative)
Overview of PREMO Modelling, Rendering and Interaction Object Types

This annex gives an overview of all PREMO object types defined in this part. This Annex does not add any new information and is here for easier reference only.

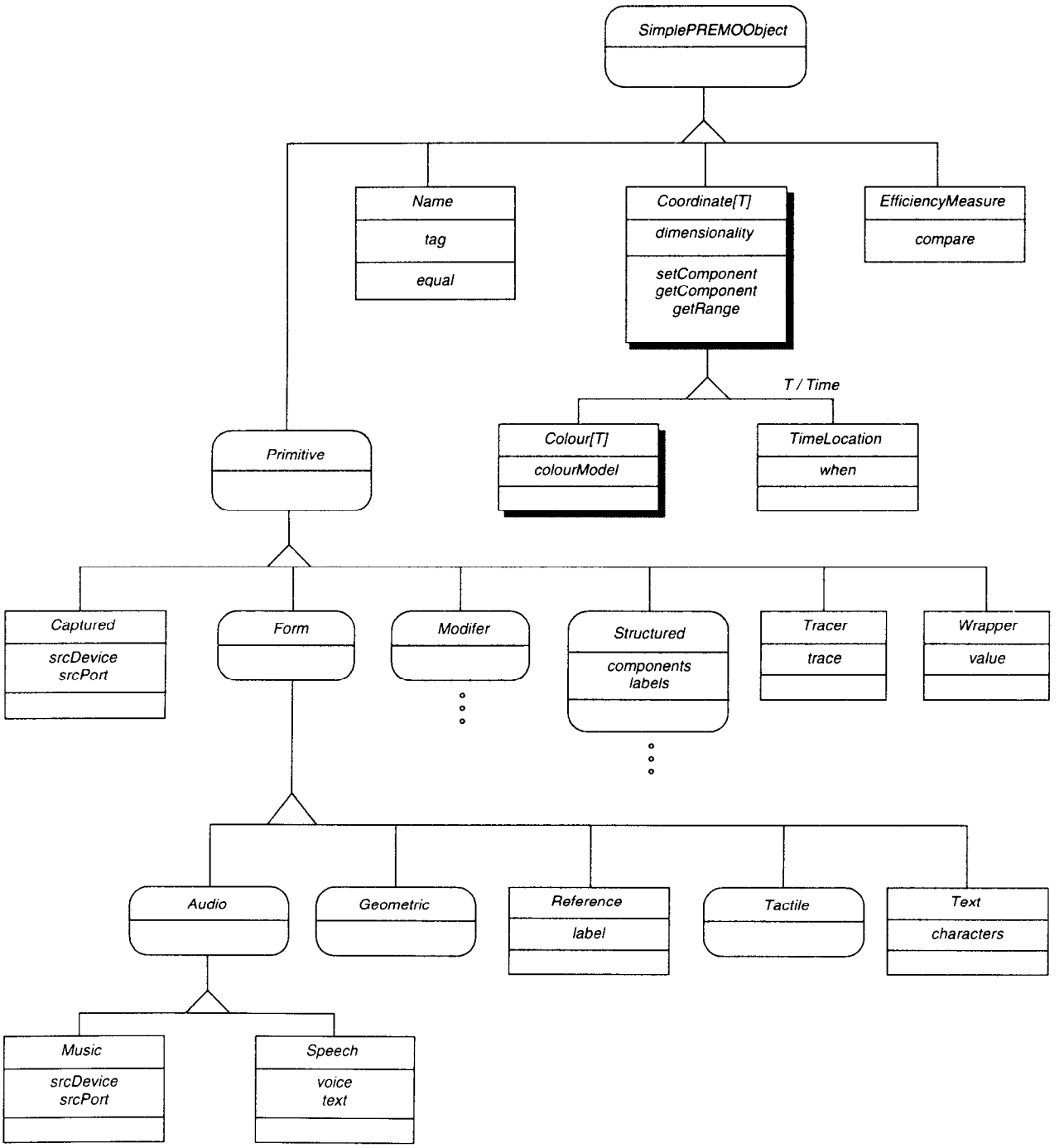

Figure 22 — PREMO primitive object types

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

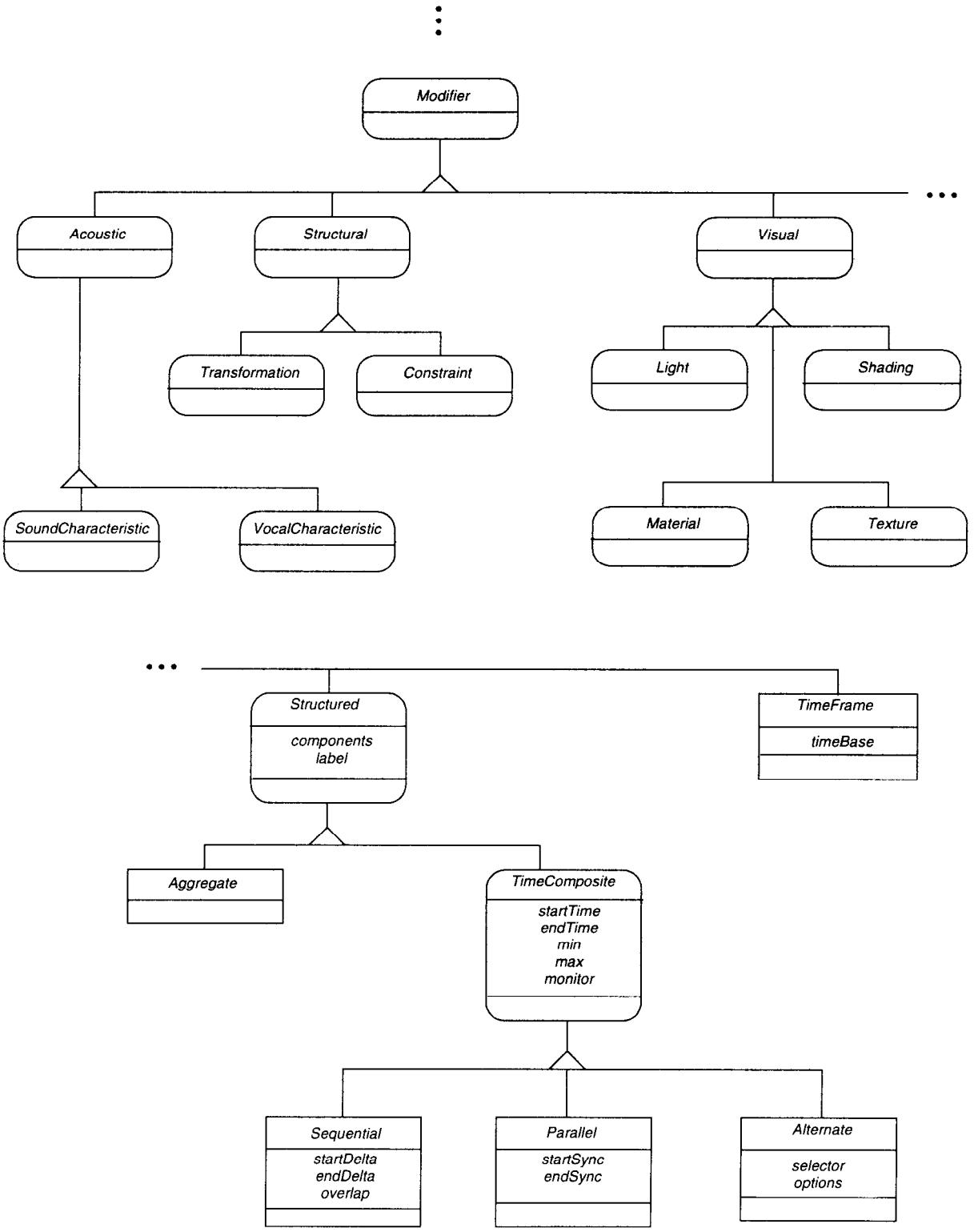

Figure 23 — PREMO Modifier and Structured primitive object types

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

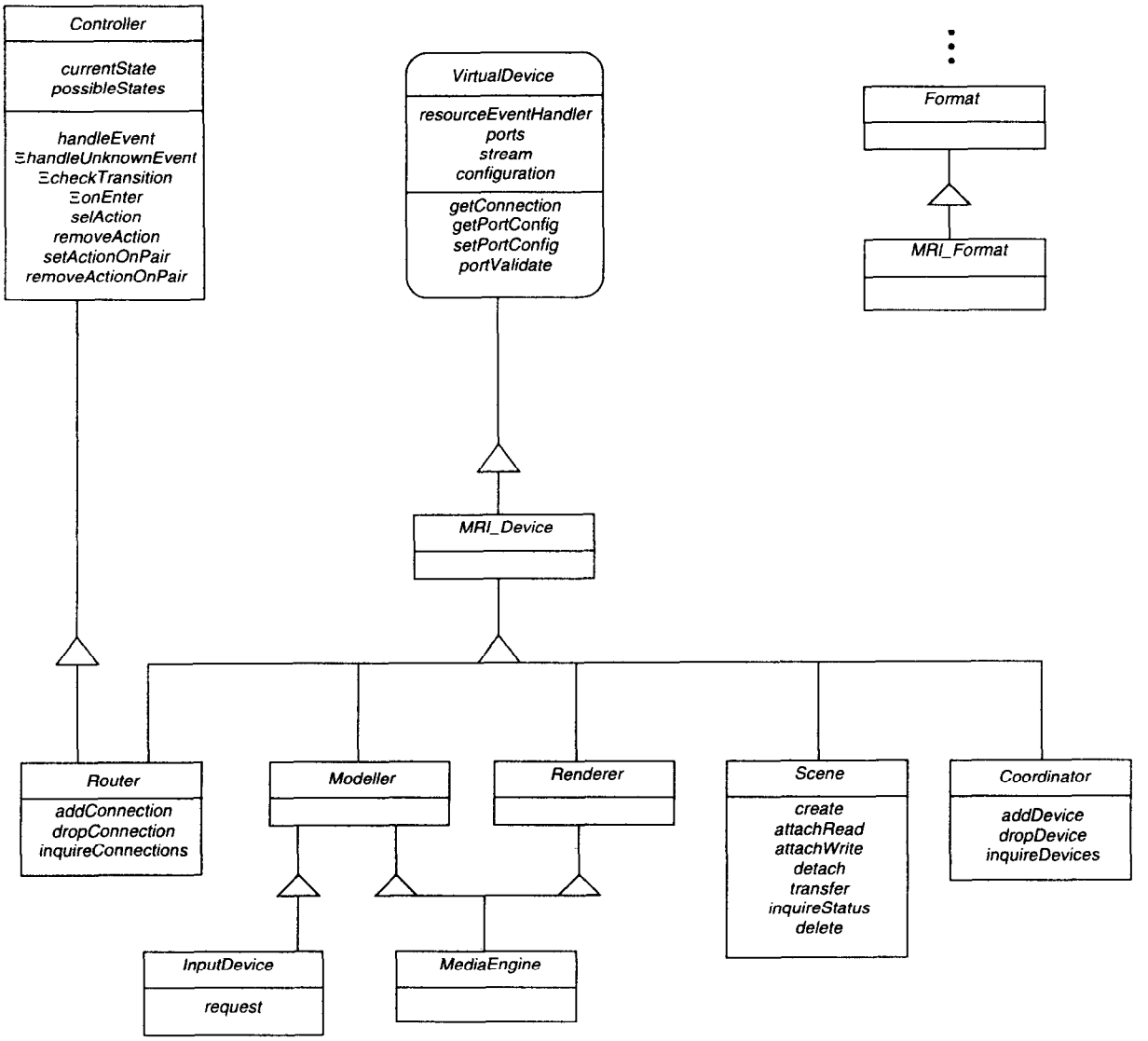

Figure 24 — PREMO MRI devices and format object types

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex B
(informative)
Diagrammatic Conventions

B.1 Introduction

ISO/IEC 14478-3 utilizes a number of graphical conventions for illustrating virtual devices, resources and the ports and streams that interconnect them. This part is concerned with a larger collection of specialized processing devices, and while these are all ultimately subtypes of *VirtualDevice*, it is convenient to use a correspondingly richer collection of shapes to represent these devices within this part. The role of this Annex is simply to summarise the notation used. This is done in the form of a subtype hierarchy that mimics appropriate parts of the structure given in Annex A. Here graphical signatures are used in place of object type structures.

B.2 General Graphical Signatures

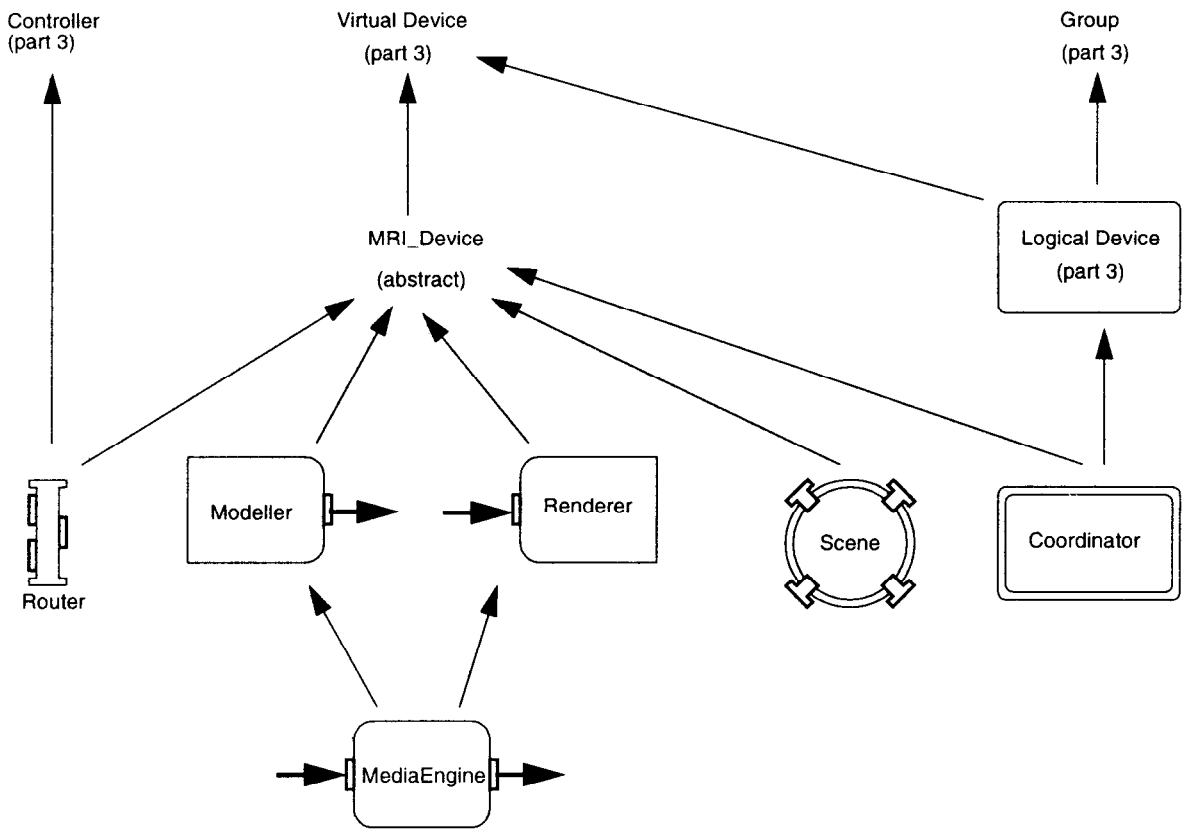

Figure 25 — Graphical signatures for part 4 device subtypes

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

B.3 Conventions for Devices and Communication

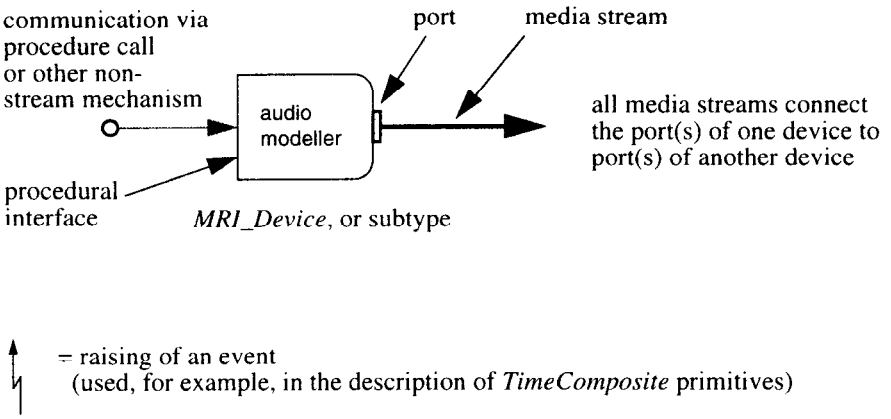

Figure 26 — Conventions for representing media flow and other forms of communication

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex C
(informative)
Relationship between Part 4 and the CGRM

C.1 Introduction

The Computer Graphics Reference Model (CGRM) [ISO/IEC 11072:1992] is a descriptive framework intended for use in understanding and explaining the entities and processes involved in computer graphics. It is not an implementation framework, but rather a conceptual model into which specific and disparate systems may be mapped. The CGRM defines an abstract architecture for graphics applications, consisting of a 'pipeline' of environments shown on the left of Figure 27. Each environment consists of a number of processing elements and data stores. These are represented as rectangles and circles (respectively) in the diagram on the right hand side of Figure 27.

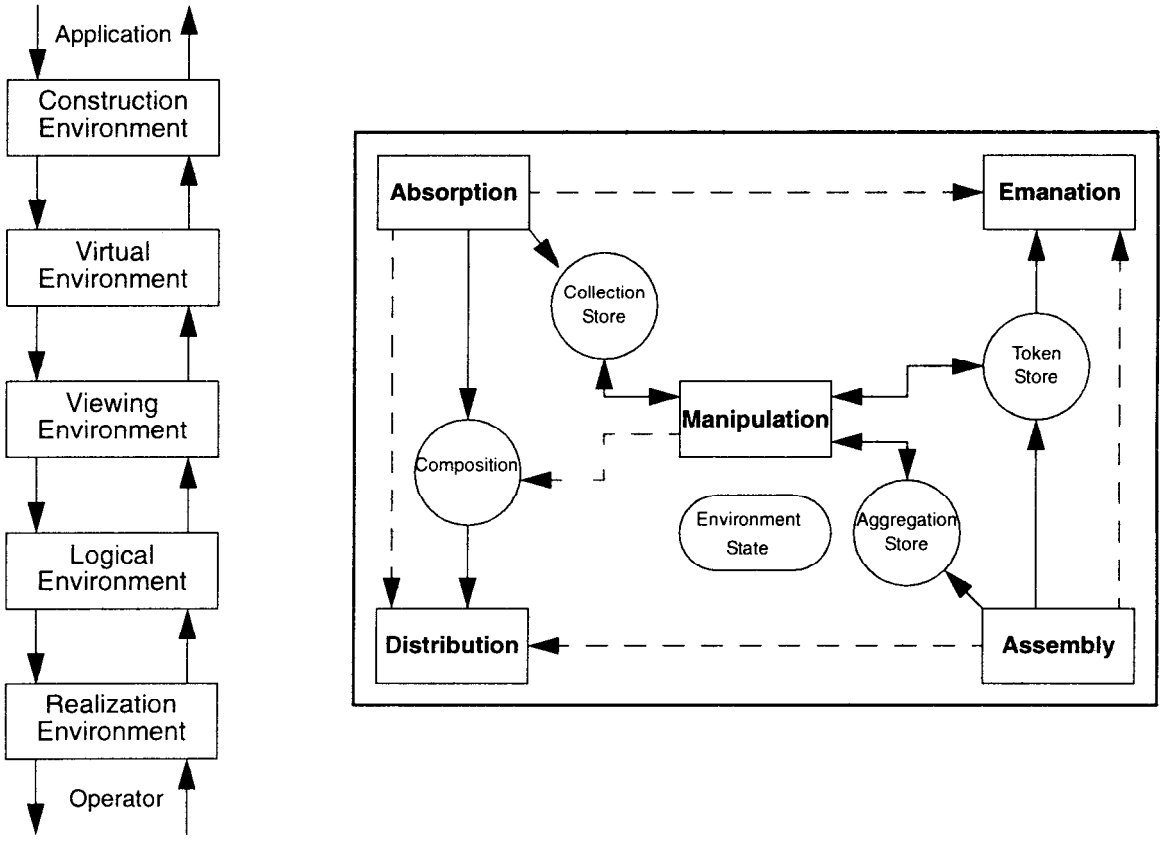

Figure 27 — CGRM pipeline and environment models

C.2 Architectural Links

There is no simple mapping between this architecture and the structures defined by this part of ISO/IEC 14478. Rather, the mapping depends on how the facilities available to the application and on how it combines them with the structures defined in this part. For example, one application might make use of an existing GKS'94 renderer as part of a larger network of modellers and

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

renderers. The GKS renderer has components that map onto the environments in the CGRM pipeline. However, other renderers in the same application may implement a subset of the pipeline. A number of processing modules, derived from the renderer object type defined in this part, might be linked together to realize a second complete pipeline.

C.3 Processing Links

In each environment, there is a single interface for incoming entities from the immediately higher environment concerning graphical output and a single interface for incoming entities from the immediately lower environment concerning graphical input. The same coordinate system is used for both input and output entities passing between each pair of adjacent environments. The coordinate systems used by the composition, collection store, token store and aggregation store within an environment are the same. Consequently, all transformations occur in the absorption and emanation processes. There are interfaces for storage and retrieval of all or parts of data elements in data capture metafiles.

C.4 Input and Output Primitives

This part of ISO/IEC 14478 does not distinguish between primitives used for input and those used for output.

C.5 Storage

In the CGRM, output is defined in terms of output primitives which make up a composition that is presented to the operator. Input is defined in terms of input tokens which make up a token store that is accumulated for the application in an appropriate form. Any connection between received input and generated output is conceptually handled by the application. The application may delegate this responsibility to specific environments. To support the creation and manipulation of complex presentations in a distributed environment, PREMO defines a storage facility — the scene object type — that can be understood as a means of realizing the various storage environments described in the CGRM. As noted previously however, there is no requirement that users of PREMO conform to the architecture of the CGRM.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex D
(informative)
A typical example scenario of MRI usage

Here is an overview of the actions that a client would carry out to construct the multimedia system shown in Figure 6 on page 22, and which was discussed in clause 6. The order given here is somewhat arbitrary, as a client may for example prefer to build the system on a node by node basis and would therefore interleave some of the steps described below.

a) Creation of devices needed for interaction. Using the factory finder mechanism described in PREMO Part 2, the client would create (or obtain a reference to) instances of the device types needed to make up the network. As the properties used when requesting object instances may include location, the objects so created may be distributed. For the system used in the example, the following devices are required:

1) Two modellers;
2) Three media engines, one each for audio, video and graphics;
3) A scene;
4) An audio renderer;
5) A graphics renderer;
6) A mouse (input device);
7) A router;
8) Three logical devices;
9) A coordinator;
10) Various Group objects and VirtualConnection objects.

b) Using the procedure illustrated in Annex B of PREMO part 3, the client uses the *VirtualConnections* and *Groups* to establish the direct connections between devices, for example, the link from the graphics renderer to the display. The structure of the system at this stage is shown below.

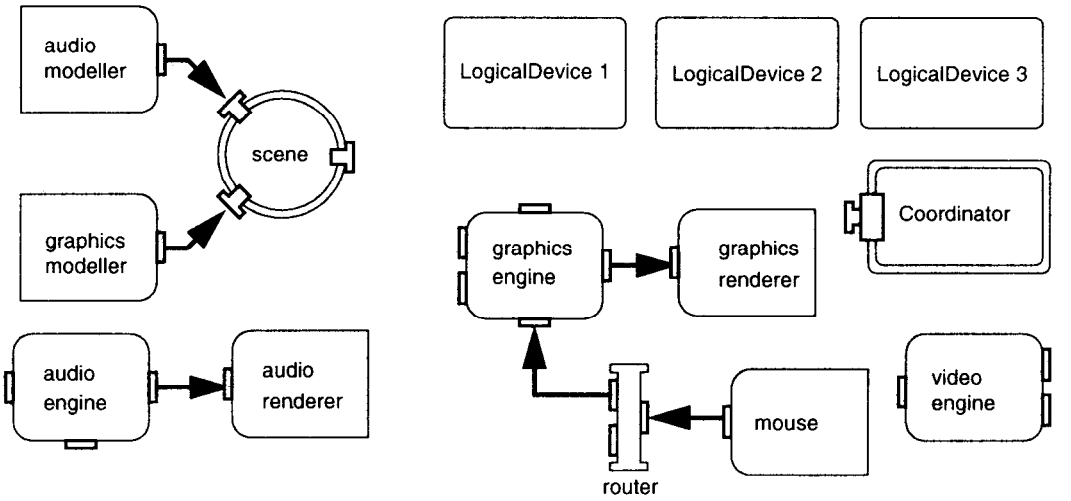

Figure 28 — AV system: basic components

c) The three instances of *LogicalDevice* are populated by the client with their respective components using the *acquireResource* operation inherited from the *Group* object type to add appropriate devices to each group (this is again illustrated in PREMO part 3). The client invokes the *definePort* operation on each *LogicalDevice* to obtain a port to access particular ports

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

of the devices contained within. The client does not explicitly connect the new port to the one on the target device; this is done by the definePort operation. The result is shown below.

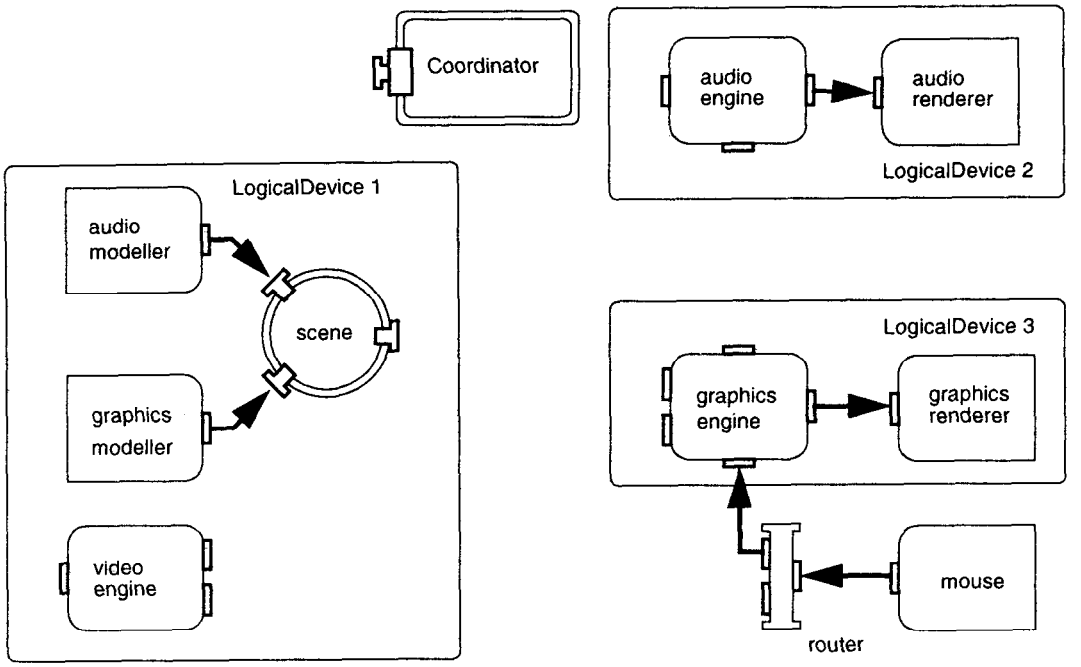

Figure 29 — AV system: adding resources to the logical devices

d) The client again uses the part 3 procedure to interconnect the appropriate ports of the LogicalDevices. The system now looks as shown below.

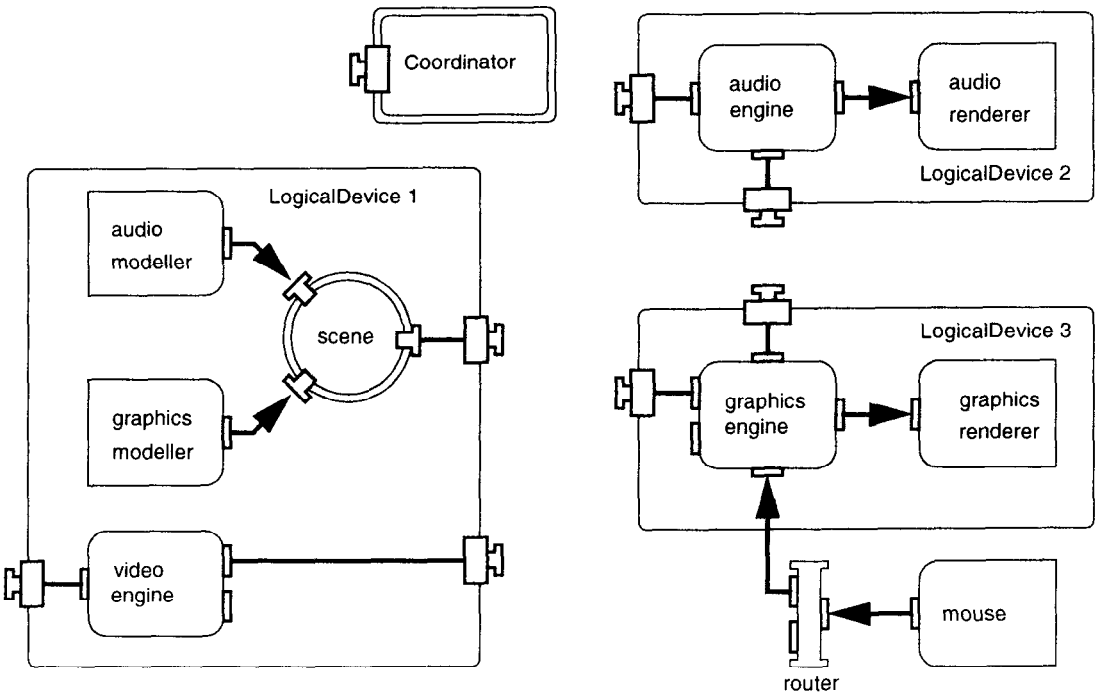

Figure 30 — AV system: establishing logical device ports

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

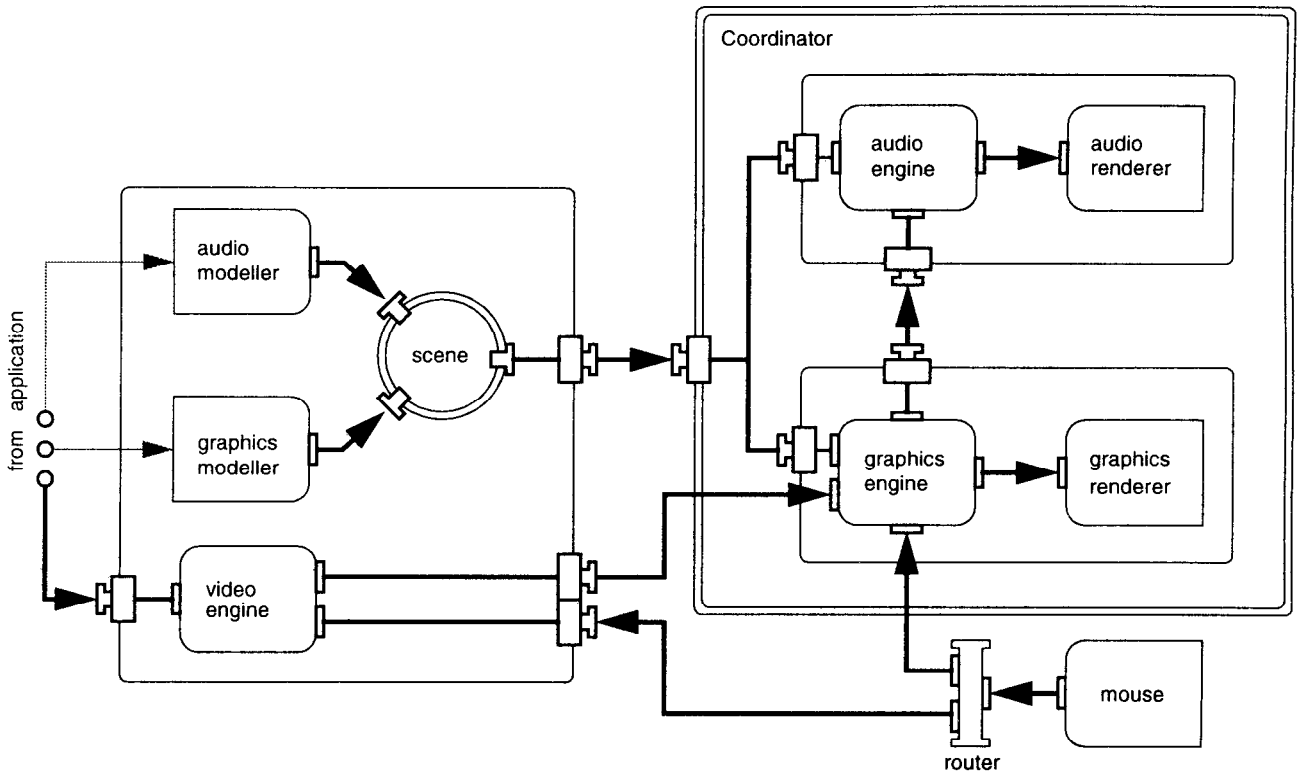

Figure 31 — AV system: introducing the coordinator

e) The three logical devices are encapsulated into the coordinator object, using the acquireResource operation as in step (c).

f) The client obtains three ports to the Synchronizer using the definePort operation. The final result is:

At this point the devices needed to carry out multimedia presentation are connected. The client may need to invoke instance-specific operations on those devices to initialize them, for example in the case of the graphics renderer to create a workstation or to establish viewing parameters.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

ICS 35.140

Descriptors: data processing, information interchange, graphic data processing, image processing, video data, audio data, coding (data conversion), coded representation.

Price based on 71 pages

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale