INTERNATIONAL STANDARD

ISO/IEC 14496-1

First edition
1999-12-15

Information technology — Coding of audio-visual objects —

Part 1: Systems

Technologies de l'information — Codage des objets audiovisuels —

Partie 1: Systèmes

Reference number
ISO/IEC 14496-1:1999(E)

Contents

Page

Figures

Tables

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, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this part of ISO/IEC 14496 may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

International Standard ISO/IEC 14496-1 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 29, *Coding of audio, picture, multimedia and hypermedia information*.

ISO/IEC 14496 consists of the following parts, under the general title *Coding of audio-visual objects*:

— *Part 1: Systems*

— *Part 2: Visual*

— *Part 3: Audio*

— *Part 4: Conformance testing*

— *Part 5: Reference software*

— *Part 6: Delivery Multimedia Integration Framework (DMIF)*

Annexes C, G and H form a normative part of this part of ISO/IEC 14496. Annexes A, B, D, E, F, I, J and K are for information only.

0 Introduction

0.1 Overview

ISO/IEC 14496 specifies a system for the communication of interactive audio-visual scenes. This specification includes the following elements:

1. the coded representation of natural or synthetic, two-dimensional (2D) or three-dimensional (3D) objects that can be manifested audibly and/or visually (audio-visual objects) (specified in part 1,2 and 3 of ISO/IEC 14496);
2. the coded representation of the spatio-temporal positioning of audio-visual objects as well as their behavior in response to interaction (scene description, specified in this part of ISO/IEC 14496);
3. the coded representation of information related to the management of data streams (synchronization, identification, description and association of stream content, specified in this part of ISO/IEC 14496); and
4. a generic interface to the data stream delivery layer functionality (specified in part 6 of ISO/IEC 14496).

The overal operation of a system communicating audio-visual scenes can be paraphrased as follows:

At the sending terminal, the audio-visual scene information is compressed, supplemented with synchronization information and passed to a delivery layer that multiplexes it into one or more coded binary streams that are transmitted or stored. At the receiving terminal, these streams are demultiplexed and decompressed. The audio-visual objects are composed according to the scene description and synchronization information and presented to the end user. The end user may have the option to interact with this presentation. Interaction information can be processed locally or transmitted back to the sending terminal. ISO/IEC 14496 defines the syntax and semantics of the bitstreams that convey such scene information, as well as the details of their decoding processes.

This part of ISO/IEC 14496 specifies the following tools:

— a terminal model for time and buffer management;

— a coded representation of interactive audio-visual scene description information (Binary Format for Scenes – BIFS);

— a coded representation of metadata for the identification, description and logical dependencies of the elementary streams (Object descriptors and other Descriptors);

— a coded representation of descriptive audio-visual content information (object content information – OCI);

— an interface to intellectual property management and protection (IPMP) systems;

— a coded representation of synchronization information (sync layer – SL); and

— a multiplexed representation of individual elementary streams in a single stream (FlexMux).

These various elements are described functionally in this subclause and specified in the normative clauses that follow.

0.2 Architecture

The information representation specified in ISO/IEC 14496-1 describes the means to create an interactive audio-visual scene in terms of coded audio-visual information and associated scene description information. The entity that composes and sends, or receives and presents such a coded representation of an interactive audio-visual scene is generically referred to as an "audio-visual terminal" or just "terminal". This terminal may correspond to a standalone application or be part of an application system.

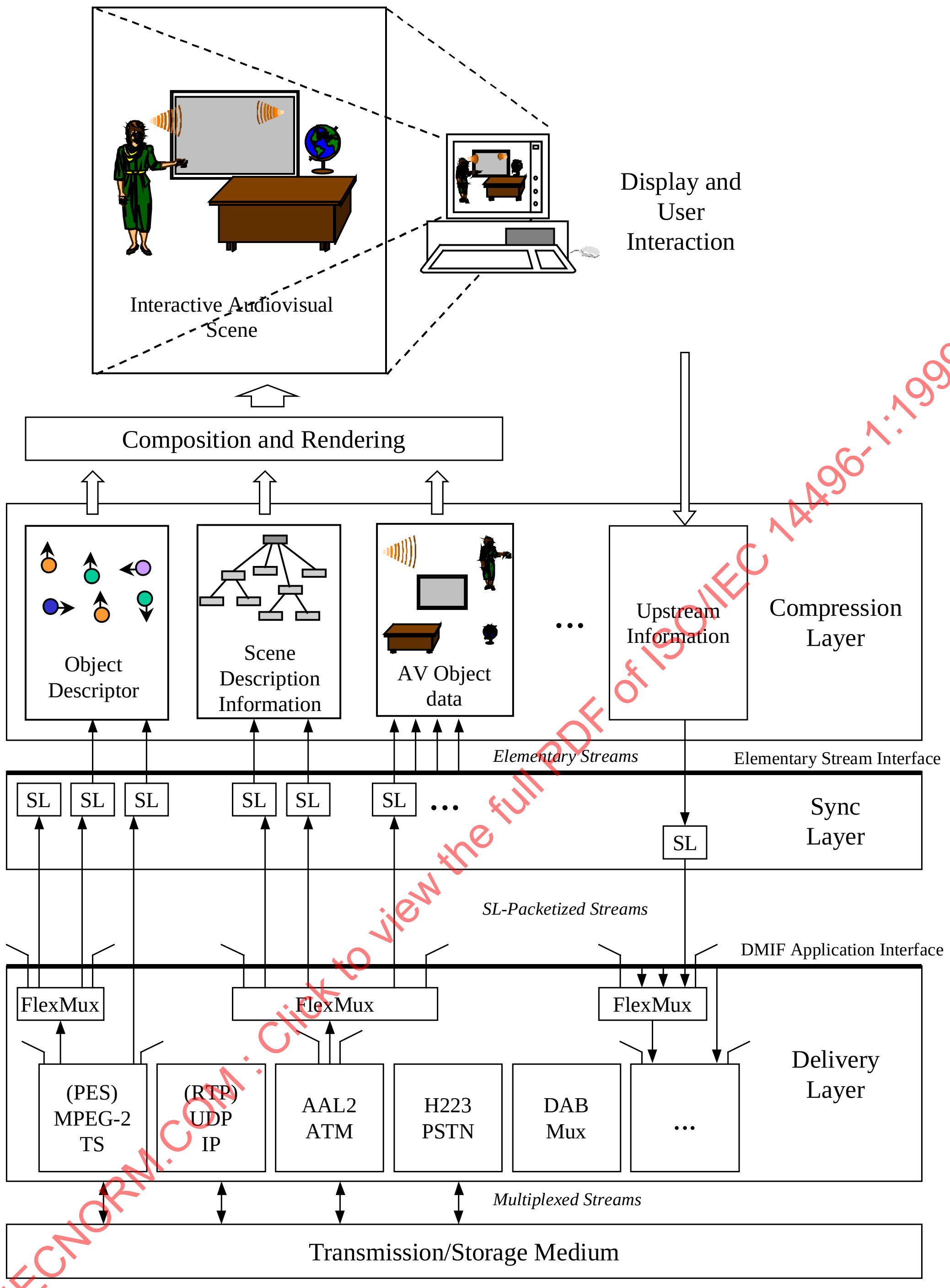


Figure 1 - The ISO/IEC 14496 terminal architecture

The basic operations performed by such a receiver terminal are as follows. Information that allows access to content complying with ISO/IEC 14496 is provided as initial session set up information to the terminal. Part 6 of ISO/IEC 14496 defines the procedures for establishing such session contexts as well as the interface to the delivery layer that generically abstracts the storage or transport medium. The initial set up information allows, in a recursive manner, to locate one or more elementary streams that are part of the coded content representation. Some of these elementary streams may be grouped together using the multiplexing tool described in ISO/IEC 14496-1.

Elementary streams contain the coded representation of either audio or visual data or scene description information. Elementary streams may as well themselves convey information to identify streams, to describe logical dependencies between streams, or to describe information related to the content of the streams. Each elementary stream contains only one type of data.

Elementary streams are decoded using their respective stream-specific decoders. The audio-visual objects are composed according to the scene description information and presented by the terminal's presentation device(s). All these processes are synchronized according to the Systems Decoder Model (SDM) using the synchronization information provided at the synchronization layer.

These basic operations are depicted in Figure 1, and are described in more detail below.

0.3 Terminal Model: Systems Decoder Model

The systems decoder model provides an abstract view of the behavior of a terminal complying with ISO/IEC 14496-1. Its purpose is to enable a sending terminal to predict how the receiving terminal will behave in terms of buffer management and synchronization when reconstructing the audio-visual information that comprises the presentation. The systems decoder model includes a systems timing model and a systems buffer model which are described briefly in the following subclauses.

0.3.1 Timing Model

The timing model defines the mechanisms through which a receiving terminal establishes a notion of time that enables it to process time-dependent events. This model also allows the receiving terminal to establish mechanisms to maintain synchronization both across and within particular audio-visual objects as well as with user interaction events. In order to facilitate these functions at the receiving terminal, the timing model requires that the transmitted data streams contain implicit or explicit timing information. Two sets of timing information are defined in ISO/IEC 14496-1: clock references and time stamps. The former convey the sending terminal's time base to the receiving terminal, while the latter convey a notion of relative time for specific events such as the desired decoding or composition time for portions of the encoded audio-visual information.

0.3.2 Buffer Model

The buffer model enables the sending terminal to monitor and control the buffer resources that are needed to decode each elementary stream in a presentation. The required buffer resources are conveyed to the receiving terminal by means of descriptors at the beginning of the presentation. The terminal can then decide whether or not it is capable of handling this particular presentation. The buffer model allows the sending terminal to specify when information may be removed from these buffers and enables it to schedule data transmission so that the appropriate buffers at the receiving terminal do not overflow or underflow.

0.4 Multiplexing of Streams: The Delivery Layer

The term delivery layer is used as a generic abstraction of any existing transport protocol stack that may be used to transmit and/or store content complying with ISO/IEC 14496. The functionality of this layer is not within the scope of ISO/IEC 14496-1, and only the interface to this layer is considered. This interface is the DMIF Application Interface (DAI) specified in ISO/IEC 14496-6. The DAI defines not only an interface for the delivery of streaming data, but also for signaling information required for session and channel set up as well as tear down. A wide variety of delivery mechanisms exist below this interface, with some of them indicated in Figure 1. These mechanisms serve for transmission as well as storage of streaming data, i.e., a file is considered to be a particular instance of a delivery layer. For applications where the desired transport facility does not fully address the needs of a service according to the specifications in ISO/IEC 14496, a simple multiplexing tool (FlexMux) with low delay and low overhead is defined in ISO/IEC 14496-1.

0.5 Synchronization of Streams: The Sync Layer

Elementary streams are the basic abstraction for any streaming data source. Elementary streams are conveyed as sync layer-packetized (SL-packetized) streams at the DMIF Application Interface. This packetized representation additionally provides timing and synchronization information, as well as fragmentation and random access information. The sync layer (SL) extracts this timing information to enable synchronized decoding and, subsequently, composition of the elementary stream data.

0.6 The Compression Layer

The compression layer receives data in its encoded format and performs the necessary operations to decode this data. The decoded information is then used by the terminal's composition, rendering and presentation subsystems.

0.6.1 Object Description Framework

The purpose of the object description framework is to identify and describe elementary streams and to associate them appropriately to an audio-visual scene description. Object descriptors serve to gain access to ISO/IEC 14496 content. Object content information and the interface to intellectual property management and protection systems are also part of this framework.

An object descriptor is a collection of one or more elementary stream descriptors that provide the configuration and other information for the streams that relate to either an audio-visual object or a scene description. Object descriptors are themselves conveyed in elementary streams. Each object descriptor is assigned an identifier (object descriptor ID), which is unique within a defined name scope. This identifier is used to associate audio-visual objects in the scene description with a particular object descriptor, and thus the elementary streams related to that particular object.

Elementary stream descriptors include information about the source of the stream data, in form of a unique numeric identifier (the elementary stream ID) or a URL pointing to a remote source for the stream. Elementary stream descriptors also include information about the encoding format, configuration information for the decoding process and the sync layer packetization, as well as quality of service requirements for the transmission of the stream and intellectual property identification. Dependencies between streams can also be signaled within the elementary stream descriptors. This functionality may be used, for example, in scalable audio or visual object representations to indicate the logical dependency of a stream containing enhancement information, to a stream containing the base information. It can also be used to describe alternative representations for the same content (e.g. the same speech content in various languages).

0.6.1.1 Intellectual Property Management and Protection

The intellectual property management and protection (IPMP) framework for ISO/IEC 14496 content consists of a normative interface that permits an ISO/IEC 14496 terminal to host one or more IPMP Systems. The IPMP interface consists of IPMP elementary streams and IPMP descriptors. IPMP descriptors are carried as part of an object descriptor stream. IPMP elementary streams carry time variant IPMP information that can be associated to multiple object descriptors.

The IPMP System itself is a non-normative component that provides intellectual property management and protection functions for the terminal. The IPMP System uses the information carried by the IPMP elementary streams and descriptors to make protected ISO/IEC 14496 content available to the terminal. An application may choose not to use an IPMP System, thereby offering no management and protection features.

0.6.1.2 Object Content Information

Object content information (OCI) descriptors convey descriptive information about audio-visual objects. The main content descriptors are: content classification descriptors, keyword descriptors, rating descriptors, language descriptors, textual descriptors, and descriptors about the creation of the content. OCI descriptors can be included directly in the related object descriptor or elementary stream descriptor or, if it is time variant, it may be carried in an elementary stream by itself. An OCI stream is organized in a sequence of small, synchronized entities called events that contain a set of OCI descriptors. OCI streams can be associated to multiple object descriptors.

0.6.2 Scene Description Streams

Scene description addresses the organization of audio-visual objects in a scene, in terms of both spatial and temporal attributes. This information allows the composition and rendering of individual audio-visual objects after the respective decoders have reconstructed the streaming data for them. For visual data, ISO/IEC 14496-1 does not mandate particular composition algorithms. Hence, visual composition is implementation dependent. For audio data, the composition process is defined in a normative manner in 9.2.2.13 and ISO/IEC 14496-3.

The scene description is represented using a parametric approach (BIFS - Binary Format for Scenes). The description consists of an encoded hierarchy (tree) of nodes with attributes and other information (including event sources and targets). Leaf nodes in this tree correspond to elementary audio-visual data, whereas intermediate nodes group this material to form audio-visual objects, and perform grouping, transformation, and other such operations on audio-visual objects (scene description nodes). The scene description can evolve over time by using scene description updates.

In order to facilitate active user involvement with the presented audio-visual information, ISO/IEC 14496-1 provides support for user and object interactions. Interactivity mechanisms are integrated with the scene description information, in the form of linked event sources and targets (routes) as well as sensors (special nodes that can trigger events based on specific conditions). These event sources and targets are part of scene description nodes, and thus allow close coupling of dynamic and interactive behavior with the specific scene at hand. ISO/IEC 14496-1, however, does not specify a particular user interface or a mechanism that maps user actions (e.g., keyboard key presses or mouse movements) to such events.

Such an interactive environment may not need an upstream channel, but ISO/IEC 14496 also provides means for client-server interactive sessions with the ability to set up upstream elementary streams and associate them to specific downstream elementary streams.

0.6.3 Audio-visual Streams

The coded representations of audio and visual information are described in ISO/IEC 14496-3 and ISO/IEC 14496-2, respectively. The reconstructed audio-visual data are made available to the composition process for potential use during the scene rendering.

0.6.4 Upchannel Streams

Downchannel elementary streams may require upchannel information to be transmitted from the receiving terminal to the sending terminal (e.g., to allow for client-server interactivity). Figure 1 indicates the flowpath for an elementary stream from the receiving terminal to the sending terminal. The content of upchannel streams is specified in the same part of the specification that defines the content of the downstream data. For example, upchannel control streams for video downchannel elementary streams are defined in ISO/IEC 14496-2.

Information technology — Coding of audio-visual objects —

Part 1: **Systems**

1 Scope

This part of ISO/IEC 14496 specifies system level functionalities for the communication of interactive audio-visual scenes. More specifically:

1. system level description of the coded representation of natural or synthetic, two-dimensional (2D) or three-dimensional (3D) objects that can be manifested audibly and/or visually (audio-visual objects);

2. the coded representation of the spatio-temporal positioning of audio-visual objects as well as their behavior in response to interaction (scene description); and

3. the coded representation of information related to the management of data streams (synchronization, identification, description and association of stream content).

2 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 14496. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this part of ISO/IEC 14496 are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the lastest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

[1] ISO 639-2:1998, *Codes for the representation of names of languages – Part 2: Alpha-3 code*.

[2] ISO 3166-1:1997, *Codes for the representation of names of countries and their subdivisions – Part 1: Country codes*.

[3] ISO/IEC 10646-1:1993, *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane*.

[4] ITU-T Rec. T.81 (1992)|ISO/IEC 10918-1:1994, *Information technology – Digital compression and coding of continuous-tone still images: Requirements and guidelines*.

[5] ISO/IEC 11172-2:1993, *Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s – Part 2: Video*.

[6] ISO/IEC 11172-3:1993, *Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s – Part 3: Audio*.

[7] ITU-T Rec. H.262 (1995)|ISO/IEC 13818-2:1996, *Information technology – Generic coding of moving pictures and associated audio information: Video*.

[8] ISO/IEC 13818-3:1998, *Information technology – Generic coding of moving pictures and associated audio information – Part 3: Audio*.

[9] ISO/IEC 13818-7:1997, *Information technology – Generic coding of moving pictures and associated audio information – Part 7: Advanced Audio Coding (AAC).*

[10] ISO/IEC 14772-1:1998, *Information technology – Computer graphics and image processing – The Virtual Reality Modeling Language – Part 1: Functional specification and UTF-8 encoding.*

[11] ISO/IEC 16262:1998, *Information technology – ECMAScript language specification.*

[12] IEEE Std 754-1985, *Standard for Binary Floating-Point Arithmetic.*

3 Additional References

None cited.

4 Definitions

4.1 Access Unit (AU)
An individually accessible portion of data within an *elementary stream*. An access unit is the smallest data entity to which timing information can be attributed.

4.2 Alpha Map
The representation of the transparency parameters associated with a texture map.

4.3 Audio-visual Object
A representation of a natural or synthetic object that has an audio and/or visual manifestation. The representation corresponds to a node or a group of nodes in the BIFS scene description. Each audio-visual object is associated with zero or more *elementary streams* using one or more *object descriptors.*

4.4 Audio-visual Scene (AV Scene)
A set of audio-visual objects together with scene description information that defines their spatial and temporal attributes including behaviors resulting from object and user interactions.

4.5 Binary Format for Scene (BIFS)
A coded representation of a parametric scene description format.

4.6 Buffer Model
A model that defines how a terminal complying with ISO/IEC 14496 manages the buffer resources that are needed to decode a presentation.

4.7 Byte Aligned
A position in a coded bit stream with a distance of a multiple of 8-bits from the first bit in the stream.

4.8 Clock Reference
A special time stamp that conveys a reading of a time base.

4.9 Composition
The process of applying scene description information in order to identify the spatio-temporal attributes and hierarchies of audio-visual objects..

4.10 Composition Memory (CM)
A random access memory that contains composition units.

4.11 Composition Time Stamp (CTS)
An indication of the nominal composition time of a composition unit.

4.12 Composition Unit (CU)
An individually accessible portion of the output that a decoder produces from access units.

4.13 Compression Layer
The layer of a system according to the specifications in ISO/IEC 14496 that translates between the coded representation of an elementary stream and its decoded representation. It incorporates the decoders.

4.14 Decoder
An entity that translates between the coded representation of an elementary stream and its decoded representation.

4.15 Decoding buffer (DB)
A buffer at the input of a decoder that contains access units.

4.16 Decoder configuration
The configuration of a decoder for processing its elementary stream data by using information contained in its elementary stream descriptor.

4.17 Decoding Time Stamp (DTS)
An indication of the nominal decoding time of an access unit.

4.18 Delivery Layer
A generic abstraction for delivery mechanisms (computer networks, etc.) able to store or transmit a number of multiplexed elementary streams or FlexMux streams.

4.19 Descriptor
A data structure that is used to describe particular aspects of an elementary stream or a coded audio-visual object.

4.20 DMIF Application Interface (DAI)
An interface specified in ISO/IEC 14496-6. It is used here to model the exchange of SL-packetized stream data and associated control information between the sync layer and the delivery layer.

4.21 Elementary Stream (ES)
A consecutive flow of mono-media data from a single source entity to a single destination entity on the compression layer.

4.22 Elementary Stream Descriptor
A structure contained in object descriptors that describes the encoding format, initialization information, sync layer configuration, and other descriptive information about the content carried in an elementary stream.

4.23 Elementary Stream Interface (ESI)
An interface modeling the exchange of elementary stream data and associated control information between the compression layer and the sync layer.

4.24 FlexMux Channel (FMC)
A label to differentiate between data belonging to different constituent streams within one FlexMux Stream. A sequence of data in one FlexMux channel within a FlexMux stream corresponds to one single SL-packetized stream.

4.25 FlexMux Packet
The smallest data entity managed by the FlexMux tool. It consists of a header and a payload.

4.26 FlexMux Stream
A sequence of FlexMux Packets with data from one or more SL-packetized streams that are each identified by their own FlexMux channel.

4.27 FlexMux tool
A tool that allows the interleaving of data from multiple data streams.

4.28 Graphics Profile
A profile that specifies the permissible set of graphical elements of the BIFS tool that may be used in a scene description stream. Note that BIFS comprises both graphical and scene description elements.

4.29 Inter
A mode for coding parameters that uses previously coded parameters to construct a prediction.

4.30 Intra
A mode for coding parameters that does not make reference to previously coded parameters to perform the encoding.

4.31 Initial Object Descriptor
A special object descriptor that allows the receiving terminal to gain initial access to portions of content encoded according to ISO/IEC 14496. It conveys profile and level information to describe the complexity of the content.

4.32 Intellectual Property Identification (IPI)
A unique identification of one or more elementary streams corresponding to parts of one or more audio-visual objects.

4.33 Intellectual Property Management and Protection (IPMP) System
A generic term for mechanisms and tools to manage and protect intellectual property. Only the interface to such systems is normatively defined.

4.34 Object Clock Reference (OCR)
A clock reference that is used by a decoder to recover the time base of the encoder of an elementary stream.

4.35 Object Content Information (OCI)
Additional information about content conveyed through one or more elementary streams. It is either aggregated to individual elementary stream descriptors or is itself conveyed as an elementary stream.

4.36 Object Descriptor (OD)
A descriptor that aggregates one or more elementary streams by means of their elementary stream descriptors and defines their logical dependencies.

4.37 Object Descriptor Command
A command that identifies the action to be taken on a list of object descriptors or object descriptor IDs, e.g., update or remove.

4.38 Object Descriptor Profile
A profile that specifies the configurations of the object descriptor tool and the sync layer tool that are allowed.

4.39 Object Descriptor Stream
An elementary stream that conveys object descriptors encapsulated in object descriptor commands.

4.40 Object Time Base (OTB)
A time base valid for a given elementary stream, and hence for its decoder. The OTB is conveyed to the decoder via object clock references. All time stamps relating to this object's decoding process refer to this time base.

4.41 Parametric Audio Decoder
A set of tools for representing and decoding speech signals coded at bit rates between 6 Kbps and 16 Kbps, according to the specifications in ISO/IEC 14496-3.

4.42 Quality of Service (QoS)
The performance that an elementary stream requests from the delivery channel through which it is transported. QoS is characterized by a set of parameters (e.g., bit rate, delay jitter, bit error rate, etc.).

4.43 Random Access
The process of beginning to read and decode a coded representation at an arbitrary point within the elementary stream.

4.44 Reference Point
A location in the data or control flow of a system that has some defined characteristics.

4.45 Rendering
The action of transforming a scene description and its constituent audio-visual objects from a common representation space to a specific presentation device (i.e., speakers and a viewing window).

4.46 Rendering Area
The portion of the display device's screen into which the scene description and its constituent audio-visual objects are to be rendered.

4.47 Scene Description
Information that describes the spatio-temporal positioning of audio-visual objects as well as their behavior resulting from object and user interactions. The scene description makes reference to elementary streams with audio-visual data by means of pointers to object descriptors.

4.48 Scene Description Stream
An elementary stream that conveys scene description information.

4.49 Scene Graph Elements
The elements of the BIFS tool that relate only to the structure of the audio-visual scene (spatio-temporal temporal positioning of audio-visual objects as well as their behavior resulting from object and user interactions) excluding the audio, visual and graphics nodes as specified in clause 13.

4.50 Scene Graph Profile
A profile that defines the permissible set of scene graph elements of the BIFS tool that may be used in a scene description stream. Note that BIFS comprises both graphical and scene description elements.

4.51 SL-Packetized Stream (SPS)
A sequence of sync layer Packets that encapsulate one elementary stream.

4.52 Structured Audio
A method of describing synthetic sound effects and music as defined by ISO/IEC 14496-3.

4.53 Sync Layer (SL)
A layer to adapt elementary stream data for communication across the DMIF Application Interface, providing timing and synchronization information, as well as fragmentation and random access information. The sync layer syntax is configurable and can be configured to be empty.

4.54 Sync Layer Configuration
A configuration of the sync layer syntax for a particular elementary stream using information contained in its elementary stream descriptor.

4.55 Sync Layer Packet (SL-Packet)
The smallest data entity managed by the sync layer consisting of a configurable header and a payload. The payload may consist of one complete access unit or a partial access unit.

4.56 Syntactic Description Language (SDL)
A language defined by ISO/IEC 14496-1 that allows the description of a bitstream's syntax.

4.57 Systems Decoder Model (SDM)
A model that provides an abstract view of the behavior of a terminal compliant to ISO/IEC 14496. It consists of the buffer model and the timing model.

4.58 System Time Base (STB)
The time base of the terminal. Its resolution is implementation-dependent. All operations in the terminal are performed according to this time base.

4.59 Terminal
A system that sends, or receives and presents the coded representation of an interactive audio-visual scene as defined by ISO/IEC 14496-1. It can be a standalone system, or part of an application system complying with ISO/IEC 14496.

4.60 Time Base
The notion of a clock; it is equivalent to a counter that is periodically incremented.

4.61 Timing Model

A model that specifies the semantic meaning of timing information, how it is incorporated (explicitly or implicitly) in the coded representation of information, and how it can be recovered at the receiving terminal.

4.62 Time Stamp

An indication of a particular time instant relative to a time base.

5 Abbreviations and Symbols

AU	Access Unit
AV	Audio-visual
BIFS	Binary Format for Scene
CM	Composition Memory
CTS	Composition Time Stamp
CU	Composition Unit
DAI	DMIF Application Interface (see ISO/IEC 14496-6)
DB	Decoding Buffer
DTS	Decoding Time Stamp
ES	Elementary Stream
ESI	Elementary Stream Interface
ESID	Elementary Stream Identifier
FAP	Facial Animation Parameters
FAPU	FAP Units
FDP	Facial Definition Parameters
FIG	FAP Interpolation Graph
FIT	FAP Interpolation Table
FMC	FlexMux Channel
FMOD	The floating point modulo (remainder) operator which returns the remainder of x/y such that: Fmod(x/y) = x – k*y, where k is an integer, sgn(fmod(x/y)) = sgn(x), and abs(fmod(x/y)) < abs(y)
IP	Intellectual Property
IPI	Intellectual Property Identification
IPMP	Intellectual Property Management and Protection
NCT	Node Coding Tables
NDT	Node Data Type
NINT	Nearest INTeger value
OCI	Object Content Information
OCR	Object Clock Reference
OD	Object Descriptor
ODID	Object Descriptor Identifier
OTB	Object Time Base
PLL	Phase Locked Loop
QoS	Quality of Service
SAOL	Structured Audio Orchestra Language
SASL	Structured Audio Score Language
SDL	Syntactic Description Language
SDM	Systems Decoder Model
SL	Synchronization Layer
SL-Packet	Synchronization Layer Packet
SPS	SL-Packetized Stream
STB	System Time Base
TTS	Text-To-Speech
URL	Universal Resource Locator
VOP	Video Object Plane
VRML	Virtual Reality Modeling Language

6 Conventions

For the purpose of unambiguously defining the syntax of the various bitstream components defined by the normative parts of ISO/IEC 14496 a *syntactic description language* is used. This language allows the specification of the mapping of the various parameters in a binary format as well as how they are placed in a serialized bitstream. The definition of the language is provided in clause 12.

7 Systems Decoder Model

7.1 Introduction

The purpose of the systems decoder model (SDM) is to provide an abstract view of the behavior of a terminal complying with ISO/IEC 14496. It may be used by the sender to predict how the receiving terminal will behave in terms of buffer management and synchronization when decoding data received in the form of elementary streams. The systems decoder model includes a timing model and a buffer model.

The systems decoder model specifies:

1. the interface for accessing demultiplexed data streams (DMIF Application Interface),
2. decoding buffers for coded data for each elementary stream,
3. the behavior of elementary stream decoders,
4. composition memory for decoded data from each decoder, and
5. the output behavior of composition memory towards the compositor.

These elements are depicted in Figure 2. Each elementary stream is attached to one single decoding buffer. More than one elementary stream may be connected to a single decoder (e.g., in a decoder of a scaleable audio-visual object).

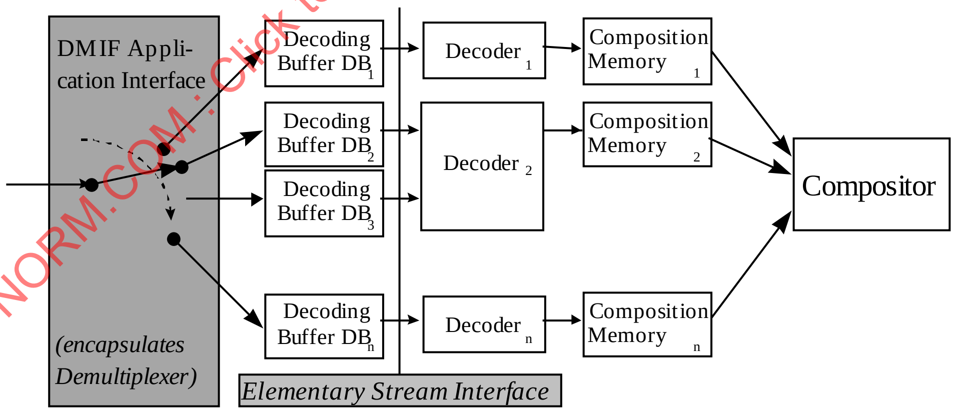


Figure 2 - Systems Decoder Model

7.2 Concepts of the Systems Decoder Model

This subclause defines the concepts necessary for the specification of the timing and buffering model. The sequence of definitions corresponds to a walk from the left to the right side of the SDM illustration in Figure 2.

7.2.1 DMIF Application Interface (DAI)

For the purposes of the systems decoder model, the DMIF Application Interface encapsulates the demultiplexer and provides access to streaming data that is consumed by the decoding buffers. The streaming data received through the DAI consists of SL-packetized streams. The required properties of the DAI are described in 10.4. The DAI semantics are fully specified in ISO/IEC 14496-6.

7.2.2 SL-Packetized Stream (SPS)

An SL-packetized stream consists of a sequence of packets, according to the syntax and semantics specified in 10.2, that encapsulate a single elementary stream. The packets contain elementary stream data partitioned in access units as well as side information, e.g., for timing and access unit labeling. SPS data enter the decoding buffers.

7.2.3 Access Units (AU)

Elementary stream data is partitioned into access units. The delineation of an access unit is completely determined by the entity that generates the elementary stream (e.g., the compression layer). An access unit is the smallest data entity to which timing information can be attributed. Two access units from the same elementary stream shall never refer to the same decoding or composition time. Any further partitioning of the data in an elementary stream is not visible for the purposes of the Systems Decoder Model. Access units are conveyed by SL-packetized streams and are received by the decoding buffers. The decoders consume access units with the necessary side information (e.g., time stamps) from the decoding buffers.

NOTE — An ISO/IEC 14496-1 compliant terminal implementation is not required to process each incoming access unit as a whole. It is furthermore possible to split an access unit into several fragments for transmission as specified in clause 10. This allows the sending terminal to dispatch partial AUs immediately as they are generated during the encoding process. Such partial AUs may have significance for improved error resilience.

7.2.4 Decoding Buffer (DB)

The decoding buffer is a buffer at the input of an elementary stream decoder in the receiving terminal that receives and stores access units. The Systems Buffer Model enables the sending terminal to monitor the decoding buffer resources that are used during a presentation.

7.2.5 Elementary Streams (ES)

Streaming data received at the output of a decoding buffer, independent of its content, is considered as an elementary stream for the purpose of ISO/IEC 14496. The elementary streams are produced and consumed by the compression layer entities (encoders and decoders, respectively). ISO/IEC 14496 assumes that the integrity of an elementary stream is preserved from end to end, from the ESI of the sending terminal to the ESI of the receiving terminal.

7.2.6 Elementary Stream Interface (ESI)

The elementary stream interface models the exchange of elementary stream data and associated control information between the compression layer and the sync layer. At the receiving terminal the ESI is located at the output of the decoding buffer. The ESI is specified in 10.3.

7.2.7 Decoder

For the purposes of this model, the decoder extracts access units from the decoding buffer at precisely defined points in time and places composition units, the results of the decoding processes, in the composition memory. A decoder may be attached to several decoding buffers.

7.2.8 Composition Units (CU)

Decoders consume access units and produce composition units. An access unit corresponds to an integer number of composition units. Composition units reside in composition memory.

7.2.9 Composition Memory (CM)

The composition memory is a random access memory that contains composition units. The size of this memory is not normatively specified.

7.2.10 Compositor

The compositor takes composition units out of the composition memory and either consumes them (e.g. composes and presents them, in the case of audio-visual data) or skips them. The compositor is not specified in ISO/IEC 14496-1, as the details of this operation are not relevant within the context of the System Decoder Model. Subclause 7.3.5 defines which composition units are available to the compositor at any instant of time.

7.3 Timing Model Specification

The timing model relies on clock references and time stamps to synchronize audio-visual data conveyed by one or more elementary streams. The concept of a clock with its associated clock references is used to convey the notion of time to a receiving terminal. Time stamps are used to indicate the precise time instants at which the receiving terminal consumes the access units in the decoding buffers or may access the composition units resident in the composition memory. The time stamps are therefore associated with access units and composition units. The semantics of the timing model are defined in the subsequent clauses. The syntax for conveying timing information is specified in 10.2.

NOTE — This timing model is designed for rate-controlled ("push") applications.

7.3.1 System Time Base (STB)

The System Time Base (STB) defines the terminal's notion of time. The resolution of the STB is implementation dependent. All actions of the terminal are scheduled according to this time base for the purpose of this timing model.

NOTE — This does not imply that all terminals compliant with ISO/IEC 14496 operate on one single STB.

7.3.2 Object Time Base (OTB)

The object time base (OTB) defines the notion of time for a given data stream. The resolution of this OTB can be selected as required by the application or as defined by a profile. All time stamps that the sending terminal inserts in a coded data stream refer to this time base. The OTB of a data stream is known at the receiving terminal either by means of object clock reference information inserted in the stream or by an indication that its time base is slaved to a time base conveyed with another stream, as specified in 10.2.3.

NOTE 1 — Elementary streams may be created for the sole purpose of conveying time base information.

NOTE 2 — The receiving terminal's System Time Base need not be locked to any of the available object time bases.

7.3.3 Object Clock Reference (OCR)

A special kind of time stamps, object clock references (OCR), are used to convey the OTB to the elementary stream decoder. The value of the OCR corresponds to the value of the OTB at the time the sending terminal generates the object clock reference time stamp. OCR time stamps are placed in the SL packet header as described in 10.2.4. The receiving terminal shall extract and evaluate the OCR when its first byte enters its decoding buffer.

7.3.4 Decoding Time Stamp (DTS)

Each access unit has an associated nominal decoding time, the time at which it must be available in the decoding buffer for decoding. The AU is not guaranteed to be available in the decoding buffer either before or after this time. Decoding is assumed to occur instantaneously when the instant of time indicated by the DTS is reached.

This point in time can be implicitly specified if the (constant) temporal distance between successive access units is indicated in the setup of the elementary stream (see 10.2.3). Otherwise a decoding time stamp (DTS) whose syntax is defined in 10.2.4 conveys this point in time.

A decoding time stamp shall only be conveyed for an access unit that carries a composition time stamp as well, and only if the DTS and CTS values are different. Presence of both time stamps in an AU may indicate a reversal between coding order and composition order.

7.3.5 Composition Time Stamp (CTS)

Each composition unit has an associated nominal composition time, the time at which it must be available in the composition memory for composition. The CU is not guaranteed to be available in the composition memory *for composition* before this time. Since the SDM assumes an instantaneous decoding process, the CU is available to the *decoder*, at that instant in time corresponding to the DTS of the corresponding AU, for further use (e.g. in prediction processes).

This instant in time is implicitly known, if the (constant) temporal distance between successive composition units is indicated in the setup of the elementary stream. Otherwise a composition time stamp (CTS) whose syntax is defined in 10.2.4 conveys this instant in time.

The current CU is instantaneously accessible by the compositor anytime between its composition time and the composition time of the subsequent CU. If a subsequent CU does not exist, the current CU becomes unavailable at the end of the lifetime of its elementary stream (i.e., when its elementary stream descriptor is removed).

7.3.6 Occurrence and Precision of Timing Information in Elementary Streams

The frequency at which DTS, CTS and OCR values are to be inserted in the bitstream as well as the precision, jitter and drift are application and profile dependent. Some usage considerations can be found in 10.2.7.

7.3.7 Time Stamps for Dependent Elementary Streams

An audio-visual object may refer to multiple elementary streams that constitute a scaleable content representation (see 8.7.1.5). Such a set of elementary streams shall adhere to a single object time base. Temporally co-located access units for such elementary streams are then identified by identical DTS or CTS values.

EXAMPLE

The example in Figure 3 illustrates the arrival of two access units at the Systems Decoder. Due to the constant delay assumption of the model (see 7.4.2 below), the arrival times correspond to the instants in time when the sending terminal has sent the respective AUs. The sending terminal must select this instant in time so that the Decoding Buffer at the receiving terminal never overflows or underflows. At the receiving terminal, an AU is instantaneously decoded, at that instant in time corresponding to its DTS, and the resulting CU(s) are placed in the composition memory and remain there until the subsequent CU(s) arrive or the associated object descriptor is removed.

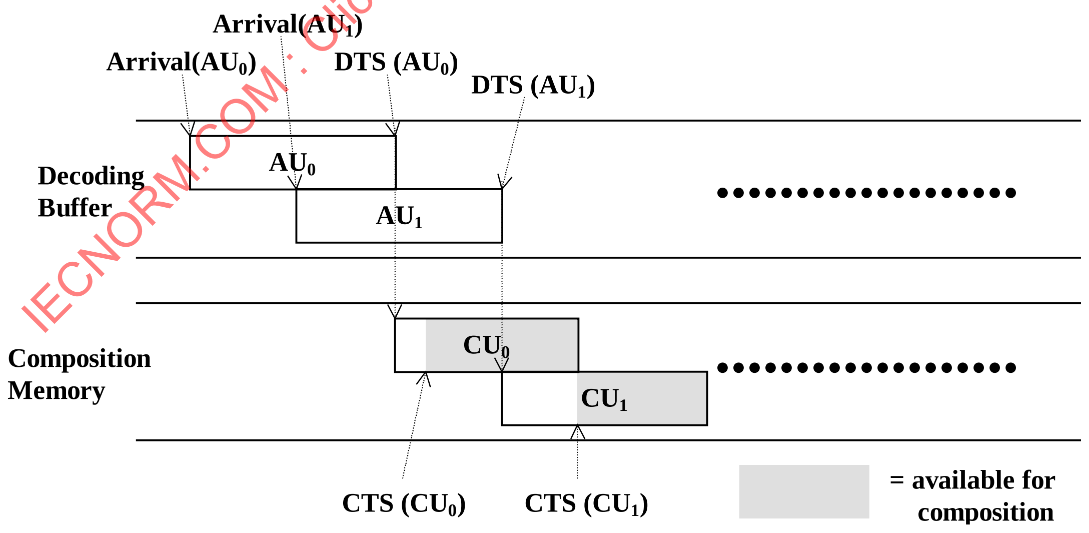


Figure 3 - Composition unit availability

7.4 Buffer Model Specification

7.4.1 Elementary Decoder Model

Figure 4 indicates one branch of the Systems Decoder Model (Figure 2). This simplified model is used to specify the buffer model. It treats each elementary stream separately and therefore, associates a composition memory with only one decoder. The legend following Figure 4 elaborates on the symbols used in this figure.

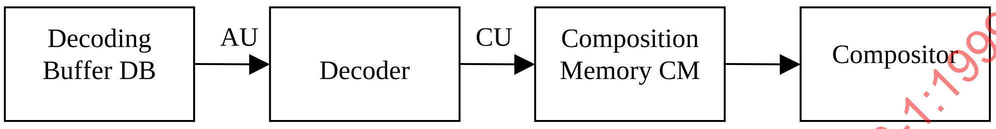


Figure 4 - Flow diagram for the Systems Decoder Model

Legend:

DB	Decoding buffer for the elementary stream.
CM	Composition memory for the elementary stream.
AU	The current access unit input to the Decoder.
CU	The current composition unit input to the composition memory. CU results from decoding AU. There may be several composition units resulting from decoding one access unit.

7.4.2 Assumptions

7.4.2.1 Constant end-to-end delay

Data transmitted in real time have a timing model in which the end-to-end delay from the encoder input at the sending terminal, to the decoder output at the receiving terminal, is constant. This delay is equal to the sum of the delay due to the encoding process, subsequent buffering, multiplexing at the sending terminal, the delay due to the delivery layers and the delay due to the demultiplexing, decoder buffering and decoding processes at the receiving terminal.

Note that the receiving terminal is free to add a temporal offset (delay) to the absolute values of all time stamps if it can cope with the additional buffering needed. However, the temporal difference between two time stamps (that determines the temporal distance between the associated AUs or CUs) has to be preserved for real-time performance.

7.4.2.2 Demultiplexer

The end-to-end delay between multiplexer output, at the sending terminal, and demultiplexer input, at the receiving terminal, is constant.

7.4.2.3 Decoding Buffer

The needed decoding buffer size is known by the sending terminal and conveyed to the receiving terminal as specified in 8.6.5.

The size of the decoding buffer is measured in bytes.

The decoding buffer is filled at the rate given by the maximum bit rate for this elementary stream (if this information is conveyed by the sending terminal), and with a zero rate otherwise. The maximum bit rate is conveyed by the sending terminal as a part of the decoder configuration information during the set up phase for each elementary stream (see 8.6.5).

Information is received from the DAI in the form of SL packets. The SL packet headers are removed at the input to the decoding buffers.

7.4.2.4 Decoder

The decoding processes are assumed to be instantaneous for the purposes of the Systems Decoder Model.

7.4.2.5 Composition Memory

The mapping of an AU to one or more CUs (by the decoder) is known implicitly at both the sending and the receiving terminals.

7.4.2.6 Compositor

The composition processes are assumed to be instantaneous for the purposes of the Systems Decoder Model.

7.4.3 Managing Buffers: A Walkthrough

In this example, we assume that the model is used in a “push” scenario. In applications where non-real time content is to be delivered, flow control by suitable signaling may be established to request access units at the time they are needed at the receiving terminal. The mechanisms for doing so are application-dependent, and are not specified in ISO/IEC 14496.

The behaviors of the various elements in the SDM are modeled as follows:

— The sending terminal signals the required decoding buffer resources to the receiving terminal before starting the delivery. This is done as specified in 8.6.5 either explicitly, by requesting the decoding buffer sizes for individual elementary streams, or implicitly, by indicating a profile (see clause 13). The decoding buffer size is measured in bytes.

— The sending terminal models the behavior of the decoding buffers by making the following assumptions :

— Each decoding buffer is filled at the maximum bitrate specified for its associated elementary stream if this information is available.

— At the instant of time corresponding to its DTS, an AU is instantaneously decoded and removed from the decoding buffer.

— At the instant of time corresponding to its DTS, a known amount of CUs corresponding to the just decoded AU are put in the composition memory.

The current CU is available to the compositor between instants of time corresponding to the CTS of the current CU and the CTS of the subsequent CU. If a subsequent CU does not exist, the current CU becomes unavailable at the end of lifetime of its data stream.

Using these assumptions on the buffer model, the sending terminal may freely use the space in the decoding buffers. For example, it may deliver data for several AUs of a stream, for non real time usage, to the receiving terminal, and pre-store them in the DB long before they have to be decoded (assuming sufficient space is available). Subsequently, the full delivery bandwidth may be used to transfer data of a real time stream just in time. The composition memory may be used, for example, as a reordering buffer. In the case of visual decoding, it may contain the decoded P-frames needed by a video decoder for the decoding of intermediate B-frames, before the arrival of the CTS of the latest P-frame.

8 Object Description Framework

8.1 Introduction

The scene description (specified in clause 9) and the elementary streams that convey streaming data are the basic building blocks of the architecture of ISO/IEC 14496-1. Elementary streams carry data for audio or visual objects as well as for the scene description itself. The object description framework provides the link between elementary streams and the scene description. The scene description declares the spatio-temporal relationship of audio-visual objects, while the object description framework specifies the elementary stream resources that provide the time-varying data for the scene. This indirection facilitates independent changes to the scene structure, the properties of the elementary streams (e.g. its encoding) and their delivery.

The object description framework consists of a set of descriptors that allows to identify, describe and properly associate elementary streams to each other and to audio-visual objects used in the scene description. Numeric identifiers, called ObjectDescriptorIDs, associate object descriptors to appropriate nodes in the scene description. Object descriptors are themselves conveyed in elementary streams to allow time stamped changes to the available set of object descriptors to be made.

Each object descriptor is itself a collection of descriptors that describe one or more elementary streams that are associated to a single node and that usually relate to a single audio or visual object. This allows to indicate a scaleable content representation as well as multiple alternative streams that convey the same content, e.g., in multiple qualities or different languages.

An elementary stream descriptor within an object descriptor identifies a single elementary stream with a numeric identifier, called ES_ID. Each elementary stream descriptor contains the information necessary to initiate and configure the decoding process for the elementary stream, as well as intellectual property identification. Optionally, additional information may be associated to a single elementary stream, most notably quality of service requirements for its transmission or a language indication. Both, object descriptors and elementary stream descriptors may use URLs to point to remote object descriptors or a remote elementary stream source, respectively.

The object description framework provides the hooks to implement intellectual property management and protection (IPMP) systems. IPMP information is conveyed both through IPMP descriptors as part of the object descriptor stream and through IPMP streams that carry time variant IPMP information. The structure of IPMP descriptors and IPMP streams is specified in this clause while their internal syntax and semantics and, hence, the operation of the IPMP system is outside the scope of ISO/IEC 14496.

Object content information allows the association of metadata with a whole presentation or with individual object descriptors or with elementary stream descriptors. A set of OCI descriptors is defined that either form an integral part of an object descriptor or elementary stream descriptor or are conveyed by means of a proper OCI stream that allows the conveyance of time variant object content information.

Access to ISO/IEC 14496 content is gained through an initial object descriptor that needs to be made available through means not defined in ISO/IEC 14496. The initial object descriptor in the simplest case points to the scene description stream and the corresponding object descriptor stream. The access scenario is outlined in 8.7.3.

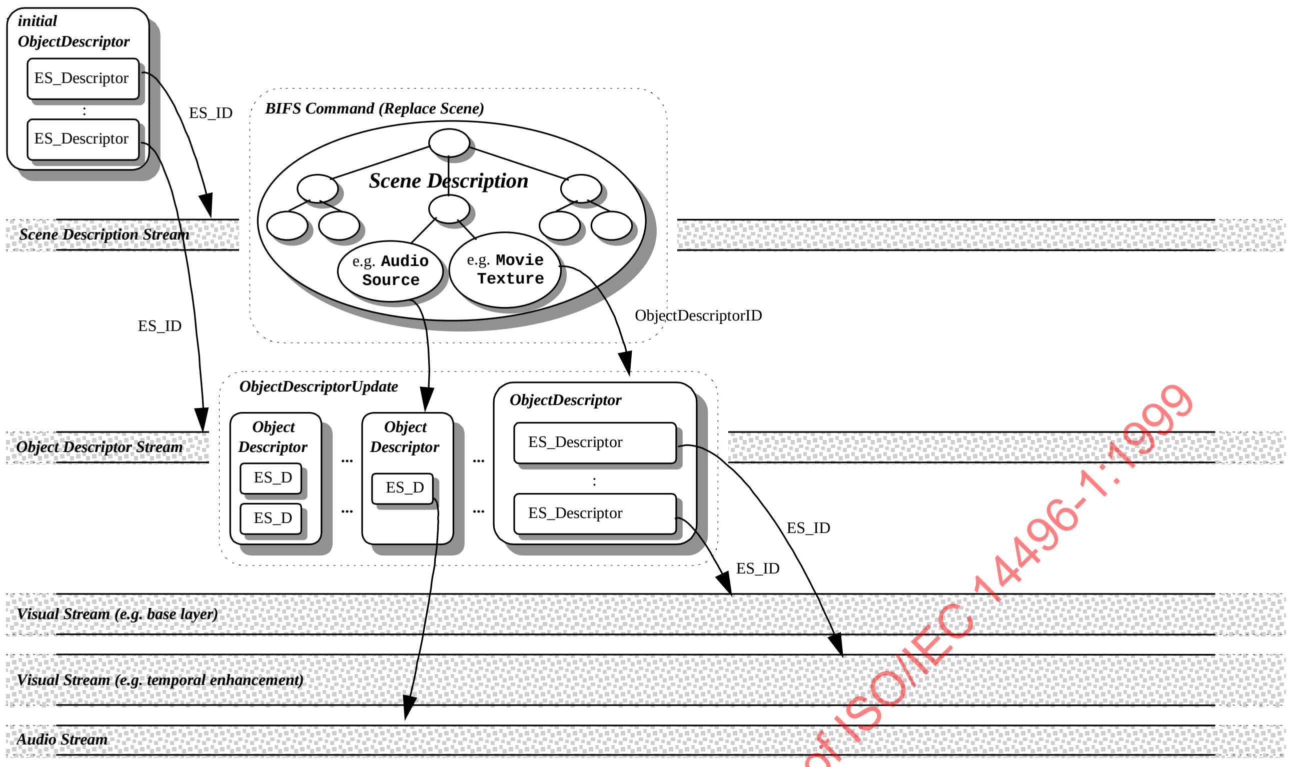


Figure 5 - Object descriptors linking scene description to elementary streams

The remainder of this clause is structured in the following way:

— Subclause 8.2 specifies the data structures on which the object descriptor framework is based.

— Subclause 8.3 specifies the concepts of the IPMP elements in the object description framework.

— Subclause 8.4 specifies the object content information elements in the object description framework.

— Subclause 8.5 specifies the object descriptor stream and the syntax and semantics of the command set that allows the update or removal of object descriptor components.

— Subclause 8.6 specifies the syntax and semantics of the object descriptor and its component descriptors.

— Subclause 8.7 specifies rules for object descriptor usage as well as the procedure to access content through object descriptors.

— Subclause 8.8 specifies the usage of the IPMP system interface.

8.2 Common data structures

8.2.1 Overview

The commands and descriptors defined in this subclause constitute self-describing classes, identified by unique class tags. Each class encodes explicitly its size in bytes. This facilitates future compatible extensions of the commands and descriptors. They may be expanded with additional syntax elements that are ignored by an OD decoder that expects an earlier revision of a class. In addition, anywhere in a syntax where a set of tagged classes is expected it is permissible to intersperse expandable classes with unknown class tag values. These classes shall be skipped, using the encoded size information.

The remainder of this clause defines the syntax and semantics of the command and descriptor classes. Some commands and descriptors contain themselves a set of component descriptors. They are said to *aggregate a set of component descriptors.*

8.2.2 BaseDescriptor

8.2.2.1 Syntax

```
abstract aligned(8) expandable(2^28-1) class BaseDescriptor : bit(8) tag=0 {
   // empty. To be filled by classes extending this class.
}
```

8.2.2.2 Semantics

This class is an abstract base class that is extended by the descriptor classes specified in 8.6. Each descriptor constitutes a self-describing class, identified by a unique class tag. This abstract base class establishes a common name space for the class tags of these descriptors. The values of the class tags are defined in Table 1. As an expandable class the size of each class instance in bytes is encoded and accessible through the instance variable sizeOfInstance (see 12.3.3).

A class that allows the aggregation of classes of type BaseDescriptor may actually aggregate any of the classes that extend BaseDescriptor.

NOTE — User private descriptors may have an internal structure, for example to identify the country or manufacturer that uses a specific descriptor. The tags and semantics for such user private descriptors may be managed by a registration authority if required.

Table 1 - List of Class Tags for Descriptors

Tag value	**Tag name**
0x00	Forbidden
0x01	ObjectDescrTag
0x02	InitialObjectDescrTag
0x03	ES_DescrTag
0x04	DecoderConfigDescrTag
0x05	DecSpecificInfoTag
0x06	SLConfigDescrTag
0x07	ContentIdentDescrTag
0x08	SupplContentIdentDescrTag
0x09	IPI_DescrPointerTag
0x0A	IPMP_DescrPointerTag
0x0B	IPMP_DescrTag
0x0C	QoS_DescrTag
0x0D	RegistrationDescrTag
0x0E-0x3F	Reserved for ISO use (descriptors)
0x40	ContentClassificationDescrTag
0x41	KeyWordDescrTag
0x42	RatingDescrTag
0x43	LanguageDescrTag
0x44	ShortTextualDescrTag
0x45	ExpandedTextualDescrTag
0x46	ContentCreatorNameDescrTag
0x47	ContentCreationDateDescrTag
0x48	OCICreatorNameDescrTag
0x49	OCICreationDateDescrTag
0x4A-0x5F	Reserved for ISO use (OCI extensions)
0x60-0xBF	Reserved for ISO use
0xC0-0xFE	User private
0xFF	Forbidden

The following additional symbolic names are introduced:

ExtDescrTagStartRange = 0x80
ExtDescrTagEndRange = 0xFE
OCIDescrTagStartRange = 0x40
OCIDescrTagEndRange = 0x5F

8.2.3 BaseCommand

8.2.3.1 Syntax

```
abstract aligned(8) expandable(2^28-1) class BaseCommand : bit(8) tag=0 {
   // empty. To be filled by classes extending this class.
}
```

8.2.3.2 Semantics

This class is an abstract base class that is extended by the command classes specified in 8.5.5. Each command constitutes a self-describing class, identified by a unique class tag. This abstract base class establishes a common name space for the class tags of these commands. The values of the class tags are defined in Table 2. As an expandable class the size of each class instance in bytes is encoded and accessible through the instance variable sizeOfInstance (see 12.3.3).

A class that allows the aggregation of classes of type BaseCommand may actually aggregate any of the classes that extend BaseCommand.

NOTE — User private commands may have an internal structure, for example to identify the country or manufacturer that uses a specific command. The tags and semantics for such user private command may be managed by a registration authority if required.

Table 2 - List of Class Tags for Commands

Tag value	Tag name
0x00	forbidden
0x01	ObjectDescrUpdateTag
0x02	ObjectDescrRemoveTag
0x03	ES_DescrUpdateTag
0x04	ES_DescrRemoveTag
0x05	IPMP_DescrUpdateTag
0x06	IPMP_DescrRemoveTag
0x07-0xBF	Reserved for ISO (command tags)
0xC0-0xFE	User private
0xFF	forbidden

8.3 Intellectual Property Management and Protection (IPMP)

8.3.1 Overview

The intellectual property management and protection (IPMP) framework for ISO/IEC 14496 content consists of a normative interface that permits an ISO/IEC 14496 terminal to host one or more IPMP Systems. An IPMP System is a non-normative component that provides intellectual property management and protection functions for the terminal.

The IPMP interface consists of IPMP elementary streams and IPMP descriptors. The normative structure of IPMP elementary streams is specified in this subclause. IPMP descriptors are carried as part of an object descriptor stream and are specified in 8.6.13. The IPMP interface allows applications (or derivative application standards) to build specialized IPMP Systems. Alternatively, an application may choose not to use an IPMP System, thereby offering no management and protection features. The IPMP System uses the information carried by the IPMP elementary streams and descriptors to make protected ISO/IEC 14496 content available to the terminal. The detailed semantics and decoding process of the IPMP System are not in the scope of ISO/IEC 14496. The usage of the IPMP System Interface, however, is explained in 8.8.

8.3.2 IPMP Streams

8.3.2.1 Structure of the IPMP Stream

The IPMP stream is an elementary stream that passes time-varying information to one or more IPMP Systems. This is accomplished by periodically sending a sequence of IPMP messages along with the content at a period determined by the IPMP System(s).

8.3.2.2 Access Unit Definition

An IPMP access unit consists of one or more IPMP messages, as defined in 8.3.2.5. All IPMP messages that are to be processed at the same instant in time shall constitute a single access unit. Access units in IPMP streams shall be labeled and time-stamped by suitable means. This shall be done via the related flags and the composition time stamps, respectively, in the SL packet header (see 10.2.4). The composition time indicates the point in time at which an IPMP access unit becomes valid, i.e., when the embedded IPMP messages shall be evaluated. Decoding and composition time for an IPMP access unit shall always have the same value.

An access unit does not necessarily convey or update the complete set of IPMP messages that are currently required. In that case it just modifies the persistent state of the IPMP system. However, if an access unit conveys the complete set of IPMP messages required at a given point in time it shall set the `randomAccessPointFlag` in the SL packet header to ‘1’ for this access unit. Otherwise, the `randomAccessPointFlag` shall be set to ‘0’.

NOTE — An SL packet with `randomAccessPointFlag=1` but with no IPMP messages in it indicates that at the current time instant no IPMP messages are required for operation.

8.3.2.3 Time Base for IPMP Streams

The time base associated to an IPMP stream shall be indicated by suitable means. This shall be done by means of object clock reference time stamps in the SL packet headers (see 10.2.4) for this stream or by indicating the elementary stream from which this IPMP stream inherits the time base (see 10.2.3). All time stamps in the SL-packetized IPMP stream refer to this time base.

8.3.2.4 IPMP Decoder Configuration

8.3.2.4.1 Syntax

```
class IPMPDecoderConfiguration extends DecoderSpecificInfo : bit(8) tag=DecSpecificInfoTag {
   // IPMP system specific configuration information
}
```

8.3.2.4.2 Semantics

An IPMP system may require information to initialize its operation. This information shall be conveyed by extending the `decoderSpecificInfo` class as specified in 8.6.6. If utilized, `IPMPDecoderConfiguration` shall be conveyed in the `ES_Descriptor` declaring the IPMP stream.

8.3.2.5 IPMP message syntax and semantics

8.3.2.5.1 Syntax

```
class IPMP_Message() extends ExpandableBaseClass
{
   bit(16)  IPMPS_Type;
   if (IPMPS_Type == 0) {
      bit(8) URLString[sizeOfInstance-2];
   } else {
      bit(8) IPMP_data[sizeOfInstance-2];
   }
}
```

8.3.2.5.2 Semantics

The `IPMP_Message` conveys control information for an IPMP System.

`IPMPS_Type` - the type of the IPMP System. A zero value does not correspond to an IPMP System, but indicates the presence of a URL. A Registration Authority as designated by the ISO shall assign valid values for this field.

`URLString[]` - contains a UTF-8 [3] encoded URL that shall point to the location of a remote IPMP_Message whose IPMP_data shall be used in place of locally provided data.

`IPMP_data` - opaque data to control the IPMP System.

8.4 Object Content Information (OCI)

8.4.1 Overview

Audio-visual objects that are associated with elementary stream data through an object descriptor may have additional object content information attached to them. For this purpose, a set of OCI descriptors is defined in 8.6.17. OCI descriptors may directly be included as part of an `ObjectDescriptor` or `ES_Descriptor` as defined in 8.6.1.

In order to accommodate time variant OCI that is separable from the object descriptor stream, OCI descriptors may as well be conveyed in an OCI stream. An OCI stream is referred to through an ES_Descriptor, with the `streamType` field set to OCI_Stream. How OCI streams may be aggregated to object descriptors is defined in 8.7.1.3. The structure of the OCI stream is defined in this subclause.

8.4.2 OCI Streams

8.4.2.1 Structure of the OCI Stream

The OCI stream is an elementary stream that conveys time-varying object content information, termed OCI events. Each OCI event consists of a number of OCI descriptors.

8.4.2.2 Access Unit Definition

An OCI access unit consists of one or more OCI_Events, as described in 8.4.2.5. Access units in OCI elementary streams shall be labelled and time stamped by suitable means. This shall be done by means of the related flags and the composition time stamp, respectively, in the SL packet header (see 10.2.4). The composition time indicates the point in time when an OCI access unit becomes valid, i.e., when the embedded OCI events shall be added to the list of events. Decoding and composition time for an OCI access unit shall always have the same value.

An access unit may or may not convey or update the complete set of OCI events that are currently valid. In the latter case, it just modifies the persistent state of the OCI decoder. However, if an access unit conveys the complete set of OCI events valid at a given point in time it shall set the `randomAccessPointFlag` in the SL packet header to ‘1’ for this access unit. Otherwise, the `randomAccessPointFlag` shall be set to ‘0’.

NOTE — An SL packet with `randomAccessPointFlag=1` but with no OCI events in it indicates that at the current time instant no valid OCI events exist.

8.4.2.3 Time Base for OCI Streams

The time base associated with an OCI stream shall be indicated by suitable means. This shall be done by the use of object clock reference time stamps in the SL packet headers (see 10.2.4) for this stream or by indicating the elementary stream from which this OCI stream inherits the time base (see 10.2.3). All time stamps in the SL-packetized OCI stream refer to this time base.

8.4.2.4 OCI Decoder Configuration

8.4.2.4.1 Syntax

```
class OCIDecoderConfiguration extends DecoderSpecificInfo  : bit(8) tag=DecSpecificInfoTag {
   const bit(8) versionLabel = 0x01;
}
```

8.4.2.4.2 Semantics

This information is needed to initialize operation of the OCI decoder. It shall be conveyed by extending the `decoderSpecificInfo` class as specified in 8.6.6. `OCIDecoderConfiguration` shall be conveyed in the `ES_Descriptor` declaring the OCI stream.

`versionLabel` **–** indicates the version of OCI specification used on the corresponding OCI data stream. Only the value 0x01 is allowed; all the other values are reserved.

8.4.2.5 OCI_Events syntax and semantics

8.4.2.5.1 Syntax

```
class OCI_Event extends ExpandableBaseClass {
   bit(15) eventID;
   bit(1)  absoluteTimeFlag;
   bit(32) startingTime;
   bit(32) duration;
   OCI_Descriptor OCI_Descr[1 .. 255];
}
```

8.4.2.5.2 Semantics

`eventID` – contains the identification number of the described event that is unique within the scope of this OCI stream.

`absoluteTimeFlag` – indicates the time base for `startingTime` as described below.

`startingTime` – indicates the starting time of the event in hours, minutes, seconds and hundredth of seconds. The format is 8 digits, the first 6 digits expressing hours, minutes and seconds with 4 bits each in binary coded decimal and the last two expressing hundredth of seconds in hexadecimal using 8 bits.

EXAMPLE — 02:36:45:89 is coded as “0x023645” concatenated with “0b0101.1001” (89 in binary), resulting to “0x02364559”.

If `absoluteTimeFlag` is set to zero, `startingTime` is relative to the object time base of the corresponding object. In that case it is the responsibility of the application to ensure that this object time base is conveyed such that `startingTime` can be identified unambiguously (see 10.2.7). If `absoluteTimeFlag` is set to one, `startingTime` is expressed as an absolute value, refering to wall clock time.

`duration` – contains the duration of the corresponding object in hours, minutes, seconds and hundredth of seconds. The format is 8 digits, the first 6 digits expressing hours, minutes and seconds with 4 bits each in binary coded decimal and the last two expressing hundredth of seconds in hexadecimal using 8 bits.

`OCI_Descr[]` – an array of one up to 255 `OCI_Descriptor` classes as specified in 8.6.17.2.

8.5 Object Descriptor Stream

8.5.1 Structure of the Object Descriptor Stream

Similar to the scene description, object descriptors are transported in a dedicated elementary stream, termed object descriptor stream. Within such a stream, it is possible to dynamically convey, update and remove complete object descriptors, or their component descriptors, the ES_Descriptors, and IPMP descriptors. The update mechanism allows, for example, to advertise new elementary streams for an audio-visual object as they become available, or to remove references to streams that are no longer available. Updates are time stamped to indicate the instant in time they take effect.

This subclause specifies the structure of the object descriptor elementary stream including the syntax and semantics of its constituent elements, the object descriptor commands (OD commands).

8.5.2 Access Unit Definition

An OD access unit consists of one or more OD commands, as described in 8.5.5. All OD commands that are to be processed at the same instant in time shall constitute a single access unit. Access units in object descriptor elementary streams shall be labelled and time stamped by suitable means. This shall be done by means of the related flags and the composition time stamp, respectively, in the SL packet header (see 10.2.4). The composition time indicates the point in time when an OD access unit becomes valid, i.e., when the embedded OD commands shall be executed. Decoding and composition time for an OD access unit shall always have the same value.

An access unit may not convey or update the complete set of object descriptors that are currently required. In that case it just modifies the persistent state of the object descriptor decoder. However, if an access unit conveys the complete set of object descriptors required at a given point in time it shall set the `randomAccessPointFlag` in the SL packet header to ‘1’ for this access unit. Otherwise, the `randomAccessPointFlag` shall be set to ‘0’.

NOTE — An SL packet with `randomAccessPointFlag=1` but with no OD commands in it indicates that at the current time instant no valid object descriptors exist.

8.5.3 Time Base for Object Descriptor Streams

The time base associated to an object descriptor stream shall be indicated by suitable means. This shall be done by means of object clock reference time stamps in the SL packet headers (see 10.2.4) for this stream or by indicating the elementary stream from which this object descriptor stream inherits the time base (see 10.2.3). All time stamps in the SL-packetized object descriptor stream refer to this time base.

8.5.4 OD Decoder Configuration

The object descriptor decoder does not require additional configuration information.

8.5.5 OD Command Syntax and Semantics

8.5.5.1 Overview

Object descriptors and their components as defined in 8.6 shall always be conveyed as part of one of the OD commands specified in this subclause. The commands describe the action to be taken on the components conveyed with the command, specifically ‘update’ or ‘remove’. Each command affects one or more object descriptors, ES_Descriptors or IPMP descriptors.

8.5.5.2 ObjectDescriptorUpdate

8.5.5.2.1 Syntax

```
class ObjectDescriptorUpdate extends BaseCommand : bit(8) tag=ObjectDescrUpdateTag {
   ObjectDescriptor OD[1 .. 255];
}
```

8.5.5.2.2 Semantics

The `ObjectDescriptorUpdate` class conveys a list of new or updated ObjectDescriptors. The components of an already existing ObjectDescriptor shall not be changed by an update, but an `ObjectDescriptorUpdate` may remove or add ES_Descriptors as components of the related object descriptor.

Removal of an `ES_Descriptor` within an `ObjectDescriptor` conveyed by this command is accomplished by omitting it from the array of `ES_Descriptors` aggregated to the `ObjectDescriptor`. Addition of an `ES_Descriptor` within an `ObjectDescriptor` conveyed by this command is accomplished by adding it to the array of `ES_Descriptors` aggregated to the `ObjectDescriptor`.

To update the characteristics of an elementary stream, it is required that its original `ES_Descriptor` be removed and the changed `ES_Descriptor` be conveyed.

`OD[]` – an array of `ObjectDescriptors` as defined in 8.6.2. The array shall have any number of one up to 255 elements.

8.5.5.3 ObjectDescriptorRemove

8.5.5.3.1 Syntax

```
class ObjectDescriptorRemove extends BaseCommand : bit(8) tag=ObjectDescrRemoveTag {
   bit(10) objectDescriptorId[(sizeOfInstance*8)/10];
}
```

8.5.5.3.2 Semantics

The `ObjectDescriptorRemove` class renders unavailable a set of object descriptors. The BIFS nodes associated to these object descriptors shall have no reference any more to the elementary streams that have been listed in the removed object descriptors. An objectDescriptorID that does not refer to a valid ObjectDescriptor is ignored.

NOTE — It is possible that a scene description node references an OD_ID which does not currently have an associated OD.

`ObjectDescriptorId[]` – an array of `ObjectDescriptorIDs` that indicates the object descriptors that are removed.

8.5.5.4 ES_DescriptorUpdate

8.5.5.4.1 Syntax

```
class ES_DescriptorUpdate extends BaseCommand : bit(8) tag=ES_DescrUpdateTag {
   bit(10) objectDescriptorId;
   ES_Descriptor ESD[1 .. 30];
}
```

8.5.5.4.2 Semantics

The `ES_DescriptorUpdate` class adds or updates references to elementary streams within the object descriptor labeled `objectDescriptorID`. Values of syntax elements of an updated `ES_Descriptor` shall remain unchanged.

To update the characterstics of an elementary stream, it is required that its original ES_Descriptor be removed and the changed ES_Descriptor be conveyed.

An elementary stream identified with a given ES_ID may be attached to more than one object descriptor. All corresponding `ES_Descriptors` refering to this ES_ID that are conveyed through either `ES_DescriptorUpdate` or `ObjectDescriptorUpdate` commands shall have identical content.

`objectDescriptorID` - identifies the `ObjectDescriptor` for which `ES_Descriptors` are updated. If the objectDescriptorID does not refer to any valid object descriptor, then this command is ignored.

`ESD[]` – an array of `ES_Descriptors` as defined in 8.6.4. The array shall have any number of one up to 30 elements.

8.5.5.5 ES_DescriptorRemove

8.5.5.5.1 Syntax

```
class ES_DescriptorRemove extends BaseCommand : bit(8) tag=ES_DescrRemoveTag {
   bit(10) objectDescriptorId;
   aligned (8) bit(16) ES_ID[1..30];
}
```

8.5.5.5.2 Semantics

The `ES_DescriptorRemove` class removes the reference to an elementary stream from an ObjectDescriptor and renders this stream unavailable for nodes referencing this ObjectDescriptor.

`objectDescriptorID` - identifies the `ObjectDescriptor` from which `ES_Descriptors` are removed. If the objectDescriptorID does not refer to a valid object descriptor in the same scope, then this command is ignored.

`ES_ID[]` – an array of `streamCount` `ES_IDs` that labels the `ES_Descriptors` to be removed from `objectDescriptorID`. If any of the ES_IDs do not refer to an `ES_Descriptor` currently referenced by the OD, then those ES_IDs are ignored.

8.5.5.6 IPMP_DescriptorUpdate

8.5.5.6.1 Syntax

```
class IPMP_DescriptorUpdate extends BaseCommand : bit(8) tag=IPMP_DescrUpdateTag {
   IPMP_Descriptor ipmpDescr[1..255];
}
```

8.5.5.6.2 Semantics

The `IPMP_DescriptorUpdate` class conveys a list of new or updated `IPMP_Descriptors`. An `IPMP_Descriptor` identified by an `IPMP_DescriptorID` that already exists shall be replaced by the new descriptor.

`IPMP_Descriptors` remain valid until they are replaced by another `IPMP_DescriptorUpdate` command or removed.

`ipmpDescr[]` – an array of `IPMP_Descriptor` as specified in 8.6.13.

8.5.5.7 IPMP_DescriptorRemove

8.5.5.7.1 Syntax

```
class IPMP_DescriptorRemove extends BaseCommand : bit(8) tag=IPMP_DescrRemoveTag {
   bit(8) IPMP_DescriptorID[1..255];
}
```

8.5.5.7.2 Semantics

The `IPMP_DescriptorRemove` class conveys a list of `IPMP_DescriptorsIDs` that identify the `IPMP_Descriptors` that shall be removed.

`IPMP_DescriptorID[]` - is a list of `IPMP_DescriptorIDs`.

8.6 Object Descriptor Components

8.6.1 Overview

Object descriptors contain various additional descriptors as their components, in order to describe individual elementary streams and their properties. They shall always be conveyed as part of one of the OD commands specified in the previous subclause. This subclause defines the syntax and semantics of object descriptors and their component descriptors.

8.6.2 ObjectDescriptor

8.6.2.1 Syntax

```
class ObjectDescriptor extends BaseDescriptor : bit(8) tag=ObjectDescrTag {
   bit(10) ObjectDescriptorID;
   bit(1) URL_Flag;
   const bit(5) reserved=0b1111.1;
   if (URL_Flag) {
      bit(8) URLlength;
      bit(8) URLstring[URLlength];
   } else {
      ES_Descriptor esDescr[1 .. 30];
      OCI_Descriptor ociDescr[0 .. 255];
      IPMP_DescriptorPointer ipmpDescrPtr[0 .. 255];
   }
   ExtensionDescriptor extDescr[0 .. 255];
}
```

8.6.2.2 Semantics

The ObjectDescriptor consists of three different parts.

The first part uniquely labels the object descriptor within its name scope (see 8.7.2.4) by means of an objectDescriptorId. Nodes in the scene description use objectDescriptorID to refer to the related object descriptor. An optional URLstring indicates that the actual object descriptor resides at a remote location.

The second part consists of a list of ES_Descriptors, each providing parameters for a single elementary as well as an optional set of object content information descriptors and pointers to IPMP descriptors for the contents for elementary stream content described in this object descriptor.

The third part is a set of optional descriptors that support the inclusion of future extensions as well as the transport of private data in a backward compatible way.

objectDescriptorId **–** This syntax element uniquely identifies the ObjectDescriptor within its name scope. The value 0 is forbidden and the value 1023 is reserved.

URL_Flag – a flag that indicates the presence of a URLstring.

URLlength – the length of the subsequent URLstring in bytes.

URLstring[] – A string with a UTF-8 [3] encoded URL that shall point to another ObjectDescriptor. Only the content of this object descriptor shall be returned by the delivery entity upon access to this URL. Within the current name scope, the new object descriptor shall be referenced by the objectDescriptorId of the object descriptor carrying the URLstring. On name scopes see 8.7.2.4. Permissible URLs may be constrained by profile and levels as well as by specific delivery layers.

esDescr[] – an array of ES_Descriptors as defined in 8.6.4. The array shall have any number of one up to 30 elements.

ociDescr[] – an array of OCI_Descriptors, as defined in 8.6.17.2, that relates to the audio-visual object(s) described by this object descriptor. The array shall have any number of zero up to 255 elements.

ipmpDescrPtr[] – an array of IPMP_DescriptorPointer, as defined in 8.6.12, that points to the IPMP_Descriptors related to the elementary stream(s) described by this object descriptor. The array shall have any number of zero up to 255 elements.

extDescr[] – an array of ExtensionDescriptors as defined in 8.6.15. The array shall have any number of zero up to 255 elements.

8.6.3 InitialObjectDescriptor

8.6.3.1 Syntax

```
class InitialObjectDescriptor extends BaseDescriptor : bit(8) tag=InitialObjectDescrTag {
   bit(10) ObjectDescriptorID;
   bit(1) URL_Flag;
   bit(1) includeInlineProfileLevelFlag;
   const bit(4) reserved=0b1111;
   if (URL_Flag) {
      bit(8) URLlength;
      bit(8) URLstring[URLlength];
   } else {
      bit(8) ODProfileLevelIndication;
      bit(8) sceneProfileLevelIndication;
      bit(8) audioProfileLevelIndication;
      bit(8) visualProfileLevelIndication;
      bit(8) graphicsProfileLevelIndication;
      ES_Descriptor ESD[1 .. 30];
      OCI_Descriptor ociDescr[0 .. 255];
      IPMP_DescriptorPointer ipmpDescrPtr[0 .. 255];
   }
   ExtensionDescriptor extDescr[0 .. 255];
}
```

8.6.3.2 Semantics

The `InitialObjectDescriptor` is a variation of the `ObjectDescriptor` specified in the previous subclause that allows to signal profile and level information for the content refered by it. It shall be used to gain initial access to ISO/IEC 14496 content (see 8.7.3).

`objectDescriptorId` **–** This syntax element uniquely identifies the `ObjectDescriptor` within its name scope. The value 0 is forbidden and the value 1023 is reserved.

`URL_Flag` – a flag that indicates the presence of a `URLstring`.

`includeInlineProfileLevelFlag` – a flag that, if set to one, indicates that the subsequent profile indications take into account the resources needed to process any content that might be inlined.

`URLlength` – the length of the subsequent `URLstring` in bytes.

`URLstring[]` – A string with a UTF-8 [3] encoded URL that shall point to another `InitialObjectDescriptor`. Only the content of this object descriptor shall be returned by the delivery entity upon access to this URL. Within the current name scope, the new object descriptor shall be referenced by the `objectDescriptorId` of the object descriptor carrying the URLstring. On name scopes see 8.7.2.4. Permissible URLs may be constrained by profile and levels as well as by specific delivery layers.

`ODProfileLevelIndication` – an indication as defined in Table 3 of the object descriptor profile and level required to process the content associated with this `InitialObjectDescriptor`.

Table 3 - ODProfileLevelIndication Values

Value	**Profile**	**Level**
0x00	Forbidden	-
0x01-0x7F	reserved for ISO use	-
0x80-0xFD	user private	-
0xFE	no OD profile specified	-
0xFF	no OD capability required	-
NOTE — Usage of the value 0xFE indicates that the content described by this InitialObjectDescriptor does not comply to any OD profile specified in ISO/IEC 14496-1. Usage of the value 0xFF indicates that none of the OD profile capabilities are required for this content.		

`sceneProfileLevelIndication` – an indication as defined in Table 4 of the scene graph profile and level required to process the content associated with this `InitialObjectDescriptor`.

Table 4 - sceneProfileLevelIndication Values

Value	**Profile**	**Level**
0x00	Reserved for ISO use	-
0x01	Simple2D profile	L1
0x02-0x7F	reserved for ISO use	-
0x80-0xFD	user private	-
0xFE	no scene graph profile specified	-
0xFF	no scene graph capability required	-
NOTE — Usage of the value 0xFE indicates that the content described by this InitialObjectDescriptor does not comply to any scene graph profile specified in ISO/IEC 14496-1. Usage of the value 0xFF indicates that none of the scene graph profile capabilities are required for this content.		

`audioProfileLevelIndication` – an indication as defined in Table 5 of the audio profile and level required to process the content associated with this `InitialObjectDescriptor`.

Table 5 - audioProfileLevelIndication Values

Value	Profile	Level
0x00	Reserved for ISO use	-
0x01	Main Profile	L1
0x02	Main Profile	L2
0x03	Main Profile	L3
0x04	Main Profile	L4
0x05	Scalable Profile	L1
0x06	Scalable Profile	L2
0x07	Scalable Profile	L3
0x08	Scalable Profile	L4
0x09	Speech Profile	L1
0x0A	Speech Profile	L2
0x0B	Synthesis Profile	L1
0x0C	Synthesis Profile	L2
0x0D	Synthesis Profile	L3
0x0E-0x7F	reserved for ISO use	-
0x80-0xFD	user private	-
0xFE	no audio profile specified	-
0xFF	no audio capability required	-
NOTE — Usage of the value 0xFE indicates that the content described by this InitialObjectDescriptor does not comply to any audio profile specified in ISO/IEC 14496-3. Usage of the value 0xFF indicates that none of the audio profile capabilities are required for this content.		

`visualProfileLevelIndication` – an indication as defined in Table 6 of the visual profile and level required to process the content associated with this `InitialObjectDescriptor`.

Table 6 - visualProfileLevelIndication Values

Value	Profile	Level
0x00	Reserved for ISO use	-
0x01	Simple	L3
0x02	Simple	L2
0x03	Simple	L1
0x04	Simple Scalable	L2
0x05	Simple Scalable	L1
0x06	Core	L2
0x07	Core	L1
0x08	Main	L4
0x09	Main	L3
0x0A	Main	L2
0x0B	N-Bit	L2
0x0C	Hybrid	L2
0x0D	Hybrid	L1
0x0E	Basic Animated Texture	L2
0x0F	Basic Animated Texture	L1
0x10	Scalable Texture	L3
0x11	Scalable Texture	L2
0x12	Scalable Texture	L1
0x13	Simple Face Animation	L2
0x14	Simple Face Animation	L1
0x15-0x7F	reserved for ISO use	-
0x80-0xFD	user private	-

0xFE	no visual profile specified	-
0xFF	no visual capability required	-
NOTE — Usage of the value 0xFE indicates that the content described by this InitialObjectDescriptor does not comply to any visual profile specified in ISO/IEC 14496-2. Usage of the value 0xFF indicates that none of the visual profile capabilities are required for this content.		

`graphicsProfileLevelIndication` – an indication as defined in Table 7 of the graphics profile and level required to process the content associated with this `InitialObjectDescriptor`.

Table 7 - graphicsProfileLevelIndication Values

Value	Profile	Level
0x00	Reserved for ISO use	
0x01	Simple2D profile	L1
0x02-0x7F	reserved for ISO use	
0x80-0xFD	user private	
0xFE	no graphics profile specified	
0xFF	no graphics capability required	
NOTE — Usage of the value 0xFE may indicate that the content described by this InitialObjectDescriptor does not comply to any conformance point specified in ISO/IEC 14496-1. Usage of the value 0xFF indicates that none of the graphics profile capabilities are required for this content.		

`ESD[]` – an array of `ES_Descriptors` as defined in 8.6.4. The array shall have any number of one up to 30 elements.

`ociDescr[]` – an array of OCI_Descriptors as defined in 8.6.17.2 that relates to the set of audio-visual objects that are described by this initial object descriptor. The array shall have any number of zero up to 255 elements.

`ipmpDescrPtr[]` – an array of `IPMP_DescriptorPointer`, as defined in 8.6.12, that points to the IPMP_Descriptors related to the elementary stream(s) described by this object descriptor. The array shall have any number of zero up to 255 elements.

`extDescr[]` – an array of `ExtensionDescriptors` as defined in 8.6.15. The array shall have any number of zero up to 255 elements.

8.6.4 ES_Descriptor

8.6.4.1 Syntax

```
class ES_Descriptor extends BaseDescriptor : bit(8) tag=ES_DescrTag {
   bit(16) ES_ID;
   bit(1) streamDependenceFlag;
   bit(1) URL_Flag;
   const bit(1) reserved=1;
   bit(5) streamPriority;
   if (streamDependenceFlag)
      bit(16) dependsOn_ES_ID;
   if (URL_Flag) {
      bit(8) URLlength;
      bit(8) URLstring[URLlength];
   }
   DecoderConfigDescriptor decConfigDescr;
   SLConfigDescriptor slConfigDescr;
   IPI_DescrPointer ipiPtr[0 .. 1];
   IP_IdentificationDataSet ipIDS[0 .. 255];
   IPMP_DescriptorPointer ipmpDescrPtr[0 .. 255];
   LanguageDescriptor langDescr[0 .. 255];
   QoS_Descriptor qosDescr[0 .. 1];
   RegistrationDescriptor regDescr[0 .. 1];
   ExtensionDescriptor extDescr[0 .. 255];
}
```

8.6.4.2 Semantics

The ES_Descriptor conveys all information related to a particular elementary stream and has three major parts.

The first part consists of the ES_ID which is a unique reference to the elementary stream within its name scope (see 8.7.2.4), a mechanism to describe dependencies of elementary streams within the scope of the parent ObjectDescriptor and an optional URL string. Dependencies and usage of URLs are specified in 8.7.

The second part consists of the component descriptors which convey the parameters and requirements of the elementary stream.

The third part is a set of optional extension descriptors that support the inclusion of future extensions as well as the transport of private data in a backward compatible way.

ES_ID – This syntax element provides a unique label for each elementary stream within its name scope. The values 0 and 0xFFFF are reserved.

streamDependenceFlag – If set to one indicates that a dependsOn_ES_ID will follow.

URL_Flag – if set to 1 indicates that a URLstring will follow.

streamPriority – indicates a relative measure for the priority of this elementary stream. An elementary stream with a higher streamPriority is more important than one with a lower streamPriority. The absolute values of streamPriority are not normatively defined.

dependsOn_ES_ID – is the ES_ID of another elementary stream on which this elementary stream depends. The stream with dependsOn_ES_ID shall also be associated to the same ObjectDescriptor as the current ES_Descriptor.

URLlength – the length of the subsequent URLstring in bytes.

URLstring[] – contains a UTF-8 [3] encoded URL that shall point to the location of an SL-packetized stream by name. The parameters of the SL-packetized stream that is retrieved from the URL are fully specified in this ES_Descriptor. See also 8.7.3.3. Permissible URLs may be constrained by profile and levels as well as by specific delivery layers.

decConfigDescr – is a DecoderConfigDescriptor as specified in 8.6.5.

slConfigDescr – is an SLConfigDescriptor as specified in 8.6.7.

ipiPtr[] – an array of zero or one IPI_DescrPointer as specified in 8.6.11.

ipIDS[] – an array of zero or more IP_IdentificationDataSet as specified in 8.6.8.

Each ES_Descriptor shall have either one IPI_DescrPointer or one up to 255 IP_IdentificationDataSet elements. This allows to unambiguously associate an IP Identification to each elementary stream.

ipmpDescrPtr[] – an array of IPMP_DescriptorPointer, as defined in 8.6.12, that points to the IPMP_Descriptors related to the elementary stream described by this ES_Descriptor. The array shall have any number of zero up to 255 elements.

langDescr[] – an array of zero or one LanguageDescriptor structures as specified in 8.6.17.6. It indicates the language attributed to this elementary stream.

NOTE — Multichannel audio streams may be treated as one elementary stream with one ES_Descriptor by ISO/IEC 14496. In that case different languages present in different channels of the multichannel stream are not identifyable with a LanguageDescriptor.

qosDescr[] – an array of zero or one QoS_Descriptor as specified in 8.6.14.

extDescr[] – an array of ExtensionDescriptor structures as specified in 8.6.15.

8.6.5 DecoderConfigDescriptor

8.6.5.1 Syntax

```
class DecoderConfigDescriptor extends BaseDescriptor : bit(8) tag=DecoderConfigDescrTag {
   bit(8) objectTypeIndication;
   bit(6) streamType;
   bit(1) upStream;
   const bit(1) reserved=1;
   bit(24) bufferSizeDB;
   bit(32) maxBitrate;
   bit(32) avgBitrate;
   DecoderSpecificInfo decSpecificInfo[0 .. 1];
}
```

8.6.5.2 Semantics

The `DecoderConfigDescriptor` provides information about the decoder type and the required decoder resources needed for the associated elementary stream. This is needed at the receiving terminal to determine whether it is able to decode the elementary stream. A stream type identifies the category of the stream while the optional decoder specific information descriptor contains stream specific information for the set up of the decoder in a stream specific format that is opaque to this layer.

`ObjectTypeIndication` – an indication of the object or scene description type that needs to be supported by the decoder for this elementary stream as per the following table. For `streamType` values other than audioStream and visualStream, the `objectTypeIndication` shall be set to 0xFF, indicating that no object type is specified.

Table 8 - objectTypeIndication Values

Value	**`ObjectTypeIndication` Description**
0x00	Forbidden
0x01-0x1F	reserved for ISO use
0x20	Visual ISO/IEC 14496-2 [a]
0x21-0x3F	reserved for ISO use
0x40	Audio ISO/IEC 14496-3 [b]
0x41-0x5F	reserved for ISO use
0x60	Visual ISO/IEC 13818-2 Simple Profile
0x61	Visual ISO/IEC 13818-2 Main Profile
0x62	Visual ISO/IEC 13818-2 SNR Profile
0x63	Visual ISO/IEC 13818-2 Spatial Profile
0x64	Visual ISO/IEC 13818-2 High Profile
0x65	Visual ISO/IEC 13818-2 422 Profile
0x66	Audio ISO/IEC 13818-7 Main Profile
0x67	Audio ISO/IEC 13818-7 LowComplexity Profile
0x68	Audio ISO/IEC 13818-7 SSR Profile
0x69	Audio ISO/IEC 13818-3
0x6A	Visual ISO/IEC 11172-2
0x6B	Audio ISO/IEC 11172-3
0x6C	Visual ISO/IEC 10918-1
0x6D - 0xBF	reserved for ISO use
0xC0 - 0xFE	user private
0xFF	no profile specified

[a] The actual object types are defined in ISO/IEC 14496-2 and are conveyed in the DecoderSpecificInfo as specified in ISO/IEC 14496-2, Annex K.

[b] The actual object types are defined in ISO/IEC 14496-3 and are conveyed in the DecoderSpecificInfo as specified in ISO/IEC 14496-3 Section 1 subclause 1.6.2.1.

`streamType` – conveys the type of this elementary stream as per this table.

Table 9 - streamType Values

streamType value	stream type description
0x00	forbidden
0x01	ObjectDescriptorStream (see 8.5)
0x02	ClockReferenceStream (see 10.2.5)
0x03	SceneDescriptionStream (see 9.2.1)
0x04	VisualStream
0x05	AudioStream
0x06	MPEG7Stream
0x07	IPMPStream (see 8.3.2)
0x08	ObjectContentInfoStream (see 8.4.2)
0x09 - 0x1F	reserved for ISO use
0x20 - 0x3F	user private

`upStream` – indicates that this stream is used for upstream information.

`bufferSizeDB` – is the size of the decoding buffer for this elementary stream in byte.

`maxBitrate` – is the maximum bitrate in bits per second of this elementary stream in any time window of one second duration.

`avgBitrate` – is the average bitrate in bits per second of this elementary stream. For streams with variable bitrate this value shall be set to zero.

`decSpecificInfo[]` – an array of zero or one decoder specific information classes as specified in 8.6.6.

8.6.6 DecoderSpecificInfo

8.6.6.1 Syntax

```
abstract class DecoderSpecificInfo extends BaseDescriptor : bit(8) tag=DecSpecificInfoTag
{
   // empty. To be filled by classes extending this class.
}
```

8.6.6.2 Semantics

The decoder specific information constitutes an opaque container with information for a specific media decoder. The existence and semantics of decoder specific information depends on the values of `DecoderConfigDescriptor.streamType` and `DecoderConfigDescriptor.objectTypeIndication`.

For values of `DecoderConfigDescriptor.objectTypeIndication` that refer to streams complying with ISO/IEC 14496-2 the syntax and semantics of decoder specific information are defined in Annex K of that part.

For values of `DecoderConfigDescriptor.objectTypeIndication` that refer to streams complying with ISO/IEC 14496-3 the syntax and semantics of decoder specific information are defined in section 1, clause 1.6 of that part.

For values of `DecoderConfigDescriptor.objectTypeIndication` that refer to scene description streams the semantics of decoder specific information is defined in 9.2.1.2.

For values of `DecoderConfigDescriptor.objectTypeIndication` that refer to streams complying with ISO/IEC 13818-7 the decoder specific information consists of the ADIF -header if it is present (or none if it is not present) and an access unit is a „raw_data_block()“ as defined in ISO/IEC 13818-7 [9].

For values of `DecoderConfigDescriptor.objectTypeIndication` that refer to streams complying with ISO/IEC 13818-3 [8] the decoder specific information is empty since all necessary data is in the bitstream frames itself. The access units in this case are the „frame()“ bitstream element as is defined in ISO/IEC 11172-3 [6].

For values of `DecoderConfigDescriptor.objectTypeIndication` that refer to streams complying with ISO/IEC 10918-1 [4], the decoder specific information is:

```
class JPEG_DecoderConfig extends DecoderSpecificInfo : bit(8) tag=DecSpecificInfoTag {
   int(16) headerLength;
   int(16) Xdensity;
   int(16) Ydensity;
   int(8) numComponents;
}
```

with

`headerLength` –indicates the number of bytes to skip from the beginning of the stream to find the first pixel of the image.

`Xdensity` and `Ydensity` – specify the pixel aspect ratio.

`numComponents` – indicates whether the image has Y component only or is Y, Cr, Cb. It shall be equal to 1 or 3.

8.6.7 SLConfigDescriptor

This descriptor defines the configuration of the sync layer header for this elementary stream. The specification of this descriptor is provided together with the specification of the sync layer in 10.2.3.

8.6.8 IP_IdentificationDataSet

8.6.8.1 Syntax

```
abstract class IP_IdentificationDataSet extends BaseDescriptor
      : bit(8) tag=ContentIdentDescrTag..SupplContentIdentDescrTag
{
   // empty. To be filled by classes extending this class.
}
```

8.6.8.2 Semantics

This class is an abstract base class that is extended by the descriptor classes that implement IP identification. A descriptor that allows to aggregate classes of type IP_IdentificationDataSet may actually aggregate any of the classes that extend IP_IdentificationDataSet.

8.6.9 ContentIdentificationDescriptor

8.6.9.1 Syntax

```
class ContentIdentificationDescriptor extends IP_IdentificationDataSet
      : bit(8) tag=ContentIdentDescrTag
{
   const bit(2) compatibility=0;
   bit(1)    contentTypeFlag;
   bit(1)    contentIdentifierFlag;
   bit(1)    protectedContent;
   bit(3)    reserved = 0b111;
   if (contentTypeFlag)
      bit(8)  contentType;
   if (contentIdentifierFlag) {
      bit(8)  contentIdentifierType;
      bit(8)  contentIdentifier[sizeOfInstance-2-contentTypeFlag];
   }
}
```

8.6.9.2 Semantics

The content identification descriptor is used to identify content. All types of elementary streams carrying content can be identified using this mechanism. The content types include audio, visual and scene description data. Multiple content identification descriptors may be associated to one elementary stream. These descriptors shall never be detached from the ES_Descriptor.

`compatibility` – must be set to 0.

`contentTypeFlag` – flag to indicate if a definition of the type of content is available.

`contentIdentifierFlag` – flag to indicate presence of creation ID.

`protectedContent` - if set to one indicates that the elementary streams that refer to this IP_IdentificationDataSet are protected by a method outside the scope of ISO/IEC 14496. The behavior of the terminal compliant with the ISO/IEC 14496 specifications when processing such streams is undefined.

`contentType` – defines the type of content using one of the values specified in the the following table.

Table 10 - contentType Values

0	Audio-visual
1	Book
2	Serial
3	Text
4	Item or Contribution (e.g. article in book or serial)
5	Sheet music
6	Sound recording or music video
7	Still Picture
8	Musical Work
9-254	Reserved for ISO use
255	Others

`contentIdentifierType` – defines a type of content identifier using one of the values specified in the following table.

Table 11 - contentIdentifierType Values

0	ISAN	International Standard Audio-Visual Number
1	ISBN	International Standard Book Number
2	ISSN	International Standard Serial Number
3	SICI	Serial Item and Contribution Identifier
4	BICI	Book Item and Component Identifier
5	ISMN	International Standard Music Number
6	ISRC	International Standard Recording Code
7	ISWC-T	International Standard Work Code (Tunes)
8	ISWC-L	International Standard Work Code (Literature)
9	SPIFF	Still Picture ID
10	DOI	Digital Object Identifier
11-255	Reserved for ISO use	

`contentIdentifier` – international code identifying the content according to the preceding `contentIdentifierType`.

8.6.10 SupplementaryContentIdentificationDescriptor

8.6.10.1 Syntax

```
class SupplementaryContentIdentificationDescriptor extends
   IP_IdentificationDataSet : bit(8) tag= SupplContentIdentDescrTag
{
   bit(24) languageCode;
   bit(8)  supplContentIdentifierTitleLength;
   bit(8)  supplContentIdentifierTitle[supplContentIdentifierTitleLength];
   bit(8)  supplContentIdentifierValueLength;
   bit(8)  supplContentIdentifierValue[supplContentIdentifierValueLength];
}
```

8.6.10.2 Semantics

The supplementary content identification descriptor is used to provide extensible identifiers for content that are qualified by a language code. Multiple supplementary content identification descriptors may be associated to one elementary stream. These descriptors shall never be detached from the ES_Descriptor.

`language code` – This 24 bits field contains the ISO 639-2:1998 [1] bibliographic three character language code of the language of the following text fields.

`supplementaryContentIdentifierTitleLength` – indicates the length of the subsequent `supplementaryContentIdentifierTitle` in bytes.

`supplementaryContentIdentifierTitle` – identifies the title of a supplementary content identifier that may be used when a numeric content identifier (see 8.6.9) is not available.

`supplementaryContentIdentifierValueLength` – indicates the length of the subsequent `supplementaryContentIdentifierValue` in bytes.

`supplementaryContentIdentifierValue` – identifies the value of a supplementary content identifer associated to the preceding `supplementaryContentIdentifierTitle`.

8.6.11 IPI_DescrPointer

8.6.11.1 Syntax

```
class IPI_DescrPointer extends BaseDescriptor : bit(8) tag=IPI_DescrPointerTag {
   bit(16) IPI_ES_Id;
}
```

8.6.11.2 Semantics

The `IPI_DescrPointer` class contains a reference to the elementary stream that includes the `IP_IdentificationDataSets` that are valid for this stream. This indirect reference mechanism allows to convey such descriptors only in one elementary stream while making references to it from any `ES_Descriptor` that shares the same information.

`ES_Descriptors` for elementary streams that are intended to be accessible regardless of the availability of a referred stream shall explicitly include their `IP_IdentificationDataSets` instead of using an `IPI_DescrPointer`.

`IPI_ES_Id` – the `ES_ID` of the elementary stream whose ES_Descriptor contains the IP Information valid for this elementary stream. If the `ES_Descriptor` for `IPI_ES_Id` is not available, the IPI status of this elementary stream is undefined.

8.6.12 IPMP_DescriptorPointer

8.6.12.1 Syntax

```
class IPMP_DescriptorPointer extends BaseDescriptor : bit(8) tag=IPMP_DescrPointerTag {
   bit(8) IPMP_DescriptorID;
}
```

8.6.12.2 Semantics

`IPMP_DescriptorID` - ID of the referenced IPMP_Descriptor (see 8.6.13).

Presence of this descriptor in an `ObjectDescriptor` indicates that all streams referred to by embedded `ES_Descriptors` are subject to protection and management by the IPMP System specified in the referenced `IPMP_Descriptor`.

Presence of this descriptor in an `ES_Descriptor` indicates that the stream associated with this descriptor is subject to intellectual property management and protection by the IPMP System specified in the referenced `IPMP_Descriptor`.

8.6.13 IPMP Descriptor

8.6.13.1 Syntax

```
class IPMP_Descriptor() extends BaseDescriptor :  bit(8) IPMP_DescrTag {
   bit(8) IPMP_DescriptorID;
   unsigned int(16)    IPMPS_Type;
   if (IPMPS_Type == 0) {
      bit(8) URLString[sizeOfInstance-3];
   } else {
      bit(8) IPMP_data[sizeOfInstance-3];
   }
}
```

8.6.13.2 Semantics

The `IPMP_Descriptor` conveys IPMP information to an IPMP System. `IPMP_Descriptors` are conveyed in object descriptor streams via `IPMP_DescriptorUpdates` as specified in 8.5.5.6. They are not directly included in `ObjectDescriptors` or `ES_Descriptors`. `IPMP_Descriptors` are referenced by `ObjectDescriptors` or `ES_Descriptors` using `IPMP_DescriptorPointers` (see 8.6.12). An `IPMP_Descriptor` may be referenced by multiple `ObjectDescriptors` or `ES_Descriptors`.

`IPMP_DescriptorID` - a unique ID for this IPMP descriptor within its name scope (see 8.7.2.4).

`IPMPS_Type` - the type of the IPMP System. A zero value does not correspond to an IPMP System but is used to indicate the presence of a URL. A Registration Authority designated by ISO shall assign valid values for this field.

`URLString[]` - contains a UTF-8 [3] encoded URL that points to the location of a remote IPMP_Descriptor whose IPMP_data shall be used in place of locally provided data.

`IPMP_data` - opaque data to control the IPMP System.

8.6.13.3 Implementation of a Registration Authority (RA)

ISO/IEC JTC 1/SC 29 shall issue a call for nominations from Member Bodies of ISO or National Committees of IEC in order to identify suitable organizations that will serve as the Registration Authority for the IPMPS_Type as defined in this clause. The selected organization shall serve as the Registration Authority. The so-named Registration Authority shall execute its duties in compliance with Annex H of the JTC 1 Directives. The registered IPMPS_Type is hereafter referred to as the Registered Identifier (RID).

Upon selection of the Registration Authority, JTC 1 shall require the creation of a Registration Management Group (RMG) that will review appeals filed by organizations whose request for an RID to be used in conjunction with ISO/IEC 14496 has been denied by the Registration Authority.

Annex D provides information on the procedure for registering a unique IPMPS_Type value.

8.6.14 QoS_Descriptor

8.6.14.1 Syntax

```
class QoS_Descriptor extends BaseDescriptor : bit(8) tag=QoS_DescrTag {
   bit(8) predefined;
   if (predefined==0) {
      QoS_Qualifier qualifiers[];
   }
}
```

8.6.14.2 Semantics

The QoS_descriptor conveys the requirements that the ES has on the transport channel and a description of the traffic that this ES will generate. A set of predefined values is to be determined; customized values can be used by setting the `predefined` field to 0.

`predefined` – a value different from zero indicates a predefined QoS profile according to the table below.

Table 12 - Predefined QoS Profiles

predefined value	**description**
0x00	Custom
0x01 - 0xff	Reserved

`qualifier` – an array of one or more `QoS_Qualifiers`.

8.6.14.3 QoS_Qualifier

8.6.14.3.1 Syntax

```
abstract class QoS_Qualifier extends ExpandableBaseClass : bit(8) tag=0x01..0xff {
   // empty. To be filled by classes extending this class.
}

class QoS_Qualifier_MAX_DELAY extends QoS_Qualifier : bit(8) tag=0x01 {
   unsigned int(32) MAX_DELAY;
}

class QoS_Qualifier_PREF_MAX_DELAY extends QoS_Qualifier : bit(8) tag=0x02 {
   unsigned int(32) PREF_MAX_DELAY;
}

class QoS_Qualifier_LOSS_PROB extends QoS_Qualifier : bit(8) tag=0x03 {
   double(32) LOSS_PROB;
}

class QoS_Qualifier_MAX_GAP_LOSS extends QoS_Qualifier : bit(8) tag=0x04 {
   unsigned int(32) MAX_GAP_LOSS;
}

class QoS_Qualifier_MAX_AU_SIZE extends QoS_Qualifier : bit(8) tag=0x41 {
   unsigned int(32) MAX_AU_SIZE;
}

class QoS_Qualifier_AVG_AU_SIZE extends QoS_Qualifier : bit(8) tag=0x42 {
   unsigned int(32) AVG_AU_SIZE;
}

class QoS_Qualifier_MAX_AU_RATE extends QoS_Qualifier : bit(8) tag=0x43 {
   unsigned int(32) MAX_AU_RATE;
}
```

8.6.14.3.2 Semantics

QoS qualifiers are defined as derived classes from the abstract `QoS_Qualifier` class. They are identified by means of their class tag. Unused tag values up to and including 0x7F are reserved for ISO use. Tag values from 0x80 up to and including 0xFE are user private. Tag values 0x00 and 0xFF are forbidden.

`MAX_DELAY` – Maximum end to end delay for the stream in microseconds.

`PREF_MAX_DELAY` – Preferred end to end delay for the stream in microseconds.

`LOSS_PROB` – Allowable loss probability of any single AU as a fractional value between 0.0 and 1.0.

`MAX_GAP_LOSS` – Maximum allowable number of consecutively lost AUs.

`MAX_AU_SIZE` – Maximum size of an AU in bytes.

`AVG_AU_SIZE` – Average size of an AU in bytes.

`MAX_AU_RATE` – Maximum arrival rate of AUs in AUs/second.

8.6.15 ExtensionDescriptor

8.6.15.1 Syntax

```
abstract class ExtensionDescriptor extends BaseDescriptor
: bit(8) tag = ExtDescrTagStartRange .. ExtDescrTagEndRange {
   // empty. To be filled by classes extending this class.
}
```

8.6.15.2 Semantics

This class is an abstract base class that may be extended for defining additional descriptors in future. The available range of class tag values allow ISO defined extensions as well as private extensions. A descriptor that allows to aggregate ExtensionDescriptor classes may actually aggregate any of the classes that extend ExtensionDescriptor. Extension descriptors may be ignored by a terminal that conforms to ISO/IEC 14496-1.

8.6.16 RegistrationDescriptor

The registration descriptor provides a method to uniquely and unambiguously identify formats of private data streams.

8.6.16.1 Syntax

```
class RegistrationDescriptor extends BaseDescriptor  : bit(8) tag=RegistrationDescrTag {
   bit(32) formatIdentifier;
   bit(8) additionalIdentificationInfo[sizeOfInstance-4];
}
```

8.6.16.2 Semantics

`formatIdentifier` – is a value obtained from a Registration Authority as designated by ISO.

`additionalIdentificationInfo` – The meaning of `additionalIdentificationInfo`, if any, is defined by the assignee of that `formatIdentifier`, and once defined, shall not change.

The registration descriptor is provided in order to enable users of ISO/IEC 14496-1 to unambiguously carry elementary streams with data whose format is not recognized by ISO/IEC 14496-1. This provision will permit ISO/IEC 14496-1 to carry all types of data streams while providing for a method of unambiguous identification of the characteristics of the underlying private data streams.

In the following subclause and Annex D, the benefits and responsibilities of all parties to the registration of private data format are outlined.

8.6.16.2.1 Implementation of a Registration Authority (RA)

ISO/IEC JTC 1/SC 29 shall issue a call for nominations from Member Bodies of ISO or National Committees of IEC in order to identify suitable organizations that will serve as the Registration Authority for the formatIdentifier as defined in this subclause. The selected organization shall serve as the Registration Authority. The so-named Registration Authority shall execute its duties in compliance with Annex H of the JTC 1 Directives. The registered private data formatIdentifier is hereafter referred to as the Registered Identifier (RID).

Upon selection of the Registration Authority, JTC 1 shall require the creation of a Registration Management Group (RMG) which will review appeals filed by organizations whose request for an RID to be used in conjunction with ISO/IEC 14496-1 has been denied by the Registration Authority.

Annex D provides information on the procedure for registering a unique format identifier.

8.6.17 Object Content Information Descriptors

8.6.17.1 Overview

This subclause defines the descriptors that constitute the object content information. These descriptors may either be included in an `OCI_Event` in an OCI stream or be part of an `ObjectDescriptor` or `ES_Descriptor` as defined in 8.6.1.

8.6.17.2 OCI_Descriptor Class

8.6.17.2.1 Syntax

```
abstract class OCI_Descriptor extends BaseDescriptor
   : bit(8) tag= OCIDescrTagStartRange .. OCIDescrTagEndRange
{
   // empty. To be filled by classes extending this class.
}
```

8.6.17.2.2 Semantics

This class is an abstract base class that is extended by the classes specified in the subsequent clauses. A descriptor or an OCI_Event that allows to aggregate classes of type OCI_Descriptor may actually aggregate any of the classes that extend OCI_Descriptor.

8.6.17.3 Content classification descriptor

8.6.17.3.1 Syntax

```
class ContentClassificationDescriptor extends OCI_Descriptor
                : bit(8) tag= ContentClassificationDescrTag {
   bit(32) classificationEntity;
   bit(16) classificationTable;
   bit(8) contentClassificationData[sizeOfInstance-6];
}
```

8.6.17.3.2 Semantics

The content classification descriptor provides one or more classifications of the event information. The `classificationEntity` field indicates the organization that classifies the content. The possible values have to be registered with a registration authority to be identified.

`classificationEntity` **–** indicates the content classification entity. The values of this field are to be defined by a registration authority to be identified.

`classificationTable` – indicates which classification table is being used for the corresponding classification. The classification is defined by the corresponding classification entity. 0x00 is a reserved value.

`contentClassificationData[]` – this array contains a classification data set using a non-default classification table.

8.6.17.4 Key Word Descriptor

8.6.17.4.1 Syntax

```
class KeyWordDescriptor extends OCI_Descriptor : bit(8) tag=KeyWordDescrTag {
   int i;
   bit(24) languageCode;
   bit(1) isUTF8_string;
   aligned(8) unsigned int(8) keyWordCount;
   for (i=0; i<keyWordCount; i++) {
      unsigned int(8) keyWordLength[[i]];
      if (isUTF8_string) then {
         bit(8) keyWord[[i]][keyWordLength[i]];
      } else {
         bit(16) keyWord[[i]][keyWordLength[i]];
      }
   }
}
```

8.6.17.4.2 Semantics

The key word descriptor allows the OCI creator/provider to indicate a set of key words that characterize the content. The choice of the key words is completely free but each time the key word descriptor appears, all the key words given are for the language indicated in `languageCode`. This means that, for a certain event, the key word descriptor must appear as many times as the number of languages for which key words are to be provided.

`languageCode` – contains the ISO 639-2:1998 [1] bibliographic three character language code of the language of the following text fields.

`isUTF8_string` – indicates that the subsequent string is encoded with one byte per character (UTF-8). Else it is two byte per character.

`keyWordCount` – indicates the number of key words to be provided.

`keyWordLength` – specifies the length in characters of each key word.

`keyWord[]` – a Unicode [3] encoded string that specifies the key word.

8.6.17.5 Rating Descriptor

8.6.17.5.1 Syntax

```
class RatingDescriptor extends OCI_Descriptor : bit(8) tag=RatingDescrTag {
   bit(32) ratingEntity;
   bit(16) ratingCriteria;
   bit(8)  ratingInfo[sizeOfInstance-6];
}
```

8.6.17.5.2 Semantics

This descriptor gives one or more ratings, originating from corresponding rating entities, valid for a specified country. The `ratingEntity` field indicates the organization which is rating the content. The possible values have to be registered with a registration authority to be identified. This registration authority shall make the semantics of the rating descriptor publicly available.

`ratingEntity` – indicates the rating entity. The values of this field are to be defined by a registration authority to be identified.

`ratingCriteria` – indicates which rating criteria are being used for the corresponding rating entity. The value 0x00 is reserved.

`ratingInfo[]` – this array contains the rating information.

8.6.17.6 Language Descriptor

8.6.17.6.1 Syntax

```
class LanguageDescriptor extends OCI_Descriptor : bit(8) tag=LanguageDescrTag {
   bit(24) languageCode;
}
```

8.6.17.6.2 Semantics

This descriptor identifies the language of the corresponding audio/speech or text object that is being described.

`languageCode` – contains the ISO 639-2:1998 [1] bibliographic three character language code of the corresponding audio/speech or text object that is being described.

8.6.17.7 Short Textual Descriptor

8.6.17.7.1 Syntax

```
class ShortTextualDescriptor extends OCI_Descriptor : bit(8) tag=ShortTextualDescrTag {
   bit(24) languageCode;
   bit(1) isUTF8_string;
   aligned(8) unsigned int(8) nameLength;
   if (isUTF8_string) then {
      bit(8) eventName[nameLength];
      unsigned int(8) textLength;
      bit(8) eventText[textLength];
   } else {
      bit(16) eventName[nameLength];
      unsigned int(8) textLength;
      bit(16) eventText[textLength];
   }
}
```

8.6.17.7.2 Semantics

The short textual descriptor provides the name of the event and a short description of the event in text form.

`languageCode` – contains the ISO 639-2:1998 [1] bibliographic three character language code of the language of the following text fields.

`isUTF8_string` – indicates that the subsequent string is encoded with one byte per character (UTF-8). Else it is two byte per character.

`nameLength` – specifies the length in characters of the event name.

`eventName[]` – a Unicode [3] encoded string that specifies the event name.

`textLength` – specifies the length in characters of the following text describing the event.

`eventText[]` – a Unicode [3] encoded string that specifies the text description for the event.

8.6.17.8 Expanded Textual Descriptor

8.6.17.8.1 Syntax

```
class ExpandedTextualDescriptor extends OCI_Descriptor : bit(8) tag=ExpandedTextualDescrTag {
   int i;
   bit(24) languageCode;
   bit(1) isUTF8_string;
   aligned(8) unsigned int(8) itemCount;
   for (i=0; i<itemCount; i++){
      unsigned int(8) itemDescriptionLength[[i]];
      if (isUTF8_string) then {
         bit(8) itemDescription[[i]][itemDescriptionLength[i];
      } else {
         bit(16) itemDescription[[i]][itemDescriptionLength[i]];
      }
      unsigned int(8) itemLength[[i]];
      if (isUTF8_string) then {
         bit(8) itemText[[i]][itemLength[i]];
      } else {
         bit(16) itemText[[i]][itemLength[i]];
      }
   }
   unsigned int(8) textLength;
   int nonItemTextLength=0;
   while( textLength == 255 ) {
      nonItemTextLength += textLength;
      bit(8) textLength;
   }
   nonItemTextLength += textLength;
   if (isUTF8_string) then {
      bit(8) nonItemText[nonItemTextLength];
   } else {
      bit(16) nonItemText[nonItemTextLength];
   }
}
```

8.6.17.8.2 Semantics

The expanded textual descriptor provides a detailed description of an event, which may be used in addition to, or independently from, the short event descriptor. In addition to direct text, structured information in terms of pairs of description and text may be provided. An example application for this structure is to give a cast list, where for example the item description field might be “Producer” and the item field would give the name of the producer.

`languageCode` **-** contains the ISO 639-2:1998 [1] bibliographic three character language code of the language of the following text fields.

`isUTF8_string` – indicates that the subsequent string is encoded with one byte per character (UTF-8). Else it is two byte per character.

`itemCount` – specifies the number of items to follow (itemised text).

`itemDescriptionLength` **–** specifies the length in characters of the item description.

`itemDescription[]` **–** a Unicode [3] encoded string that specifies the item description.

`itemLength` **–** specifies the length in characters of the item text.

`itemText[]` **–** a Unicode [3] encoded string that specifies the item text.

`textLength` **–** specifies the length in characters of the non itemised expanded text. The value 255 is used as an escape code, and it is followed by another `textLength` field that contains the length in bytes above 255. For lengths greater than 511 a third field is used, and so on.

`nonItemText[]` – a Unicode [3] encoded string that specifies the non itemised expanded text.

8.6.17.9 Content Creator Name Descriptor

8.6.17.9.1 Syntax

```
class ContentCreatorNameDescriptor extends OCI_Descriptor
             : bit(8) tag= ContentCreatorNameDescrTag {
   int i;
   unsigned int(8) contentCreatorCount;
   for (i=0; i<contentCreatorCount; i++){
      bit(24) languageCode[[i]];
      bit(1) isUTF8_string[[i]];
      aligned(8) unsigned int(8) contentCreatorLength[[i]];
      if (isUTF8_string[[i]]) then {
         bit(8) contentCreatorName[[i]][contentCreatorLength[i]];
      } else {
         bit(16) contentCreatorName[[i]][contentCreatorLength[i]];
      }
   }
}
```

8.6.17.9.2 Semantics

The content creator name descriptor indicates the name(s) of the content creator(s). Each content creator name may be in a different language.

`contentCreatorCount` – indicates the number of content creator names to be provided.

`languageCode` – contains the ISO 639-2:1998 [1] bibliographic three character language code of the language of the following text fields. Note that for languages that only use Latin characters, just one byte per character is needed in Unicode [3].

`isUTF8_string` – indicates that the subsequent string is encoded with one byte per character (UTF-8). Else it is two byte per character.

`contentCreatorLength[[i]]` – specifies the length in characters of each content creator name.

`contentCreatorName[[i]][]` – a Unicode [3] encoded string that specifies the content creator name.

8.6.17.10 Content Creation Date Descriptor

8.6.17.10.1 Syntax

```
class ContentCreationDateDescriptor extends OCI_Descriptor
             : bit(8) tag= ContentCreationDateDescrTag {
   bit(40) contentCreationDate;
}
```

8.6.17.10.2 Semantics

This descriptor identifies the date of the content creation.

`contentCreationDate` – contains the content creation date of the data corresponding to the event in question, in Universal Time, Co-ordinated (UTC) and Modified Julian Date (MJD) (see Annex F). This field is coded as 16 bits giving the 16 least significant bits of MJD followed by 24 bits coded as 6 digits in 4-bit Binary Coded Decimal (BCD). If the content creation date is undefined all bits of the field are set to 1.

8.6.17.11 OCI Creator Name Descriptor

8.6.17.11.1 Syntax

```
class OCICreatorNameDescriptor extends OCI_Descriptor
              : bit(8) tag=OCICreatorNameDescrTag {
   int i;
   unsigned int(8) OCICreatorCount;
   for (i=0; i<OCICreatorCount; i++) {
      bit(24) languageCode[[i]];
      bit(1) isUTF8_string;
      aligned(8) unsigned int(8) OCICreatorLength[[i]];
      if (isUTF8_string) then {
         bit(8) OCICreatorName[[i]][OCICreatorLength];
       } else {
         bit(16) OCICreatorName[[i]][OCICreatorLength];
      }
   }
}
```

8.6.17.11.2 Semantics

The name of OCI creators descriptor indicates the name(s) of the OCI description creator(s). Each OCI creator name may be in a different language.

`OCICreatorCount` – indicates the number of OCI creators.

`languageCode[[i]]` – contains the ISO 639-2:1998 [1] bibliographic three character language code of the language of the following text fields.

`isUTF8_string` – indicates that the subsequent string is encoded with one byte per character (UTF-8). Else it is two byte per character.

`OCICreatorLength[[i]]` – specifies the length in characters of each OCI creator name.

`OCICreatorName[[i]]` – a Unicode [3] encoded string that specifies the OCI creator name.

8.6.17.12 OCI Creation Date Descriptor

8.6.17.12.1 Syntax

```
class OCICreationDateDescriptor extends OCI_Descriptor
              : bit(8) tag=OCICreationDateDescrTag {
   bit(40) OCICreationDate;
}
```

8.6.17.12.2 Semantics

This descriptor identifies the creation date of the OCI description.

`OCICreationDate` - This 40-bit field contains the OCI creation date for the OCI data corresponding to the event in question, in Co-ordinated Universal Time (UTC) and Modified Julian Date (MJD) (see Annex F). This field is coded as 16 bits giving the 16 least significant bits of MJD followed by 24 bits coded as 6 digits in 4-bit Binary Coded Decimal (BCD). If the OCI creation date is undefined all bits of the field are set to 1.

8.7 Rules for Usage of the Object Description Framework

8.7.1 Aggregation of Elementary Stream Descriptors in a Single Object Descriptor

8.7.1.1 Overview

An object descriptor shall aggregate the descriptors for the set of elementary streams that is intended to be associated to a single node of the scene description and that usually relate to a single audio-visual object. The set of streams may convey a scaleable content representation as well as multiple alternative content representations, e.g., multiple qualities or different languages. Additional streams with IPMP and object content information may be attached.

These options are described by the ES_Descriptor syntax elements `streamDependenceFlag`, `dependsOn_ES_ID`, as well as `streamType`. The semantic rules for the aggregation of elementary stream descriptors within one `ObjectDescriptor` (OD) are specified in this subclause.

8.7.1.2 Aggregation of Elementary Streams with the same streamType

An OD may aggregate multiple ES_Descriptors with the same `streamType` of either visualStream, audioStream or SceneDescriptionStream. However, descriptors for streams with two of these types shall not be mixed within one OD.

8.7.1.3 Aggregation of Elementary Streams with Different streamTypes

In the following cases ESs with different `streamType` may be aggregated:

— An OD may aggregate zero or one additional ES_Descriptor with `streamType` = ObjectContentInfoStream (see 8.4.2). This ObjectContentInfoStream shall be valid for the content conveyed through the other visual, audio or scene description streams whose descriptors are aggregated in this OD.

— An OD may aggregate zero or one additional ES_Descriptors with `streamType` = ClockReferenceStream (see 10.2.5). This ClockReferenceStream shall be valid for the ES within the name scope that refer to the ES_ID of this ClockReferenceStream in their SLConfigDescriptor.

— An OD may aggregate zero or more additional ES_Descriptors with `streamType` = IPMPStream (see 8.3.2). This IPMPStream shall be valid for the content conveyed through the other visual, audio or scene description streams whose descriptors are aggregated in this OD.

8.7.1.4 Aggregation of scene description streams and object descriptor streams

An object descriptor that aggregates one or more ES_Descriptors of `streamType` = SceneDescriptionStream may aggregate any number of additional ES_Descriptors with `streamType` = ObjectDescriptorStream. ES_Descriptors of `streamType` = ObjectDescriptorStream shall not be aggregated in object descriptors that do not contain ES_Descriptors of `streamType` = SceneDescriptionStream.

This means that scene description and object descriptor streams are always combined within one object descriptor. The dependencies between these streams are defined in 8.7.1.5.2.

8.7.1.5 Elementary Stream Dependencies

8.7.1.5.1 Independent elementary streams

ES_Descriptors within one OD with the same `streamType` of either audioStream, visualStream or SceneDescriptionStream that have `streamDependenceFlag=0` refer to independent elementary streams. Such independent elementary streams shall convey alternative representations of the same content. Only one of these representations shall be selected for use in the scene.

NOTE — Independent ESs should be ordered within an OD according to the content creator’s preference. The ES that is first in the list of ES aggregated to one object descriptor should be preferable over an ES that follows later. In case of audio streams, however, the selection should for obvious reasons be done according to the prefered language of the receiving terminal.

8.7.1.5.2 Dependent elementary streams

ES_Descriptors within one OD with the same `streamType` of either audioStream, visualStream, SceneDescriptionStream or ObjectDescriptorStream that have `streamDependenceFlag=1` refer to dependent elementary streams. The ES_ID of the stream on which the dependent elementary stream depends is indicated by `dependsOn_ES_ID`. The ES_Descriptor with this ES_ID shall be aggregated to the same OD. One independent elementary stream per object descriptor and all its dependent elementary streams may be selected for concurrent use in the scene.

Stream dependencies are governed by the following rules:

— For dependent ES of `streamType` equal to either audioStream or visualStream the dependent ES shall have the same `streamType` as the ES on which it depends. This implies that the dependent stream contains enhancement information to the one it depends on. The precise semantic meaning of the dependencies is opaque at this layer.

— An ES with a `streamType` of SceneDescriptionStream shall only depend on an ES with `streamType` of SceneDescriptionStream or ObjectDescriptorStream.

— Dependency on an ObjectDescriptorStream implies that the ObjectDescriptorStream contains the object descriptors that are refered to by this SceneDescriptionStream.

— Dependency on a SceneDescriptionStream implies that the dependent stream contains enhancement information to the one it depends on. The dependent SceneDescriptionStream shall depend on the same ObjectDescriptorStream on which the other SceneDescriptionStream depends.

— An ES with a `streamType` of ObjectDescriptorStream shall only depend on an ES with a `streamType` of SceneDescriptionStream. This dependency does not have implications for the object descriptor stream.

— Only if a second stream with `streamType` of SceneDescriptionStream depends on this stream with `streamType` = ObjectDescriptorStream, it implies that the second SceneDescriptionStream depends on the first SceneDescriptionStream. The object descriptors in the ObjectDescriptorStream shall only be valid for the second SceneDescriptionStream.

— An ES that flows upstream, as indicated by `DecoderConfigDescriptor.upStream` = 1 shall always depend upon another ES that has the `upStream` flag set to zero. This implies that this upstream is associated to the downstream it depends on.

— The availability of the dependent stream is undefined if an ES_Descriptor for the stream it depends upon is not available.

8.7.2 Linking Scene Description and Object Descriptors

8.7.2.1 Associating Object Descriptors to BIFS Nodes

Some BIFS nodes contain an **url** field. Such nodes are associated to their elementary stream resources (if any) via an object descriptor. The association is established by means of the `objectDescriptorID`, as specified in 9.3.7.18.2. The name scope for this ID is specified in 8.7.2.4.

Each BIFS node requires a specific streamType (audio, visual, inlined scene description, etc.) for its associated elementary streams. The associated object descriptor shall contain ES_Descriptors with this streamType. The behavior of the terminal is undefined if an object descriptor contains ES_Descriptors with stream types that are incompatible with the associated BIFS node.

Note that commands adding or removing object descriptors need not be co-incident in time with the addition or removal of BIFS nodes in the scene description that refer to such an object descriptor. However, the behavior of the terminal is undefined if a BIFS node in the scene description references an object descriptor that is no longer valid.

The terminal shall gracefully handle references from the scene description to object descriptors that are not currently available.

8.7.2.2 Multiple scene description and object description streams

An object descriptor that is associated to an **Inline** node of the scene description or that represents the primary access to content compliant with the ISO/IEC 14496 specifications (initial object descriptor) aggregates as a minimum, one scene description stream and the corresponding object descriptor stream (if additional elementary streams need to be referenced).

However, it is permissible to split both the scene description and the object descriptors in multiple streams. This allows a bandwidth-scaleable encoding of the scene description. Each stream shall contain a valid sequence of access units as defined in 9.2.1.3 and 8.5.2, respectively. All resulting scene description streams and object descriptor streams shall remain aggregated in a single object descriptor. The dependency mechanism shall be used to indicate how the streams depend on each other.

All streams shall continue to be processed by a single scene description and object descriptor decoding process, respectively. The time stamps of the access units in different streams shall be used to re-establish the original order of access units.

NOTE — This form of partitioning of the scene description and the object descriptor streams in multiple streams is not visible in the scene description itself.

8.7.2.3 Scene and Object Description in Case of Inline Nodes

The BIFS scene description allows to recursively partition a scene through the use of **Inline** nodes (see 9.4.2.52). Each **Inline** node is associated to an object descriptor that points to at least one additional scene description stream as well as another object descriptor stream (if additional elementary streams need to be referenced). An example for such a hierarchical scene description can be found in Figure 6.

8.7.2.4 Name Scope of Identifiers

The scope of the `objectDescriptorID`, `ES_ID` and `IPMP_DescriptorID` identifiers that label the object descriptors, elementary stream descriptors and IPMP descriptors, respectively, is defined as follows. This definition is based on the restriction that associated scene description and object descriptor streams shall always be aggregated in a single object descriptor, as specified in 8.7.1.4. The following rule defines the name scope:

— Two `objectDescriptorID`, `ES_ID` or `IPMP_DescriptorID` as well as `nodeID` and `ROUTEID` identifiers belong to the same name scope if and only if these identifiers occur in elementary streams with a `streamType` of either ObjectDescriptorStream or SceneDescriptionStream that are aggregated in a single object descriptor.

NOTE 1 — Hence, the difference between the two methods specified in 8.7.2.2 and 8.7.2.3 above to partition a scene description in multiple streams is that the first method allows multiple scene description streams that refer to the same name scope while an **Inline** node opens a new name scope.

NOTE 2 — This implies that a URL in an object descriptor opens a new name scope since it points to an object descriptor that is not carried in the same ObjectDescriptorStream.

8.7.2.5 Reuse of identifiers

For reasons of error resilience, it is recommended not to reuse `objectDescriptorID` and `ES_ID` identifiers to identify more than one object or elementary stream, respectively, within one presentation. That means, if an object descriptor or elementary stream descriptor is removed by means of an OD command and later on reinstalled with another OD command, then it shall still point to the same content item as before.

8.7.3 ISO/IEC 14496 Content Access

8.7.3.1 Introduction

In order to access ISO/IEC 14496 compliant content it is a pre-condition that an initial object descriptor to such content is known through means outside the scope of ISO/IEC 14496. The subsequent content access procedure is specified conceptually, using a number of walk throughs. Its precise definition depends on the chosen delivery layer.

For applications that implement the DMIF Application Interface (DAI) specified in ISO/IEC 14496-6 which abstracts the delivery layer, a mapping of the conceptual content access procedure to calls of the DAI is specified in 8.7.3.9.

The content access procedure determines the set of required elementary streams, requests their delivery and associates them to the scene description. The selection of a subset of elementary streams suitable for a specific ISO/IEC 14496 terminal is possible, either based on profiles or on inspection of the set of object descriptors.

8.7.3.2 The Initial Object Descriptor

Initial object descriptors convey information about the profiles required by the terminal compliant with ISO/IEC 14496 specifications to be able to process the described content. This profile information summarizes the complexity of the content referenced directly or indirectly through this initial object descriptor, i.e., it indicates the overall terminal capabilities required to decode and present this content. Therefore initial object descriptors constitute self-contained access points to content compliant with ISO/IEC 14496 specifications.

There are two constraints to this general statement:

— If the `includeInlineProfileLevelFlag` of the initial object descriptor is not set, the complexity of any inlined content is not included in the profile indications.

— In addition to the elementary streams that are decodable by the terminal conforming to the indicated profiles, alternate content representations might be available. This is further explained in 8.7.3.4.

An initial object descriptor may be conveyed by means not defined in ISO/IEC 14496. The content may be accessed starting from the elementary streams that are described by this initial object descriptor, usually one or more scene description streams and zero or more object descriptor streams.

Content refered to by an initial object descriptor may itself be referenced from another piece of ISO/IEC 14496 content. In this case, the initial object descriptor will be conveyed in an object descriptor stream.

Ordinary object descriptors may be used as well to describe scene description and object descriptor streams. However, since they do not carry profile information, they can only be used to access content if that information is either not required by the terminal or is obtained by other means.

8.7.3.3 Usage of URLs in the Object Descriptor Framework

URLs in the object description framework serve to locate either inlined ISO/IEC 14496 content or the elementary stream data associated to individual audio-visual objects.

URLs in ES_Descriptors locate elementary stream data that shall be delivered as SL-packetized stream by the delivery entity associated to the current name scope. The complete description of the stream (its ES_Descriptor) is available locally.

URLs in object descriptors locate an object descriptor at a remote location. Only the content of this object descriptor shall be returned by the delivery entity upon access to this URL. This implies that the description of the resources for the associated BIFS node or the inlined content is only available at the remote location. Note, however, that depending on the value of `includeInlineProfileLevelFlag` in the initial object descriptor, the global resources needed may already be known (i.e., including remote, inlined portions).

8.7.3.4 Selection of Elementary Streams for an Audio-Visual Object

Elementary streams are attached through their object descriptor to appropriate BIFS nodes which, in most cases, constitute the representation of a single audio-visual object in the scene. The selection of one or more ESs for each BIFS node may be governed by the profile indications that are conveyed in the initial object descriptor. All object descriptors shall at least include one elementary stream with suitable object type to satisfy the initially signaled profiles.

Additionally, object descriptors may aggregate ES_Descriptors for elementary streams that require more computing or bandwidth resources. Those elementary streams may be used by the receiving terminal if it is capable of processing them.

In case initial object descriptors do not indicate any profile and level or if profile and level indications are disregarded, an alternative to the profile driven selection of streams exists. The receiving terminal may evaluate the ES_Descriptors of all available elementary streams for each BIFS node and choose by some non-standardized way for which subset it has sufficient resources to decode them while observing the constraints specified in this subclause.

NOTE — Some restrictions on the selection of and access to elementary streams might exist if a set of elementary streams shares a single object time base (see 10.2.6).

8.7.3.5 Content access in “push” and “pull” scenarios

In an interactive, or “pull” scenario, the receiving terminal actively requests the establishment of sessions and the delivery of content, i.e., streams. This usually involves a session and channel set up protocol between sender and receiver. This protocol is not specified here. However, the conceptual steps to be performed are the same in all cases and are specified in the subsequent clauses.

In a broadcast, or “push” scenario, the receiving terminal passively processes what it receives. Instead of issuing requests for session or channel set up the receiving terminal shall evaluate the relevant descriptive information that associates ES_IDs to their transport channel. The syntax and semantics of this information is outside the scope of ISO/IEC 14496, however, it needs to be present in any delivery layer implementation. This allows the terminal to gain access to the elementary streams forming part of the content.

8.7.3.6 Content access through a known Object Descriptor

8.7.3.6.1 Pre-conditions

— An object descriptor has been acquired. This may be an initial object descriptor.

— The object descriptor contains ES_Descriptors pointing to object descriptor stream(s) and scene description stream(s) using ES_IDs.

— A communication session to the source of these streams is established.

— A mechanism exists to open a channel that takes user data as input and provides some returned data as output.

8.7.3.6.2 Content Access Procedure

The content access procedure shall be equivalent to the following:

1. The object descriptor is evaluated and the ES_ID for the streams that are to be opened are determined.
2. Requests for opening the selected ESs are made, using a suitable channel set up mechanism with the ES_IDs as parameter.
3. The channel set up mechanism shall return handles to the streams that correspond to the requested list of ESs.
4. Requests for delivery of the selected ESs are made.
5. Interactive scenarios: Delivery of streams starts. All scenarios: The streams now become accessible.
6. Scene description and object descriptor stream are evaluated.
7. Further streams are opened as needed with the same procedure, starting at step 1.

8.7.3.7 Content access through a URL in an Object Desciptor

8.7.3.7.1 Pre-conditions

— A URL to an object descriptor or an initial object descriptor has been acquired.

— A mechanism exists to open a communication session that takes a URL as input and provides some returned data as output.

8.7.3.7.2 Content access procedure

The content access procedure shall be equivalent to the following:

1. A connection to the source of the URL is made, using a suitable service set up call.

2. The service set up call shall return data consisting of a single object descriptor.

3. Continue at step 1 in 8.7.3.6.2.

8.7.3.8 Content access through a URL in an elementary stream descriptor

8.7.3.8.1 Pre-conditions

— An ES_Descriptor pointing to a stream through a URL has been aquired. (Note that the ES_Descriptor fully specifies the configuration of the stream.)

— A mechanism exists to open a channel that takes a URL as input and provides some returned data as output.

8.7.3.8.2 Content access procedure

The content access procedure shall be equivalent to the following:

1. Request to open the stream is made, using a suitable channel set up mechanism with the URL as parameter.

2. The channel set up mechanism shall return a handle to the stream that corresponds to the requested URL.

3. Requests for delivery of the selected stream are made.

4. Interactive scenarios: Delivery of stream starts. All scenarios: The stream now becomes accessible.

EXAMPLE — Access to Complex Content

The example in Figure 6 shows a complex piece of ISO/IEC 14496 content, consisting of three parts. The upper part is a scene accessed through its initial object descriptor. It contains, among others a visual and an audio stream. A second part of the scene is inlined and accessed through its initial object descriptor that is pointed to (via URL) in the object descriptor stream of the first scene. Utilization of the initial object descriptor allows the signaling of profile information for the second scene. Therefore this scene may also be used without the first scene. The second scene contains, among others, a scaleably encoded visual object and an audio object. A third scene is inlined and accessed via the ES_IDs of its object descriptor and scene description streams. These ES_IDs are known from an object descriptor conveyed in the object descriptor stream of the second scene. Note that this third scene is not accessed through an initial object descriptor. Therefore the profile information for this scene need to be included in the profile information for the second scene.

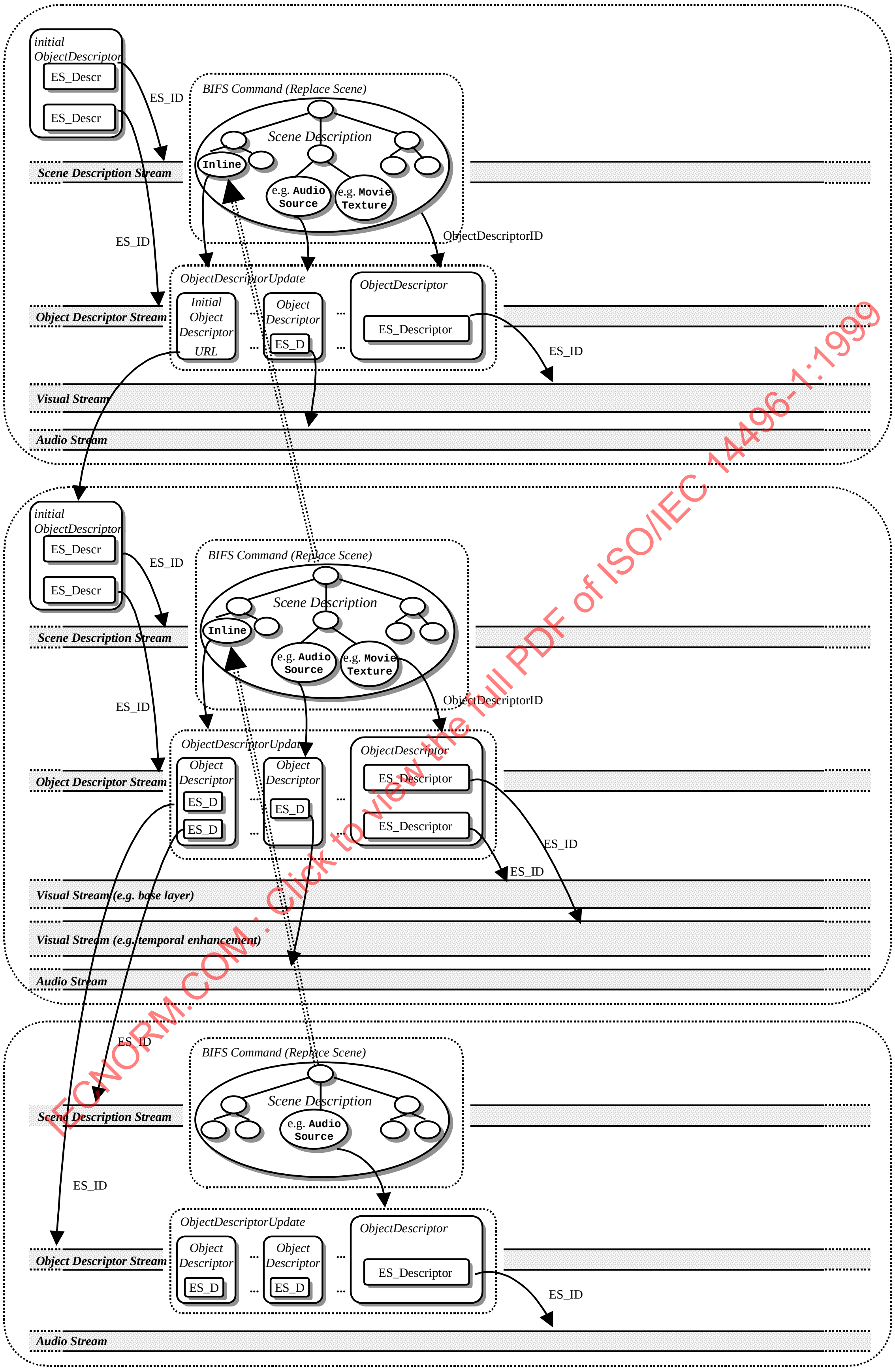


Figure 6 - Complex content example

8.7.3.9 Mapping of Content Access Procedure to DAI calls

The following two DAI primitives, quoted from ISO/IEC 14496-6, subclause 10.4, are required to implement the content access procedure described in 8.7.3.6 to 8.7.3.8:

DA_ServiceAttach (IN: URL, uuDataInBuffer, uuDataInLen;
OUT: response, serviceSessionId, uuDataOutBuffer, uuDataOutLen)

DA_ChannelAdd (IN: serviceSessionId, loop(qosDescriptor, direction, uuDataInBuffer, uuDataInLen);
OUT: loop(response, channelHandle, uuDataOutBuffer, uuDataOutLen))

DA_ServiceAttach is used to implement steps 1 and 2 of 8.7.3.7.2. The URL shall be passed to the IN: URL parameter. UuDataInBuffer shall remain empty. The returned serviceSessionId shall be kept for future reference to this URL. UuDataOutBuffer shall contain a single object descriptor.

DA_ChannelAdd is used to implement steps 2 and 3 of 8.7.3.6.2. serviceSessionId shall be the identifier for the service session that has supplied the object descriptor that includes the ES_Descriptor that is currently processed. QosDescriptor shall be the QoS_Descriptor of this ES_Descriptor, direction shall indicate upstream or downstream channels according to the `DecoderConfigDescriptor.upstream` flag. UuDataInBuffer shall contain the ES_ID of this ES_Descriptor. On successful return, channelHandle shall contain a valid, however, not normative handle to the accessible stream.

DA_ChannelAdd is used to implement steps 1 and 2 of 8.7.3.8.2. serviceSessionId shall be the identifier for the service session that has supplied the object descriptor that includes the ES_Descriptor that is currently processed. QosDescriptor shall be the QoS_Descriptor of this ES_Descriptor, direction shall indicate upstream or downstream channels according to the `DecoderConfigDescriptor.upstream` flag. UuDataInBuffer shall contain the URL of this ES_Descriptor. On successful return, channelHandle shall contain a valid, however, not normative handle to the accessible stream.

NOTE1 — It is a duty of the service to discriminate between the two cases with either ES_ID or URL as parameters to uuDataInBuffer in DA_ChannelAdd.

NOTE2 — Step 4 in 8.7.3.6.2 and step 3 in 8.7.3.8.2 are currently not mapped to a DAI call in a normative way. It may be implemented using the DA_UserCommand() primitive.

The set up example in the following figure conveys an initial object descriptor that points to one SceneDescriptionStream, an optional ObjectDescriptorStream and additional optional SceneDescriptionStreams or ObjectDescriptorStreams. The first request to the DAI will be a DA_ServiceAttach() with the content address as a parameter. This call will return an initial object descriptor. The ES_IDs in the contained ES_Descriptors will be used as parameters to a DA_ChannelAdd() that will return handles to the corresponding channels.

Additional streams (if any) that are identified when processing the content of the object descriptor stream(s) are subsequently opened using the same procedure. The object descriptor stream is not required to be present if no further audio- or visual streams or inlined scene description streams form part of the content.

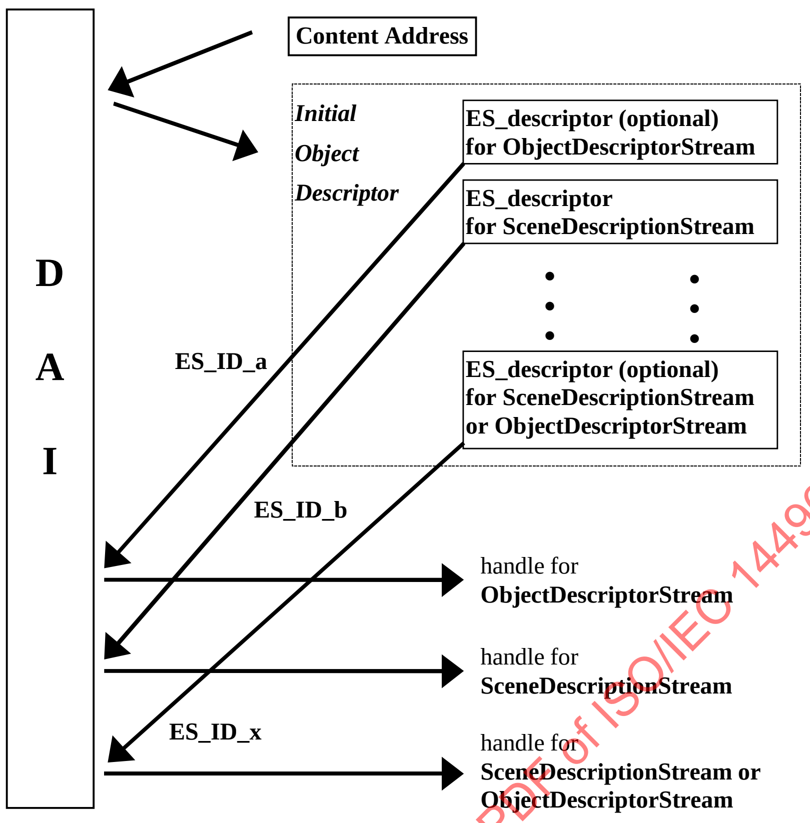


Figure 7 - Requesting stream delivery through the DAI

8.8 Usage of the IPMP System interface

8.8.1 Overview

IPMP elementary streams and descriptors may be used in a variety of ways. For instance, IPMP elementary streams may convey time-variant IPMP information such as keys that change periodically. An IPMP elementary stream may be associated with a given elementary stream or set of elementary streams. Similarly, IPMP descriptors may be used to convey time-invariant or slowly changing IPMP information associated with a given elementary stream or set of elementary streams. This subclause specifies methods how to associate an IPMP system to an elementary stream or a set of elementary streams.

8.8.2 Association of an IPMP System with IS0/IEC 14496 content

8.8.2.1 Association in the initial object descriptor

An IPMP System may be associated with ISO/IEC 14496 content in the initial object descriptor. In that case the initial object descriptor shall aggregate in addition to the ES_Descriptors for scene description and object descriptor streams one or more ES_Descriptors that reference one or more IPMP elementary streams. This implies that all the elementary streams that are described through this initial object descriptor are governed by the one or more IPMP Systems that are identified within the one or more IPMP streams.

8.8.2.2 Association in other object descriptors

An IPMP System may be associated with ISO/IEC 14496 content in an object descriptor in three ways:

In the first case, the object descriptor aggregates in addition to the ES_Descriptors for the content elementary streams one or more ES_Descriptors that reference one or more IPMP elementary streams. This implies that all the content elementary streams described through this object descriptor are governed by the one or more IPMP Systems that are identified within the one or more IPMP streams. Note that an ES_Descriptor that describes an IPMP stream may contain references to IPMP_Descriptors.

The second method is to include one or more IPMP_DescriptorPointers in the object descriptor. This implies that all content elementary streams described by this object descriptor are governed by the IPMP System(s) that is/are identified within the referenced IPMP descriptor(s).

The third method is to include IPMP_DescriptorPointers in the ES_Descriptors embedded in this object descriptor. This implies that the elementary stream referenced by such an ES_Descriptor is controlled by an IPMP System.

8.8.3 IPMP of Object Descriptor streams

Object Descriptor streams shall not be affected by IPMP Systems, i.e., they shall always be available without protection.

An IPMP_Descriptor associated with an object descriptor stream through an IPMP_DescriptorPointer implies that an IPMP System controls all elementary streams that are referred to by this object descriptor stream.

8.8.4 IPMP of Scene Description streams

Scene description streams are treated like any media stream, i.e. they may be managed by an IPMP System.

An IPMP_Descriptor associated with a scene description stream implies that the IPMP System controls this scene description stream.

There are two ways to protect part of a scene description (or to apply different IPMP Systems to different components of a given scene):

The first method exploits the fact that it is permissible to have more than one scene description stream associated with one object descriptor (see 8.7.2.2). Such a split of the scene description can be freely designed by a content author, for example, putting a basic scene description into the first stream and adding one or more additional scene description streams that enhance this basic scene using BIFS updates.

The second method is to structure the scene using one or more **Inline** nodes (see 9.4.2.52). Each **Inline** node refers to one or more additional scene description streams, each of which might use a different IPMP System.

8.8.5 Usage of URLs in managed and protected content

8.8.5.1 URLs in the BIFS Scene Description

ISO/IEC 14496 does not specify compliance points for content that uses BIFS URLs that do not point to an object descriptor. Equally, no normative way to apply an IPMP System to such links exists. The behavior of an IPMP-enabled terminal that encounters such links is undefined.

8.8.5.2 URLs in Object Descriptors

URLs in object descriptors point to other remote object descriptors. This merely constitutes an indirection and should not adversely affect the behavior of the IPMP System that might be invoked through this remote object descriptor.

NOTE — The only difference is that while the original site might be trusted, the referred one might not. Further corrective actions to guard against this condition are not in the scope of ISO/IEC 14496.

8.8.5.3 URLs in ES_Descriptors

URLs in ES descriptors are used to access elementary streams remotely. This merely constitutes an indirection and therefore does not adversely affect the behavior of the IPMP System that might be invoked through this remote object descriptor.

NOTE — The only difference is that while the original site might be trusted, the referred one might not. Further corrective actions to guard against this condition are not in the scope of ISO/IEC 14496.

8.8.6 IPMP Decoding Process

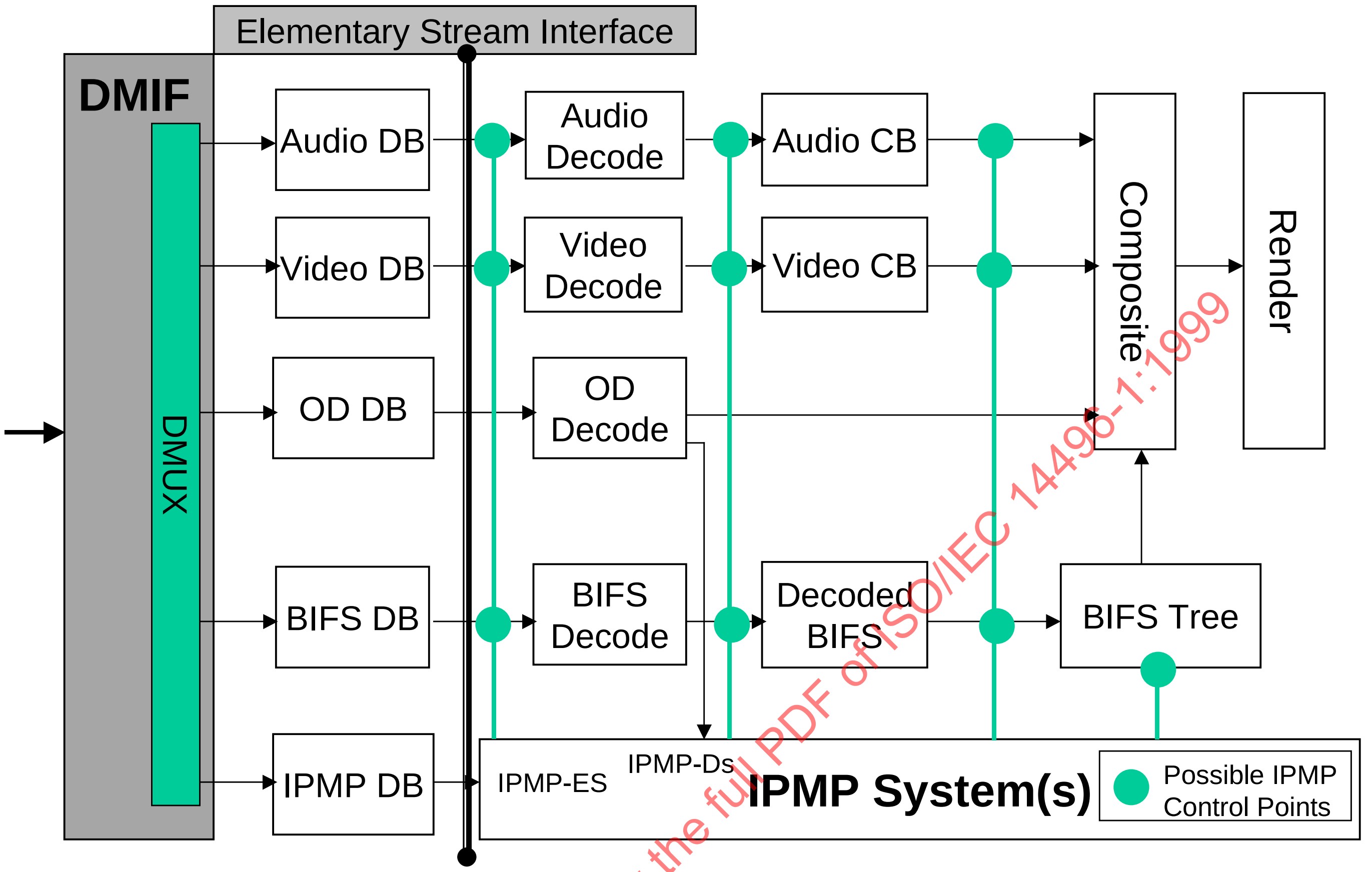


Figure 8 - IPMP system in the ISO/IEC 14496 terminal architecture

Figure 8 depicts the injection of an IPMP System with respect to the MPEG-4 terminal. IPMP System specific data is supplied to the IPMP System via IPMP streams and/or IPMP descriptors, and the IPMP system releases protected content *after the sync layer.*

Each elementary stream under the control of an IPMP System has the conceptual element of a *stream flow controller.* Stream flow control can take place between the the SyncLayer decoder and the decoder buffer. As the figure indicates, elements of IPMP control may take place at other points in the terminal including, after decoding (as with some watermarking systems) or in the decoded BIFS stream, or after the composition buffers have been written, or in the BIFS scene tree. Stream flow controllers either enable or disable processing of an elementary stream in a non-normative way that depends on the status information provided by the IPMP System.

Finally, the IPMP System must at a minimum:

1. Process the IPMP stream and descriptor

2. Appropriately manage (e.g. decrypt and release) protected elementary streams.

The initialization process of the IPMP System is not specified except that it shall not unduly delay the content access process as specified in 8.7.3.

9 Scene Description

9.1 Introduction

9.1.1 Scope

ISO/IEC 14496 addresses the coding of audio-visual objects of various types: natural video and audio objects as well as textures, text, 2- and 3-dimensional graphics, and also synthetic music and sound effects. To reconstruct a multimedia scene at the terminal, it is hence not sufficient to transmit the raw audio-visual data to a receiving terminal. Additional information is needed in order to combine this audio-visual data at the terminal and construct and present to the end user a meaningful multimedia scene. This information, called scene description, determines the placement of audio-visual objects in space and time and is transmitted together with the coded objects as illustrated in Figure 9. Note that the scene description only describes the structure of the scene. The action of assembling these objects in the same representation space is called composition. The action of transforming these audio-visual objects from a common representation space to a specific presentation device (i.e., speakers and a viewing window) is called rendering.

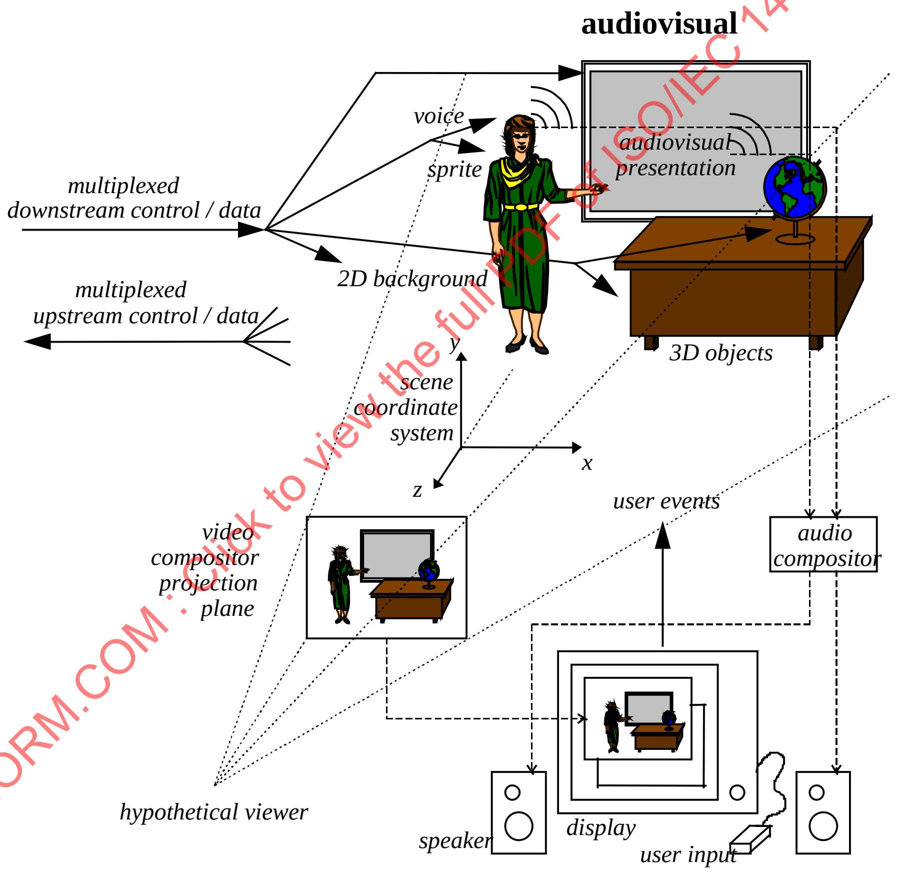


Figure 9 - An example of an object-based multimedia scene

Independent coding of different objects may achieve higher compression, and also brings the ability to manipulate content at the terminal. The behaviors of objects and their response to user inputs can thus also be represented in the scene description.

The scene description framework used in ISO/IEC 14496-1 is based largely on ISO/IEC 14772-1:1998 (Virtual Reality Modeling Language – VRML) [10].

9.1.2 Composition and Rendering

ISO/IEC 14496-1 defines the syntax and semantics of bitstreams that describe the spatio-temporal relationships of audio-visual objects. For visual data, particular composition algorithms are not mandated since they are implementation-dependent; for audio data, subclause 9.2.2.13 and the semantics of the AudioBIFS nodes normatively define the composition process. The manner in which the composed scene is presented to the user is not specified for audio or visual data. The scene description representation is termed “BInary Format for Scenes” (BIFS).

9.1.3 Scene Description

In order to facilitate the development of authoring, editing and interaction tools, scene descriptions are coded independently from the audio-visual media that form part of the scene. This permits modification of the scene without having to decode or process in any way the audio-visual media. The following clauses detail the scene description capabilities that are provided by ISO/IEC 14496-1.

9.1.3.1 Grouping of audio-visual objects

A scene description follows a hierarchical structure that can be represented as a graph. Nodes of the graph form audio-visual objects, as illustrated in Figure 10. The structure is not necessarily static; nodes may be added, deleted or be modified.

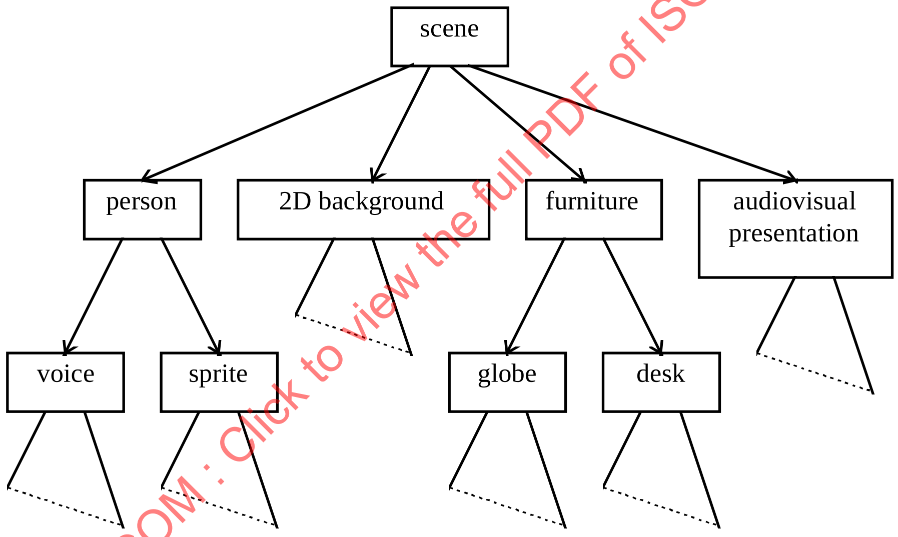


Figure 10 - Logical structure of example scene

9.1.3.2 Spatio-Temporal positioning of objects

Audio-visual objects have both a spatial and a temporal extent. Complex audio-visual objects are constructed by combining appropriate scene description nodes to build up the scene graph. Audio-visual objects may be located in 2D or 3D space. Each audio-visual object has a local co-ordinate system. A local co-ordinate system is one in which the audio-visual object has a pre-defined (but possibly varying) spatio-temporal location and scale (size and orientation). Audio-visual objects are positioned in a scene by specifying a co-ordinate transformation from the object’s local co-ordinate system into another co-ordinate system defined by a parent node in the scene graph.

9.1.3.3 Attributes of audio-visual objects

Scene description nodes expose a set of parameters through which aspects of their appearance and behavior can be controlled.

EXAMPLE — the volume of a sound; the color of a synthetic visual object; the source of a streaming video.

9.1.3.4 Behavior of audio-visual objects

ISO/IEC 14496-1 provides tools for enabling dynamic scene behavior and user interaction with the presented content. User interaction can be separated into two major categories: client-side and server-side. Client-side interaction is an integral part of the scene description described herein. Server-side interaction is not dealt with.

Client-side interaction involves content manipulation that is handled locally at the end-user's terminal. It consists of the modification of attributes of scene objects according to specified user actions.

EXAMPLE — A user can click on a scene to start an animation or video sequence. The facilities for describing such interactive behavior are part of the scene description, thus ensuring the same behavior in all terminals conforming to ISO/IEC 14496-1.

9.2 Concepts

9.2.1 BIFS Elementary Streams

9.2.1.1 Overview

BIFS is a compact binary format representing a pre-defined set of audio-visual objects, their behaviors, and their spatio-temporal relationships. The BIFS scene description may, in general, be time-varying. Consequently, BIFS data is carried in a dedicated elementary stream and is subject to the provisions of the systems decoder model (see clause 7). Portions of BIFS data that become valid at a given point in time are contained in BIFS `CommandFrames` or `AnimationFrames` and are delivered within time-stamped access units. Note that the initial BIFS scene is sent as a BIFS-Command, although it is not required, in general, that a BIFS `CommandFrame` contains a complete BIFS scene description.

9.2.1.2 BIFS Decoder Configuration

BIFS configuration information is contained in a `BIFSConfig` (see 9.3.5.2) syntax structure, which is transmitted as `DecoderSpecificInfo` for the BIFS elementary stream in the corresponding object descriptor (see 8.6.6). This gives basic information that must be known by the terminal in order to parse the BIFS elementary stream. In particular, it indicates whether the stream consists of BIFS-Command or BIFS-Anim entities.

9.2.1.3 BIFS Access Units

A BIFS data access unit consists of one BIFS `CommandFrame` or `AnimationFrame`, as defined in 9.3.6.2 and 9.3.8.2, respectively. The BIFS `CommandFrame` or `AnimationFrame` shall convey all the data that is to be processed at any given instant in time. Access units in BIFS streams shall be labelled and time-stamped by suitable means. This shall be done via the related flags and the composition time stamps (CTS), respectively, in the SL packet header (see 10.2.4). The composition time indicates the point in time at which the `CommandFrame` or `AnimationFrame` embedded in a BIFS access unit shall become valid. This means that any changes to audio-visual objects that are described in the BIFS access unit will become visible or audible at precisely this time in an ideal compositor, unless a different behavior is specified by the fields of their nodes. Decoding and composition time for a BIFS access unit shall always have the same value.

An access unit does not necessarily convey a complete scene. In that case it just modifies the persistent state of the scene description. However, if an access unit conveys a complete scene as required at a given point in time it shall set the `randomAccessPointFlag` in the SL packet header to '1' for this access unit. Otherwise, the `randomAccessPointFlag` shall be set to '0'.

9.2.1.4 Time base for BIFS streams

The time base associated to a BIFS stream shall be indicated by suitable means. This shall be done by means of object clock reference time stamps in the SL packet headers (see 10.2.4) for this stream or by indicating the elementary stream from which this BIFS stream inherits the time base (see 10.2.3). All time stamps in the SL-packetized BIFS stream refer to this time base.

9.2.1.5 Multiple BIFS streams

Scene description data may be conveyed in more than one BIFS elementary streams. Two distinct mechanisms exist to associate a set of BIFS elementary streams to a single scene.

The first method uses **Inline** nodes (see 9.4.2.52) in a BIFS scene description. Each such node refers to further BIFS elementary streams. In this case, multiple BIFS streams have a hierarchical dependency. Each **Inline** node opens a new name scope for the identifiers used to label BIFS elements (`nodeID`, `ROUTEID`, `objectDescriptorID`). Therefore, it is not possible to pass events between parts of a scene that reside below different **Inline** nodes.

EXAMPLE 1 — An application of hierarchical BIFS streams is a multi-user virtual conferencing scene, where sub-scenes originate from different sources. Usually, it is neither possible nor useful to specify interaction between two such disjoint parts of the scene.

The second method to associate multiple BIFS elementary streams to a single scene is to group their elementary stream descriptors in a single object descriptor (see 8.7.2.2). In this case, these BIFS streams share the same scope for the identifiers they use (`nodeID`, `ROUTEID`, `objectDescriptorID`). This allows a single scene to be partitioned into multiple streams.

EXAMPLE 2 — An application may offer a presentation with different levels of detail, corresponding to different data rates and different computational complexity. By sharing the same name scope, the more detailed scene description can build on the simple one, rather than sending the entire scene again.

9.2.1.6 Time

9.2.1.6.1 Time-dependent nodes

The semantics of the **loop**, **startTime** and **stopTime** exposedFields and the **isActive** eventOut in time-dependent nodes are as described in ISO/IEC 14772-1:1998, subclause 4.6.9 [10]. **startTime**, **stopTime** and **loop** apply only to the local start, pause and restart of media and do not affect the delivery of the stream attached to the time dependent node. ISO/IEC 14496-1 has the following time-dependent nodes: **AnimationStream**, **AudioBuffer**, **AudioClip**, **AudioSource**, **MovieTexture** and **TimeSensor**.

9.2.1.6.2 Time fields in BIFS nodes

Several BIFS nodes have fields of type SFTime that identify a point in time at which an event occurs (change of a parameter value, start of a media stream, etc). Depending on the individual field semantics, these fields may contain time values that refer either to an absolute position on the time line of the BIFS stream or that define a time duration.

As defined in 9.2.1.4, the speed of the flow of time for events in a BIFS stream is determined by the time base of the BIFS stream. This determines unambiguously durations expressed by relative SFTime values like the **cycleTime** field of the **TimeSensor** node.

The semantics of some SFTime fields is such that the time values shall represent an absolute position on the time line of the BIFS stream (e.g. **startTime** in **MovieTexture**). This absolute position is defined as follows:

Each node in the scene description has an associated point in time at which it is inserted in the scene graph or at which an SFTime field in such a node is updated through a `CommandFrame` in a BIFS access unit (see 9.2.1.3). The value in the SFTime field is the positive offset from this point in time in seconds. Negative values are not permitted. The absolute position on the time line shall therefore be calculated as the sum of the CTS value of the BIFS access unit and the value of the SFTime field.

NOTE 1 — Absolute time in ISO/IEC 14772-1:1998 is defined slightly differently. Due to the non-streamed nature of the scene description in that case, absolute time corresponds to wallclock time in [10].

NOTE 2 — The SFTime fields that define the start or stop of a media stream are relative to the BIFS time base. If the time base of the media stream is a different one, it is not generally possible to set a **startTime** that corresponds exactly to the composition time of a composition unit of this media stream.

EXAMPLE — The example in Figure 11 shows a BIFS access unit that is to become valid at CTS. It conveys a node that has an associated media elementary stream. The startTime of this node is set to a positive value Δt. Hence, startTime will occur Δt seconds after the CTS of the BIFS access unit that has incorporated this node (or the value of the startTime field) in the scene graph.

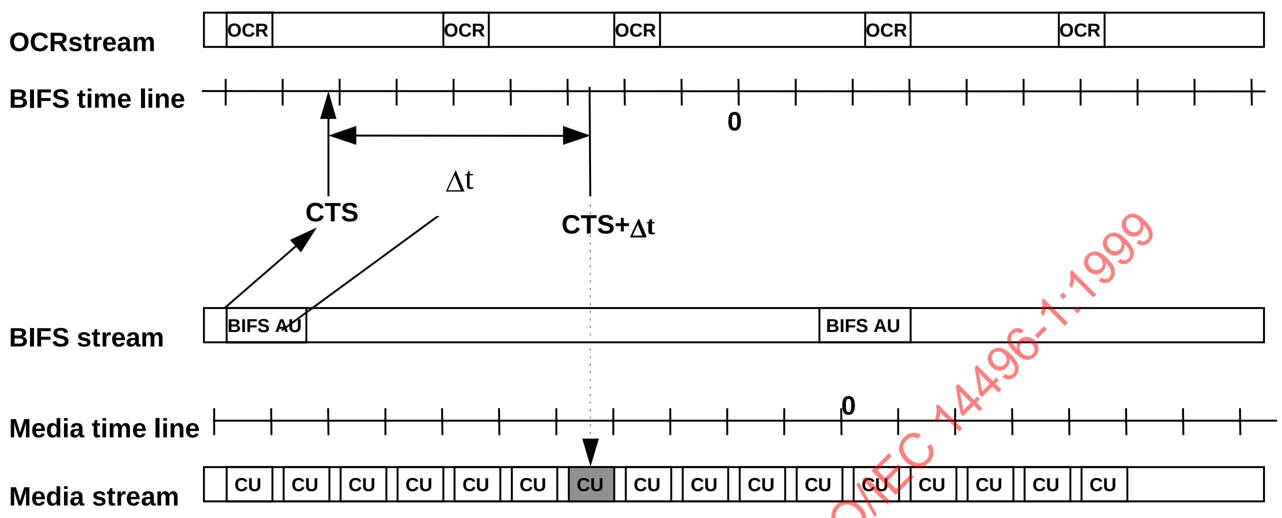


Figure 11 - Media start times and CTS

9.2.2 BIFS Scene Graph

9.2.2.1 Structure of the BIFS scene graph

Conceptually, BIFS scenes represent (as in ISO/IEC 14772-1:1998 [10]) a set of visual and audio primitives distributed in a directed acyclic graph, in a 3D space. However, BIFS scenes may fall into several sub-categories representing particular cases of this conceptual model. In particular, BIFS scene descriptions support scenes composed of:

— 2D primitives (only)

— 3D primitives (only)

— A combination of 2D and 3D primitives

— Audio primitives (only)

In scenes combining 2D and 3D primitives, the following possibilities exist:

— Complete 2D and 3D scenes layered in a 2D space with depth

— 2D and 3D scenes used as texture maps for 2D or 3D primitives

— 2D scenes drawn in the local X-Y plane of the local co-ordinate system in a 3D scene

Figure 12 describes a typical BIFS scene structure.

A BIFS scene shall start with a one of the following nodes: **OrderedGroup**, **Group**, **Layer2D**, **Layer3D**. When the profile used enables visual elements to be composed, the first node indicates the co-ordinate system and context (2D or 3D) to be used for the children of that node. The following rules apply:

— Scene starts with a **Layer2D** or **OrderedGroup** node: A 2D co-ordinate system and context is assumed.

— Scene starts with a **Layer3D** or **Group** node : A 3D co-ordinate system and context is assumed.

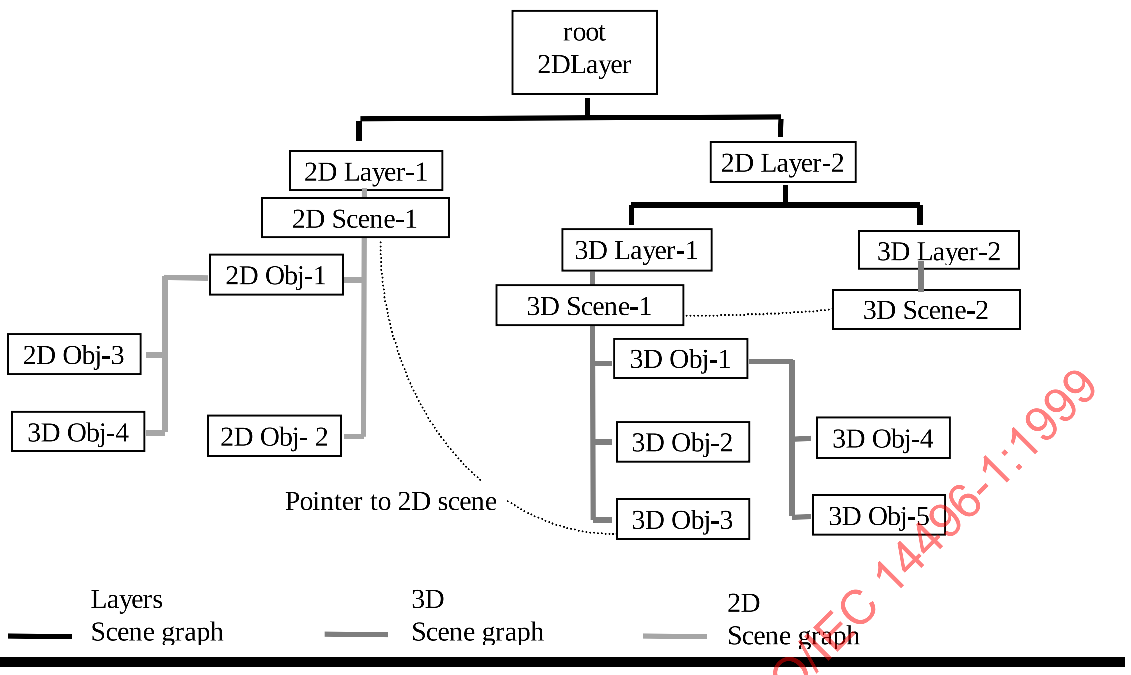


Figure 12 - Scene graph example.

The hierarchy of three different scene graphs is shown: a 2D graphics scene graph and two 3D graphics scene graphs combined with the 2D scene via layer nodes. As shown in the picture, the 3D Layer-2 is the same scene as 3D Layer-1, but the viewpoint may be different. The 3D Obj-3 is an Appearance node that uses the 2D Scene-1 as a texture node.

9.2.2.2 2D Co-ordinate System

The origin of the 2D co-ordinate system is positioned in the center of the rendering area, the x-axis is positive to the right, and the y-axis is positive upwards.

The width of the rendering area represents -1.0 to +1.0 (meters) on the x-axis (see Figure 13). The extent of the y-axis in the positive and negative directions is determined by the aspect ratio of the rendering area so that the unit of distance is equal in both directions. The rendering area is either the entire screen, or window on a computer screen, when viewing a single 2D scene, or the rectangular area defined by the texture used in a **CompositeTexture2D** node, or a **Layer2D** node that contains a subordinate 2D scene description.

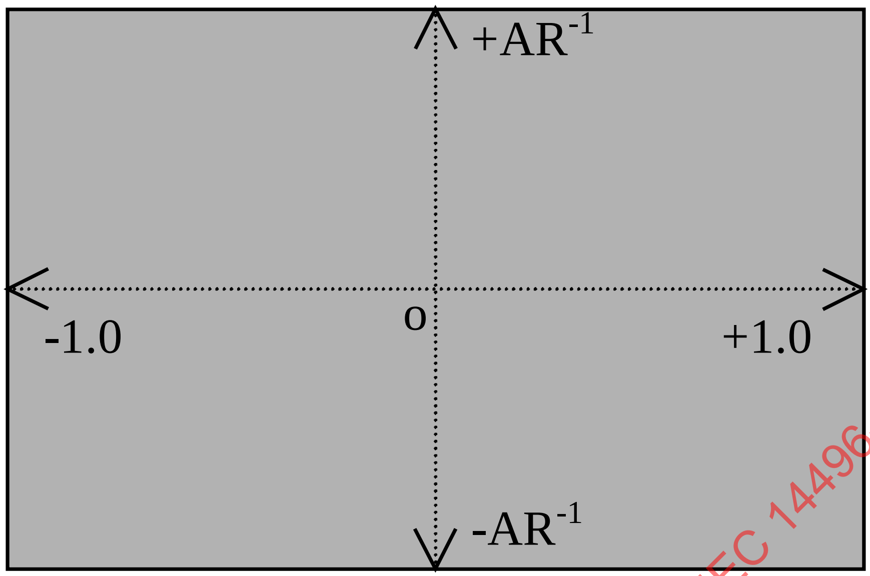


Figure 13 - 2D co-ordinate system (AR = Aspect Ratio)

9.2.2.3 3D Co-ordinate System

The 3D co-ordinate system is as described in ISO/IEC 14772-1:1998, subclause 4.4.5 [10]. When 2D objects are described in a 3D space, they are drawn in the local (x,y) plane (z=0), and the units used are those of the 3D co-ordinate system for the x and y directions.

9.2.2.4 Mixing 2D and 3D scenes

— A single BIFS scene may contain both 2D and 3D elements. The following methods exist:

— 2D primitives may be placed in a 3D scene graph. In this cased, the 2D primitives are drawn in the local (x,y) plane, and use the local coordinate system, restricted to this (x,y) plane.

— 2D and 3D scenes may be composed and overlapped on the screen using **Layer2D** and **Layer3D** nodes. This is useful, for instance, when it is desirable to have 2D interfaces to 3D worlds ("head up" display), or a 3D insert in a 2D scene.

— 2D and 3D scenes may be mapped onto any given geometry using the **CompositeTexture2D** and **CompositeTexture3D** nodes. For instance, 2D scenes may be mapped onto animated 3D geometry to perform special effects.

9.2.2.5 Drawing Order

It is possible to specify the drawing order of elements of the scene, using the **OrderedGroup** node. This feature may be used for 2D or 3D scenes. 2D scenes are considered to have zero depth. Nonetheless, it is important to be able to specify the order in which 2D objects are composed, in order to describe their apparent depths. 3D scenes may use the drawing order facility to solve conflicts of coplanar polygons or other rendering optimizations.

The following rules determine the drawing order, including conflict resolution for objects having the same drawing order:

1. The object having the lowest drawing order shall be drawn first (taking into account negative values).

2. Objects having the same drawing order shall be drawn in the order in which they appear in the scene description.

9.2.2.6 Pixel and Meter metrics

In addition to meter-based metrics, it is also possible to use pixel-based metrics. In this case, 1 meter is set to be equal to the distance between two pixels. This applies to both the horizontal (x-axis) and vertical (y-axis) directions.

The selection of the appropriate metrics is performed by the content creator. In particular, it is controlled by the `BIFSConfig` syntax (see 9.3.5.2): when `pixelMetric` is set to 1, pixel metrics shall be used for the entire scene.

9.2.2.7 Nodes and fields

9.2.2.7.1 Nodes

The BIFS scene description consists of a collection of nodes that describe the scene structure. An audio-visual object in the scene is described by one or more nodes, which may be grouped together (using a grouping node). Nodes are grouped into node data types (NDTs) and the exact type of the node is specified using a `nodeType` field.

An audio-visual object may be completely described within the BIFS information, e.g. **Box** with **Appearance**, or may also require elementary stream data from one or more audio-visual objects, e.g. **MovieTexture** or **AudioSource**. In the latter case, the node includes a reference to an object descriptor that indicates which elementary stream(s) is (are) associated with the node, or directly to a URL description (see ISO/IEC 14772-1:1998, subclause 4.5.2 [10]). With the exception of the **Anchor** and **Script** nodes, a **url** field may only refer to content that conforms to a valid profile and level for the terminal.

9.2.2.7.2 Fields and Events

See ISO/IEC 14772-1:1998, subclause 5.1 [10].

9.2.2.8 Internal, ASCII and Binary Representation of Scenes

ISO/IEC 14496-1 describes the attributes of audio-visual objects using node structures and fields. These fields can be one of several types (see 9.2.2.7.2). To facilitate animation of the content and modification of the objects' attributes in time, within the terminal, it is necessary to use an internal representation of nodes and fields as described in the node specifications (see 9.4). This is essential to ensure deterministic behaviour in the terminal's compositor, for instance when applying ROUTEs or differentially coded BIFS-Anim frames. The observable behaviour of compliant terminals shall not be affected by the way in which they internally represent and transform data; that is, they shall behave as if their internal representation is as defined herein.

However, when encoding the BIFS scene description, different attributes may need to be quantized or compressed appropriately. Thus, the binary representation of fields may differ according to the types of fields, or according to the precision needed to represent a given audio-visual object's attributes. The semantics of nodes are described in 9.4. The binary syntax which represents the binary format as transported in streams conforming to ISO/IEC 14496-1 is provided in 9.3 and uses the node coding parameters provided in Annex H.

9.2.2.8.1 Binary Syntax Overview

9.2.2.8.1.1 Scene Description

The entire scene is represented by a binary encoding of the scene graph. This encoding restricts the VRML grammar as defined in ISO/IEC 14778-1:1997, Annex A [10], but still enables the representation of any scene that can be generated by this grammar.

EXAMPLE — One example of the grammatical differences is the fact that all ROUTEs are represented at the end of a BIFS scene, and that a global grouping node is required at the top level of the scene.

9.2.2.8.1.2 Node Description

Node types are encoded according to the context of the node. This improves efficiency by exploiting the fact that not all nodes are valid at all places in the scene graph. In many instances, only one of a subset of all BIFS nodes is valid at a particular place in the scene graph, and hence in the bitstream.

9.2.2.8.1.3 Fields description

Fields may be quantized to improve compression efficiency. Several aspects of the inverse quantization process can be controlled by adjusting the parameters of the **QuantizationParameter** node.

9.2.2.8.1.4 ROUTE description

All ROUTEs are described at the end of the scene. This improves bit efficiency by grouping these elements in a single location in the bitstream and removes the need for switches in the syntax to allow ROUTEs and nodes to be described in a mixed format.

9.2.2.9 Basic Data Types

There are two general classes of fields and events: fields/events that contain a single value (e.g. a single number or a vector), and fields/events that contain multiple values. Multiple-valued fields/events have names that begin with MF, whereas single valued begin with SF.

9.2.2.9.1 Numerical data and string data types

9.2.2.9.1.1 Introduction

For each basic data type, single field and multiple field data types are defined in ISO/IEC 14772-1:1998, subclause 5.2 [10]. Some further restrictions are described herein.

9.2.2.9.1.2 SFInt32/MFInt32

When routing values between two SFInt32s note shall be taken of the valid range of the destination. If the value being conveyed is outside the valid range, it shall be clipped to be equal to either the maximum or minimum value of the valid range, as follows:

if x > max, x := max

if x < min, x := min

9.2.2.9.1.3 SFTime

The SFTime field and event specifies a single time value. Time values shall consist of 64-bit floating point numbers indicating a duration in seconds or the number of seconds elapsed since the origin of time as defined in the semantics for each SFTime field.

9.2.2.9.2 Node data types

Nodes in the scene are also represented by a data type, namely SFNode and MFNode types. ISO/IEC 14496-1 also defines a set of sub-types, such as SFColorNode, SFMaterialNode. These node data types (NDTs) allow efficient binary representation of BIFS scenes, taking into account the usage context to achieve better compression. However, the generic SFNode and MFNode types are sufficient for internal representations of BIFS scenes.

9.2.2.10 Attaching nodeIDs to nodes

Each node in a BIFS scene graph may have a `nodeID` associated with it, to be used for referencing. ISO/IEC 14772-1:1998, subclause 4.6.2 [10], describes the DEF statement which is used to attach names to nodes. In BIFS scenes, an integer value is used for the same purpose for `nodeIDs`. The number of bits used to represent these integer values is specified in the `BIFSConfig` syntax (see 9.3.5.2).

The following restrictions apply:

a) Nodes are identified by the use of `nodeIDs`, which are binary numbers conveyed in the BIFS bitstream.

b) The scope of `nodeIDs` is given in 9.2.1.5.

c) No two nodes in the scene graph may have the same `nodeID` at any point in time.

Nodes that have been assigned a `nodeID` may be re-used, as described in ISO/IEC 14772-1:1998, subclause 4.6.3 [10]. Note that this mechanism results in a scene description that is a directed acyclic graph, rather than a simple tree.

The mechanisms that allow modifications to the BIFS scene also depend on the use of `nodeIDs` (see 9.2.2.10).

9.2.2.11 Standard Units

As described in ISO/IEC 14772-1:1998, subclause 4.4.5 [10], the standard units used in the scene description are the following:

Table 13 - Standard units

Category	**Unit**
Distance	Meter
Color Space	RGB [0,1] [0,1] [0,1]
Time	Seconds
Angle	Radians

9.2.2.12 Mapping of Scenes to Screens

BIFS scenes may contain still images and videos that are to be pixel-copied to the rendering device using their native dimensions as produced at the output of their terminals. The **Bitmap** node (see 9.4.2.14) provides a screen-aligned geometry that has the pixel dimensions of the texture that is mapped onto it.

NOTE — When **Bitmap** is used, the same scene will appear differently on screens with different resolutions. BIFS scenes that do not use the **Bitmap** node are independent from the screen on which they are viewed.

9.2.2.12.1 Transparency of visual objects

Content complying with ISO/IEC 14496-1 may include still images or video sequences with representations that include alpha values. These values provide transparency information and are to be treated as specified in ISO/IEC 14772-1:1998, subclause 4.14 [10]. For video sequences represented according to ISO/IEC 14496-2, transparency is handled as specified in ISO/IEC 14496-2.

9.2.2.13 Special considerations for audio

9.2.2.13.1 Audio sub-graphs

Audio nodes are used to build audio scenes in the terminal from audio sources coded with tools specified in ISO/IEC 14496-3. The audio scene description capabilities provide two functionalities:

— “Physical modelling” composition for virtual-reality applications, where the goal is to recreate the acoustic space of a real or virtual environment.

— “Post-production” composition for traditional content applications, where the goal is to apply high-quality signal processing transformations.

Audio may be included in either 2D or 3D scene graphs. In a 3D scene, the audio may be spatially presented to sound as though it originates from a particular 3D direction, according to the positions of the object and the listener.

The **Sound** node is used to attach audio to 3D scene graphs and the **Sound2D** node is used to attach audio to 2D scene graphs. As with visual objects, an audio object represented by one of these nodes has a position in space and time, and is transformed by the spatial and grouping transforms of nodes hierarchically above it in the scene.

The nodes below the **Sound**/**Sound2D** nodes, however, constitute an audio sub-graph. This sub-graph is used to describe a particular audio object through the mixing and processing of several audio streams. Rather than representing a hierarchy of spatio-temporal transformations, the nodes within the audio sub-graph represent a signal flow graph that describes how to create the audio object from the audio coded in the **AudioSource** streams. That is, each audio sub-graph node (**AudioSource**, **AudioMix**, **AudioSwitch**, **AudioFX**, **AudioClip**, **AudioBuffer**, **AudioDelay**) accepts one or several channels of input audio, and describes how to turn these channels of input audio into one or more channels of output. The only sounds presented in the audio-visual scene are those which are the output of audio nodes that are children of a **Sound**/**Sound2D** node (that is, the "highest" outputs in the audio sub-graph). The remaining nodes represent "intermediate results" in the sound computation process and the sound represented therein is not presented to the user.

The normative semantics of each of the audio sub-graph nodes describe the exact manner in which to compute the output audio the input audio for each node based on its parameters.

9.2.2.13.2 Overview of sound node semantics

This subclause describes the concepts for normative calculation of the audio objects in the scene in detail, and describes the normative procedure for calculating the audio signal which is the output of a **Sound/Sound2D** node given the audio signals which are its input.

Recall that the audio nodes present in an audio sub-graph do not each represent a sound to be presented in the scene. Rather, the audio sub-graph represents a signal-flow graph which computes a single (possibly multi-channel) audio object based on a set of audio inputs (in **AudioSource** nodes) and parametric transformations. The only sounds which are presented to the listener are those which are the "output" of these audio sub-graphs, as connected to a **Sound**/**Sound2D** node. This subclause describes the proper computation of this signal-flow graph and resulting audio object.

As each audio source is decoded, it produces data that is stored in composition memory (CM). At a particular time instant in the scene, the compositor shall receive from each audio decoder a CM such that the decoded time of the first audio sample of the CM for each audio source is the same (that is, the first sample is synchronized at this time instant). Each CM will have a certain length, depending on the sampling rate of the audio source and the clock rate of the system. In addition, each CM has a certain number of channels, depending on the audio source .

Each node in the audio sub-graph has an associated input buffer and output buffer, except for the **AudioSource** node which has no input buffer. The CM for the audio source acts as the input buffer of audio for the **AudioSource** with which the decoder is associated. As with CM, each input and output buffer for each node has a certain length, and a certain number of channels.

As the signal-flow graph computation proceeds, the output buffer of each node is placed in the input buffer of its parent node, as follows:

If an audio node, *N*, has *n* children, and each of the children produces *k(i)* channels of output, for 1 <= *i* <= *n*, then the node, *N*, shall have *k*(1) + *k*(2) + ... + *k*(*n*) channels of input, where the first *k*(1) channels [number 1 through *k*(1)] shall be the channels of the first child, the next *k*(2) channels [number *k*(1)+1 through *k*(1)+*k*(2)] shall be the channels of the second child, and so forth.

Then, the output buffer of the node is calculated from the input buffer based on the particular rules for that node.

9.2.2.13.2.1 Sample-rate conversion

If the various children of a **Sound**/**Sound2D** node do not produce output at the same sampling rate, then the lengths of the output buffers of the children do not match, and the sampling rates of the children's' output must be brought into alignment in order to place their output buffers in the input buffer of the parent node. The sampling rate of the input buffer for the node shall be the fastest of the sampling rates of the children. The output buffers of the children shall be resampled to be at this sampling rate. The particular method of resampling is non-normative, but the quality shall be close in accuracy to the DAC that the signal is targeted for, i.e. according to the rule `dB SNR = 6 * (nbits –1)`, where `nbits` is the number of bits corresponding to the maximum bit depth of any of the signals being so converted and/or composited. Aliasing artifacts may be at this level of signal-to-noise ratio. The noise level due to arithmetic accuracy and other uncorrelated noise sources should be below the rule `dB SNR = 6* nbits`.

The output sampling rate of a node shall be the output sampling rate of the input buffers after this resampling procedure is applied.

Content authors are advised that content which contains audio sources operating at many different sampling rates, especially sampling rates which are not related by simple rational values, may produce scenes with a high computational complexity.

EXAMPLE — Suppose that node N has children M1 and M2, all three audio nodes, and that M1 and M2 produce output at S1 and S2 sampling rates respectively, where S1 > S2. Then if the decoding frame rate is F frames per second, then M1's output buffer will contain S1/F samples of data, and M2's output buffer will contain S2/F samples of data. Then, since M1 is the faster of the children, its output buffer values are placed in the input buffer of N. The output buffer of M2 is resampled by the factor S1/S2 to be S1/F samples long, and these values are placed in the input buffer of N. The output sampling rate of N is S1.

9.2.2.13.2.2 Number of output channels

If the **numChan** field of an audio node, which indicates the number of output channels, differs from the number of channels produced according to the calculation procedure in the node description, or if the **numChan** field of an **AudioSource** node differs in value from the number of channels of an input audio stream, then the **numChan** field shall take precedence when including the source in the audio sub-graph calculation, as follows:

a) If the value of the **numChan** field is strictly less than the number of channels produced, then only the first **numChan** channels shall be used in the output buffer.

b) If the value of the **numChan** field is strictly greater than the number of channels produced, then the "extra" channels shall be set to all 0's in the output buffer.

9.2.2.13.3 Audio-specific BIFS Nodes

In the following table, nodes that are related to audio scene description are listed.

Table 14 – Audio-Specific BIFS Nodes

Node	Purpose	Subclause
AudioBuffer	Interactively trigger snippets of sound	9.4.2.4
AudioClip	Insert an audio clip into a scene	9.4.2.5
AudioDelay	Add delay to sound	9.4.2.6
AudioMix	Mix sounds	9.4.2.8
AudioSource	Define audio source input to a scene	9.4.2.9
AudioFX	Apply post-production effects to sound	9.4.2.7
AudioSwitch	Switching of audio sources in a scene	9.4.2.10
ListeningPoint	Define listening point in a scene	9.4.2.57
Sound, **Sound2D**	Define properties of sound	9.4.2.82, 9.4.2.83

9.2.3 Sources of modification to the scene

9.2.3.1 Interactivity and behaviors

To describe interactivity and behavior of scene objects, the event architecture defined in ISO/IEC 14772-1:1998, subclause 4.10 [10], is used. Sensors and routes describe interactivity and behaviors. Sensor nodes generate events based on user interaction or a change in the scene. These events are routed to interpolator or other nodes to change the attributes of these nodes. If routed to an interpolator, a new parameter is interpolated according to the input value, and is finally routed to the node which must process the event

9.2.3.1.1 Attaching ROUTEIDs to routes

`ROUTEIDs` may be attached to routes using the DEF mechanism, described in ISO/IEC 14772-1:1998, subclause 4.6.2 [10]. This allows routes to be subsequently referenced in BIFS-Command structures. `ROUTEIDs` are integer values and the namespace for routes is distinct from that of `nodeIDs`. The number of bits used to represent these integer values is specified in the BIFS `DecoderConfigDescriptor`.

The scope of `ROUTEIDs` is defined in see 9.2.1.5. The following restrictions apply:

a) Routes are identified by the use of `ROUTEIDs`, which are binary numbers conveyed in the BIFS bitstream.

b) The scope of `ROUTEIDs` is given in 9.2.1.5.

c) No two routes in the scene graph may have the same `ROUTEID` at any point in time.

The mechanisms that allow modifications to the BIFS scene also depend on the use of `nodeIDs` (see 9.2.2.10). The USE mechanism shall not be used with routes.

9.2.3.1.2 Conditional node

The **Conditional** node (see 9.4.2.22) allows BIFS-Commands to be described in the scene which shall only be applied to the scene graph when an event is received on one of the **Conditional** node's inputs.

9.2.3.2 External modification of the scene: BIFS-Commands

The BIFS-Command mechanism enables the change of properties of the scene graph, its nodes and behaviors.

EXAMPLE — **Transform** nodes can be modified to move objects in space; **Material** nodes can be changed to modify an object's appearance, and fields of geometric nodes can be totally or partially changed to modify the geometry of objects.

9.2.3.2.1 Overview

BIFS-Commands are used to modify a set of properties of the scene at a given time instant in time. Commands are grouped into `CommandFrames` (see 9.3.6.2) in order to be able to send several commands in a single access unit. The following four basic commands are defined:

1. Replacement of an entire scene
2. Insertion
3. Deletion
4. Replacement

The first of these commands allows the replacement of the entire BIFS scene. The replacement of the entire scene requires a scene graph representing a valid BIFS scene to be transmitted. The `SceneReplace` command is the only random access point in the BIFS stream.

The other three commands can be used to update the following structures:

1. A node
2. An eventIn, exposedField or an indexed value in an MFField
3. A ROUTE

In order to modify the scene the sender must transmit a BIFS `CommandFrame` that contains one or more update commands. A single source of BIFS-Commands is assumed. The identification of a node in the scene is provided by a `nodeID`. Note that it is the sender's responsibility to provide this `nodeID`, which must be unique (see 9.2.1.5). The identification of a node's fields is provided by sending the `INid` of the field (see Annex H).

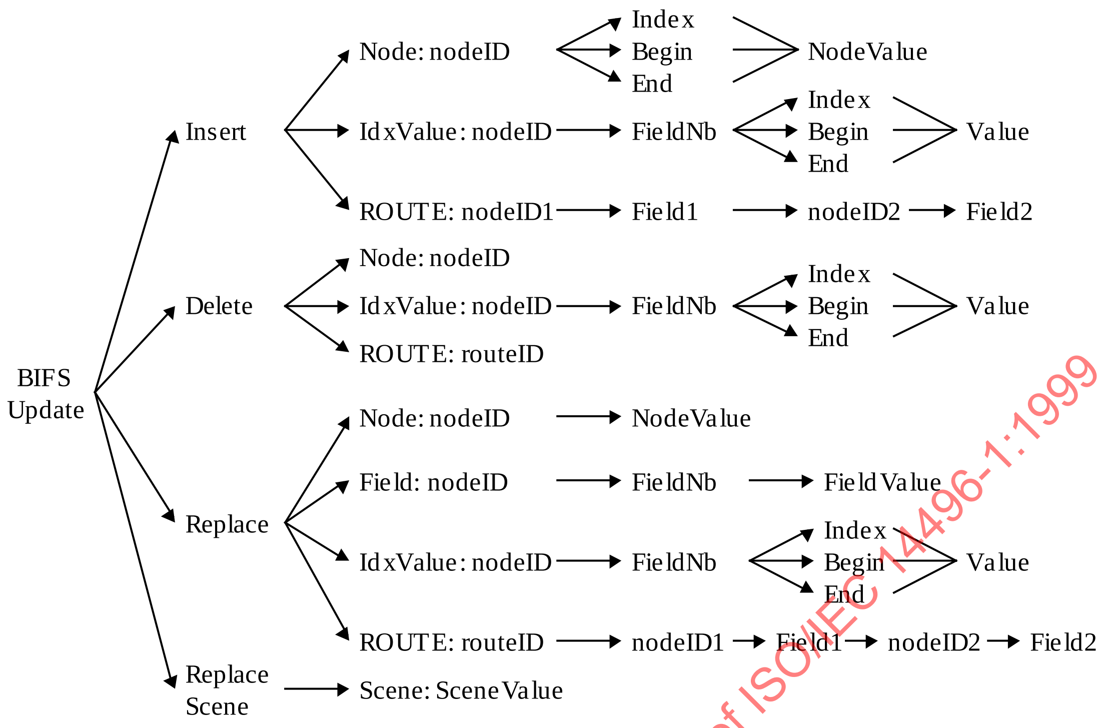


Figure 14 - BIFS-Command Types

9.2.3.2.2 Modification of indexed values

Insertion of an indexed value in a field implies that all later values in the field have their indices incremented and the length of the field increases accordingly. Appending a value to an indexed value field also increases the length of the field but the indices of existing values in the field do not change.

Deletion of an indexed value in a field implies that all later values in the field have their indices decremented and the length of the field decreases accordingly.

9.2.3.2.3 Timing of BIFS-Commands

The time at which a BIFS-Command is applied shall be the composition time stamp of the access unit in which the command is contained, as defined in the sync layer (see 10.2).

9.2.3.3 External animation of the scene: BIFS-Anim

BIFS-Anim provides for the continuous update of the certain fields of nodes in the scene graph. BIFS-Anim is used to integrate different kinds of animation, including the ability to animate face models as well as meshes, 2D and 3D positions, rotations, scale factors, and color attributes. Although BIFS-Anim and BIFS-Command have the same elementary stream type (see Table 9) they may not occupy the same elementary stream. BIFS-Anim information is conveyed in a separate elementary stream from that which carries BIFS-Command elements.

9.2.3.3.1 Overview

BIFS-Anim elementary streams consist of a sequence of `AnimationFrames`. The `AnimationMask`, which is required to interpret these `AnimationFrames`, is transmitted in the `DecoderSpecificInfo` for the BIFS-Anim elementary stream in the corresponding object descriptor (see 8.6.6).

9.2.3.3.2 BIFS-Anim configuration

The `AnimationMask` contains one `ElementaryMask` for each node that is to be animated. These `ElementaryMasks` specify the fields that are contained in the `AnimationFrames` for a given animated node, and their associated quantization parameters. Only eventIn or exposedField fields that have an animation method (see

Annex H and 9.2.3.3.3) can be modified using BIFS-Anim. Such fields are called dynamic fields. In addition, the animated field must be part of an updateable node; that is, a node that has been assigned a `nodeID`. The `AnimationMask` is composed of several elementary masks defining these parameters.

9.2.3.3.3 BIFS-Anim animation parameters

Animation parameters are transmitted as a sequence of `AnimationFrames`. `AnimationFrames` specify the values of the dynamic fields of updateable nodes that are being animated in BIFS-Anim streams. An `AnimationFrame` contains the new values of all animated parameters at a specified time, unless if it is specified that, for some frames, these parameters are not sent. The parameters can be sent in Intra (the absolute value is sent) and Predictive modes (the difference between the current and previous values is sent).

Animation parameters can be applied to any eventIn or exposedField of any updateable node of a scene which has an assigned animation method (see Annex H).

NOTE — Some node tables in Annex H contain an eventIn or exposedField that has an animation method but for which there is no associated `dynID`. This is the case when only one exposedField or eventIn in a node has an animation method. In such cases, it is not necessary for the field to have a `dynID` since the terminal can assume that BIFS-Anim animations for this type of node refer to the only dynamic field of the node.

The types of dynamic fields are:

— SFInt32/MFInt32

— SFFloat/MFFloat

— SFRotation/MFRotation

— SFColor/MFColor

— SFVec2f/MFVec2f

— SFVec3f/MFVec3f

9.2.3.4 Order of application of modifications to the scene

Where modifications to the scene graph, resulting from the use of more than one of the permitted methods, must be applied simultaneously, the following order of application shall be observed:

1. BIFS-Anim
2. **Conditional** node
3. BIFS-Command

9.3 BIFS Syntax

9.3.1 Introduction

BIFS data consists of two distinct elements in the multiplexed bitstream. Terminal configuration information is first sent in the object descriptor. The remaining BIFS information is sent in a separate elementary stream.

The syntax and semantics of the terminal configuration is described in 9.3.5.2. Two different kinds of session can take place: a BIFS-Command session or a BIFS-Anim session.

If the session is a BIFS-Command session, a sequence of commands to modify the scene is sent. The syntax and semantics of these commands are described in 9.3.6.

If the session is a BIFS-Anim session, a sequence of animation data to change the values of specific fields in the scene is sent. The syntax and semantics of this session is described in 9.3.8.

9.3.2 Decoding tables, data structures and associated functions

9.3.2.1 Function of decoding tables, data structures and functions

This subclause describes tables and data structures used to contain necessary data, along with the associated functions, for decoding the BIFS elementary streams. These are not syntax elements but are descriptions, often in code or pseudo-code, of data and functions that are required to decode the bitstream. The tables and data structures may be known a priori at the terminal or may be constructed from data parsed from the bitstream. They are referenced throughout the syntax.

NOTE — The code or pseudo-code for the non-syntax data elements is purely notational and does not imply a normative requirement to use these code fragments in implementations.

Coding of individual nodes and field values is very regular, and follows a depth-first order (children or sub-nodes of a node are present in the bitstream before its siblings).

9.3.2.2 Node Data Type Tables

Identification of nodes and fields within a BIFS scene graph is context-dependent. Each field of a BIFS node that accepts nodes as fields can only accept a specific set of nodes. Each of these sets of nodes is stored in a node data type table and is referenced by a node data type (NDT).

A field of type SFNode is fully described by its NDT. Each node belongs to one or more NDT tables. These tables are provided in Annex H and identify the various nodes and node types they contain.

Identification of a particular node depends on the context of the NDT specified for its parent field. The value 0 is always reserved for future extensions.

EXAMPLE — **Anchor** is identified by the 5-bit code 0b0000.1 when the context of its parent's field is SF2DNode, whereas the 7-bit code 0b0000.001 is used when the context of its parent's field is SFWorldNode.

9.3.2.3 Node Coding Tables and field indexing

The syntactic description of fields is context-dependent. For a given node, its fields are indexed using a code called a `fieldID`. This `fieldID` is not unique for each field of a node but varies according to the "mode" in which the field is referenced. There are five modes in which a field may be referenced and, thus, five types of `fieldID`. For each field of each node, the binary values of the `fieldID`s for each mode are defined in the node coding tables.

`defID`
The `defID`s refer to the `fieldID`s for those fields that may have a value when nodes are declared. They refer to fields of type exposedField and field. This indexing scheme is further referred to as the "def" mode.

`inID`
The `inID`s refer to the `fieldID`s for those events and fields that can be modified from outside the node. They refer to fields of type exposedField and eventIn types. This indexing scheme is further referred to as the "in" mode.

`outID`
The `outID`s refer to the `fieldID`s for those events and fields that can be output from the node. They refer to fields of type exposedField and eventOut types. This indexing scheme is further referred to as the "out" mode.

`dynID`
The `dynID`s refer to the `fieldID`s for those fields that can be animated using the BIFS-Anim scheme. They refer to a subset of the fields designated by `inID`s. This indexing scheme is further referred to as the "dyn" mode.

`allID`
The `allID`s refer to all events and fields of the node. That is, there is an `allID` for each field of a node. This indexing scheme is further referred to as the "all" mode.

The length of each of the `fieldID` types for each node depends on the number of fields of that type for the given node.

EXAMPLE — The **AnimationStream** node has four fields of type defID. Therefore, three bits are required to code the defIDs for this node. The **Appearance node**, however, has just three fields of type defID. Therefore, two bits are sufficient to code the defIDs for this node.

9.3.2.4 BIFSConfig

This data structure is a global data structure referred to in every BIFS access unit. The data contained in the `BIFSConfig` data structure is transmitted in the syntax element of the same name (see 9.3.5.2).

```
class BIFSConfig extends DecoderSpecificInfo :
bit(8) tag=DecSpecificInfoTag{
   int nodeIDbits;              The number of bits used to encode the nodeIDs.
   int routeIDbit;              The number of bits used to encode the routeIDs.
   boolean randomAccess;        The randomAccess boolean is set in the
                                BIFSConfig to distinguish between BIFS-Anim
                                elementary streams in which support random access
                                at any intra frame, and those where random access
                                may not be possible at all intra frames. In the latter
                                case, greater compression efficiency may be achieved
                                because a given intra frame may re-use quantization
                                settings and statistics from the previous intra frame.
   AnimationMask animMask;      The AnimationMask used for BIFS-Anim
}
```

9.3.2.5 AnimationMask

The `AnimationMask` structure contains all the relevant information to describe a BIFS-Anim session. It is constructed, upon receipt of the `BIFSConfig` syntax element, during the configuration of the BIFS decoder, and updated for every received `AnimationFrame`.

```
Class AnimationMask {
   int numNodes;                   The number of nodes to be animated
   NodeData animNode[numNodes];    The array of animated nodes.
   boolean isIntra;                The status of the current frame: intra if isIntra is true,
                                   predictive otherwise.
   boolean isActive[numNodes];     The mask of active animated node for the current frame.
                                   If the node is not animated in the current frame, the
                                   boolean shall be false.
}
```

9.3.2.6 NodeData

This data structure is built to decode the relevant information for one node. It is created from the node coding tables in Annex H. The following functions support relevant operations on this data structure:

```
NodeData MakeNode(int nodeType)
```

This function creates a `NodeData` structure from the node coding table matching the given `nodeType`.

```
NodeData GetNodeFromID (int nodeID)
```

This function returns the `NodeData` structure matching the given `nodeID`.

```
class NodeData {
   int nodeType;              The nodeType of the node.
   FieldData field[];         The fields of this node whose construction is described below.
                              This array is indexed in "all" mode.
   boolean isAnimField[];     The mask of animated fields for the entire BIFS-Anim session,
                              indexed in "dyn" mode. This array is only used in BIFS-Anim.
```

	The following data describes the indexing of the fields in “in”, “out”, “def”, “dyn” and “all” modes
`int nDEFbits;`	The number of bits used for “def” field codes (the width of the codewords in the 2nd column of the node coding tables).
`int nINbits;`	The number of bits used for “in” field codes (the width of the codewords in the 3rd column of the node coding tables).
`int nOUTbits;`	The number of bits used for “out” field codes (the width of the codewords in the 4th column of the node coding tables).
`int numDEFfields;`	The number of “def” fields available for this node
`int numDYNfields;`	The number of “dyn” fields available for this node.
`int in2all[];`	The ids of eventIns and exposedFields in “all” mode, indexed with the ids in “in” mode.
`int def2all[];`	The ids of fields and exposedFields in “all” mode, indexed with the ids in “def” mode.
`int dyn2all[];`	The ids of dynamic fields in “all” mode, indexed with the ids in “dyn” mode.

```
}
```

9.3.2.7 FieldData

This data structure is built to decode the relevant information for one field. It is created from the field’s entry in the relevant node coding table (see Annex H).

```
Class FieldData {
```

`int fieldType;`	The type of the field (e.g., `SFInt32Type`). This is given by the “Field Type” column of the node coding table for the node to which it belongs.
`int quantType;`	The type of quantization used for the field. This is given by the “Q” column of the node coding table of the node to which it belongs. Types refer to Table 17 in 9.3.3.1.1.
`int animType;`	The animation method for the field. This is given by the “A” column of the node coding table. Types refer to animation type in Table 23 in 9.3.3.2.1.
`boolean useEfficientCoding;`	Set to true if the efficient coding is to be used. This value is FALSE by default. If there is a local **QuantizationParameter** node this value is the same as its **useEfficientCoding** field.
	The following data structures are used in the quantization process:
`FieldCodingTable      fct;`	This field is determined from the node coding table as described in 9.3.2.9.
`AnimFieldQP           aqp;`	This field is only used in BIFS-Anim. It references an `AnimFieldQP` stucture described in 9.3.2.10.
`QuantizationParameter lqp;`	This field points to the local **QuantizationParameter** node.
`boolean isQuantized;`	Set to true if the corresponding field is quantized, false otherwise.
`int nbBits;`	The number of bits used for the quantization of the field.
`float floatMin[];`	The minimum bounds for the quantization of vector fields. These values are obtained from the `FieldCodingTable` (described in 9.3.2.9) and the current **QuantizationParameter** node (for BIFS-Scene) or the `animField` (for BIFS-Anim).
`float floatMax[];`	The maximum bounds for the quantization of vector fields. These values are obtained from the `FieldCodingTable`

```
                                   (described in 9.3.2.9) and the current
                                   QuantizationParameter node (for BIFS-Scene) or the
                                   animField (for BIFS-Anim).
   int intMin[];                   The minimum bounds for integers (SFInt32 and MFInt32).
                                   These values are obtained from the FieldCodingTable
                                   (described in 9.3.2.9) and the current
                                   QuantizationParameter node (for BIFS-Scene) or the
                                   animField (for BIFS-Anim).
}
```

It is assumed that the following functions are available:

`int isSF(FieldData field)`
Returns 1 if the field's `fieldType` corresponds to a single field and 0 otherwise.

`int getNbComp(FieldData field)`
Returns the number of quantized components for the field as given below:

Table 15 – Return values of `getNbComp`

fieldType	quantType	animType	value returned
SFFloat SFInt32	any	6,7,8 13	1
SFVec2f SFVec3f	any 9	2,12 9	2
SFVec3f SFRotation	!=9 any	1,4,11 10	3

The number of quantized components is the same as the natural number of components (three for SFVec3f, two for SFVec2f, and so on) except for normals (2) and rotations (3) because of the quantization process (see 9.3.3.3).

9.3.2.8 Node Data Type Table Parameters

The following functions provide access to the node data type tables (described in Annex H):

`int GetNodeType(int nodeDataType, int localNodeType)`
Returns the `nodeType` of the node indexed by `localNodeType` in the node data type table. The `nodeType` of a node is its index in the `SFWorldNode` NDT Table.

`int GetNDTnbBits(int nodeDataType)`
Returns the number of bits used to index the nodes of the matching node data type table (this number is indicated in the last column of the first row of the node data type table).

`int GetNDTFromID(int id)`
Returns the `nodeDataType` for the **children** field of the node identified by the `nodeID`, `id`. Nodes having a **children** field may have restrictions on the types of node that may occupy the field. These node types are indicated in the node semantics (see 9.4 and ISO/IEC 14772-1:1998 , Table 4.3 [10]).

9.3.2.9 Field Coding Table

This data structure contains parameters relating to the quantization of the field. It is created from the field's entry in the relevant node coding table (Annex H).

```
Class FieldCodingTable {
   float floatMin[];          The minimum default bounds for fields of type SFFloat, SFVec2f and
                              SFVec3f. These values are obtained from the "[m, M]" column of the
                              node coding table.
   float floatMax[];          The minimum default bounds for fields of type SFFloat, SFVec2f and
                              SFVec3f. These values are obtained from the "[m, M]" column of the
                              node coding table.
```

`float intMin[];`	The minimum default bounds for fields of type SFInt32. These values are obtained from the “[m, M]” column of the node coding table.
`float intMax[];`	The minimum default bounds for fields of type SFInt32. These values are obtained from the “[m, M]” column of the node coding table.
`int defaultNbBits;`	The number of bits used by default for each field. Only used when the quantization category of the field is 13. For quantization category 13, the number of bits used for coding is also specified in the node coding (e.g “13 16” in the node coding table means category 13 with 16 bits).

```
}
```

9.3.2.10 AnimFieldQP

This data structure contains the necessary quantization parameters and information for the animation of a field. It is updated throughout the BIFS-Anim session.

```
class AnimFieldQP {
```

`int animType;`	The animation method for the field. This is given by the “A” column of the node coding table for each node. Types refer to animation type in Table 23 in 9.3.3.2.1.
`boolean useDefault;`	If this bit is set to TRUE, then the bounds used in intra mode are those specified in the “[m, M]” column of the node coding table. The default value is FALSE.
`boolean isTotal;`	If the field is a multiple field and if this boolean is set to TRUE, all the components of the multiple field are animated.
`int numElement;`	The number of elements being animated in the field. This is 1 for all single fields, and equal to or greater than 1 for multiple fields.
`int indexList[];`	If the field is a multiple field and if `isTotal` is false, this is the list of the indices of the animated `SFFields`. For instance, if the field is an `MFField` with elements 3,4 and 7 being animated, the valuse of `indexList` will be {3,4,7}.
`float[] Imin;`	The minimum values for bounds of the field in intra mode. This value is obtained from the “[m, M]” column of the node coding table (if `useDefault` is TRUE), the `InitialAnimQP` (if `useDefault` is FALSE and the last intra did not hold any new `AnimQP`), or the `AnimQP`.
`float[] Imax;`	The maximum values for bounds of the field in intra mode. This value is obtained from the “[m, M]” column of the semantics table (if `useDefault` is TRUE), the `InitialAnimQP` (if `useDefault` is FALSE and if the last intra did not hold any new `AnimQP`), or the `AnimQP`.
`int[] IminInt;`	The minimum value for bounds of variations of integer fields in intra mode. This value is obtained from the `InitialAnimQP` (if the last intra did not hold any new `AnimQP`) or `AnimQP` structure.
`int[] Pmin;`	The minimum value for bounds of variations of the field in predictive mode. This value is obtained from the `InitialAnimQP` (if the last intra did not hold any new `AnimQP`) or `AnimQP`.
`int INbBits;`	The number of bits used in intra mode for the field. This value is obtained from the `InitialAnimQP` or `AnimQP`.
`int PNbBits;`	The number of bits used in predictive mode for the field. This value is obtained from the `InitialAnimQP` (if the last intra did not hold any new `AnimQP`) or `AnimQP` structure.

```
}
```

It is assumed that the following function is available :

```
int getNbBounds(AnimFieldQP aqp)
```
Returns the number of set of bounds matching the animation type (see 9.3.2.3), as follows :

Table 16 - Return values of getNbBounds

aqp.animType	value returned

`4,6,7,8`
`9,10`
`11,12,13`
	`1`

`2`	`2`
`1`	`3`

Note that only `Position2D` and `Position3D` have specific sets of bounds for each of their components. The number of bounds is also the number of independent models used in predictive mode during the BIFS-Anim session.

9.3.3 Quantization

In BIFS scenes, the values of the fields may be quantized. BIFS-Anim data is always quantized. This subclause describes this quantization process. A number of parameters control the quantization of a field. Here, these parameters are used to construct a notational data structure called `FieldData`. In this subclause, the semantics of how to determine these parameters for BIFS scenes and BIFS-Anim are first described, followed by a description of the actual quantization process.

9.3.3.1 Quantization of BIFS scenes

9.3.3.1.1 Quantization categories

Single fields are coded according to the type of the field. The fields have a default syntax that specifies a non-quantized encoding. When quantization is used, the quantization parameters are obtained from a special node called **QuantizationParameter**. The following quantization categories are specified, providing suitable quantization procedures for the various types of quantities represented by the various fields of the BIFS nodes.

Table 17 - Quantization Categories

Category	Description
0	None
1	3D position
2	2D positions
3	Drawing order
4	SFColor
5	Texture Coordinate
6	Angle
7	Scale
8	Interpolator keys
9	Normals
10	Rotations
11	Object Size 3D (1)
12	Object Size 2D (2)
13	Linear Scalar Quantization
14	CoordIndex
15	Reserved

Each field that may be quantized is assigned to one of the quantization categories (see Annex H). Along with quantization parameters, minimum and maximum values are specified for each field of each node.

9.3.3.1.2 Determining the quantization parameters for a given field

The scope of quantization is constrained to a single BIFS access unit. A field is quantized when:

— The field is of type SFInt32, SFFloat, SFRotation, SFVec2f or SFVec3f.

— The quantization category of the field is not 0.

— The node to which the field belongs has a **QuantizationParameter** (see 9.4.2.77) node in its context

— The quantization for this type of field is activated (by setting the corresponding boolean to TRUE in the **QuantizationParameter** node.

The `isQuantized`, `nbBits`, `floatMin`, `floatMax` and `intMin` fields of the `FieldData` structure pertain to the quantization of the field. The values of these fields are determined from the local **QuantizationParameter** (`lqp`) and the `FieldCodingTable` (`fct`) stored in the `FieldData`. This is done in the following way:

`isQuantized`

`isQuantized` is set to true when the three following conditions are met :

— `lqp!=0` (there is a **QuantizationParameter** node in the scope of the field)

— `quantType !=0` (the field value is of a type that may be quantized), and

— the following condition is met for the relevant quantization type:

Table 18 - Condition for setting `isQuantized` to true

quantType	Condition
1	lqp.position3DQuant == TRUE
2	lqp.position2DQuant == TRUE
3	lqp.drawOrderQuant == TRUE
4	lqp.colorQuant == TRUE
5	lqp.textureCoordinateQuant == TRUE
6	lqp.angleQuant == TRUE
7	lqp.scaleQuant == TRUE
8	lqp.keyQuant == TRUE
9	lqp.normalQuant == TRUE
10	lqp.normalQuant == TRUE
11	lqp.sizeQuant == TRUE
12	lqp.sizeQuant == TRUE
13	Always TRUE
14	Always TRUE
15	Always TRUE

`nbBits`

In the BIFS scene quantization process, `nbBits` is set in the following way :

Table 19 - Value of `nbBits` depending on `quantType`

`quantType`	`nbBits`
1	`lqp.position3DNbBits`
2	`lqp.position2DNbBits`
3	`lqp.drawOrderNbBits`
4	`lqp.colorNbBits`
5	`lqp.textureCoordinateNbBits`
6	`lqp.angleNbBits`
7	`lqp.scaleNbBits`
8	`lqp.keyNbBits`
9,10	`lqp.normalNbBits`
11,12	`lqp.sizeNbBits`
13	`fct.defaultNbBits`
14	This value is set according to the number of points received in the last received coord field of the node. Let N that number, then: $\text{nbBits} = \text{Ceil}(\log_2(N))$ where the function Ceil returns the smallest integer greater than its argument
15	0

`floatMin[]`

In the BIFS scene quantization process, `floatMin` is set in the following way:

Table 20 - Value of `floatMin`, depending on `quantType` and `fieldType`

`quantType`	`fieldType`	`floatMin`
1	SFVec3fType	`lqp.position3Dmin`
2	SFVec2fType	`lqp.position2Dmin`
3	SFFloatType	`max(fct.min[0],lqp.drawOrderMin)`
4	SFFloatType	`lqp.colorMin`
	SFColorType	`lqp.colorMin, lqp.colorMin, lqp.colorMin`
5	SFVec2fType	`lqp.textureCoordinateMin`
6	SFFloatType	`Max(fct.min[0],lqp.angleMin)`
7	SFFloatType	`lqp.scaleMin`
	SFVec2fType	`lqp.scaleMin, lqp.scaleMin`
	SFVec3fType	`lqp.scaleMin, lqp.scaleMin, lqp.scaleMin`
8	SFFloatType	`Max(fct.min[0],lqp.keyMin)`
9	SFVec3fType	0.0
10	SFRotationTyp e	0.0
11,12	SFFloatType	`lqp.sizeMin`
	SFVec2fType	`lqp.sizeMin, lqp.sizeMin`
	SFVec3fType	`lqp.sizeMin, lqp.sizeMin, lqp.sizeMin`
13,14,15		NULL

`floatMax[]`

In the BIFS scene quantization process, `floatMax` is set in the following way:

Table 21 - Value of `floatMax`, depending on `quantType` and `fieldType`

quantType	**fieldType**	**floatMax**
1	SFVec3fType	lqp.position3Dmax
2	SFVec2fType	lqp.position2Dmax
3	SFFloatType	min(fct.max[0],lqp.drawOrderMax)
4	SFFloatType	lqp.colorMax
	SFColorType	lqp.colorMax, lqp.colorMax, lqp.colorMax
5	SFVec2fType	lqp.textureCoordinateMax
6	SFFloatType	min(fct.max[0],lqp.angleMax)
7	SFFloatType	lqp.scaleMax
	SFVec2fType	lqp.scaleMax, lqp.scaleMax
	SFVec3fType	lqp.scaleMax, lqp.scaleMax, lqp.scaleMax
8	SFFloatType	min(fct.max[0],lqp.keyMax)
9	SFVec3fType	1.0
10	SFRotationType	1.0
11,12	SFFloatType	lqp.sizeMax
	SFVec2fType	lqp.sizeMax, lqp.sizeMax
	SFVec3fType	lqp.sizeMax, lqp.sizeMax, lqp.sizeMax
13,14,15		NULL

`intMin[]`

In the BIFS scene quantization process, `intMin` is set in the following way:

Table 22 - Value of `intMin`, depending on `quantType`

quantType	**intMin**
1,2,3,4,5,6,7,8 9,10,11,12	NULL
13,14	fct.intMin[0]
15	NULL

9.3.3.2 Quantization of BIFS-Anim

9.3.3.2.1 Animation Categories

The fields are grouped in the following categories for animation:

Table 23 – Animation Categories

Category	Description
0	None
1	Position 3D
2	Positions 2D
3	Reserved
4	Color
5	Reserved
6	Angle
7	Float
8	BoundFloat
9	Normals
10	Rotation
11	Size 3D
12	Size 2D
13	Integer
14	Reserved
15	Reserved

9.3.3.2.2 Determining the quantization parameters for a given field

The `isQuantized`, `nbBits, floatMin, floatMax and intMin` fields of the `FieldData` structure pertain to the quantization of the field. The values of these fields are determined from the local `AnimFieldQP` (`aqp`) and the `FieldCodingTable` (`fct`) stored in the `FieldData`. This is done in the following way:

`isQuantized`

In the BIFS-Anim quantization process, `isQuantized` is always TRUE.

`nbBits`

In the BIFS-Anim quantization process, `nbBits` is set in the following way :

Table 24 - Value of `nbBits`, depending on `animType`

`animType`	`nbBits`

`1,2,4,6,7,8,9`
`10,11,12,13`
	`animType.INbBits`

`floatMin[]`

In the BIFS-Anim quantization process, `floatMin` is set in the following way:

Table 25 - Value of `floatMin`, depending on `animType`

animType		**aqp.useDefault**	**floatMin**
4	Color	true	fct.min[0], fct.min[0], fct.min[0]
		false	aqp.IMin[0], aqp.IMin[0], aqp.IMin[0]
8	BoundFloat	true	fct.min[0]
		false	aqp.IMin[0]
1	Position 3D	false	aqp.IMin
2	Position 2D	false	aqp.IMin
11	Size 3D	false	aqp.IMin[0], aqp.IMin[0]
12	Size 2D	false	aqp.IMin[0], aqp.IMin[0], aqp.IMin[0]
7	Float	false	aqp.IMin[0]
6 9 10	Angle Normal Rotation	false	0.0
13	Integer	false	NULL
14,15	Reseved		NULL

`floatMax[]`

In the BIFS-Anim quantization process, `floatMax` is set in the following way:

Table 26 - Value of `floatMax`, depending on `animType`

animType		**aqp.useDefault**	**floatMax**
4	Color	true	fct.max[0], fct.max[0], fct.max[0]
		false	aqp.IMax[0], aqp.IMax[0], aqp.IMax[0]
8	BoundFloat	true	fct.max[0]
		false	aqp.IMax[0]
1	Position 3D	false	aqp.IMax
2	Position 2D	false	aqp.IMax
11	Size 3D	false	aqp.IMax[0], aqp.IMax[0]
12	Size 2D	false	aqp.IMax[0], aqp.IMax[0], aqp.IMax[0]
7	Float	false	aqp.IMax[0]
6 9 10	Angle Normal Rotation	false false	2*Pi 1.0
13	Integer	false	NULL
14,15	Reseved		NULL

`intMin[]`

In the BIFS-Anim quantization process, `intMax` is set in the following way:

Table 27 - Value of `intMin`, depending on `animType`

`animType`	`intMin`

`1,2,4,6,7,8`
`9,10,11,12`
	`NULL`

`13`	`aqp.IminInt[0]`
`14,15`	`NULL`

9.3.3.3 Quantization process

Let $v_q(t)$ be the value decoded from the bitstream at an instant *t*. Then, the inverse-quantized value at time *t* is:

$$v(t) = \mathrm{InvQuant}\left(v_q(t)\right)$$

The linear quantization and inverse quantization are:

```
int quantize (float Vmin, float Vmax, float v, int Nb)
```

which returns $$v_q = \frac{v - V_{\min}}{V_{\max} - V_{\min}}(2^{Nb} - 1)$$

```
float invQuantize (float Vmin,float Vmax,int vq, int Nb)
```

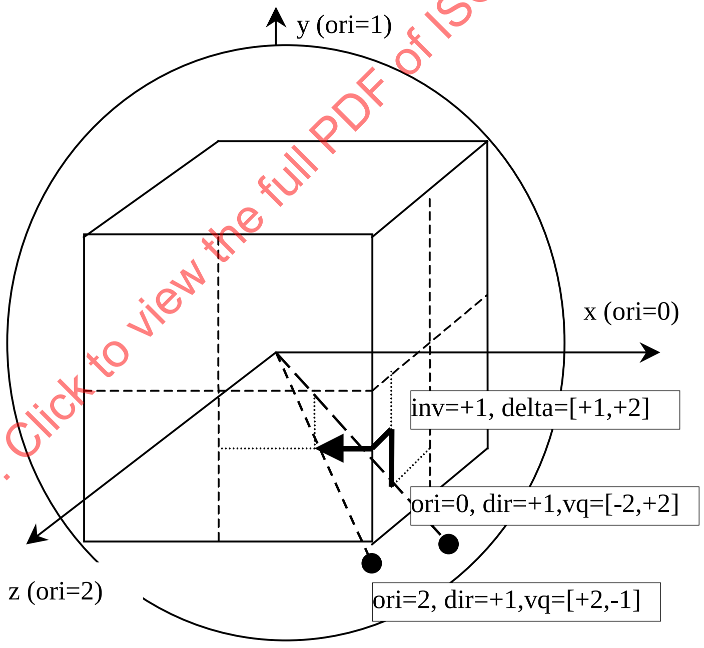
which returns $$\hat{v} = V_{\min} + v_q \frac{V_{\max} - V_{\min}}{2^{\max(Nb,1)} - 1}$$

If `isQuantized` is true, the quantization/inverse quantization process is the following :

Table 28 - Quantization and inverse quantization process

`quantType`	`animType`	**Quantization/Inverse Quantization Process**
1,2,3,4,5 6,7,8 11,12	1,2,4 6,7,8 11,12	For each component of the vector, the float quantization is applied: $v_q[i] = \mathrm{quantize}(\mathrm{floatMin}[i], \mathrm{floatMax}[i], \mathrm{v}[i], \mathrm{nbBits})$ For the inverse quantization: $\hat{v}[i] = \mathrm{invQuantize}(\mathrm{floatMin}[i], \mathrm{floatMax}[i], \mathrm{v}_q[i], \mathrm{nbBits})$
9,10	9,10	For normals and rotations, the quantization method is as follows. Normals are first renormalized : $v[0] = \frac{n_x}{\sqrt{{n_x}^2 + {n_u}^2 + {n_z}^2}}$, $v[1] = \frac{n_y}{\sqrt{{n_x}^2 + {n_u}^2 + {n_z}^2}}$, $v[2] = \frac{n_z}{\sqrt{{n_x}^2 + {n_u}^2 + {n_z}^2}}$ Rotations (axis $\vec{n}$, angle α) are first written as quaternions : $v[0] = \cos(\frac{\alpha}{2})$ $v[1] = \frac{n_x}{\Vert\vec{n}\Vert}.\sin(\frac{\alpha}{2})$ $v[2] = \frac{n_y}{\Vert\vec{n}\Vert}.\sin(\frac{\alpha}{2})$ $v[3] = \frac{n_z}{\Vert\vec{n}\Vert}.\sin($ The number of reduced components is defined to be N: 2 for normals, and 3 for rotations. Note that v is then of dimension N+1. The compression and quantization process is the same for both :

quantType	animType	Quantization/Inverse Quantization Process
		The orientation k of the unit vector v is determined by the largest component in absolute value: $k = \text{argMax}(\lvert v[i]\rvert)$. This is an integer between 0 and N that is encoded using two bits. The direction of the unit vector v is 1 or –1 and is determined by the sign of the component $v[k]$.Note that this value is not written for rotations (because of the properties of quaternions). The *N* components of the compressed vector are computed by mapping the square on the unit sphere $\left\{ v \;\middle\vert\; 0 \le \frac{v[i]}{v[k]} \le 1 \right\}$ into a N dimensional square : $$v_c[i] = \frac{4}{\pi}\tan^{-1}\left(\frac{v[(i+k+1)\,\text{mod}(N+1)]}{v[k]}\right) \quad i = 0,\ldots,N$$ If nbBits=0, the process is complete. Otherwise, each component of v_c (which lies between –1 and 1) is quantized as a signed integer as follows : $v_q[i] = 2^{\text{nbBits}-1} + \text{quantize}(\text{floatMin}[0], \text{floatMax}[0], v_c[i], \text{nbBits}-1)$ The value encoded in the bitstream is $2^{.\text{nbBits}-1} + v_q[i]$ The decoding process is the following : The value decoded from the stream is converted to a signed value $v_q[i] = v_{decoded} - 2^{.\text{nbBits}-1}$ The inverse quantization is performed $v_c[i] = \text{invQuantize}(\text{floatMin}[0], \text{floatMin}[0], v_q[i], \text{nbBits}-1)$ After extracting the orientation (k) and direction (dir) , the inverse mapping can be performed : $$\hat{v}[k] = \text{dir}.\frac{1}{\sqrt{1 + \sum_{i=0}^{i<N} \tan^2 \frac{\pi.v_c[i]}{4}}}$$

quantType	animType	Quantization/Inverse Quantization Process
		$$\hat{v}[(i+k+1)\bmod(N+1)] = \tan\left(\frac{\pi . v_c[i]}{4}\right) . \hat{v}[k] \quad i = 0,\ldots,N$$ If the object is a rotation, v can be either used directly or converted back from a quaternion to a SFRotation : $$\alpha = 2.\cos^{-1}(\hat{v}[0]) \quad n_x = \frac{\hat{v}[1]}{\sin(\alpha/2)} \quad n_y = \frac{\hat{v}[2]}{\sin(\alpha/2)} \quad n_z = \frac{\hat{v}[3]}{\sin(\alpha/2)}$$ The entire compression process therefore consists in projecting a vector of the unit sphere onto the face of a cube inscribed inside the sphere, and transmitting separately the face's index (orientation: x, y or z – and direction : + or -) and the coordinates on the face. EXAMPLE — How two different normals are encoded in the case nbBits=3. The compensation process (described in 9.3.4) is also illustrated.  y (ori=1) x (ori=0) z (ori=2) inv=+1, delta=[+1,+2] ori=0, dir=+1,vq=[-2,+2] ori=2, dir=+1,vq=[+2,-1] Note that two quaternions that lie in opposite directions on the unit sphere actually represent the same rotation. This is the reason why the direction is not coded for rotations.
13,14	13	For integers, the quantized value is the integer shifted to fit the interval [0, 2^{nbBits} -1]. $$v_q = v - \text{intMin}$$ The inverse quantization process in then : $$\hat{v} = \text{intMin} + v_q$$

`quantType`	`animType`	**Quantization/Inverse Quantization Process**
fieldType SFImage		For SFImage types, the width and height of the image are sent. **`numComponents`** defines the image type. The following four types are enabled: If the value is ‘00’, then a grey scale image is defined. If the value is ‘01’, a grey scale with alpha channel is used. If the value is ‘10’, then an RGB image is used. If the value is ‘11’, then an RGB image with alpha channel is used.

9.3.4 Compensation process

This subclause describes the mechanism used to compensate a quantized value for a given `FieldData` structure. In other words, how to add a delta to a quantized value to yield the result of addition, which is another quantized value. For vectorial types, this is simply an addition component by component, but for normals and rotations special care has to be taken when performing this addition. This process is used in predictive mode in BIFS-Anim sessions.

Let v_q^1 be the initial quantized value, v^δ be the delta value and v_q^2 be the quantized value resulting from the addition. The general inverse compensation process is :

$$v_q^2 = \mathrm{AddDelta}(v_q^1, v^\delta)$$

v_q^1 and v^δ are interpreted as follows:

A quantized value v_q contains an array of integers vq[]. Additionally, for normals and rotations, v_q^1 contains an orientation and, for normals only, a direction (see 9.3.3.3).

A delta value v^δ contains an array of integers vDelta[]. Additionnally, for normals, it contains an integer inverse whose value is –1 or 1.

The size of these arrays is that returned by the function `getNbComp(field)`, as described in 9.3.2.7.

The result v_q^2 is then computed in the following way :

Table 29 - Compensation process

animType	**Compensation Process**		
1,2,4,6,7,8 11,12,13	The components of v_q^2 are: `vq2[i] = vq1[i] + vDelta[i]`		
9,10	The addition is first performed component by component and stored in a temporary array: `vqTemp[i] = vq1[i] + vDelta[i].` Let `scale` = $2^{\max(0,nbBits-1)} - 1$. Let `N` the number of reduced components (2 for normals, 3 for rotations) There are then three cases are to be considered:		

	For every index I,
	$\lvert vqTemp[i] \rvert \le scale$
		v_q^2 is defined by,
		`vq2[i]       = vqTemp[i]`
		`orientation2= orientation1`
		`direction2   = direction1 * inverse`	

	There is one and only one index k such that $\lvert vqTemp[k] \rvert > scale$	v_q^2 is rescaled as if gliding on the faces of the mapping cube. Let inv = 1 if `vqTemp[k]>=0` and –1 else Let `dOri` = k+1 The components of vq2 are computed as follows	
		$0 \le i < N - dOri$	`vq2[i] = inv*vqTemp[(i+dOri) mod N]`
		$i = N - dOri$	`vq2[i] = inv*2*scale–vqTemp[dOri–1]`
		$N - dOri < i < N$	`vq2[i] = inv*vqTemp[(i+dOri-1) mod N]`

		`orientation2 = (orientation1 + dOri) mod (N+1)`
		`direction2   = direction1 * inverse * inv`	

	There are several indices k such that $\lvert vqTemp[k] \rvert > scale$	The result is undefined	

9.3.5 BIFS Configuration

9.3.5.1 Overview

This subclause describes the terminal configuration for the BIFS elementary stream. It is encapsulated within the `specificInfo` fields of the general `DecoderSpecificInfo` structure (see 8.6.6), which is contained in the `DecoderConfigDescriptor` that is carried in `ES_Descriptors`. If the session is a BIFS-Anim session, the BIFS configuration contains some specific information to describe the animation mask, which specifies the elements of the scene to be animated.

9.3.5.2 BIFSConfig

9.3.5.2.1 Syntax

```
class BIFSConfig extends DecoderSpecificInfo : bit(8) tag=DecSpecificInfoTag {
   unsigned int(5) nodeIDbits;
   unsigned int(5) routeIDbits;
   bit(1) isCommandStream;
   if(isCommandStream) {
      bit(1) pixelMetric;
      bit(1) hasSize;
      if(hasSize) {
         unsigned int(16) pixelWidth;
         unsigned int(16) pixelHeight;
      }
   }
   else {
      bit(1)        randomAccess;
      AnimationMask   animMask();
   }
}
```

9.3.5.2.2 Semantics

BIFSConfig is the terminal configuration for the BIFS elementary stream. It is encapsulated within the specificInfo fields of the general DecoderSpecificInfo structure (see 8.6.6), which is contained in the DecoderConfigDescriptor that is carried in ES_Descriptors.

The parameter nodeIDbits sets the number of bits used to represent nodeIDs. Similarly, routeIDbits sets the number of bits used to represent ROUTEIDs.

The boolean isCommandStream identifies whether the BIFS stream is a BIFS-Command stream or a BIFS-Anim stream. If the BIFS-Command stream is selected (isCommandStream set to TRUE), the following parameters are contained in BIFSConfig:

— The boolean isPixelMetric indicates whether pixel metrics or meter metrics are used.

— The boolean hasSize indicates whether a desired scene size (in pixels) is specified. If hasSize is set to true, pixelWidth and pixelHeight provide to the receiving terminal the desired horizontal and vertical dimensions (in pixels) of the scene.

If isCommandStream is false, the following information is contained in BIFSConfig:

— The randomAccess boolean signals the mode of the BIFS-Anim stream. If the bit is set to TRUE, it is possible to perform random access in the BIFS-Anim stream at any intra frame. At each intra frame, the statistics of the arithmetic decoder shall be reset. New quantization parameters shall be coded in the bistream or the default parameters sent in the BIFS-Anim mask are used. If the randomAccess bit is set to FALSE, compression may be more efficient, but random access may not be possible at each intra frame. See 9.3.8 for detailed semantics.

— The AnimationMask specifies the animation parameters of the BIFS-Anim elementary stream.

9.3.5.3 AnimationMask

9.3.5.3.1 Syntax

```
class AnimationMask() {
   int numNodes = 0;
   do {
      ElementaryMask elemMask();
      numNodes++;
        bit(1) moreMasks;
   } while (moreMasks);
}
```

9.3.5.3.2 Semantics

The `AnimationMask` describes the nodes and fields to be animated, along with the quantization parameters to help decode their values. It consists of a list of `ElementaryMasks`.

If the boolean `moreMasks` is TRUE, another `ElementaryMask` shall be present.

9.3.5.4 Elementary mask

9.3.5.4.1 Syntax

```
Class ElementaryMask() {
   bit(BIFSConfig.nodeIDbits) nodeID;
      NodeUpdateField node = GetNodeFromID(nodeID);
   switch (node.nodeType) {
      case FaceType:
      break;
      case BodyType:
      break;
      case IndexedFaceSet2DType:
      break;
   default:
      InitialFieldsMask initMask(node);
   }
}
```

9.3.5.4.2 Semantics

The `ElementaryMask` describes how to animate the elements of a node.

The integer `nodeID` identifies the animated node.

If the node's `nodeType` is **FDP**, **BDP** or **IndexedFaceSet2D**, no further information is expected.

NOTE — The **BDP** node ("`case BodyType`") is not specified in ISO/IEC 14496 nor in ISO/IEC 14772-1:1998 [10]. This syntax switch has been provided to allow support for future extensions.

If any other case, an `InitialFieldsMask` shall be present.

9.3.5.5 InitialFieldsMask

9.3.5.5.1 Syntax

```
class InitialFieldsMask(NodeUpdateField node) {
   for(i=0; i<node.numDYNfields; i++)
      bit(1) node.isAnimField[i];
   int i;
   for(i=0; i<node.numDYNfields; i++) {
      if (node.isAnimField[i]) {
         FieldData field = node.field[node.dyn2all[i]];
         AnimFieldQP aqp = field.aqp;
         if (!isSF(field) {
            bit(1) aqp.isTotal;
            if (!aqp.isTotal) {
               unsigned int(5) nbBits;
               do {
                  int(nbBits) aqp.indexList[aqp.numElement++];
                  bit(1) moreIndices;
               } while (moreIndices);
         }
         InitialAnimQP QP[i](field.aqp);
      }
   }
}
```

9.3.5.5.2 Semantics

The InitialFieldsMask specifies which fields of a given node are animated.

The array of booleans isAnimField describes whether the fields (indexed with dynIDs) are animated.

If a multiple field is animated and if the boolean isTotal is TRUE, all the of the field's individual elements are animated.

If a multiple field is animated and if the boolean isTotal is FALSE, the indices of the animated individual field are sent and stored in aqp.indexList[]. The number of bits used to encode them is specified by nbBits. If the boolean moreIndices is TRUE, another index shall be present.

An InitialAnimQP shall then be expected.

9.3.5.6 InitialAnimQP

9.3.5.6.1 Syntax

```
InitialAnimQP(animFieldQP aqp) {

   aqp.useDefault=FALSE;
   switch(aqp.animType) {

         case  4:           // Color
         case  8:           // BoundFloats
            bit(1)     aqp.useDefault
         case  1:           // Position 3D
         case  2:           // Position 2D
         case 11:           // Size 3D
         case 12:           // Size 2D
         case  7:           // Floats
            if (!aqp.useDefault) {
               for (i=0;i<getNbBounds(aqp);i++) {
                  bit(1)       useEfficientCoding
                  GenericFloat aqp.Imin[i](useEfficientCoding);
               }
               for (i=0;i<getNbBounds(aqp);i++) {
                  bit(1)       useEfficientCoding
                  GenericFloat aqp.Imax[i](useEfficientCoding);
               }
            }
            break;

         case 13:           // Integers
            int(32)    aqp.IminInt[0];
         break;
   }
   unsigned int(5)      aqp.INbBits;

    for (i=0;i<getNbBounds(aqp);i++) {
         int(INbBits+1) vq
         aqp.Pmin[i] = vq-2^aqp.INbBits;
   }

   unsigned int(4)      aqp.PNbBits;

}
```

9.3.5.6.2 Semantics

The InitialAnimQP specifies the field's default quantization parameters.

The quantization bounds are first coded. For animTypes that have default finite bounds (Colors, BoundFloats), the default bounds of the field coding tables data structures can optionally be used by setting useDefautBounds to TRUE. For all other animTypes, this boolean is set to FALSE. For all vectorial animTypes (Position3D, Position2D, Size3D, Size2D, Float, BoundFloat, Color), if useDefaultBounds is FALSE, the quantization bounds aqp.Imin[] and aqp.Imax[] are coded. Depending on the value of useEfficientCoding, these bounds are coded using GenericFloat as floats of 32 bits or less. For the animTypes Angle, Normal and Rotation, no quantization bounds are coded.

The number of bits used in the quantization process, aqp.INbBits, is then coded. The quantization process (see 9.3.3.3) is used in intra mode only.

The minimal bounds used to offset the values obtained from the compensatiation process in predictive mode, Pmin[], are then coded. Pmins may have values in the range -2^{INbBits} to $2^{\mathrm{INbBits}}-1$. The value is coded as an unsigned integer using INbBits+1 bits and has the value PMin+2^{INbBits}.

The number of bits used for the predictive values, aqp.PNbBits, is then coded. The compensation process (see 9.3.4) is used in predictive mode only.

9.3.6 BIFS Command Syntax

9.3.6.1 Overview

This subclause describes the commands that can be sent to act on the scene. They allow insertion, modification, and deletion of elements of the scene (new scenes, nodes, fields). All BIFS information is encapsulated in BIFS command frames. Each frame may contain commands that perform a number of operations, such as insertion, deletion, or modification of scene nodes, their fields, or routes.

9.3.6.2 Command Frame

9.3.6.2.1 Syntax

```
class CommandFrame() {
   do {
      Command command();
      bit(1) continue;
   } while (continue);
}
```

9.3.6.2.2 Semantics

A CommandFrame is a collection of BIFS-Commands, and corresponds to one access unit. A sequence of commands may be sent. The boolean value continue, when TRUE, indicates that another command follows the current one.

9.3.6.3 Command

9.3.6.3.1 Syntax

```
class Command() {
   bit(2) code;
   switch (code) {
   case 0:
      InsertionCommand insert();
      break;
   case 1:
      DeletionCommand delete();
      break;
   case 2:
      ReplacementCommand replace();
      break;
   case 3:
      SceneReplaceCommand sceneReplace();
      break;
}
```

9.3.6.3.2 Semantics

For each Command, the 2-bit flag, code, signals one of the four basic commands: insertion, deletion, replacement, and scene replacement.

9.3.6.4 Insertion Command

9.3.6.4.1 Syntax

```
class InsertionCommand() {
   bit(2) parameterType ;
   switch parameterType {
   case 0:
      NodeInsertion nodeInsert();
      break;
   case 2:
      IndexedValueInsertion idxInsert();
      break;
   case 3:
      ROUTEInsertion ROUTEInsert();
      break ;
   }
}
```

9.3.6.4.2 Semantics

There are four basic insertion commands, signaled by the 2-bit flag parameterType.

If parameterType is 0, a NodeInsertion is expected.

If parameterType is 2, an IndexedValueInsertion is expected.

If parameterType is 3, a ROUTEInsertion is expected.

9.3.6.5 Node Insertion

9.3.6.5.1 Syntax

```
class NodeInsertion() {
   bit(BIFSConfig.nodeIDbits) nodeID ;
      int ndt=GetNDTFromID(nodeID);
   bit(2) insertionPosition;
   switch (insertionPosition) {
   case 0:    // insertion at a specified position
      bit (8) position;
      SFNode node(ndt);
      break;

   case 2:       // insertion at the beginning of the field
      SFNode node(ndt);
      break;

   case 3:    // insertion at the end of the field
      SFNode node(ndt);
      break;
   }
}
```

9.3.6.5.2 Semantics

The insertion of a node may be performed on a node that has an MFNode children field. Inserting a node adds the node at the desired position in the children multiple field. The command is thus valid only if the node referred to by nodeID contains a children field of type MFNode.

A node may be inserted in the children field of a grouping node. The `nodeID` of this grouping node is first coded.

The NDT of the inserted node can be determined from the NDT of the children field in which the node is inserted.

The position in the children field where the node shall be inserted, insertionPosition is then coded on two bits :

— If the `insertionPosition` is 0, the node is inserted at a specified position coded on 8 bits.

— If the `insertionPosition` is 2, the node is inserted at the beginning of the field.

— If the `insertionPosition` is 3, the node is inserted at the end of the field.

The node is then coded.

9.3.6.6 IndexedValue Insertion

9.3.6.6.1 Syntax

```
class IndexedValueInsertion() {
   bit(BIFSConfig.nodeIDbits) nodeID;
   NodeUpdateField node=GetNodeFromID(nodeID);
   int(node.nINbits) inID;
   bit(2) insertionPosition;
   switch (insertionPosition) {
   case 0: // insertion at a specified position
      bit (16) position;
      SFField value(node.field[node.in2all[inID]]);
      break;

   case 2:    // insertion at the beginning of the field
      SFField value(node.field[node.in2all[inID]]);
      break;

   case 3:    // insertion at the end of the field
      SFField value(node.field[node.in2all[inID]]);
      break;
   }
}
```

9.3.6.6.2 Semantics

The `IndexedValueInsertion` syntax allows the insertion of a new value in a multiple field at the desired position.

The `nodeID` of the node in whose field the value is to be inserted is first coded.

The field in which the value is inserted must be a multiple field type. The field is signaled with an `inID`. The `inID` is parsed using the table for the node type of the node in which the value is inserted. The node type may be determined from the `nodeID`

The position in the children field where the node shall be inserted, `insertionPosition`, is then coded:

— If the `insertionPosition` is 0, the node is inserted at a specified position coded using 16 bits.

— If the `insertionPosition` is 2, the node is inserted at the beginning of the field.

— If the `insertionPosition` is 3, the node is inserted at the end of the field.

The node is then coded.

9.3.6.7 ROUTE Insertion

9.3.6.7.1 Syntax

```
class ROUTEInsertion() {
   bit(1) isUpdatable;
   if (isUpdatable)
      bit(BIFSConfig.routeIDbits) routeID;

   bit(BIFSConfig.nodeIDbits) departureNodeID;
      NodeData nodeOUT=GetNodeFromID(departureNodeID);
   int(nodeOUT.nOUTbits) departureID;
   bit(BIFSConfig.nodeIDbits) arrivalNodeID;
      NodeData nodeIN=GetNodeFromID(arrivalNodeID);
   int(nodeIN.nINbits) arrivalID;
}
```

9.3.6.7.2 Semantics

The ROUTE insertion syntax permits the addition of a new ROUTE in the list of ROUTEs for the current scene.

A ROUTE is inserted in the list of ROUTEs by specifying a new ROUTE.

If the boolean `isUpdatable` is TRUE, a `routeID` is coded to allow the ROUTE to be referenced.

The `nodeID` of the route's departure, `departureNodeID`, is first coded.

The `outID` of the departure field in the departure node, `departureID`,is then coded.

The `nodeID` of the route's arrival, `arrivalNodeID`, is then coded.

The `inID` of the arrival field in the arrival node, `arrivalID`, is then coded.

9.3.6.8 Deletion Command

9.3.6.8.1 Syntax

```
class DeletionCommand() {
   bit(2) parameterType ;
   switch (parameterType) {
   case 0:
      NodeDeletion nodeDelete();
      break ;

   case 2:
      IndexedValueDeletion idxDelete();
      break ;

   case 3:
      ROUTEDeletion ROUTEDelete();
      break ;
   }
}
```

9.3.6.8.2 Semantics

There are three types of deletion commands, signalled by the 2-bit flag `parameterType`.

If `parameterType` is 0, a `NodeDeletion` is expected.

If `parameterType` is 2, an `IndexedValueDeletion` is expected.

If `parameterType` is 3, a `ROUTEDeletion` is expected.

9.3.6.9 Node Deletion

9.3.6.9.1 Syntax

```
class NodeDeletion() {
   bit(BIFSConfig.nodeIDbits) nodeID;
}
```

9.3.6.9.2 Semantics

The `NodeDeletion` syntax permits the deletion of a node with a specific `nodeID`. The node deletion deletes the node and all its instances, if it was referenced elsewhere in the scene with a USE statement.

The node deletion is signalled by the `nodeID` of the node to be deleted. When deleting a node, all fields shall also deleted, as well as all ROUTEs related to the node or its fields.

9.3.6.10 IndexedValue Deletion

9.3.6.10.1 Syntax

```
class IndexedValueDeletion() {
   bit(BIFSConfig.nodeIDbits) nodeID;
      NodeData node=GetNodeFromID(nodeID);
   int(node.nINbits) inID;
   bit(2) deletionPosition;
   switch (deletionPosition) {
   case 0: // deletion at a specified position
      bit(16) position;
      break;
   case 2:    // deletion at the beginning of the field
      break;
   case 3:    // deletion at the end of the field
      break;
   }
}
```

9.3.6.10.2 Semantics

The `IndexedValueDeletion` syntax permits the deletion of an element of a multiple value field.

The `nodeID` of the node to be deleted is first coded.

The `inID` of the field to be deleted is then coded.

The position in the children field from where the value shall be deleted, `deletionPosition`, is then coded:

— If the `insertionPosition` is 0, the value at specified position, coded using 16 bits, shall be deleted.

— If the `insertionPosition` is 2, the value at the beginning of the field shall be deleted.

— If the `insertionPosition` is 3, the value at the end of the field shall be deleted.

9.3.6.11 ROUTE Deletion

9.3.6.11.1 Syntax

```
class ROUTEDeletion() {
   bit(BIFSConfig.routeIDbits) routeID;
}
```

9.3.6.11.2 Semantics

The `ROUTEDeletion` syntax permits the deletion of a ROUTE with a given `routeID` from the list of active ROUTEs.

Deleting a ROUTE is performed by specifying its `routeID`. This is similar to the deletion of a node.

9.3.6.12 Replacement Command

9.3.6.12.1 Syntax

```
class ReplacementCommand() {
   bit(2) parameterType ;
   switch (parameterType) {
   case 0:
      NodeReplacement nodeReplace();
      break;

   case 1:
      FieldReplacement fieldReplace();
      break;

   case 2:
      IndexedValueReplacement idxReplace();
      break ;

   case 3:
      ROUTEReplacement ROUTEReplace();
      break;
   }
}
```

9.3.6.12.2 Semantics

There are 4 replacement commands, signalled by the 2-bit flag `parameterType`.

If `parameterType` is 0, a `NodeReplacement` is expected.

If `parameterType` is 1, a `FieldReplacement` is expected.

If `parameterType` is 2, a `IndexedValueReplacement` is expected.

If `parameterType` is 3, a `ROUTEReplacement` is expected.

9.3.6.13 Node Replacement

9.3.6.13.1 Syntax

```
class NodeReplacement() {
   bit(BIFSConfig.nodeIDbits) nodeID;
   SFNode node(SFWorldNode);
}
```

9.3.6.13.2 Semantics

The `NodeReplacement` syntax permits the deletion of an existing node and its replacement with a new node. All ROUTEs pointing to the deleted node as well as any instances of the node created through the USE mechanism shall be deleted.

The node to be replaced is signalled by its `nodeID`. The new node is encoded with the `SFWorldNode` node data type, which is valid for all BIFS nodes, in order to avoid necessitating the NDT of the replaced node to be established.

9.3.6.14 Field Replacement

9.3.6.14.1 Syntax

```
class FieldReplacement() {
   bit(BIFSConfig.nodeIDbits) nodeID ;
   NodeData node = GetNodeFromID(nodeID);
   int(node.nINbits) inID;
   Field value(node.field[node.in2all[inID]]);
}
```

9.3.6.14.2 Semantics

This `FieldReplacement` syntax permits the modification of the value of a field of an existing node. The existing value shall be deleted and replaced with the new value.

The `nodeID` of the node whose field is to be modified is first coded

The `inID` of the field to be modified is then coded

The new field is then coded

9.3.6.15 IndexedValueReplacement

9.3.6.15.1 Syntax

```
class IndexedValueReplacement() {
   bit(BIFSConfig.nodeIDbits) nodeID;
      NodeData node = GetNodeFromID(nodeID);
   int(node.nINbits) inID;
   bit(2) replacementPosition;
   switch (replacementPosition) {
   case 0: // replacement at a specified position
      bit (16) position;
      SFField value(node.field[node.in2all[inID]]);
      break;

   case 2: // replacement at the beginning of the field
      SFField value(node.field[node.in2all[inID]]);
      break;

   case 3: // replacement at the end of the field
      SFField value(node.field[node.in2all[inID]]);
      break;
   }
}
```

9.3.6.15.2 Semantics

The `IndexedValueReplacement` syntax permits the modification of the value of an element of a multiple field. As for any multiple field access, it is possible to replace at the beginning, the end or at a specified position in the multiple field.

The `nodeID` of the node whose field is to be modified is first coded

The `inID` of the field whose value is to be modified is then coded

The position in the children field where value has to be modified, `replacementPosition`, is then coded:

— If the `insertionPosition` is 0, the value at specified position, coded using 16 bits, is modified.

— If the `insertionPosition` is 2, the value at the beginning of the field is modified.

— If the `insertionPosition` is 3, the value at the end of the field is modified.

The new value is then coded as a SFField.

9.3.6.16 ROUTE Replacement

9.3.6.16.1 Syntax

```
class ROUTEReplacement() {
   bit(BIFSConfig.routeIDbits) routeID;
   bit(BIFSConfig.nodeIDbits) departureNodeID;
      NodeData nodeOUT = GetNodeFromID(nodeID);
   int(nodeOUT.nOUTbits) departureID;
   bit(BIFSConfig.nodeIDbits) arrivalNodeID;
      NodeData nodeIN = GetNodeFromID(nodeID);
   int(nodeIN.nINbits) arrivalID;
}
```

9.3.6.16.2 Semantics

Replacing a ROUTE deletes the replaced ROUTE and replaces it with the new ROUTE.

The `routeID` of the ROUTE to be replaced is first coded.

The `nodeID` of the new route's departure, `departureNodeID`, is then coded.

The `outID` of the departure field in the departure node, `departureID`, is then coded.

The `nodeID` of the route's arrival, `arrivalNodeID`, is then coded.

The `inID` of the arrival field in the arrival node, `arrivalID`, is then coded.

9.3.6.17 Scene ReplaceCommand

9.3.6.17.1 Syntax

```
class SceneReplaceCommand() {
   BIFSScene scene();
}
```

9.3.6.17.2 Semantics

Replacing a scene results in the entire BIFS scene being replaced with a new `BIFSScene` scene. When used in the context of an **Inline** node, this corresponds to replacement of the sub-scene (previously assumed to be empty). In a BIFS elementary stream, the `SceneReplacement` commands are the only random access points.

9.3.7 BIFS Scene

9.3.7.1 BIFSScene

9.3.7.1.1 Syntax

```
class BIFSScene() {
   bit(8) reserved;
   SFNode nodes(SFTopNode);
   bit(1) hasROUTEs;
   if (hasROUTEs) {
      ROUTEs routes();
   }
}
```

9.3.7.1.2 Semantics

The integer `reserved` may be used in future extensions. It shall be set to 0.

The `BIFSScene` structure represents the global scene. A `BIFSScene` is always associated to a `ReplaceScene` BIFS-Command message. The `BIFSScene` is structured in the following way:

The nodes of the scene are described first as an SFNode. The first node in the scene shall be of type SFTopNode (see Annex H).

ROUTEs are described after all nodes

All BIFS scenes shall begn with a node of type `SFTopNode`. This implies that the top node may be one of **Layer2D**, **OrderedGroup**, **Group** or **Layer3D**.

9.3.7.2 SFNode

9.3.7.2.1 Syntax

```
class SFNode(int nodeDataType) {
   bit(1) isReused ;
   if (isReused) {
      bit(BIFSConfig.nodeIDbits) nodeID;
   }
   else {
      bit(GetNDTnbBits(nodeDataType)) localNodeType;
      nodeType = GetNodeType(nodeDataType,localNodeType);
      bit(1) isUpdateable;
      if (isUpdateable) {
         bit(BIFSConfig.nodeIDbits) nodeID;
      }
      bit(1) MaskAccess;
      if (MaskAccess) {
         MaskNodeDescription mnode(MakeNode(nodeType));
      }
      else {
         ListNodeDescription lnode(MakeNode(nodeType));
      }
   }
}
```

9.3.7.2.2 Semantics

The `SFNode` syntax represents a generic node. The encoding depends on the context of the parent field of the node. This context is described by the parent field's node data type (NDT).

If `isReused` is TRUE then this node is a reference to another node, identified by its `nodeID`. This is equivalent to the use of the USE statement in ISO/IEC 14772-1:1998 [10].

If `isReused` is FALSE, then a complete node is provided in the bitstream. This requires that the `nodeType` be inferred from the node data type. The node is referenced by its `localNodeType` in the node data type table. Then, this information is converted into the node's `nodeType` (e.g. its `localNodeType` in the `SFWorldNode` NDT table).

The `isUpdatable` flag enables the assignment of a `nodeID` to the node. This is equivalent to the DEF statement of ISO/IEC 14772-1:1998 [10].

The node definition follows using either a `MaskNodeDescription`, or a `ListNodeDescription`.

The `nodeType` is a number that represents the type of the node. This `nodeType` is coded using a variable number of bits for efficiency reasons. The exact type of node may be determined from the `nodeType` as follows:

1. The data type of the field parsed indicates the node data type. The root node is always of type `SFTopNode`.
2. From the node data type expected and the total number of nodes type in the category, the number of bits representing the `nodeType` is obtained (this number is given in the node data type tables in Annex H).

3. The nodeType gives the nature of the node to be parsed.

EXAMPLE — The **Shape** node has 2 fields defined as:

```
   exposedField SFAppearanceNode        Appearance         NULL
   exposedField SFGeometry3DNode        geometry           NULL
```

When decoding a **Shape** node, if the first field is transmitted, a node of type SFAppearanceNode is expected. The only node with SFAppearanceNode type is the **Appearance** node, and hence the nodeType can be coded using 0 bits. When decoding the **Appearance** node, the following fields can be found:

```
   exposedField SFMaterialNode          Material          NULL
   exposedField SFTextureNode           texture           NULL
   exposedField SFTextureTransformNode  TextureTransform  NULL
```

9.3.7.3 MaskNodeDescription

9.3.7.3.1 Syntax

```
class MaskNodeDescription(NodeData node) {
   for (i=0; i<node.numDEFfields; i++) {
      bit(1) Mask;
      if (Mask)
         Field value(node.field[node.def2all[i]]);
   }
}
```

9.3.7.3.2 Semantics

In the MaskNodeDescription, a mask indicates, for each "def" mode field (those having a defID) of this node type, if the field value is specified. Fields are sent in the order indicated in Annex H. The field types are thus known and permit the field's value to be decoded.

9.3.7.4 ListNodeDescription

9.3.7.4.1 Syntax

```
class ListNodeDescription (NodeData node) {
   bit(1) endFlag;
   while (!EndFlag){
      int(node.nDEFbits) fieldRef;
      Field value(node.field[node.def2all[i]]);
      bit(1) endFlag;
   }
}
```

9.3.7.4.2 Semantics

In the ListNodeDescription, fields are directly addressed by their field reference, fieldRef. The reference is sent as a defID and its parsing depends on the node type (see 9.3.2.3).

9.3.7.5 Field

9.3.7.5.1 Syntax

```
class Field(FieldData field) {
   if (isSF(field))
      SFField svalue(field);
   else
      MFField mvalue(field);
}
```

9.3.7.5.2 Semantics

A field is encoded according to its type: single (SFField) or multiple (MFField). A multiple field is a collection of single fields.

9.3.7.6 MFField

9.3.7.6.1 Syntax

```
class MFField(FieldData field)  {
   bit(1) reserved;
   if (!reserved) {
      bit(1) isListDescription;
      if (isListDescription)
         MFListDescription lfield(field);
      else
         MFVectorDescription vfield(field);
   }
}
```

9.3.7.6.2 Semantics

The bit `reserved` is reserved for future extension. The bit shall be set to 0.

MFField types can be encoded with a list (`MFListDescription`) or vector (`MFVectorDescription`) description.

9.3.7.7 MFListDescription

9.3.7.7.1 Syntax

```
class MFListDescription(FieldData field)  {
   bit(1) endFlag;
   while (!endFlag) {
      SFField field(field);
      bit(1) endFlag;
   }
}
```

9.3.7.7.2 Semantics

The MFField type is encoded as a list of single fields.

9.3.7.8 MFVectorDescription

9.3.7.8.1 Syntax

```
class MFVectorDescription(FieldData field) {
   int(5) NbBits;
   int(NbBits)  numberOfFields;
   SFField field[numberOfFields](field);
}
```

9.3.7.8.2 Semantics

The MFField type is encoded as a vector of fields whose dimension is specified.

The number of bits, `NbBits`, used to specify the dimension of the vector is first coded. The actual dimension is then coded as an unsigned integer using `NbBits`. The fields are then coded in order.

9.3.7.9 SFField

9.3.7.9.1 Syntax

```
class SFField(FieldData field) {
   switch (field.fieldType) {

      case SFNodeType:
         SFNode nValue(field.fieldType);
         break;

      case SFBoolType:
         SFBool bValue;
         break;

      case SFColorType:
         SFColor cValue(field);
         break;

      case SFFloatType:
         SFFloat fValue(field);
         break;

      case SFInt32Type:
         SFInt32 iValue(field);
         break;

      case SFRotationType:
         SFRotation rValue(field);
         break;

      case SFStringType:
         SFString sValue;
         break;

      case SFTimeType:
         SFTime tValue;
         break;

      case SFUrlType:
         SFUrl uValue;
         break;

      case SFVec2fType:
         SFVec2f v2Value(field);
         break;

      case SFVec3fType:
         SFVec3f v3Value(field);
         break;

      case SFImageType:
         SFImage imageValue(field);
         break;

      case SFCommandBufferType:
         SFCommandBuffer commandValue(field);
         break;

      case SFScriptType:
         SFScript scriptValue();
         break;
   }
}
```

9.3.7.9.2 Semantics

Each field is encoded according to its `fieldType`.

9.3.7.10 GenericFloat

9.3.7.10.1 Syntax

```
class GenericFloat(boolean useEfficientCoding) {
   if (!useEfficientCoding)
      float(32) value;
   else {
      EfficientFloat  value;
}
```

9.3.7.10.2 Semantics

If the parameter `useEfficientCoding` is true, the float is coded using the `EfficientFloat` scheme. Otherwise, the IEEE 32 bit format for float coding is used.

9.3.7.11 EfficientFloat

9.3.7.11.1 Syntax

```
class EfficientFloat {
   unsigned int(4) mantissaLength;
   if (mantissaLength != 0) {
      int(3) exponentLength;
      int(1) mantissaSign;
      int(mantissaLength-1) mantissa;
      if (exponentLength != 0) {
         int(1) exponentSign;
         int(exponentLength-1) exponent;
      }
   }
}
```

9.3.7.11.2 Semantics

For floating point values it is possible to use a more economical representation than the standard 32-bit format, as specified in the `EfficientFloat` structure. This representation separately encodes the size of the exponent (base 2) and mantissa of the number.

If the `mantissaLength` is 0, the decoded value is 0 and further parameters are not coded.

If the `mantissaLength` is not 0, the `exponentLength`, `mantissaSign` and `mantissa` are coded. The mantissa sign is 1 when the mantissa is negative, otherwise it is 0.

The `mantissa` syntax element contains the actual mantissa with the leading 1 removed, hence only (`mantissaLength`-1) bits are needed to encode it.

If the `exponentLength` is 0 then exponent is not parsed, and the decoded exponent is set, by default, to 0. Otherwise, the sign is read, with `exponentSign=1` used to denote a negative exponent. The leading 1 of the exponent is not coded, so that `exponent` can be encoded using `exponentLength`-1 bits.

The actual mantissa and exponent are, respectively, $(2^{\text{mantissaLength-1}} + \texttt{mantissa})$ and

$(2^{\text{exponentLength-1}} + \texttt{exponent})$, thus in all other cases the decoded value shall be:

$$(1-2.\text{mantissaSign}).(2^{\text{mantissaLength-1}}+\text{mantissa}).2^{(1\text{-}2.\text{exponentSign}).(2^{\text{exponentLength}-1}+\text{exponent})}$$

9.3.7.12 SFBool

9.3.7.12.1 Syntax

```
class SFBool {
   bit(1) value;
}
```

9.3.7.12.2 Semantics

If `value` is 1 the decoded boolean is set to TRUE. If `value` is 0, the decoded boolean is set to FALSE.

9.3.7.13 SFColor

9.3.7.13.1 Syntax

```
class SFColor(FieldData field) {
   if (field.isQuantized)
      QuantizedField qvalue(field);
   else {
      GenericFloat rValue(field.useEfficientCoding);
      GenericFloat gValue(field.useEfficientCoding);
      GenericFloat bValue(field.useEfficientCoding);
   }
}
```

9.3.7.13.2 Semantics

If the field's `isQuantized` bit is TRUE, the `QuantizedField` scheme shall be used. Otherwise each component of the `SFColor` is coded using the `GenericFloat` scheme.

9.3.7.14 SFFloat

9.3.7.14.1 Syntax

```
class SFFloat(FieldData field) {
   if (field.isQuantized)
      QuantizedField qvalue(field);
   else
      GenericFloat value(field.useEfficientCoding);
}
```

9.3.7.14.2 Semantics

If the field's `isQuantized` bit is TRUE, the `QuantizedField` scheme shall be used. Otherwise the SFFloat is coded using the `GenericFloat` scheme.

9.3.7.15 SFInt32

9.3.7.15.1 Syntax

```
class SFInt32(FieldData field) {
   if (field.isQuantized)
      QuantizedField qvalue(field);
   else
      int(32) value;
}
```

9.3.7.15.2 Semantics

If the field's `isQuantized` bit is TRUE, the `QuantizedField` scheme shall be used. Otherwise the SFInt32 is coded as a signed value using 32 bits.

9.3.7.16 SFRotation

9.3.7.16.1 Syntax

```
class SFRotation(FieldData field) {
   if (field.isQuantized)
      QuantizedField qvalue(field);
   else {
      GenericFloat xAxis(field.useEfficientCoding);
      GenericFloat yAxis(field.useEfficientCoding);
      GenericFloat zAxis(field.useEfficientCoding);
      GenericFloat angle(field.useEfficientCoding);
   }
}
```

9.3.7.16.2 Semantics

If the field’s `isQuantized` bit is TRUE, the `QuantizedField` scheme shall be used. Otherwise each component of the SFRotation is coded indepedently using the `GenericFloat` scheme.

9.3.7.17 SFString

9.3.7.17.1 Syntax

```
class SFString {
   unsigned int(5) lengthBits;
   unsigned int(lengthBits) length;
   char(8) value[length];
}
```

9.3.7.17.2 Semantics

The `SFString` is coded as an array of characters whose length is first specified.

`lengthBits` is the number of bits used to encode the string length.

`length` is the length of the string coded using `lengthBits`.

All characters are coded using the UTF-8 character encoding [3].

9.3.7.18 SFTime

9.3.7.18.1 Syntax

```
class SFTime {
   double(64) value;
}
```

9.3.7.18.2 Semantics

The SFTime value is coded as a 64-bit double.

9.3.7.19 SFUrl

9.3.7.19.1 Syntax

```
class SFUrl {
   bit(1) isOD;
   if (isOD)
      bit(10) ODid;
   else
      SFString urlValue;
}
```

9.3.7.19.2 Semantics

If the `SFUrl` refers to an object descriptor, the `ObjectDescriptorID` is coded as a 10-bit integer. Otherwise the URL is sent as an `SFString`.

9.3.7.20 SFVec2f

9.3.7.20.1 Syntax

```
class SFVec2f(FieldData field) {
   if (field.isQuantized)
      QuantizedField qvalue(field);
   else {
      GenericFloat value1;
      GenericFloat value2;
   }
}
```

9.3.7.20.2 Semantics

If the field’s `isQuantized` bit is TRUE, the `QuantizedField` scheme shall be used. Otherwise each component of the SFVec2f is coded using the `GenericFloat` scheme.

9.3.7.21 SFVec3f

9.3.7.21.1 Syntax

```
class SFVec3f(FieldData field) {
   if (field.isQuantized)
      QuantizedField qvalue(field);
   else {
      GenericFloat value1(field.useEfficientCoding);
      GenericFloat value2(field.useEfficientCoding);
      GenericFloat value3(field.useEfficientCoding);
   }
}
```

9.3.7.21.2 Semantics

If the field’s `isQuantized` bit is TRUE, the `QuantizedField` scheme shall be used. Otherwise each component of the SFVec3f is coded using the `GenericFloat` scheme.

9.3.7.22 SFImage

9.3.7.22.1 Syntax

```
class SFImage {
   unsigned int(12)  width;
   unsigned int(12)  height;
   bit(2)  numComponents;
   bit(8)  pixels[(numComponents+1)*width*height];
}
```

9.3.7.22.2 Semantics

The `width` and `height` in pixels of the image are coded as 12-bit unsigned integers.

`numComponents` defines the image type. The following types are permitted:

— If the value is ‘00’, then a grey scale image shall be decoded.

— If the value is ‘01’, then a grey scale with alpha channel shall be decoded.

— If the value is ‘10’, then an RGB image shall be decoded.

— If the value is ‘11’, then an RGB image with alpha channel shall be decoded.

Pixels shall be decoded as unsigned char, 8-bit encoded pixel values.

9.3.7.23 SFCommandBuffer

9.3.7.23.1 Syntax

```
class SFCommandBuffer {
   unsigned int(5) lengthBits;
   unsigned int(lengthBits) length;
   bit(8) value[length];
}
```

9.3.7.23.2 Semantics

The `SFCommandBuffer syntax element` is coded as an array of bytes whose length is first specified.

`lengthBits` is the number of bits used to encode the buffer length.

`length` is the length of the buffer coded using `lengthBits`.

`value` is an array of bytes of length `length`. It shall contain a `CommandFrame`, padded if necessary to complete the last byte.

9.3.7.24 QuantizedField

9.3.7.24.1 Syntax

```
class QuantizedField(FieldData field) {
   switch (field.quantType) {

      case 9:
         int(1) direction
      case 10:
         int(2) orientation
      default:
         break;
    }
    for (i=0;i<getNbComp(field);i++)
      int(field.nbBits) vq[i]
}
```

9.3.7.24.2 Semantics

The value is quantized using the quantization process described in 9.3.3.

For normals, the direction and orientation values specified in the quantization process are first coded. For rotations, only the orientation value is coded.

The compressed components, `vq[i]`, of the field’s value are then coded in sequence as unsigned integers using the number of bits specified in the field data structure.

9.3.7.25 SFScript

9.3.7.25.1 Syntax

```
class SFScript() {
   bit(1) isListDescription;
   if (isListDescription)
      ScriptFieldsListDescripion();
   else
      ScriptFieldsVectorDescripion();
   bit(1) EncodeURL;
   if(!EncodeURL)
      MFURL URLstring;
   else
      SFNode EncodedScript();
}
```

9.3.7.25.2 Semantics

The `Script` class is used to represent a **Script** node. This can be done as a list description or as a vector description, depending on the value in `isListDescription`. If `EncodeURL` is TRUE, the `URLString` field is read as a regular URL. Otherwise, the `URLString` field shall contain a script, which is encoded using the bitstream syntax for `EncodedScript`, given below. This bitstream is a tree representation of the BNF grammar for ECMAScript [11]. Each node determines the parse decision selected in parsing the script, and thus the resulting bitstream can be used to interpret the script directly.

9.3.7.26 ScriptFieldsListDescription

9.3.7.26.1 Syntax

```
class ScriptFieldsListDescription() {
   bit(1) endFlag; // List description of the fields
   while (!EndFlag) {
      ScriptField();
      bit(1) endFlag;
   }
}
```

9.3.7.26.2 Semantics

`ScriptFieldsListDescription` reads a list description of the fields in the **Script** node. When `endFlag` has value 1, the list has ended and no more values are read.

9.3.7.27 ScriptFieldsVectorDescription

9.3.7.27.1 Syntax

```
class ScriptFieldsVectorDescription() {
   bit(4) fieldBits;  // Number of bits for number of fields
   bit(fieldBits)  numFields; // Number of fields in the script
   for (i=0; i<numFields; ++i) {
      ScriptField();
   }
}
```

9.3.7.27.2 Semantics

`ScriptFieldsVectorDescription` reads a value `numFields`, to determine how many fields are in the **Script** node, and these are read sequentially. The number of bits used to give the number of fields is first read as 4 bits in `fieldBits`.

9.3.7.28 ScriptField

9.3.7.28.1 Syntax

```
class ScriptField() {
   bit(2) eventType;
   bit(6) fieldType;
   if (eventType == FIELD) {
      bit(1) hasInitialValue;
      if (hasInitialValue){
         NodeData node = makeNode(ScriptNodeType);
         Field(node.field[fieldType]) value;
      }
   }
}
```

9.3.7.28.2 Semantics

The `ScriptField` contains one field for the **Script** node. The `eventType` specifies the type of field, with values 0, 1, and 2 representing fields, eventIns and eventOuts, respectively. The `fieldType` is an integer value that holds the same value as the `nodeData` structure's `fieldType`. This determines the type of the field.

When the event is a field, it may have a default value. This presence of this value is indicated by `hasInitialValue` being 1. In this case, the field value is read using the `Field` class. In order to be able to use the `Field` class, a node of type `NodeData` is created which then has the appropriate field value for each `fieldType` (the `fieldType` index can be used to reference field structures of the appropriate type).

9.3.7.29 EncodedScript

9.3.7.29.1 Syntax

```
class EncodedScript {
   bit(1) hasFunction
   while (hasFunction) {
      Function function;
      bit(1) hasFunction
   }
}
```

9.3.7.29.2 Semantics

A script is a collection of functions, listed sequantially while `hasFunction` is TRUE.

9.3.7.30 Function

9.3.7.30.1 Syntax

```
class Function {
   Identifier identifier;
   Arguments arguments;
   StatementBlock statementBlock;
}
```

9.3.7.30.2 Semantics

Each function consists of an `identifier`, a list of `arguments`, and a `statementBlock` which contains the script statements executed when the function is called.

9.3.7.31 Arguments

9.3.7.31.1 Syntax

```
class Arguments {
   bit(1) hasArgument
   while (hasArgument) {
      Identifier identifier;
      bit(1) hasArgument
   }
}
```

9.3.7.31.2 Semantics

The argument list is of arbitrary length, and terminates when `hasArgument` is 0. Each argument consists of one `identifier`.

9.3.7.32 StatementBlock

9.3.7.32.1 Syntax

```
class StatementBlock {
   bit(1) isCompoundStatement
   if (isCompoundStatement) {
      bit(1) hasStatement
      while (hasStatement) {
         Statement statement;
         bit(1) hasStatement
      }
      else {
         Statement statement;
      }
   }
}
```

9.3.7.32.2 Semantics

A `statementBlock` consists of either a `compoundStatement`, which holds several script statements, or a single statement, indicated by the value of `isCompoundStatement`. When the `statementBlock` consists of several statements, the `hasStatement` bit is used to signal either the end of the list or the existance of another statement.

9.3.7.33 Statement

9.3.7.33.1 Syntax

```
class Statement {
   bit(3) statementType
   switch statementType {
      case ifStatementType:
         IFStatement ifStatement;
         break;
      case forStatementType:
         FORStatement forStatement;
          break;
      case whileStatementType:
         WHILEStatement whileStatement;
         break;
      case returnStatementType:
         RETURNStatement returnStatement;
         break;
      case compoundExpressionType:
         CompoundExpression compoundExpression;
         break;
      case breakStatementType:
```

```
      case continueStatementType:
            break;
   }
}
```

9.3.7.33.2 Semantics

A `Statement` may consist of one of the following specific statement types:

— `ifStatement`

— `forStatement`

— `whileStatement`

— `returnStatement`

— `compoundExpression`

— `breakStatement`

— `continueStatement`.

These statement types are indicated by a value from 0-6, respectively, called `statementType`.

9.3.7.34 IFStatement

9.3.7.34.1 Syntax

```
class IFStatement {
   CompoundExpression compoundExpression;
   StatementBlock statementBlock;
   bit(1) hasELSEStatement
   if (hasELSEStatement) {
      StatementBlock statementBlock;
   }
}
```

9.3.7.34.2 Semantics

An `IFStatement` is used for conditional execution of a `statementBlock`. It consists of a `CompoundExpression` followed by a `statementBlock`. The `statementBlock` is interpreted when the `CompoundExpression` evaluates to a non-zero or non-empty value. The `IFStatement` has an optional additional `statementBlockwhich` is included when `hasElseStatement` is 1. This second, optional `compoundStatement` is interpreted when the `CompoundExpression` evaluates to a zero or empty value.

9.3.7.35 FORStatement

9.3.7.35.1 Syntax

```
class FORStatement {
   OptionalExpression optionalExpression;
   OptionalExpression optionalExpression;
   OptionalExpression optionalExpression;
   StatementBlock statementBlock;
}
```

9.3.7.35.2 Semantics

A `FORStatement` is used to iterate over values, stopping when a conditional expression fails. The first `optionalExpression` shall be executed when the statement is interpretted. The second `optionalExpression` shall then be evaluated, and if it returns a non-zero or non-empty value, the statementBlock shall be executed. The third optionalExpression shall then be executed. After this process shall repeat starting with the execution of the second optionalExpression again, the statementBlock, and the third optionalExpression.

9.3.7.36 WHILEStatement

9.3.7.36.1 Syntax

```
class WHILEStatement {
   CompoundExpression compoundExpression;
   StatementBlock statementBlock;
}
```

9.3.7.36.2 Semantics

The WHILEStatement is used to conditionally execute a statementBlock for so long as the compoundExpression evaluates to a non-zero or non-empty value.

9.3.7.37 RETURNStatement

9.3.7.37.1 Syntax

```
class RETURNStatement {
   bit(1) returnValue
   if (returnValue) {
      CompoundExpression compoundExpression;
   }
}
```

9.3.7.37.2 Semantics

The RETURNStatement is used to return a value from a function. When a function has no return value, returnValue shall be 0. Otherwise, the returned value shall be the last value evaluated for compoundExpression.

9.3.7.38 CompoundExpression

9.3.7.38.1 Syntax

```
class CompoundExpression {
   do {
      Expression expression;
      bit(1) hasExpression
   } while (hasExpression);
}
```

9.3.7.38.2 Semantics

A CompoundExpression is a list of expressions, terminated when hasExpression has value 0. The value of the compound expression shall be the value of the last evaluated expression.

9.3.7.39 optionalExpression

9.3.7.39.1 Syntax

```
class optionalExpression {
   bit(1) hasCompoundExpression
   if (hasCompoundExpression) {
      CompoundExpression compoundExpression;
   }
}
```

9.3.7.39.2 Semantics

An optionalExpression may be an empty expression, containing no executable statements, or a compoundExpression. This is indicated by the value of hasCompoundExpression.

9.3.7.40 Expression

9.3.7.40.1 Syntax

```
class Expression {
   bit(6) expressionType
   switch expressionType {
      case curvedExpressionType:          // (compoundExpression)
         CompoundExpression compoundExpression;
         break;
      case negativeExpressionType:            // -expression
      case notExpressionType:             // !expression
      case onescompExpressionType:            // ~expression
      case incrementExpressionType:             // ++expression
      case decrementExpressionType:             // --expression
      case postIncrementExpressionType:        // expression++
      case postDecrementExpressionType:        // expression--
         Expression expression;
         break;
      case conditionExpressionType:   // expression ? expression : expression
         Expression expression;
         Expression expression;
         Expression expression;
         break;
      case stringExpressonType:
         String string;
         break;
      case numberExpressionType:
         Number number;
         break;
      case variableExpressionType:
         Identifier identifier;
         break;
      case functionCallExpressionType:
      case objectConstructExpressionType:
         Identifier identifier;
         Params params;
         break;
      case objectMemberAccessExpressionType:
         Expression expression;
         Identifier identifier;
         break;
      case objectMethodCallExpressionType:
         Expression expression;
         Identifier identifier;
         Params params;
         break;
      case arrayDereferenceExpressionType:
         Expression expression;
         CompoundExpression compoundExpression;
         break;
      default: // =, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=,
         // ==, !=, <, <=, >, >=, +, -, *, /, %, &&, ||, &, |,
         // ^, <<, >>, >>>
         Expression expression;
         Expression expression;
         break;
   }
}
```

9.3.7.40.2 Semantics

An expression may contain one of a number of possible executed statements, specified by the value in `expressionType`. These are listed below, according the value of `expressionType`.

`curvedExpressionType=0:`
The expression consists of a `compoundExpression`.

`NegativeExpressionType=1:`
An `expression` shall be evaluated and the value returned shall be negated.

`NotExpressionType=2:`
An `expression` shall be evaluated and its returned value shall be logically negated (empty values return non-empty, zero values return non-zero, and vice-versa).

`OnescompExpressionType=3:`
An `expression` shall be evaluated numerically (string values will yield an undefined result) and the value returned shall be bitwise negated.

`IncrementExpressionType=4:`
An `expression` shall be evaluated numerically (string values will yield an undefined result) and the value returned shall incremented by 1.

`DecrementExpressionType=5:`
An `expression` shall be evaluated numerically (string values will yield an undefined result) and the value returned shall be decremented by 1.

`PostIncrementExpressionType=6:`
An `expression` shall be evaluated numerically (string values will yield an undefined result) and its returned value shall be incremented by 1. The returned value of this `expression` shall be the value prior to the increment being applied.

`PostDecrementExpressionType=7:`
An `expression` shall be evaluated numerically (string values will yield an undefined result) and its returned value shall be decremented by 1. The returned value of this `expression` shall be the value prior to the decrement being applied.

`ConditionExpressionType=8:`
Three `expressions` shall be evaluated. If the first expression returns a non-zero or non-empty value, then the returned value of this `expression` shall be the value of the second `expression`. Otherwise, the returned value of this `expression` shall be the value of the third `expression`.

`StringExpressonType=9:`
The `expression` contains a string.

`NumberExpressionType=10:`
The `expression` is a number.

`VariableExpressionType=11:`
The `expression` is a variable and shall return the value held by the variable determined by `identifier`.

`FunctionCallExpressionType=12:`
An `identifier` determines which `function` shall be evaluated. The `params` shall be passed to the `function` by value. The returned value of the `expression` shall be the value returned by the function in its `returnStatement`.

`ObjectConstructExpressionType=13:`
A new object shall be created (using a ‘new’ statement in the script) and the object shall be held in the variable determined by `identifier`. A list of `params` shall be passed to any constructors that exist for the object.

`ObjectMemberAccessExpressionType=14:`
A member variable of an object shall be accessed and the returned value of the `expression` shall be the value in this member variable. Normally, the first `expression` will evaluate to a node in the scene graph (which is accessed through a script variable). This means that the first `expression` will normally evaluate to an `identifier` reference. The following `identifier` will then refer to a field of the node.

`ObjectMethodCallExpressionType=15:`
A method of an object shall be evaluated. The first `expression` shall evaluate to an object. The following `identifier` shall specify a method of this object. The following `params` shall be passed to the method. The value of this expression shall be the value returned by the method.

`ArrayDereferenceExpressionType=16:`
The `expression` shall be an array element reference. The first `expression` shall evaluate to an array reference. The following `compoundExpression` shall evaluate to a number that shall then be used to index the array. The returned value of this `expression` shall be the value held in the referenced array element.

The following binary operands evaluate two expressions and return a value based on a binary operation of these two expressions. The binary operation and value of `expressionType` is listed below for each binary operation. Unless explicitely stated, a string value for either of the expressions will yield an undefined result.

`BinaryOperand(=) = 17:`
The first `expression` shall evaluate to an `identifier` which shall be assigned the value of the second `expression`.

`BinaryOperand(+=) = 18:`
The first `expression` shall evaluate to an `identifier`. If the value held by the variable is numerical, the variable value shall be incremented by the value of the second `expression`, which shall also evaluate to a numerical value. If the variable is a string, then its new value shall be its original value with the second expression (which shall be a string) appended.

`BinaryOperand(-=) = 19:`
The first `expression` shall evaluate to an `identifier` whose value shall be decremented by the value of the second `expression`.

`BinaryOperand(*=) = 20:`
The first `expression` shall evaluate to an `identifier` whose value shall be set to its current value multiplied by the value of the second `expression`.

`BinaryOperand(/=) = 21:`
The first `expression` shall evaluate to an `identifier` whose value shall be set to its current value divided by the value of the second `expression`.

`BinaryOperand(%=) = 22:`
The first `expression` shall evaluate to an `identifier` whose value shall be set to its current value modulo the value of the second `expression`. The expressions shall both evaluate to integer values.

`BinaryOperand(&=) = 23:`
The first `expression` shall evaluate to an `identifier` whose value shall be set to its current value logically bitwise ANDed with the value of the second `expression`.

`BinaryOperand(|=) = 24:`
The first `expression` shall evaluate to an `identifier` whose value shall be set to its current value logically bitwise ORed with the value of the second `expression.`

`BinaryOperand(^=) = 25:`
The first `expression` shall evaluate to an `identifier` whose value shall be set to its current value logically bitwise EXCLUSIVE-ORed with the value of the second `expression.`

`BinaryOperand(<<=) = 26:`
The first `expression` shall evaluate to an `identifier` whose value shall be set to its current value bitwise shifted to the left a number of bits specified by the second `expression`.

`BinaryOperand(>>=) = 27:`
The first `expression` shall evaluate to an `identifier` whose value shall be set to its current value bitwise shifted to the right a number of bits specified by the second `expression`.

`BinaryOperand(>>>=) = 28:`
The first `expression` shall evaluate to an `identifier` whose value shall be set to its current value bitwise shifted to the right (with the least significant bits looped) a number of bits specified by the second `expression`.

`BinaryOperand(==) = 29:`
This expression shall return a non-zero value when the first and second `expression` are identical. Otherwise, the result of this expression shall be zero.

`BinaryOperand(!=) = 30:`
This expression shall return a non-zero value when the first and second `expression` are not identical. Otherwise, the result of this expression shall be zero.

`BinaryOperand(<) = 31:`
This expression shall return a non-zero value when the first `expression` is numerically or lexicographically less than the second. Otherwise, the result of this expression shall be zero.

`BinaryOperand(<=) = 32:`
This expression shall return a non-zero value when the first `expression` is numerically or lexicographically less than or equal to the second. Otherwise, the result of this expression shall be zero.

`BinaryOperand(>) = 33:`
This expression shall return a non-zero value when the first `expression` is numerically or lexicographically greater than the second. Otherwise, the result of this expression shall be zero.

`BinaryOperan(>=) = 34:`
This expression shall return a non-zero value when the first `expression` is numerically or lexicographically greater than or equal to the second. Otherwise, the result of this expression shall be zero.

`BinaryOperand(+) = 35:`
This expression shall return the sum of the first and second `expressions`. If both `expressions` are strings, then the result shall be the first `string` concatenated with the second.

`BinaryOperand(-) = 36:`
This expression shall return the difference of the first and second `expressions`.

`BinaryOperand(*) = 37:`
This expression shall return the product of the first and second `expressions`.

`BinaryOperand(/) = 38:`
This expression shall returns the quotient of the first and second `expressions`.

`BinaryOperand(%) = 39:`
This expression shall return the value of the first `expression` modulo the second `expression`.

`BinaryOperand(&&) = 40:`
This expression shall return the logical AND of the first and second `expressions`.

`BinaryOperand(||) = 41:`
This expression shall return the logical OR of the first and second `expressions`.

`BinaryOperand(&) = 42:`
This expression shall return the logical bitwise AND of the first and second `expressions`.

`BinaryOperand(|) = 43:`
This expression shall return the logical bitwise OR of the first and second `expressions`.

`BinaryOperand(^) = 44:`
This expression shall return the logical bitwise XOR of the first and second `expressions`.

`BinaryOperand(<<) = 45:`
This expression shall return the value of the first `expression` shifted to the left by the number of bits specified as the value of the second `expression`.

`BinaryOperand(>>) = 46:`
Returns the value of the first `expression` shifted to the right by the number of bits specified as the value of the second `expression`.

`BinaryOperand(>>>) = 47:`
This expression shall return the value of the first `expression` shifted to the right (with the least significant bit looped to the most significant bit) by the number of bits specified as the value of the second `expression`.

9.3.7.41 Params

9.3.7.41.1 Syntax

```
class Params {
   bit(1) hasParam
   while(hasParam) {
      Expression expression;
      bit(1) hasParam
   }
}
```

9.3.7.41.2 Semantics

The `Params` class consists of a (possibly empty) list of `expressions`. The `hasParam` bit indicates either the end of the list, or the existance of another `expression`.

9.3.7.42 Identifier

9.3.7.42.1 Syntax

```
class Identifier {
   bit(1) received
   if (received) {
      bit(num) identifierCode // num is calculated by counting
         // number of distinguished identifiers
         // received so far
   }
   else {
      String string;
   }
}
```

9.3.7.42.2 Semantics

An `identifier` is used to identify a variable. If the `identifier` has not occured before in the script, a `String` is sent holding the name of the identifier. This is indicated by the `received` bit. If the `identifier` has occured before in the script, then an `identifierCode` value is sent using `num` bits. The value of `num`, that is, the number of bits needed to send the index of the identifier in a list of all previously occuring identifiers, is variable and is determined by the minimum number of bits needed to specify the length of the list of all previously occuring identifiers.

9.3.7.43 String

9.3.7.43.1 Syntax

```
class String {
   bit(8) char
   while (char!=0) {
      bit(8) char
   }
}
```

9.3.7.43.2 Semantics

A `String` type consist of a null-terminated list of 8 bit characters.

9.3.7.44 Number

9.3.7.44.1 Syntax

```
class Number {
   bit(1) isInteger
   if (isInteger) {
      bit(5) numbits // number of bits the integer is represented
      bit(numbits) value // integer value
   }
   else {
      bit(4) floatChar // 0-9, ., E, END_SYMBOL
      while (floatChar!=END_SYMBOL) {
         bit(4) floatChar
      }
   }
}
```

9.3.7.44.2 Semantics

A number shall be represented as an integer, indicated by `isInteger`, or as a list of 4 bit characters, represending (in order) the characters 0 ,1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,0 , . , E , END-SYMBOL. The END-SYMBOL value is used to signal the end of the float value list. The list of characters shall result in a human readable float value in scientific notation.

9.3.7.45 ROUTEs

9.3.7.45.1 Syntax

```
class ROUTEs() {
   bit(1) ListDescription;
   if (ListDescription)
      ListROUTEs lroutes();
   else
      VectorROUTEs vroutes();
}
```

9.3.7.45.2 Semantics

`ROUTEs` may be encoded with a list (`ListROUTEs`) or vector (`VectorROUTEs`) description.

9.3.7.46 ListROUTEs

9.3.7.46.1 Syntax

```
class ListROUTEs() {
   do {
      ROUTE route();
      bit(1) moreROUTEs;
   }
   while (moreROUTEs);
}
```

9.3.7.46.2 Semantics

The ROUTEs are coded as a list, with the `moreROUTEs` flag used to indicate the end of the list (when set to false).

9.3.7.47 VectorROUTEs

9.3.7.47.1 Syntax

```
class VectorROUTEs() {
   int(5) nBits;
   int(nBits) length;
   ROUTE route[length]();
}
```

9.3.7.47.2 Semantics

The ROUTEs are coded as a vector whose dimension, `length`, is first specified.

9.3.7.48 ROUTE

9.3.7.48.1 Syntax

```
class ROUTE() {
   bit(1) isUpdateable;
   if (isUpdateable)
      bit(BIFSConfig.routeIDbits) routeID;

   bit(BIFSConfig.nodeIDbits) outNodeID;
      NodeData  nodeOUT = GetNodeFromID(outNodeID);
   int(nodeOUT.nOUTbits) outFieldRef;
   bit(BIFSConfig.nodeIDbits) inNodeID;
      NodeData nodeIN = GetNodeFromID(inNodeID);
   int(nodeIN.nINbits) inFieldRef;
}
```

9.3.7.48.2 Semantics

This is the basic syntax element used to represent a ROUTE. If `isUpdateable` is TRUE (‘1’) then a `routeID` is sent to enable further reference to this route. The ROUTE description is then sent. The `nodeID` of the target node is coded, followed by the target field’s `outID`. The `nodeID` of the source node is then coded, followed by the source field’s `inID`.

9.3.8 BIFS-Anim

9.3.8.1 Overview

The BIFS-Anim session has two parts: the `AnimationMask` and the `AnimationFrames`. The `AnimationMask` specifies the nodes and fields to be animated. It is sent in BIFS configuration, in the object descriptor for the BIFS elementary stream. The animation frames are sent in a separate BIFS stream. When parsing the BIFS-Anim stream, the node structure and related functions as described in Annex H are known at the receiving terminal. The decoding data structure `AnimationMask` (see 9.3.2.5) is constructed when the `AnimationMask` syntax is read, and further used in the decoding process of the BIFS-Anim frames.

`AnimationFrames` contain update information for thevalues of the animated fields described in the `AnimationMask`. They are the access units of the BIFS-Anim stream. An `AnimationFrame` can send information in intra or in predictive mode. In intra mode, the values are quantized and coded directly. In predictive mode, the difference between the quantized value of the current and the last transmitted value of the field are coded. The encoding is performed using an adaptative arithmetic coder described in Annex G.

The use of the adaptive arithmetic coder is as follows:

At the beginning of each predictive frame, the adaptive arithmetic coder is reset. At the end of each frame, it is flushed.

Each animated field has its own set of models. At each intra frame, if the stream has been declared in random access mode (see 9.3.5.2), the models are reset to the uniform statistics. If the stream is not in random access mode, the models are not reset unless the decoding structures (`AnimQP`) are modified.

9.3.8.2 AnimationFrame

9.3.8.2.1 Syntax

```
class AnimationFrame() {
   AnimationFrameHeader header(BIFSConfig.animMask);
   AnimationFrameData data(BIFSConfig.animMask);
}
```

9.3.8.2.2 Semantics

The `AnimationFrame` is the access unit of the BIFS-Anim stream. It contains the `AnimationFrameHeader`, which specifies timing, and specifies which nodes are animated in the list of animated nodes, and the `AnimationFrameData`, which contains the data for all nodes being animated.

9.3.8.3 AnimationFrameHeader

9.3.8.3.1 Syntax

```
class AnimationFrameHeader(AnimationMask mask) {
   bit(23)* next;
   if (next==0)
      bit(32) AnimationStartCode;

   bit(1) mask.isIntra;
   bit(1) mask.isActive[mask.numNodes];
   if (isIntra) {
      bit(1) isFrameRate;
      if (isFrameRate)
         FrameRate rate;
      bit(1) isTimeCode;
      if (isTimeCode)
         unsigned int(18) timeCode;
   }
   bit(1) hasSkipFrames;
   if (hasSkipFrames)
      SkipFrames skip;
}
```

9.3.8.3.2 Semantics

In the `AnimationFrameHeader`, a start code may be sent at each intra or prdictive frame to enable resynchronization. The first 23 bits are read ahead, and stored as the integer `next`.

If `next` is 0 (in other words, the first 23 bits if the `AnimationFrame` are 0), the first 32 bits of the `AnimationFrame` shall be read and interpreted as a start code that precedes the `AnimationFrame`.

If the boolean `isIntra` is TRUE, the current animation frame contains intra-coded values, otherwise it is a predictive frame.

The array of booleans `isActive` specifies which nodes shall be animated for this frame. `isActive` shall contain one boolean for each node in the `AnimationMask`. The boolean is set to TRUE if the node is to be animated; FALSE otherwise.

In intra mode, some additional timing information is also specified. The timing information obeys the syntax of the Facial Animation specification in ISO/IEC 14496-2. Finally, it is possible to skip a number of `AnimationFrames` by using the `FrameSkip` syntax specified in ISO/IEC 14496-2.

9.3.8.4 FrameRate

9.3.8.4.1 Syntax

```
class FrameRate {
   unsigned int(8) frameRate;
   unsigned int(4) seconds;
   bit(1) frequencyOffset;
}
```

9.3.8.4.2 Semantics

`frame_rate` is an 8-bit unsigned integer indicating the reference frame rate of the sequence.

`seconds` is a 4-bit unsigned integer indicating the fractional reference frame rate. The frame rate is computed as follows:

frame rate = (`frame_rate` + `seconds`/16).

`frequency_offset` is a 1-bit flag which when set to '1' indicates that the frame rate uses the NTSC frequency offset of 1000/1001. This bit would typically be set when `frame_rate` = 24, 30 or 60, in which case the resulting frame rate would be 23.97, 29.94 or 59.97 respectively. When set to '0' no frequency offset is present, i.e. if (`frequency_offset` ==1), frame rate = (1000/1001) * (`frame_rate` + `seconds`/16).

9.3.8.5 SkipFrame

9.3.8.5.1 Syntax

```
class SkipFrame {
   int nFrame = 0;
   do {
      bit(4) number_of_frames_to_skip;
      nFrame = number_of_frames_to_skip + nFrame;
   } while (number_of_frames_to_skip == 0b1111);
}
```

9.3.8.5.2 Semantics

`number_of_frames_to_skip` is a 4-bit unsigned integer indicating the number of frames skipped. If the `number_of_frames_to_skip` is equal to 15 (pattern "1111") then another 4-bit word follows allowing a skip of up to 29 frames (pattern "11111110") to be specified. If the 8-bits pattern equals "11111111", then another 4-bits word shall follow and so on, and the number of frames skipped is incremented by 30. Each 4-bit pattern of '1111' increments the total number of frames to skip with 15.

9.3.8.6 AnimationFrameData

9.3.8.6.1 Syntax

```
class AnimationFrameData (AnimationMask mask) {

   int i;
   for (i=0; i<mask.numNodes; i++) {
      if (mask.isActive[i]) {
         NodeData node = mask.animNode[i]
         switch (node.nodeType) {
         case FaceType:
            FaceFrameData fdata;
            break;
         case BodyType:
          BodyFrameData bdata;
          break;
       case IndexedFaceSet2DType:
          Mesh2DframeData mdata;
          break;
       default
          int j;
          for(j=0; j<node.numDYNfields; j++) {
             if (node.isAnimField[j])
                AnimationField AField(node.field[node.dyn2all[j]],mask.isIntra);
          }
       }
    }
  }
}
```

9.3.8.6.2 Semantics

The `AnimationFrameData` corresponds to the field data for the nodes being animated. In the case of an **IndexedFaceSet2D**, a **Face**, or a **Body** node, the syntax used is that defined ISO/IEC 14496-2. In other cases, for each field declared as an animated field is the `AnimationMask`, the `AnimationField` is sent.

NOTE — The **Body** node ("`case BodyType`") is not specified in ISO/IEC 14496 nor in ISO/IEC 14772-1:1998 [10]. This syntax switch has been provided to allow support for future extensions.

In predictive mode, at the beginning of the `AnimationFrameData`, an adaptive arithmetic coder session is initiated by resetting the adaptive arithmetic coder in the way defined by the procedure `decoder_reset( )` in Annex G. Then, the animated values are sent using this adaptive arithmetic coder, using and updating their own models.

9.3.8.7 AnimationField

9.3.8.7.1 Syntax

```
class AnimationField(FieldData field, boolean isIntra) {
  AnimFieldQP aqp = field.aqp;
  if (isIntra) {
       bit(1) hasQP;
       if(hasQP) {
          AnimQP QP(aqp);
       }
       int i;
       for (i=0; i<aqp.numElements; i++)
          if (aqp.indexList[i])
             AnimIValue ivalue(field);
  } else {
       int i;
       for (i=0; i<aqp.numElements; i++)
          if (aqp.indexList[i])
             AnimPValue pvalue(field);
  }
}
```

9.3.8.7.2 Semantics

In an `AnimationField`, if in intra mode, a new animation quantization parameter value may be sent. The intra or predictive frame follows.

In intra mode, if `hasQP` is TRUE, a new `AnimQP` is sent, it shall be valid until the next intra frame is received. If `hasQP` is FALSE, the value of the `randomAccess` boolean shall be considered.

— If `randomAccess` is set to TRUE, then the `InitialAnimQP` shall be used until the next intra frame.

— If randomAccess is set to FALSE, then the AnimQP that was valid at the previous intra frame shall be used. In this case, no random access is possible ato this particular frame.

In intra mode, if BIFSConfig.randomAccess is TRUE , the field's predictive models shall then be reset to be uniform models as defined by the procedure model_reset(PNbBits) in Annex G. If BIFSConfig.randomAccess is FALSE, the field's models are reset only if a new AnimQP is received.

The value is then sent: in intra mode, an AnimIValue is expected, in predictive mode an AnimPValue is expected.

9.3.8.8 AnimQP

9.3.8.8.1 Syntax

```
class AnimQP(AnimFieldQP aqp) {

  bit (1) IMinMax ;
  if (IMinMax) {

   aqp.useDefault=FALSE;
  switch(aqp.animType) {

       case  4:   // Color
       case  8:   // BoundFloats
          bit(1)  aqp.useDefault
       case  1:   // Position 3D
       case  2:   // Position 2D
       case 11:   // Size 3D
       case 12:   // Size 2D
       case  7:   // Floats
          if (!aqp.useDefault) {
             for (i=0;i<getNbBounds(aqp);i++) {
                bit(1)       useEfficientCoding
                GenericFloat aqp.Imin[i](useEfficientCoding);
             }
             for (i=0;i<getNbBounds(aqp);i++)
                bit(1)       useEfficientCoding
                GenericFloat aqp.Imax[i](useEfficientCoding);
          }
          break;

       case 13:   // Integers
          int(32)    aqp.IminInt[0];
       break;

  }

  bit (1) hasINbBits;
  if (hasINbBits)
     unsigned int(5)         aqp.INbBits;

  bit (1) PMinMax ;
  if (PMinMax) {
     for (i=0;i<getNbBounds(aqp);i++) {
           int(INbBits+1) vq
           aqp.Pmin[i] = vq-2^aqp.INbBits;
     }
   }

  bit (1)hasPNbBits;
  if (hasPNbBits)
     unsigned int(4)         aqp.PNbBits;

}
```

9.3.8.8.2 Semantics

The AnimQP specifies the quantization parameters that shall be used until the next intra frame is received. AnimQP is identical to InitialAnimQP (subclause 9.3.5.6) with the exception that each quantization parameter may or may not be sent.

If BIFSConfig.randomAccess is TRUE and if the parameter is not coded, then the parameter defined in the InitialAnimQP in the AnimationMask is used by default.

If BIFSConfig.randomAccess is FALSE and if the parameter is not coded, then the parameter defined in the latest AnimQP (or InitialAnimQP if this parameter was never modified) is used.

9.3.8.9 AnimIValue

9.3.8.9.1 Syntax

```
class AnimIValue(FieldData field) {
   switch (field.animType) {
      case  9:   // Normal
         int(1)       direction
      case 10:   // Rotation
         int(2)       orientation
         break;
      default:
         break;
    }
   for (j=0;j<getNbComp(field);j++)
         int(field.nbBits)  vq[j];
}
```

9.3.8.9.2 Semantics

The AnimIValue represents the quantized intra value of a field. The value is coded according to the quantization process described in 9.3.3.3.

For normals the direction and orientation values specified in the quantization process are first coded. For rotations only the orientation value is coded. If the bit representing the direction is 0, the normal's direction is set to 1, if the bit is 1, the normal's direction is set to –1. The value of the orientation is coded as an unsigned integer using 2 bits.

The compressed components vq[i] of the field's value are then coded as a sequence of unsigned integers using the number of bits specified in the field data structure.

The decoding process in intra mode computes the animation values by applying the inverse quantization process.

9.3.8.10 AnimPValue

9.3.8.10.1 Syntax

```
class AnimPValue(FieldData field) {
   switch (field.animType) {
      case  9:   // Normal
         int(1)     inverse
         break;
      default:
         break;
    }
   for (j=0;j<getNbComp(field);j++)
   int(aacNbBits)  vqDelta[j];
}
```

9.3.8.10.2 Semantics

The `AnimPValue` represents the difference between the previously received quantized value and the current quantized value of a field. The value is coded using the compensation process `AddDelta` described in 9.3.4.

The values are decoded from the adaptive arithmetic coder bitstream with the procedure v_{aac}=`aa_decode(model)` defined in Annex G. The model is updated with the procedure `model_update(model,` v_{aac}`).`

For normals the inverse value is decoded through the adaptive arithmetic coder with a uniform, non-updated model. If the bit is 0, then `inverse` is set to 1, the bit it is 1, `inverse` is set to –1.

The compensation values `vqDelta[i]` are then decoded in sequence. Let $v_q(t-1)$ be the quantized value decoded at the previous frame and $v_{aac}(t)$ be the value decoded by the frame's adaptive arithmetic decoder at instant *t* with the field's models. The value a time *t* is obtained from the previous value as follows:

$$v_\delta(t) = v_{aac}(t) + PMin$$
$$v_q(t) = \mathrm{AddDelta}(v_q(t-1), v_\delta(t))$$
$$v(t) = \mathrm{InvQuant}\left(v_q(t)\right)$$

The field's models are updated each time a value is decoded through the adaptive arithmetic coder.

If the `animType` is `1` (`Position3D`) or `2` (`Position2D`), each component of the field's value is using its own model and offset `PMin[i].` In all other cases the same model and offset `PMin[0]` is used for all the components.

`aacNbBits` is the variable number of bits needed for the adaptive arithemtic coder to decode the symbol (see Annex G).

9.4 Node Semantics

9.4.1 Overview

The BIFS nodes include nodes that have been defined in ISO/IEC 14772-1:1998 [10]. For these nodes, the semantic information is given by normative reference with any restrictions defined herein.

9.4.2 Node specifications

9.4.2.1 Anchor

9.4.2.1.1 Node interface

```
Anchor {
    eventIn         MFNode      addChildren
    eventIn         MFNode      removeChildren
    exposedField    MFNode      children            []
    exposedField    SFString    description         ""
    exposedField    MFString    parameter           []
    exposedField    MFString    url                 []
}
```

NOTE — For the binary encoding of this node see Annex H.1.1.

9.4.2.1.2 Functionality and semantics

The semantics of the **Anchor** node are specified in ISO/IEC 14772-1:1998, subclause 6.2 [10]. ISO/IEC 14496-1 does not support the bounding box parameters (**bboxCenter** and **bboxSize**). Constraints on URLs are defined by profiles and levels.

9.4.2.2 AnimationStream

9.4.2.2.1 Node interface

```
AnimationStream {
    exposedField    SFBool      loop        FALSE
    exposedField    SFFloat     speed       1.0
    exposedField    SFTime      startTime   0
    exposedField    SFTime      stopTime    0
    exposedField    MFString    url         [""]
    eventOut        SFBool      isActive
}
```

NOTE — For the binary encoding of this node see Annex H.1.2.

9.4.2.2.2 Functionality and semantics

The **AnimationStream** node is designed to implement control parameters for a BIFS-Anim stream.

The **loop**, **startTime**, and **stopTime** exposedFields and the **isActive** eventOut, and their effects on the **AnimationStream** node are described in 9.2.1.6.1.

The semantics of the **speed** exposedField are identical to those for the **MovieTexture** node (see 9.4.2.61).

The **url** field specifies the data source to be used. The data source referred to shall be a BIFS-Anim stream (see also 9.2.3.3).

9.4.2.3 Appearance

9.4.2.3.1 Node interface

```
Appearance {
    exposedField    SFNode      material            NULL
    exposedField    SFNode      texture             NULL
    exposedField    SFNode      textureTransform    NULL
}
```

NOTE — For the binary encoding of this node see Annex H.1.3.

9.4.2.3.2 Functionality and semantics

The semantics of the **Appearance** node are specified in ISO/IEC 14772-1:1998, subclause 6.3 [10].

The **material** field, if non-NULL, shall contain either a **Material** node or a **Material2D** node depending on the type of geometry node used in the geometry field of the **Shape** node that contains the **Appearance** node. The list below shows the geometry nodes that require a **Material** node, those that require a **Material2D** node and those where either may apply:

— **Material2D** only: **Circle, Curve2D, IndexedFaceSet2D, IndexedLineSet2D, PointSet2D, Rectangle**;

— **Material** only: **Box, Cone, Cylinder, ElevationGrid, Extrusion, IndexedFaceSet, IndexedLineSet, PointSet, Sphere;**

— **Material2D** or **Material**: **Bitmap, TermCap, Text.**

9.4.2.4 AudioBuffer

9.4.2.4.1 Node interface

```
AudioBuffer {
    exposedField   SFBool     loop               FALSE
    exposedField   SFFloat    length             0.0
    exposedField   SFFloat    pitch              1.0
    exposedField   SFTime     startTime          0
    exposedField   SFTime     stopTime           0
    exposedField   MFNode     children           []
    exposedField   SFInt      numChan            1
    exposedField   MFInt      phaseGroup         [1]
    eventOut       SFTime     duration_changed
    eventOut       SFBool     isActive
}
```

NOTE — For the binary encoding of this node see Annex H.1.4.

9.4.2.4.2 Functionality and semantics

The **AudioBuffer** node provides an interface to short snippets of sound to be used in an interactive scene.

EXAMPLE — Sounds triggered as “auditory icons” upon mouse clicks.

It buffers the audio generated by its children to support random restart capability upon interaction events. It differs from the **AudioClip** node in the following ways:

— **AudioBuffer** can be used in broadcast and other one-way applications in which URLs from remote locations cannot be retrieved interactively

— **AudioBuffer** can be used to trigger sounds made from processed sound (ie, with the other sound nodes) rather than only raw sound data as transmitted in the elementary stream

The **loop**, **startTime**, and **stopTime** exposedFields and the **isActive** eventOut, and their effects on the **AnimationStream** node are described in 9.2.1.6.1.

The semantics of the **speed** exposedField are identical to those for the **MovieTexture** node (see 9.4.2.61).

The **length** field specifies the length in seconds of the audio buffer. Audio shall be buffered at the instantiation of the node, and whenever the **length** field changes.

The **pitch** field specifies a pitch-shift to apply to the output sound. The pitch-shift is calculated by simple resampling; that is, a pitch-shift of 2 corresponds to playing the sound twice as fast and an octave higher. If **pitch** is negative, the buffer is played backwards at the indicated speed, beginning at the last sample in the buffer and proceeding to the first, then returning to the last sample if **loop** is TRUE.

The **children** field specifies the child nodes that provide the sound for this node. Each child shall be an AudioBIFS node; that is, one of the following: **AudioSource**, **AudioDelay**, **AudioMix**, **AudioSwitch**, **AudioFX**, **AudioClip** or **AudioBuffer**.

An event shall be generated via the **duration_changed** field whenever a change is made to the **startTime** or **stopTime** fields. An event shall also be triggered if these fields are changed simultaneously, even if the duration does not actually change.

The **numChan** field specifies the number of output channels of this node. If there are more output channels than input channels, the “extra” channels shall contain all 0s; if there are more input channels than output channels, the “extra” channels shall be ignored.

The **phaseGroup** field specifies phase relationships in the output of the node, see 9.2.2.13 and 9.4.2.9.

The output of this node is not calculated based on the current input values, but according to the **startTime** event, the **pitch** field and the contents of the clip buffer. When the **startTime** is reached (that is, the current scene time is greater than or equal to **startTime**), the sound output shall begin at the beginning of the clip buffer and **isActive** shall be set to TRUE. At each time step thereafter, the value of the output buffer shall be the value of the next portion of the clip buffer, upsampled or downsampled as necessary according to **pitch**. When the end of the clip buffer according to the value of **length** is reached, if **loop** is TRUE, the audio shall begin again from the beginning of the clip buffer; if **loop** is FALSE, the playback shall cease. This playback shall be continued until **stopTime** is reached. When the current scene time is greater than or equal to **stopTime**, the node shall cease to produce sound.

The clip buffer shall be calculated as follows. When the node is instantiated, or whenever the **length** field is changed, the first **length** seconds of the audio input to the **AudioBuffer** node shall be copied to the clip buffer. That is, after t seconds, where t < length, audio sample number t * S of channel i (where 0 <= i < **numChan**) in the buffer is set to contain the audio sample corresponding to time t of channel i of the input, where S is the sampling rate of this node. After the first length seconds, the input to this node has no effect. Changes to the **length** field that are received when **isActive** is TRUE shall be ignored.

When the playback is not active, the audio output of the node is all 0s.

9.4.2.5 AudioClip

9.4.2.5.1 Node interface

```
AudioClip {
   exposedField  SFString   description       ""
   exposedField  SFBool     loop              FALSE
   exposedField  SFFloat    pitch             1.0
   exposedField  SFTime     startTime         0
   exposedField  SFTime     stopTime          0
   exposedField  MFString   url               []
   eventOut      SFTime     duration_changed
   eventOut      SFBool     isActive
}
```

NOTE — For the binary encoding of this node see Annex H.1.5.

9.4.2.5.2 Functionality and semantics

The semantics of the **Audioclip** node are specified in ISO/IEC 14772-1:1998, subclause 6.4 [10].

The **loop**, **startTime**, and **stopTime** exposedFields and the **isActive** eventOut, and their effects on the **AudioClip** node are described in 9.2.1.6.1.

The **url** field specifies the data source to be used (see 9.2.2.7.1).

9.4.2.6 AudioDelay

9.4.2.6.1 Node interface

```
AudioDelay {
   eventIn       MFNode     addChildren
   eventIn       MFNode     removeChildren
   exposedField  MFNode     children          []
   exposedField  SFTime     delay             0
   field         SFInt32    numChan           1
   field         MFInt32    phaseGroup        []
}
```

NOTE — For the binary encoding of this node see Annex H.1.6.

9.4.2.6.2 Functionality and semantics

The **AudioDelay** node allows sounds to be started and stopped under temporal control. The start time and stop time of the child sounds are delayed or advanced accordingly.

The **addChildren** eventIn specifies a list of nodes that shall be added to the **children** field.

The **removeChildren** eventIn specifies a list of nodes that shall be removed from the **children** field.

The **children** array specifies the nodes affected by the delay. Each child shall be an AudioBIFS node; that is, one of the following: **AudioSource**, **AudioDelay**, **AudioMix**, **AudioSwitch**, **AudioFX**, **AudioClip** or **AudioBuffer**.

The **delay** field specifies the delay to apply to each child node.

The **numChan** field specifies the number of channels of audio output by this node.

The **phaseGroup** field specifies the phase relationships among the various output channels; see 9.2.1.6.1.

Implementation of the **AudioDelay** node requires the use of a buffer of size $d * S * n$, where d is the length of the delay in seconds, S is the sampling rate of the node, and n is the number of output channels from this node. At scene startup, a multichannel delay line of length d and width n is initialized to reside in this buffer.

At each time step, the $k * S$ audio samples in each channel of the input buffer, where k is the length of the system time step in seconds, are inserted into this delay line. If the number of input channels is strictly greater than the number of output channels, the extra input channels are ignored; if the number of input channels is strictly less than the number of output channels, the extra channels of the delay line shall be taken as all 0's.

The output buffer of the node is the $k * S$ audio samples which fall off the end of the delay line in this process. Note that this definition holds regardless of the relationship between k and d.

If the **delay** field is updated during playback, discontinuties (audible artefacts or "clicks") in the output sound may result. If the **delay** field is updated to a greater value than the current value, the delay line is immediately extended to the new length, and zero values inserted at the beginning, so that $d * S$ seconds later there will be a short gap in the output of the node. If the **delay** field is updated to a lesser value than the current value, the delay line is immediately shortened to the new length, truncating the values at the end of of the line, so that there is an immediate discontinuity in sound output. Manipulation of the **delay** field in this manner is not recommended unless the audio is muted within the terminal or by appropriate use of an **AudioMix** node at the same time, since it gives rise to impaired sound quality.

9.4.2.7 AudioFX

9.4.2.7.1 Node interface

```
AudioFX {
    eventIn        MFNode      addChildren
    eventIn        MFNode      removeChildren
    exposedField   MFNode      children           []
    exposedField   SFString    orch               ""
    exposedField   SFString    score              ""
    exposedField   MFFloat     params             []
    field          SFInt32     numChan            1
    field          MFInt32     phaseGroup         []
}
```

NOTE — For the binary encoding of this node see Annex H.1.7.

9.4.2.7.2 Functionality and semantics

The **AudioFX** node is used to allow arbitrary signal-processing functions defined using structured audio tools to be included and applied to its **children** (see ISO/IEC 14496-3 section 5, clause 5.15).

The **addChildren** eventIn specifies a list of nodes that shall be added to the **children** field.

The **removeChildren** eventIn specifies a list of nodes that shall be removed from the **children** field.

The **children** array contains the nodes operated upon by this effect. Each child shall be an AudioBIFS node; that is, one of the following: **AudioSource**, **AudioDelay**, **AudioMix**, **AudioSwitch**, **AudioFX**, **AudioClip** or **AudioBuffer**. If this array is empty, the node has no function (the node may not be used to create new synthetic audio in the middle of a scene graph).

The **orch** string contains a tokenised block of signal-processing code written in SAOL (Structured Audio Orchestra Language). This code block shall contain an orchestra header and some instrument definitions, and conform to the bitstream syntax of the orchestra class as defined in ISO/IEC 14496-3 section 5 subclause 5.5.2.2 and clause 5.8.

The **score** string may contain a tokenized score for the given orchestra written in SASL (Structured Audio Score Language). This score may contain control operators to adjust the parameters of the orchestra, or even new instrument instantiations. A **score** is not required. If present it shall conform to the bitstream syntax of the `score_file` class as defined in ISO/IEC 14496-3 section 5 subclause 5.5.2 and clause 5.11.

The **params** field allows BIFS commands and events to affect the sound-generation process in the orchestra. The values of **params** are available to the FX orchestra as the global array `global ksig params[128]`; see ISO/IEC 14496-3 section 5 clause 5.15.

The **numchan** field specifies the number of channels of audio output by this node.

The **phaseGroup** field specifies the phase relationships among the various output channels; see 9.2.1.6.1.

The node is evaluated according to the semantics of the orchestra code contained in the **orch** field. See ISO/IEC 14496-3, section 5, for the normative description of this process. Within the orchestra code, the multiple channels of input sound are placed on the global bus, *input_bus*; first, all channels of the first child, then all the channels of the second child, and so on. The orchestra header shall 'send' this bus to an instrument for processing. The **phaseGroup** arrays of the children are made available as the *inGroup* variable within the instrument(s) to which the *input_bus* is sent.

The orchestra code block shall not contain the `spatialize` statement.

The output buffer of this node is the sound produced as the final output of the orchestra applied to the input sounds, as described in ISO/IEC 14496-3 section 5, subclauses 5.7.3.

9.4.2.8 AudioMix

9.4.2.8.1 Node interface

```
AudioMix {
    eventIn        MFNode     addChildren
    eventIn        MFNode     removeChildren
    exposedField   MFNode     children          []
    exposedField   SFInt32    numInputs         1
    exposedField   MFFloat    matrix            []
    field          SFInt32    numChan           1
    field          MFInt32    phaseGroup        []
}
```

NOTE — For the binary encoding of this node see Annex H.1.8.

9.4.2.8.2 Functionality and semantics

This node is used to mix together several audio signals in a simple, multiplicative way. Any relationship that may be specified in terms of a mixing matrix may be described using this node.

The **addChildren** eventIn specifies a list of nodes that shall be added to the **children** field.

The **removeChildren** eventIn specifies a list of nodes that shall be removed from the **children** field.

The **children** field specifies which nodes' outputs to mix together. Each child shall be an AudioBIFS node; that is, one of the following: **AudioSource**, **AudioDelay**, **AudioMix**, **AudioSwitch**, **AudioFX**, **AudioClip** or **AudioBuffer**.

The **numInputs** field specifies the number of input channels. It shall be the sum of the number of channels of the children.

The **matrix** array specifies the mixing matrix which relates the inputs to the outputs. **matrix** is an unrolled **numInputs** x **numChan** matrix which describes the relationship between **numInputs** input channels and **numChan** output channels. The **numInputs** * **numChan** values are in row-major order. That is, the first **numInputs** values are the scaling factors applied to each of the inputs to produce the first output channel; the next **numInputs** values produce the second output channel, and so forth.

That is, if the desired mixing matrix is $\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$, specifying a "2 into 3" mix, the value of the **matrix** field shall be [*a b c d e f*].

The **numchan** field specifies the number of channels of audio output by this node.

The **phaseGroup** field specifies the phase relationships among the various output channels; see 9.2.1.6.1.

The value of the output buffer for an **AudioMix** node is calculated as follows. For each sample number *x* of output channel *i*, 1 <= *i* <= **numChan**, the value of that sample is

matrix[(0) * **numChan** + **i**] * input[1][*x*] +

matrix[(1) * **numChan** + **i**] * input[2][*x*] + ...

matrix[(**numInputs** – 1) * **numChan** + **i**] * input[**numInputs**][*x*],

where input[*i*][*j*] represents the *j*th sample of the *i*th channel of the input buffer, and the **matrix** elements are indexed starting from 1.

9.4.2.9 AudioSource

9.4.2.9.1 Node interface

```
AudioSource {
    eventIn         MFNode      addChildren
    eventIn         MFNode      removeChildren
    exposedField    MFNode      children            []
    exposedField    MFString    url                 []
    exposedField    SFFloat     pitch               1.0
    exposedField    SFFloat     speed               1.0
    exposedField    SFTime      startTime           0
    exposedField    SFTime      stopTime            0
    field           SFInt32     numChan             1
    field           MFInt32     phaseGroup          []
}
```

NOTE — For the binary encoding of this node see Annex H.1.9.

9.4.2.9.2 Functionality and semantics

This node is used to add sound to a BIFS scene. See ISO/IEC 14496-3 for information on the various audio tools available for coding sound.

The **addChildren** eventIn specifies a list of nodes that shall be added to the **children** field.

The **removeChildren** eventIn specifies a list of nodes that shall be removed from the **children** field.

The **children** field allows buffered **AudioBuffer** data to be used as sound samples within a structured audio decoding process. Only **AudioBuffer** nodes shall be children to an **AudioSource** node, and only in the case where **url** indicates a structured audio bitstream.

The **pitch** field controls the playback pitch for the structured audio and the parametric speech (HVXC) decoder. It is specified as a ratio, where 1 indicates the original bitstream pitch, values other than 1 indicate pitch-shifting by the given ratio. This field is available through the gettune() core opcode in the structured audio decoder (see ISO/IEC 14496-3, section 5). The structured audio is the only decoder that may be controlled in this manner; to adjust the pitch of other decoder types, use the **AudioFX** node with an appropriate effects orchestra.

The **speed** field controls the playback speed for the structured audio decoder (see ISO/IEC 14496-3, section 5). It is specified as a ratio, where 1 indicates the original speed; values other than 1 indicate multiplicative time-scaling by the given ratio (i.e. 0.5 specifies twice as fast). The value of this field shall be made available to the structured audio decoder indicated by the **url** field. ISO/IEC 14496-3, section 5, subclause 5.7.3.3.6, list item 8, describe the use of this field to control the structured audio decoder. The structured audio decoder is the only decoder that may be controlled in this manner; to adjust the speed of other decoder types, use the **AudioFX** node with an appropriate orchestra.

The **startTime** and **stopTime** exposedFields and their effects on the **AudioSource** node are described in 9.2.1.6.1.

The **numChan** field describes how many channels of audio are in the decoded bitstream.

The **phaseGroup** array specifies whether or not there are important phase relationships between the multiple channels of audio. If there are such relationships – for example, if the sound is a multichannel spatialized set or a “stereo pair” – it is in general dangerous to do anything more complex than scaling to the sound. Further filtering or repeated “spatialization” will destroy these relationships. The values in the array divide the channels of audio into groups; if **phaseGroup[i] = phaseGroup[j]** then channel **i** and channel **j** are phase-related. Channels for which the **phaseGroup** value is 0 are not related to any other channel.

The **url** field specifies the data source to be used (see 9.2.2.7.1).

The audio output from the decoder according to the bitstream(s), referenced in the specified URL, at the current scene time is placed in the output buffer for this node, unless the current scene time is earlier than the current value of **startTime** or later than the current value of **stopTime**, in which case 0 values are placed in the output buffer for this node for the current scene time.

For audio sources decoded using the main object of the structured audio decoder (ISO/IEC 14496-3, section 5), several variables from the scene description must be mapped into standard names in the orchestra. See ISO/IEC 14496-3, section 5, clause 5.15 and subclause 5.8.6.8.

If **AudioClip** children are provided for a structured audio decoder, the audio data buffered in the **AudioClip**(s) must be made available to the decoding process. See Subclause ISO/IEC 14496-3, section 5, subclause 5.10.2.

9.4.2.10 AudioSwitch

9.4.2.10.1 Node interface

```
AudioSwitch {
    eventIn        MFNode     addChildren
    eventIn        MFNode     removeChildren
    exposedField   MFNode     children          []
    exposedField   MFInt32    whichChoice       []
    field          SFInt32    numChan           1
    field          MFInt32    phaseGroup        []
}
```

NOTE — For the binary encoding of this node see Annex H.1.10.

9.4.2.10.2 Functionality and semantics

The **AudioSwitch** node is used to select a subset of audio channels from the child nodes specified.

The **addChildren** eventIn specifies a list of nodes that shall be added to the **children** field.

The **removeChildren** eventIn specifies a list of nodes that shall be removed from the **children** field.

The **children** field specifies a list of child options. Each child shall be an AudioBIFS node; that is, one of the following: **AudioSource**, **AudioDelay**, **AudioMix**, **AudioSwitch**, **AudioFX**, **AudioClip** or **AudioBuffer**.

The **whichChoice** field specifies which channels shall be passed through. If **whichChoice[i]** is 1, then the i-th child channel shall be passed through.

The **numchan** field specifies the number of channels of audio output by this node; ie, the number of channels in the passed child.

The **phaseGroup** field specifies the phase relationships among the various output channels; see 9.2.1.6.1.

The values for the output buffer are calculated as follows:

For each sample number *x* of channel number *i* of the output buffer, 1 <= *i* <= **numChan**, the value in the buffer is the same as the value of sample number *x* in the *j*th channel of the input, where *j* is the least value such that **whichChoice[0] + whichChoice[1] + ... + whichChoice[*j*] =** *i*.

9.4.2.11 Background

9.4.2.11.1 Node interface

```
Background {
    eventIn        SFBool     set_bind
    exposedField   MFFloat    groundAngle       []
    exposedField   MFColor    groundColor       []
    exposedField   MFString   backURL           []
    exposedField   MFString   bottomURL         []
    exposedField   MFString   frontURL          []
    exposedField   MFString   leftURL           []
    exposedField   MFString   rightURL          []
    exposedField   MFString   topURL            []
    exposedField   MFFloat    skyAngle          []
    exposedField   MFColor    skyColor          0, 0, 0
    eventOut       SFBool     isBound
}
```

NOTE — For the binary encoding of this node see Annex H.1.11.

9.4.2.11.2 Functionality and semantics

The semantics of the **Background** node are specified in ISO/IEC 14772-1:1998, subclause 6.5 [10].

The **backUrl**, **bottomURL**, **frontUrl**, **leftUrl**, **rightUrl**, **topUrl** fields specify the data sources to be used (see 9.2.2.7.1).

9.4.2.12 Background2D

9.4.2.12.1 Node interface

```
Background2D {
    eventIn       SFBool      set_bind
    exposedField  SFColor     backColor             0 0 0
    exposedField  MFString    url                   []
    eventOut      SFBool      isBound
}
```

NOTE — For the binary encoding of this node see Annex H.1.12.

9.4.2.12.2 Functionality and semantics

There exists a **Background2D** stack, in which the top-most background is the current active background one. The **Background2D** node allows a background to be displayed behind a 2D scene. The functionality of this node can also be accomplished using other nodes, but use of this node may be more efficient in some implementations.

If **set_bind** is set to TRUE the **Background2D** is moved to the top of the stack.If **set_bind** is set to FALSE, the **Background2D** is removed from the stack so the previous background which is contained in the stack is on top again.

The **isBound** event is sent as soon as the backdrop is put at the top of the stack, so becoming the current backdrop.

The **url** field specifies the data source to be used (see 9.2.2.7.1).

The **backColor** field specifies a colour to be used as the background.

This is not a geometry node and the top-left corner of the image is displayed at the top-left corner of the screen, regardless of the current transformation. Scaling and/or rotation do not have any effect on this node.

EXAMPLE — Changing the background for 5 seconds.

```
Group {
   children [
      …
      DEF TIS TimeSensor {
         startTime 5.0
         stopTime  10.0
      }
      DEF BG1 Background2D {
         …
      }
   ]
}
ROUTE TIS.isActive TO BG1.set_bind
```

9.4.2.13 Billboard

9.4.2.13.1 Node interface

```
Billboard {
    eventIn         MFNode      addChildren
    eventIn         MFNode      removeChildren
    exposedField    SFVec3f     axisOfRotation      0, 1, 0
    exposedField    MFNode      children            []
}
```

NOTE — For the binary encoding of this node see Annex H.1.13.

9.4.2.13.2 Functionality and semantics

The semantics of the **Billboard** node are specified in ISO/IEC 14772-1:1998, subclause 6.6 [10]. ISO/IEC 14496-1 does not support the bounding box parameters (**bboxCenter** and **bboxSize**).

9.4.2.14 Bitmap

9.4.2.14.1 Node interface

```
Bitmap {
    exposedField    SFVec2f     scale               -1, -1
}
```

NOTE — For the binary encoding of this node see Annex H.1.14.

9.4.2.14.2 Functionality and semantics

Bitmap is a geometry node, to be placed in the geometry field of a **Shape** node. In general, it is a screen-aligned rectangle with the dimensions of the texture that is mapped onto it, as specified in the **Appearance** node of its parent **Shape** node. However, the effective geometry of **Bitmap** is defined by the non-transparent pixels of the image or video that is mapped onto it. When no scaling is specified, a trivial texture-mapping (pixel copying) is performed.

The **scale** field specifies a scaling of the geometry in the x and y dimensions, respectively. The **scale** values shall be strictly positive or equal to -1. A **scale** value of -1 indicates that no scaling shall be applied in the relevant dimension. The special case where both scale dimensions are -1 indicates that the natural dimensions of the texture that is mapped onto the **Bitmap** shall be used.

Bitmap shall not be rotated but may be subject to translation.

Geometry sensors shall respond to the effective geometry of the **Bitmap**, which is defined by the non-transparent pixels of the texture that is mapped onto it.

Example — To specify semi-transparent video:

```
Shape {
   appearance Appearance {
      texture MovieTexture {  // Visual object
         …
      }
      material Material2D {
         transparency  0.5   // semi-transparent
      }
   }
   geometry Bitmap {}
}
```

9.4.2.15 Box

9.4.2.15.1 Node interface

```
Box {
    field          SFVec3f     size                  2, 2, 2
}
```

NOTE — For the binary encoding of this node see Annex H.1.15.

9.4.2.15.2 Functionality and semantics

The semantics of the **Box** node are specified in ISO/IEC 14772-1:1998, subclause 6.7 [10].

9.4.2.16 Circle

9.4.2.16.1 Node interface

```
Circle {
    exposedField   SFFloat     radius                1.0
}
```

NOTE — For the binary encoding of this node see Annex H.1.16.

9.4.2.16.2 Functionality and semantics

This node specifies a circle centred at (0,0) in the local coordinate system. The **radius** field specifies the radius of the circle and shall be greater than 0.

9.4.2.17 Collision

9.4.2.17.1 Node interface

```
Collision {
    eventIn        MFNode      addChildren
    eventIn        MFNode      removeChildren
    exposedField   MFNode      children              []
    exposedField   SFBool      collide               TRUE
    field          SFNode      proxy                 NULL
    eventOut       SFTime      collideTime
}
```

NOTE — For the binary encoding of this node see Annex H.1.17.

9.4.2.17.2 Functionality and semantics

The semantics of the Collision node are specified in ISO/IEC 14772-1:1998, subclause 6.8 [10]. ISO/IEC 14496-1 does not support the bounding box parameters (bboxCenter and bboxSize).

9.4.2.18 Color

9.4.2.18.1 Node interface

```
Color {
    exposedField   MFColor     color                 []
}
```

NOTE — For the binary encoding of this node see Annex H.1.18.

9.4.2.18.2 Functionality and semantics

The semantics of the **Color** node are specified in ISO/IEC 14772-1:1998, subclause 6.9 [10].

9.4.2.19 ColorInterpolator

9.4.2.19.1 Node interface

```
ColorInterpolator {
    eventIn         SFFloat     set_fraction
    exposedField    MFFloat     key                    []
    exposedField    MFColor     keyValue               []
    eventOut        SFColor     value_changed
}
```

NOTE — For the binary encoding of this node see Annex H.1.19.

9.4.2.19.2 Functionality and semantics

The semantics of the **ColorInterpolator** node are specified in ISO/IEC 14772-1:1998, subclause 6.10 [10].

9.4.2.20 CompositeTexture2D

9.4.2.20.1 Node interface

```
CompositeTexture2D {
    eventIn         MFNode      addChildren
    eventIn         MFNode      removeChildren
    exposedField    MFNode      children               []
    exposedField    SFInt32     pixelWidth             -1
    exposedField    SFInt32     pixelHeight            -1
    exposedField    SFNode      background             NULL
    exposedField    SFNode      viewport               NULL
}
```

NOTE — For the binary encoding of this node see Annex H.1.20.

9.4.2.20.2 Functionality and semantics

The **CompositeTexture2D** node represents a texture that is composed of a 2D scene, which may be mapped onto another object.

This node may only be used as the texture field of an **Appearance** node. All behaviors and user interaction are enabled when using a **CompositeTexture2D**.

The **addChildren** eventIn specifies a list of nodes that shall be added to the **children** field.

The **removeChildren** eventIn specifies a list of nodes that shall be removed from the **children** field.

The **children** field contains a list of 2D children nodes that define the 2D scene that is to form the texture map.

The **pixelWidth** and **pixelHeight** fields specify the ideal size in pixels of this map. The default values result in an undefined size being used. This is a hint for the content creator to define the quality of the texture mapping.

The semantics of the **background** and **viewport** fields are identical to the semantics of the **Layer2D** (see 9.4.2.53) fields of the same name.

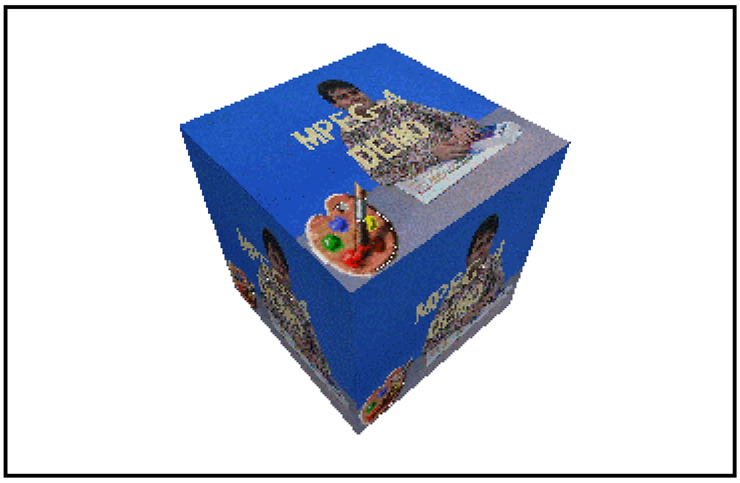


Figure 15 - A CompositeTexture2D **example. The 2D scene is projcted onto the 3D cube.**

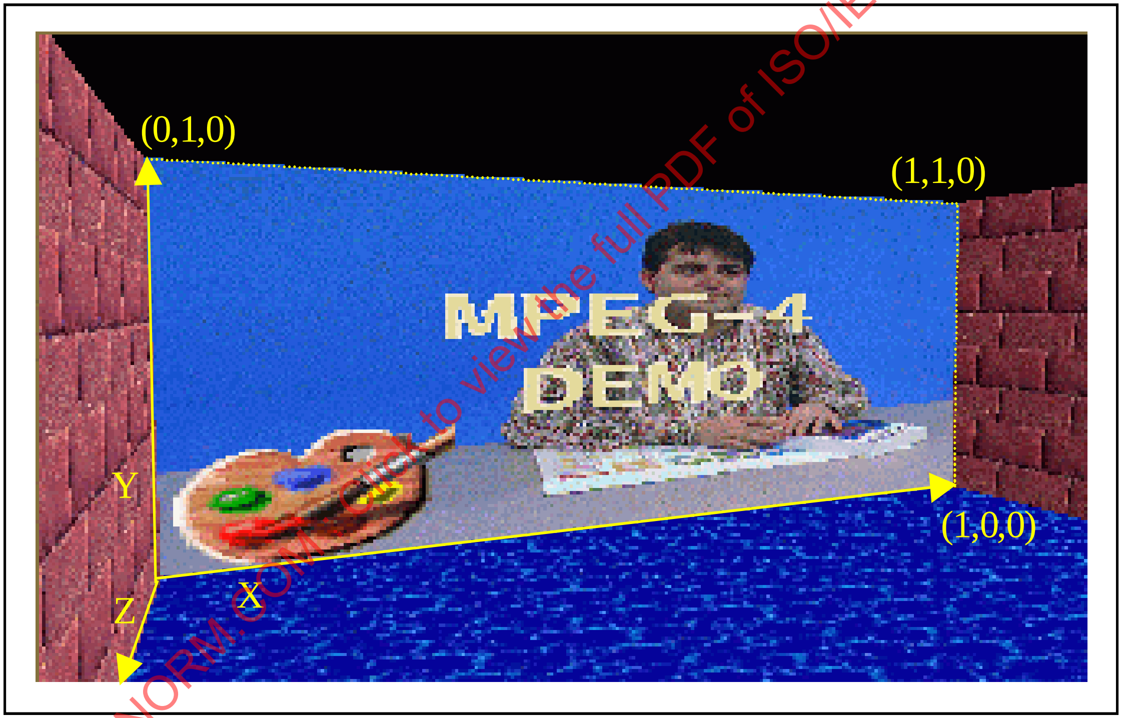


Figure 16 - A CompositeTexture2D **example.**

Here the 2D scene as defined in Figure 15 composed of an image, a logo, and a text, is textured on a rectangle n in the local X,Y plane of the back wall. A similar effect may be obtained by simply placing the 2D objects in the (3D) **Transform**. However, **CompositeTexture2D** and **CompositeTexture3D** shall be used when maping onto non-flat geometries.

9.4.2.21 CompositeTexture3D

9.4.2.21.1 Node interface

```
CompositeTexture3D {
    eventIn        MFNode        addChildren
    eventIn        MFNode        removeChildren
    exposedField   MFNode        children                  []
    exposedField   SFInt32    pixelWidth          -1
    exposedField   SFInt32    pixelHeight         -1
    exposedField   SFNode     background
    exposedField   SFNode     fog
    exposedField   SFNode     navigationInfo
    exposedField   SFNode     viewpoint
}
```

NOTE — For the binary encoding of this node see Annex H.1.21.

9.4.2.21.2 Functionality and semantics

The **CompositeTexture3D** node represents a texture mapped onto a 3D object that is composed of a 3D scene.

Behaviors and user interaction are enabled when using a **CompositeTexture3D**. However, the standard user navigation on the textured scene is disabled. Instead, sensors contained in the scene which forms the **CompositeTexture3D** may be used to define behaviours. This node may only be used as a **texture** field of an **Appearance** node.

The **addChildren** eventIn specifies a list of nodes that shall be added to the **children** field.

The **removeChildren** eventIn specifies a list of nodes that shall be removed from the **children** field.

The **children** field is the list of 3D children nodes that define the 3D scene that forms the texture map.

The **pixelWidth** and **pixelHeight** fields specify the ideal size in pixels of this map. The default values result in an undefined size being used. This is a hint for the content creator to define the quality of the texture mapping.

The **background**, **fog**, **navigationInfo** and **viewpoint** fields represent the current values of the bindable children nodes used in the 3D scene. This node may only be used as the **texture** field of an **Appearance** node. All behaviors and user interaction are enabled when using a **CompositeTexture2D**.

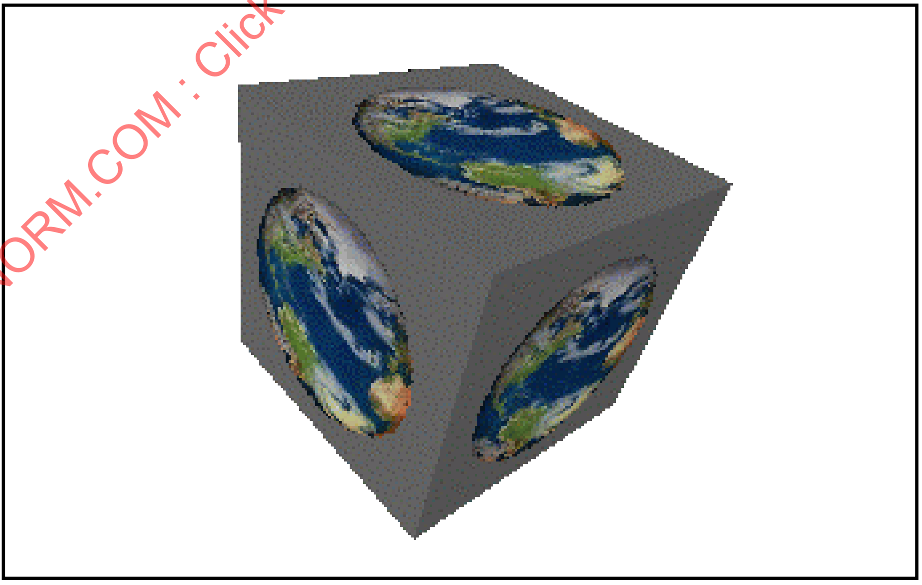


Figure 17 - CompositeTexture3D example. The 3D view of the earth is projected onto the 3D cube

9.4.2.22 Conditional

9.4.2.22.1 Node interface

```
Conditional {
    eventIn         SFBool      activate
    eventIn         SFBool      reverseActivate
    exposedField    SFString    buffer                    ""
    eventOut        SFBool      isActive
}
```

NOTE — For the binary encoding of this node see Annex H.1.22.

9.4.2.22.2 Functionality and semantics

The **Conditional** node interprets a buffered bit string of BIFS-Commands when it is activated. This allows events to trigger node updates, deletions, and other modifications to the scene. The buffered bit string is interpreted as if it had just been received.

Upon reception of either an SFBool event of value TRUE on the **activate** eventIn, or an SFBool event of value FALSE on the **reverseActivate** eventIn, the contents of the buffer field shall be interpreted as a BIFS `CommandFrame` (see 9.3.6.2). These updates are not time-stamped; they are executed at the time of the event, assuming a zero-decoding time.

EXAMPLE — A typical use of this node is for the implementation of the action of a button. The button geometry is enclosed in a grouping node which also contains a **TouchSensor** node. The **isActive** eventOut of the **TouchSensor** is routed to the activate eventIn of **Conditional** C1 and to the **reverseActivate** eventIn of **Conditional** C2; C1 then implements the “mouse-down” action and C2 implements the “mouse-up” action.

9.4.2.23 Cone

9.4.2.23.1 Node interface

```
Cone {
    field           SFFloat     bottomRadius              1.0
    field           SFFloat     height                    2.0
    field           SFBool      side                      TRUE
    field           SFBool      bottom                    TRUE
}
```

NOTE — For the binary encoding of this node see Annex H.1.23.

9.4.2.23.2 Functionality and semantics

The semantics of the **Cone** node are specified in ISO/IEC 14772-1:1998, subclause 6.11 [10].

9.4.2.24 Coordinate

9.4.2.24.1 Node interface

```
Coordinate {
    exposedField    MFVec3f     point                     []
}
```

NOTE — For the binary encoding of this node see Annex H.1.24.

9.4.2.24.2 Functionality and semantics

The semantics of the **Coordinate** node are specified in ISO/IEC 14772-1:1998, subclause 6.12 [10].

9.4.2.25 Coordinate2D

9.4.2.25.1 Node interface

```
Coordinate2D {
    exposedField    MFVec2f        point                    []
}
```

NOTE — For the binary encoding of this node see Annex H.1.25.

9.4.2.25.2 Functionality and semantics

This node defines a set of 2D coordinates to be used in the **coord** field of geometry nodes.

The **point** field contains a list of points in the 2D coordinate space (see 9.2.2.2).

9.4.2.26 CoordinateInterpolator

9.4.2.26.1 Node interface

```
CoordinateInterpolator {
    eventIn         SFFloat        set_fraction
    exposedField    MFFloat        key                      []
    exposedField    MFVec3f        keyValue                 []
    eventOut        MFVec3f        value_changed
}
```

NOTE — For the binary encoding of this node see Annex H.1.26.

9.4.2.26.2 Functionality and semantics

The semantics of the **CoordinateInterpolator** node are specified in ISO/IEC 14772-1:1998, subclause 6.13 [10].

9.4.2.27 CoordinateInterpolator2D

9.4.2.27.1 Node interface

```
CoordinateInterpolator2D {
    eventIn         SFFloat        set_fraction
    exposedField    MFFloat        key                      []
    exposedField    MFVec2f        keyValue                 []
    eventOut        MFVec2f        value_changed
}
```

NOTE — For the binary encoding of this node see Annex H.1.27.

9.4.2.27.2 Functionality and semantics

CoordinateInterpolator2D is the 2D equivalent of **CoordinateInterpolator** (see 9.4.2.26).

9.4.2.28 Curve2D

9.4.2.28.1 Node interface

```
Curve2D {
    exposedField    SFNode         point                    NULL
    exposedField    SFInt32        fineness                 0
    exposedField    MFInt32        type                     []
}
```

NOTE — For the binary encoding of this node see Annex H.1.28.

9.4.2.28.2 Functionality and semantics

This node is used to describe the Bezier approximation of a polygon in the scene at an arbitrary level of precision. It behaves as other “lines”, which means it is sensitive to modifications of line width and “dotted-ness”, and can be filled or not.

The given parameters are a control polygon and a parameter setting the quality of approximation of the curve. Internally, another polygon of fineness points is computed on the basis of the control polygon. The coordinates of that internal polygon are given by the following formula:

$$x[j] = \sum_{i=0}^{n} xc[i] \times \frac{(n-1)!}{i!(n-1-i)!} \times \left(\frac{j}{f}\right)^{i} \times \left(1-\frac{j}{f}\right)^{n-1-i},$$

where *x[j]* is the jth x coordinate of the internal polygon, *n* is the number of points in the control polygon, *xc[i]* is the ith x coordinate of the control polygon and *f* is short for the above fineness parameter which is also the number of points in the internal polygon. A similar formula yields the y coordinates.

The **point** field shall list the vertices of the control polygon.

The **fineness** parameter is an SFFloat value that indicates how finely to tessellate the Bezier curves. A value of 1 means that the curve shall be fine enough that no edges are visible. A value of 0 indicates that a straight line shall be drawn between the two points of the curve. The default value of 0.5 gives an intermediate level of smoothness. The amount of tessellation may be adjusted according to scale of the shape, making it possible to avoid visible edges appearing when the shape is zoomed. When the field **type** is specified, the above functionality is extended as follows: the curve is now defined piecewise either with the above equation or as straight segments or as non-segments, depending on the values in **type**. The **point** field is now taken to contain all key-points (points where the curve passes) and control-points (points defining the aspect of the curve around them). The values in the **type** field define the semantics of the elements of **point**.

The **point** field contains a **Coordinate2D** field with the list of points. If the **type** field is non-empty, then it shall contain tokens indicating how the point list is to be interpreted, according to the following algorithm (expressed in pseudo-code):

```
SFInt32 i     = 0;
SFInt32 j     = 0;
SFVec2f cur   = point[i++];
SFVec2f first = cur;
SFVec2f curctl;

while (i < point.length)
   SFInt32 t = 0;
   if (type.length > j) t = type[j++];

   switch(t) {
      case 0: // move, use 1 point
         if (is_filled) draw_line(cur, point[i]);
         cur = point[i];
         i++;
         break;

      case 1: // line, use 1 point
         draw_line(cur, point[i]);
         cur = point[i];
         i++;
         break;

      case 2: // bezier curve, use 3 points
         draw_curve(cur, point[i], point[i+1], point[i+2]);
         cur = point[i+2];
         curctl =point[i+1];
         i += 3;
         break;
      case 3: // tangent curve, use 2 points
        SFVec2f tanctl;
        tanctl.x = 2*cur.x – curctl.x;
        tanctl.y = 2*cur.y – curctl.y;
        draw_curve(cur, tanctl, point[i], point[i+1]);
        cur = point[i+1];
        curctl = point[i];
        i += 2;
        break;
    }
  }
  if (is_filled) draw_line(cur, first);
```

In the above pseudo-code, `draw_line(a,b)` draws a line from a to b and `draw_curve(a,b,c,d)` draws a Bezier curve from a to d, using b as the control point for a and c as the control point for d. Note that, because of the move command (`type` = 0) multiple disjoint segments are possible. In the case of a filled shape, each segment is closed by drawing a straight line from the last point in the segment to the first. Shapes are filled using the odd-even winding fill rule. If one segment is contained within another, the inside of the inner shape is not filled, allowing shapes with holes.

The first coordinate pair in **point** is the starting point of the curve. The first value in **type** describes the treatment to be applied to the subsequent coordinate pairs. At any time, a value in **type** describes the characteristics of the next curve segment. If **P is** the starting point or the last point of the previous segment of the curve; **N** the ending point of the current curve segment; $\mathbf{C_1}$ the control point on the side of **P** and $\mathbf{C_2}$ the control point on the side of **N.**

The permitted values of **type** are:

- 0 = MoveTo: One coordinate pair in the **point** list is consumed, defining **N**. **P** ends the curve. The curve shall start again at **N**. Sequences of two or more MoveTos shall not occur. MoveTo shall not occur as the first element in **type.**

- 1 = LineTo: One coordinate pair in the **point** list is consumed, defining **N**. A straight line is drawn from **P** to **N**.

- 2 = CurveTo: Three coordinate pairs in the **point** list are consumed, defining $\mathbf{C_1}$, $\mathbf{C_2}$ and **N** respectively. The first coordinate pair specifies the control point the start of this curve segment ($\mathbf{C_1}$), the second specifies the control point for end of the curve segment ($\mathbf{C_2}$) and the third specifies the ending point of the curve segment (**N**).

- 3 = NextCurveto: Two coordinate pairs in the **point** list are consumed, defining $\mathbf{C_2}$ and **N** in this order. The first coordinate pair specifies the control point for the end of the curve segment ($\mathbf{C_2}$), and the second specifies the ending point of the curve segment (**N**). The control point $\mathbf{C_1}$ for the start of the curve segment is derived from the previous control point. If the previous segment was formed with CurveTo or NextCurveTo, the start control point $\mathbf{C_1}$ is symmetrical to the end control point $\mathbf{C'_2}$ of the previous curve segment with respect to point **P**. This control type shall not occur immediately following a MoveTo or LineTo.

The formula for obtaining the coordinates of $\mathbf{C_1}$ in the case of a NextCurveTo is:

$$\mathbf{C}_{1x} = 2.\mathbf{P}_x - \mathbf{C'}_{2x} \text{and} \qquad \mathbf{C}_{1y} = 2.\mathbf{P}_y - \mathbf{C'}_{2y}$$

The first point in **point**, as the first point in the curve, is implicitly a MoveTo.

For CurveTo and NextCurveTo, the piece of curve is constructed using the above formula as applied to a polygon constructed from four points, that is the starting point **P**, the first control point $\mathbf{C_1}$, the second control point $\mathbf{C_2}$ and the end point **N**, which is the next point in the point list.

The curve shall be continuous except at points tagged with MoveTo. The tangent of the curve is only continuous at points tagged with NextCurveTo, or at points where the previous second control point $\mathbf{C'_2}$, the key point **P** and the next first control point $\mathbf{C_1}$ are aligned.

If there are more values in **point** than specified by **type**, then the unused points shall describe a curve as if no **type** was defined.

EXAMPLE —

```
geometry Curve2D {
   point Coordinate2D {
         points [ 0 0 0 100 200 100 200 200 210 200 220 200 ]
      }
      type [ 2 0 1 ]
}
```

The first segment of curve starts at 0,0 goes to 200,200 and control points are 0,100 and 200,100. The Bezier curve drawn is the one with the polygon [0 0 0 100 200 100 200 200] (represented in dotted gray) when types=null, with the same fineness. When types is specified, the fineness parameter is applied to each curve segment. Then we have a "move to", from 200,200 to 210,200. Then we have a "line to", from 210,200 to 220,200 (small segment in upper right corner).

In Figure 18, the curve is drawn in wide black, and the control polygon is drawn in dotted gray. The curve has two connex components.

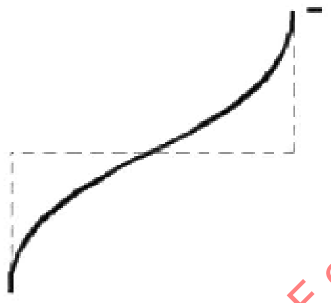


Figure 18 - Curve node example

9.4.2.29 Cylinder

9.4.2.29.1 Node interface

Cylinder {

field	SFBool	**bottom**	TRUE
field	SFFloat	**height**	2.0
field	SFFloat	**radius**	1.0
field	SFBool	**side**	TRUE
field	SFBool	**top**	TRUE

}

NOTE — For the binary encoding of this node see Annex H.1.29.

9.4.2.29.2 Functionality and semantics

The semantics of the **Cylinder** node are specified in ISO/IEC 14772-1:1998, subclause 6.14 [10].

9.4.2.30 CylinderSensor

9.4.2.30.1 Node interface

CylinderSensor {

exposedField	SFBool	**autoOffset**	TRUE
exposedField	SFFloat	**diskAngle**	0.262
exposedField	SFBool	**enabled**	TRUE
exposedField	SFFloat	**maxAngle**	-1.0
exposedField	SFFloat	**minAngle**	0.0
exposedField	SFFloat	**offset**	0.0
eventOut	SFBool	**isActive**	

```
    eventOut        SFRotation      rotation_changed
    eventOut        SFVec3f         trackPoint_changed
}
```

NOTE — For the binary encoding of this node see Annex H.1.30.

9.4.2.30.2 Functionality and semantics

The semantics of the **CylinderSensor** node are specified in ISO/IEC 14772-1:1998, subclause 6.15 [10].

9.4.2.31 DiscSensor

9.4.2.31.1 Node interface

```
DiscSensor {
    exposedField    SFBool          autoOffset              TRUE
    exposedField    SFBool          enabled                 TRUE
    exposedField    SFFloat         maxAngle                -1.0
    exposedField    SFFloat         minAngle                -1.0
    exposedField    SFFloat         offset                  0.0
    eventOut        SFBool          isActive
    eventOut        SFFloat         rotation_changed
    eventOut        SFVec2f         trackPoint_changed
}
```

NOTE — For the binary encoding of this node see Annex H.1.31.

9.4.2.31.2 Functionality and semantics

This sensor enables the rotation of an object in the 2D plane around an axis specified in the local coordinate system. The semantics are as similar to those for **CylinderSensor**, but restricted to a 2D case.

9.4.2.32 DirectionalLight

9.4.2.32.1 Node interface

```
DirectionalLight {
    exposedField    SFFloat         ambientIntensity        0.0
    exposedField    SFColor         color                   1, 1, 1
    exposedField    SFVec3f         direction               0, 0, -1
    exposedField    SFFloat         intensity               1.0
    exposedField    SFBool          on                      TRUE
}
```

NOTE — For the binary encoding of this node see Annex H.1.32.

9.4.2.32.2 Functionality and semantics

The semantics of the **DirectionalLight** node are specified in ISO/IEC 14772-1:1998, subclause 6.16 [10].

9.4.2.33 ElevationGrid

9.4.2.33.1 Node interface

```
ElevationGrid {
    eventIn         MFFloat         set_height
    exposedField    SFNode          color                   NULL
    exposedField    SFNode          normal                  NULL
    exposedField    SFNode          texCoord                NULL
    field           MFFloat         height                  []
    field           SFBool          ccw                     TRUE
    field           SFBool          colorPerVertex          TRUE
    field           SFFloat         creaseAngle             0.0
    field           SFBool          normalPerVertex         TRUE
    field           SFBool          solid                   TRUE
    field           SFInt32         xDimension              0
    field           SFFloat         xSpacing                1.0
    field           SFInt32         zDimension              0
    field           SFFloat         zSpacing                1.0
}
```

NOTE — For the binary encoding of this node see Annex H.1.33.

9.4.2.33.2 Functionality and semantics

The semantics of the **ElevationGrid** node are specified in ISO/IEC 14772-1:1998, subclause 6.17 [10].

9.4.2.34 Expression

9.4.2.34.1 Node interface

```
Expression {
    field           SFInt32         expression_select1      0
    field           SFInt32         expression_intensity1   0
    field           SFInt32         expression_select2      0
    field           SFInt32         expression_intensity2   0
    field           SFBool          init_face               FALSE
    field           SFBool          expression_def          FALSE
}
```

NOTE — For the binary encoding of this node see Annex H.1.34.

9.4.2.34.2 Functionality and semantics

The **Expression** node is used to define the expression of the face as a combination of two expressions from the standard set of expressions defined ISO/IEC 14496-2, Annex C, Table C-3.

The **expression_select1** and **expression_select2** fields specify the expression types. The **expression_intensity1** and **expression_intensity2** fields specify the corresponding expression intensities.

If **init_face** is set, a neutral face may be modified before applying FAPs 1 and 3-68.

If **expression_def** is set, current FAPs are used to define an expression and store it.

9.4.2.35 Extrusion

9.4.2.35.1 Node interface

```
Extrusion {
    eventIn         MFVec2f         set_crossSection
    eventIn         MFRotation      set_orientation
    eventIn         MFVec2f         set_scale
    eventIn         MFVec3f         set_spine
    field           SFBool          beginCap                TRUE
    field           SFBool          ccw                     TRUE
    field           SFBool          convex                  TRUE
    field           SFFloat         creaseAngle             0.0
    field           MFVec2f         crossSection            1, 1, 1, -1, -1, -1, -1, 1, 1, 1
    field           SFBool          endCap                  TRUE
    field           MFRotation      orientation             0, 0, 1, 0
    field           MFVec2f         scale                   1, 1
    field        SFBool     solid                TRUE
    field        MFVec3f    spine                0, 0, 0, 0, 1, 0
}
```

NOTE — For the binary encoding of this node see Annex H.1.35.

9.4.2.35.2 Functionality and semantics

The semantics of the **Extrusion** node are specified in ISO/IEC 14772-1:1998, subclause 6.18 [10].

9.4.2.36 Face

9.4.2.36.1 Node interface

```
Face {
    exposedField  SFNode    fit                  NULL
    exposedField  SFNode    fdp                  NULL
    exposedField  SFNode    fap                  NULL
    exposedField  SFNode    ttsSource            NULL
    exposedField  MFNode    renderedFace         NULL
}
```

NOTE — For the binary encoding of this node see Annex H.1.36.

9.4.2.36.2 Functionality and semantics

The **Face** node is used to define and animate a face in the scene. In order to animate the face with a facial animation stream, ut us necessary to link the **Face** node to a BIFS-Anim stream. The node shall be assigned a `nodeID,` through the DEF mechanism. Then, as for any BIFS-Anim stream, an animation mask is sent in the object descriptor of the BIFS-Anim stream (`specificInfo` field). The animation mask points to the **Face** node using its `nodeID`. The terminal shall then connect the facial animation decoder to the appropriate **Face** node.

The **FAP** field shall contain a **FAP** node, describing the facial animation parameters (FAPs). Each **Face** node shall contain a non-NULL **FAP** field.

The **FDP** field, which defines the particular look of a face by means of downloading the position of face definition points or an entire model, is optional. If the **FDP** field is not specified, the default face model of the terminal shall be used.

The **FIT** field, when specified, allows a set of FAPs to be defined in terms of another set of FAPs. When this field is non-NULL, the terminal shall use **FIT** to compute the maximal set of FAPs before using the FAPs to compute the mesh.

The **ttsSource** field shall only be non-NULL if the facial animation is to determine the facial animation parameters from an audio TTS source (see ISO/IEC 14496-3, section 6). In this case the **ttsSource** field shall contain an **AudioSource** node and the face shall be animated using the phonemes and bookmarks received from the TTS. See also Annex I.

renderedFace is the scene graph of the face after it is rendered (all FAP's applied).

9.4.2.37 FaceDefMesh

9.4.2.37.1 Node interface

```
FaceDefMesh {
    field        SFNode     faceSceneGraphNode   NULL
    field        MFInt32    intervalBorders      []
    field        MFInt32    coordIndex           []
    field        MFVec3f    displacements        []
}
```

NOTE — For the binary encoding of this node see Annex H.1.37.

9.4.2.37.2 Functionality and semantics

The **FaceDefMesh** node allows for the deformation of an **IndexedFaceSet** as a function of the amplitude of a FAP as specified in the related **FaceDefTable** node. The **FaceDefMesh** node defines the piece-wise linear motion trajectories for vertices of the **faceSceneGraphNode** field, which shall contain an **IndexedFaceSet** node. This **IndexedFaceSet** node belongs to the scenegraph of the **faceSceneGraph** field of the **FDP** node.

The **intervalBorders** field specifies interval borders for the piece-wise linear approximation in increasing order. Exactly one interval border shall have the value 0.

The **coordIndex** field shall contain a list of indices into the **Coordinate** node of the **IndexedFaceSet** node specified by the **faceSceneGraphNode** field.

For each vertex indexed in the **coordIndex** field, displacement vectors are given in the **displacements** field for the intervals defined in the **intervalBorders** field. There must be exactly (num(**intervalBorders**)-1)*num(**coordIndex**) values in this field.

In most cases, the animation generated by a FAP cannot be specified by updating a **Transform** node. Thus, a deformation of an **IndexedFaceSet** node needs to be performed. In this case, the **FaceDefTables** shall define which **IndexedFaceSets** are affected by a given FAP and how the **coord** fields of these nodes are updated. This is done by means of tables.

If a FAP affects an **IndexedFaceSet**, the **FaceDefMesh** shall specify a table of the following format for this **IndexedFaceSet**:

Table 30 - Vertex displacements

Vertex no.	1st Interval [I1, I2]	2nd Interval [I2, I3]	…
Index 1	Displacement D11	Displacement D12	…
Index 2	Displacement D21	Displacement D22	…
…	…	…	…

Exactly one interval border I_k must have the value 0:

$$[I_1, I_2], [I_2, I_3], \ldots [I_{k-1}, 0], [0, I_{k+1}], [I_{k+1}, I_{k+2}], \ldots [I_{max-1}, I_{max}]$$

During animation, when the terminal receives a FAP, which affects one or more **IndexedFaceSets** of the face model, it shall piece-wise linearly approximate the motion trajectory of each vertex of the affected **IndexedFaceSets** by using the appropriate table.

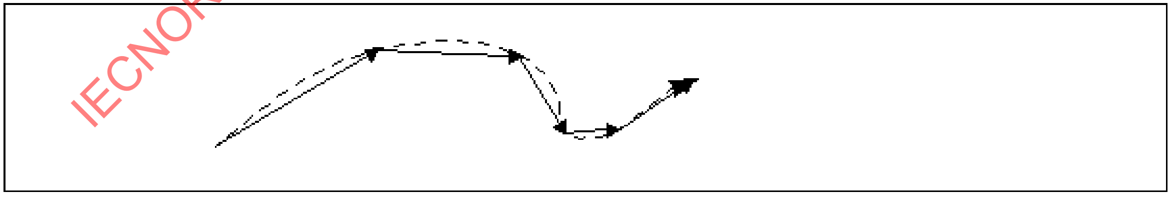


Figure 19 - An arbitrary motion trajectory is approximated as a piece-wise linear one.

If $\boldsymbol{P}_m$ is the position of the mth vertex in the **IndexedFaceSet** in neutral state (FAP = 0), $\boldsymbol{P'}_m$ the position of the same vertex after animation with the given FAP and $\boldsymbol{D}_{mk}$ the 3D displacement in the kth interval, the following algorithm shall be applied to determine the new position $\boldsymbol{P'}_m$.

Determine, in which of the intervals listed in the table the received FAP is lying.

If the received FAP is lying in the jth interval [Ij, Ij+1] and 0=Ik $\leq$ Ij, the new vertex position P'm of the mth vertex of the **IndexedFaceSet** is given by:

$P'_m = FAPU * ((I_{k+1}-0) * D_{m,k} + (I_{k+2}-I_{k+1}) * D_{m,\,k+1} + \ldots + (I_j - I_{j-1}) * D_{m,\,j-1} + (FAP-I_j) * D_{m,\,j}) + P_m.$ (Eq. 1)

If FAP > I_{max}, then P'_m is calculated by using equation Eq. 1 and setting the index j = max.

If the received FAP is lying in the jth interval $[I_j, I_{j+1}]$ and $I_{j+1} \leq I_k=0$, the new vertex position P'_m is given by:

$P'_m = FAPU * ((I_{j+1} - FAP) * D_{m,\,j} + (I_{j+2} - I_{j+1}) * D_{m,\,j+1} + \ldots + (I_{k-1} - I_{k-2}) * D_{m,\,k-2} + (0 - I_{k-1}) * D_{m,\,k-1}) + P_m$ (Eq. 2)

If FAP < I_1, then P'_m is calculated by using equation Eq. 1 and setting the index j+1 = 1.

If for a given FAP and **IndexedFaceSet** the table contains only one interval, the motion is strictly linear:

P'm = FAPU * FAP * Dm1 + Pm.

EXAMPLE —

```
FaceDefMesh {
   objectDescriptorID UpperLip
   intervalBorders [ -1000, 0, 500, 1000 ]
   coordIndex [ 50, 51]
   displacements [1 0 0,  0.9 0 0, 1.5 0 4, 0.8 0 0, 0.7 0 0, 2 0 0 ]
}
```

This **FaceDefMesh** defines the animation of the mesh "UpperLip". For the piecewise-linear motion function three intervals are defined: [-1000, 0], [0, 500] and [500, 1000]. Displacements are given for the vertices with the indices 50 and 51. The displacements for the vertex 50 are: (1 0 0), (0.9 0 0) and (1.5 0 4), the displacements for vertex 51 are (0.8 0 0), (0.7 0 0) and (2 0 0). Given a FAPValue of 600, the resulting displacement for vertex 50 would be:

displacement(vertex 50) = $500*(0.9\ 0\ 0)^T + 100 * (1.5\ 0\ 4)^T = (600\ 0\ 400)^T$.

If the FAPValue is outside the given intervals, the boundary intervals are extended to +I or -I, as appropriate.

9.4.2.38 FaceDefTables

9.4.2.38.1 Node interface

FaceDefTables {

field	SFInt32	**fapID**	0
field	SFInt32	**highLevelSelect**	0
exposedField	MFNode	**faceDefMesh**	[]
exposedField	MFNode	**faceDefTransform**	[]

}

NOTE — For the binary encoding of this node see Annex H.1.38.

9.4.2.38.2 Functionality and semantics

The **FaceDefTables** node defines the behavior of a facial animation parameter FAP on a downloaded face model in **faceSceneGraph** by specifying the displacement vectors for moved vertices inside **IndexedFaceSet** objects as a function of the FAP **fapID** and/or specifying the value of a field of a **Transform** node as a function of FAP **fapID**.

The **FaceDefTables** node is transmitted directly after the BIFS bitstream of the **FDP** node. The **FaceDefTables** lists all FAPs that animate the face model. The FAPs animate the downloaded face model by updating the **Transform** or **IndexedFaceSet** nodes of the scene graph in **faceSceneGraph**. For each listed FAP, the **FaceDefTables** node describes which nodes are animated by this FAP and how they are animated. All FAPs that occur in the bitstream have to be specified in the **FaceDefTables** node. The animation generated by a FAP can be specified either by updating a **Transform** node (using a **FaceDefTransform**), or as a deformation of an **IndexedFaceSet** (using a **FaceDefMesh**).

The FAPUs shall be calculated by the terminal using the feature points that shall be specified in the FDP. The FAPUs are needed in order to animate the downloaded face model.

9.4.2.38.3 Semantics

The **fapID** field specifies the FAP, for which the animation behavior is defined in the **faceDefMesh** and **faceDefTransform** fields.

If **fapID** has value 1 or 2, the **highLevelSelect** field specifies the type of viseme or expression. In other cases this field has no meaning and shall be ignored.

The **faceDefMesh** field shall contain a **FaceDefMesh** node.

The **faceDefTransform** field shall contain a **FaceDefTransform** node.

9.4.2.39 FaceDefTransform

9.4.2.39.1 Node interface

```
FaceDefTransform {
    field     SFNode        faceSceneGraphNode     NULL
    field     SFInt32       fieldId                1
    field     SFRotation    rotationDef            0, 0, 1, 0
    field     SFVec3f       scaleDef               1, 1, 1
    field     SFVec3f       translationDef         0, 0, 0
}
```

NOTE — For the binary encoding of this node see Annex H.1.39.

9.4.2.39.2 Functionality and semantics

The **FaceDefTransform** node defines which field (**rotation**, **scale** or **translation**) of a **Transform** node (**faceSceneGraphNode)** of **faceSceneGraph** (defined in an **FDP** node) is updated by a facial animation parameter, and how the field is updated. If the face is in its neutral position, the **faceSceneGraphNode** has its **translation**, **scale**, and **rotation** fields set to the neutral values $(0,0,0)^T$, $(1,1,1)^T$, (0,0,1,0), respectively.

The **faceSceneGraphNode** field specifies the **Transform** node for which the animation is defined. The node shall be part of **faceScenegraph** as defined in the **FDP** node.

The **fieldId** field specifies which field in the **Transform** node, specified by the **faceSceneGraphNode** field, is updated by the FAP during animation. Possible fields are **translation**, **rotation**, **scale**.

— If **fieldID**==1, **rotation** shall be updated using **rotationDef** and FAPValue.

— If **fieldID**==2, **scale** shall be updated using **scaleDef** and FAPValue.

— If **fieldID**==3, **translation** shall be updated using **translationDef** and FAPValue.

The **rotationDef** field is of type SFRotation. With **rotationDef**=(r_x,r_y,r_z,θ), the new value of the **rotation** field of the **Transform** node **faceSceneGraphNode** is:

rotation: **=**$(r_x,r_y,r_z,\theta$*FAPValue*AU) [AU is defined in ISO/IEC 14496-2]

The **scaleDef** field is of type SFVec3f. The new value of the **scale** field of the **Transform** node **faceSceneGraphNode** is:

scale:= FAPValue***scaleDef**

The **translationDef** field is of type SFVec3f. The new value of the **translation** field of the **Transform** node **faceSceneGraphNode** is:

translation:= FAPValue***translationDef**

9.4.2.40 FAP

9.4.2.40.1 Node interface

```
FAP {
    exposedField    SFNode     viseme                  NULL
    exposedField    SFNode     expression              NULL
    exposedField    SFInt32    open_jaw                +I
    exposedField    SFInt32    lower_t_midlip          +I
    exposedField    SFInt32    raise_b_midlip          +I
    exposedField    SFInt32    stretch_l_corner        +I
    exposedField    SFInt32    stretch_r_corner        +I
    exposedField    SFInt32    lower_t_lip_lm          +I
    exposedField    SFInt32    lower_t_lip_rm          +I
    exposedField    SFInt32    lower_b_lip_lm          +I
    exposedField    SFInt32    lower_b_lip_rm          +I
    exposedField    SFInt32    raise_l_cornerlip       +I
    exposedField    SFInt32    raise_r_cornerlip       +I
    exposedField    SFInt32    thrust_jaw              +I
    exposedField    SFInt32    shift_jaw               +I
    exposedField    SFInt32    push_b_lip              +I
    exposedField    SFInt32    push_t_lip              +I
    exposedField    SFInt32    depress_chin            +I
    exposedField    SFInt32    close_t_l_eyelid        +I
    exposedField    SFInt32    close_t_r_eyelid        +I
    exposedField    SFInt32    close_b_l_eyelid        +I
    exposedField    SFInt32    close_b_r_eyelid        +I
    exposedField    SFInt32    yaw_l_eyeball           +I
    exposedField    SFInt32    yaw_r_eyeball           +I
    exposedField    SFInt32    pitch_l_eyeball         +I
    exposedField    SFInt32    pitch_r_eyeball         +I
    exposedField    SFInt32    thrust_l_eyeball        +I
    exposedField    SFInt32    thrust_r_eyeball        +I
    exposedField    SFInt32    dilate_l_pupil          +I
    exposedField    SFInt32    dilate_r_pupil          +I
    exposedField    SFInt32    raise_l_i_eyebrow       +I
    exposedField    SFInt32    raise_r_i_eyebrow       +I
    exposedField    SFInt32    raise_l_m_eyebrow       +I
    exposedField    SFInt32    raise_r_m_eyebrow       +I
    exposedField    SFInt32    raise_l_o_eyebrow       +I
    exposedField    SFInt32    raise_r_o_eyebrow       +I
    exposedField    SFInt32    squeeze_l_eyebrow       +I
    exposedField    SFInt32    squeeze_r_eyebrow       +I
    exposedField    SFInt32    puff_l_cheek            +I
    exposedField    SFInt32    puff_r_cheek            +I
    exposedField    SFInt32    lift_l_cheek            +I
    exposedField    SFInt32    lift_r_cheek            +I
    exposedField    SFInt32    shift_tongue_tip        +I
    exposedField    SFInt32    raise_tongue_tip        +I
    exposedField    SFInt32    thrust_tongue_tip       +I
    exposedField    SFInt32    raise_tongue            +I
    exposedField    SFInt32    tongue_roll             +I
    exposedField    SFInt32    head_pitch              +I
    exposedField    SFInt32    head_yaw                +I
    exposedField    SFInt32    head_roll               +I
    exposedField    SFInt32    lower_t_midlip_o        +I
    exposedField    SFInt32    raise_b_midlip_o        +I
    exposedField    SFInt32    stretch_l_cornerlip     +I
    exposedField    SFInt32    stretch_r_cornerlip_o   +I
    exposedField    SFInt32    lower_t_lip_lm_o        +I
    exposedField   SFInt32      lower_t_lip_rm_o        +I
    exposedField   SFInt32      raise_b_lip_lm_o        +I
    exposedField   SFInt32      raise_b_lip_rm_o        +I
    exposedField   SFInt32      raise_l_cornerlip_o     +I
    exposedField   SFInt32      raise_r_cornerlip_o     +I
    exposedField   SFInt32      stretch_l_nose          +I
    exposedField   SFInt32      stretch_r_nose          +I
    exposedField   SFInt32      raise_nose              +I
    exposedField   SFInt32      bend_nose               +I
    exposedField   SFInt32      raise_l_ear             +I
    exposedField   SFInt32      raise_r_ear             +I
    exposedField   SFInt32      pull_l_ear              +I
    exposedField   SFInt32      pull_r_ear              +I
}
```

NOTE — For the binary encoding of this node see Annex H.1.40.

9.4.2.40.2 Functionality and semantics

This node defines the current look of the face by means of expressions and FAPs and gives a hint to TTS controlled systems on which viseme to use. For a definition of the facial animation parameters see ISO/IEC 14496-2, Annex C.

The **viseme** field shall contain a **Viseme** node.

The **expression** field shall contain an **Expression** node.

The semantics for the remaining fields are described in the ISO/IEC 14496-2, Annex C and in particular in Table C-1.

A FAP of value +I shall be interpreted as indicating that the particular FAP is uninitialized.

9.4.2.41 FDP

9.4.2.41.1 Node interface

```
FDP {
    exposedField   SFNode    featurePointsCoord   NULL
    exposedField   SFNode    textureCoords        NULL
    exposedField   SFBool    useOrthoTexture      FALSE
    exposedField   MFNode    faceDefTables        []
    exposedField   MFNode    faceSceneGraph       []
}
```

NOTE — For the binary encoding of this node see Annex H.1.41.

9.4.2.41.2 Functionality and semantics

The **FDP** node defines the face model to be used at the terminal. Two options are supported:

1. If **faceDefTables** is NULL, calibration information is downloaded, so that the proprietary face of the terminal can be calibrated using facial feature points and, optionally, the texture information. In this case, the **featurePointsCoord** field shall be set. **featurePointsCoord** contains the coordinates of facial feature points, as defined in ISO/IEC 14496-2, Annex C, Figure C-1, corresponding to a neutral face. If a coordinate of a feature point is set to +I, the coordinates of this feature point shall be ignored. The **textureCoord** field, if set, is used to map a texture on the model calibrated by the feature points. The **textureCoord** points correspond to the feature points. Tthat is, each defined feature point shall have corresponding texture coordinates. In this case, the **faceSceneGraph** shall contain exactly one texture image, and any geometry it might contain shall be ignored. The terminal shall interpret the feature points, texture coordinates, and the **faceSceneGraph** in the following way:

— Feature points of the terminal's face model shall be moved to the coordinates of the feature points supplied in **featurePointsCoord**, unless a feature point is to be ignored, as explained above.

— If **textureCoord** is set, the texture supplied in the **faceSceneGraph** shall be mapped onto the terminal's default face model. The texture coordinates are derived from the texture coordinates of the feature points supplied in **textureCoords**. The **useOrthoTexture** field provides a hint to the decoding terminal that, when FALSE, indicates that the texture image is best obtained by cylindrical projection of the face. If **useOrthoTexture** is TRUE, the texture image is best obtained by orthographic projection of the face.

2. A face model as described in the **faceSceneGraph** is decoded. This face model replaces the terminal's default face model in the terminal. The **faceSceneGraph** shall contain the face in its neutral position (all FAPs = 0). If desired, the **faceSceneGraph** shall contain the texture maps of the face. The functions defining the way in which the **faceSceneGraph** shall be modified, as a function of the FAPs, shall also be decoded. This information is described by **faceDefTables** that define how the **faceSceneGraph** is to be modified as a function of each FAP. By means of **faceDefTables**, **IndexedFaceSets** and **Transform** nodes of the **faceSceneGraph** can be animated. Since the amplitude of FAPs is defined in units that are dependent on the size of the face model, the **featurePointsCoord** field defines the position of facial features on the surface of the face described by **faceSceneGraph**. From the location of these feature points, the terminal computes the units of the FAPs. Generally, only two node types in the scene graph of a decoded face model are affected by FAPs: **IndexedFaceSet** and **Transform** nodes. If a FAP causes a deformation of an object (e.g. lip stretching), then the coordinate positions in the affected **IndexedFaceSets** shall be updated. If a FAP causes a movement which can be described with a **Transform** node (e.g. FAP 23, yaw_l_eyeball), then the appropriate fields in this **Transform** node shall be updated. It shall be assumed that this **Transform** node has its **rotation**, **scale**, and **translation** fields set to neutral values if the face is in its neutral position. A unique `nodeId` shall be assigned via the DEF statement to all **IndexedFaceSet** and **Transform** nodes which are affected by FAPs so that they can be accessed unambiguously during animation.

The **featurePointsCoord** field shall contain a **Coordinate** node that specifies feature points for the calibration of the terminal's default face. The coordinates are specified in the **point** field of the **Coordinate** node in the prescribed order, that a feature point with a lower label number is listed before a feature point with a higher label naumber.

EXAMPLE — Feature point 3.14 before feature point 4.1

The **textureCoords** field shall contain a **Coordinate** node that specifies texture coordinates for the feature points. The coordinates are listed in the **point** field in the **Coordinate** node in the prescribed order, that a feature point with a lower label is listed before a feature point with a higher label.

The **useOrthoTexture** field may contain a hint to the terminal as to the type of texture image, in order to allow better interpolation of texture coordinates for the vertices that are not feature points. If **useOrthoTexture** is FALSE, the terminal may assume that the texture image was obtained by cylindrical projection of the face. If **useOrthoTexture** is 1, the terminal may assume that the texture image was obtained by orthographic projection of the face.

The **faceDefTables** field shall contain **FaceDefTables** nodes. The behavior of FAPs is defined in this field for the face in **faceSceneGraph**.

The **faceSceneGraph** field shall contain a **Group** node. In the case of option 1 (above), this may be used to contain a texture image as described above. In the case of option 2, this shall be the grouping node for the face model rendered in the compositor and shall contain the face model. In this case, the effect of facial animation parameters is defined in the **faceDefTables** field.

9.4.2.42 FIT

9.4.2.42.1 Node interface

```
FIT {
    exposedField   MFInt32   FAPs               []
    exposedField   MFInt32   graph              []
    exposedField   MFInt32   numeratorTerms     []
    exposedField   MFInt32   denominatorTerms   []
    exposedField   MFInt32   numeratorExp       []
    exposedField   MFInt32   denominatorExp     []
    exposedField   MFInt32   numeratorImpulse   []
    exposedField   MFFloat   numeratorCoefs     []
    exposedField   MFFloat   denominatorCoefs   []
}
```

NOTE — For the binary encoding of this node see Annex H.1.42.

9.4.2.42.2 Functionality and semantics

The **FIT** node allows a smaller set of FAPs to be sent during a facial animation. This small set can then be used to determine the values of other FAPs, using a rational polynomial mapping between parameters. In a **FIT** node, rational polynomials are used to specify interpolation functions.

EXAMPLE — The top inner lip FAPs can be sent and then used to determine the top outer lip FAPs. Another example is that only viseme and/or expression FAPs are sent to drive the face. In this case, low-level FAPs are interpolated from these two high-level FAPs.

To make the scheme general, sets of FAPs are specified, along with a FAP interpolation graph (FIG) between the sets that specifies which sets are used to determine which other sets. The FIG is a graph with directed links. Each node contains a set of FAPs. Each link from a parent node to a child node indicates that the FAPs in the child node can be interpolated from the parent node. **Expression** (FAP#1) or **Viseme** (FAP #2) and their fields shall not be interpolated from other FAPs.

In a FIG, a FAP may appear in several nodes, and a node may have multiple parents. For a node that has multiple parent nodes, the parent nodes are ordered as 1st parent node, 2nd parent node, etc. During the interpolation process, if this child node needs to be interpolated, it is first interpolated from 1st parent node if all FAPs in that parent node are available. Otherwise, it is interpolated from 2nd parent node, and so on.

An example of FIG is shown in Figure 20. Each node has a `nodeID`. The numerical label on each incoming link indicates the order of these links.

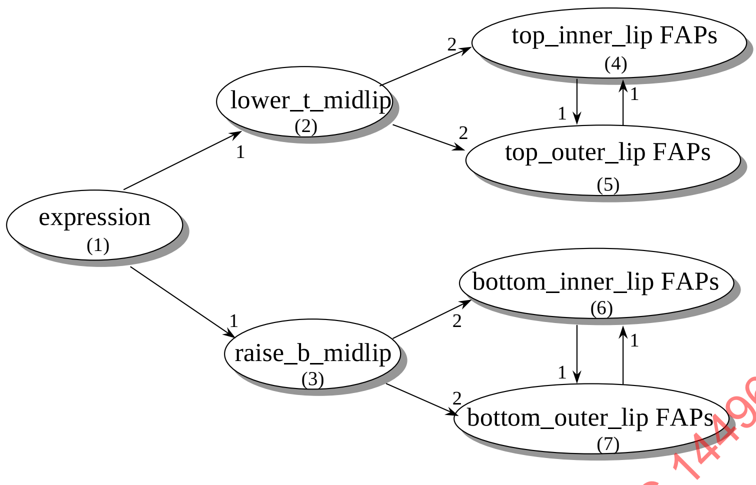


Figure 20 - A FIG example

The interpolation process based on the FAP interpolation graph is described using pseudo-C code as follows:

```
do {
   interpolation_count = 0;
   for (all Node_i) {   // from Node_1 to Node_N
      for (ordered Node_i's parent Node_k) {
         if (FAPs in Node_i need interpolation and
                  FAPs in Node_k have been interpolated or are available) {
            interpolate Node_i from Node_k;  //using interpolation function
                                             // table here
            interpolation_count ++;
            break;
         }
      }
   }
} while (interpolation_count != 0);
```

Each directed link in a FIG is a set of interpolation functions. Suppose F_1, F_2, …, F_n are the FAPs in a parent set and f_1, f_2, …, f_m are the FAPs in a child set.

Then, there are m interpolation functions denoted as:

$f_1 = I_1(F_1, F_2, …, F_n)$

$f_2 = I_2(F_1, F_2, …, F_n)$

…

$f_m = I_m(F_1, F_2, …, F_n)$

Each interpolation function I_k () is in a rational polynomial form if the parent node does not contain viseme FAP or expression FAP.

$$I(F_1, F_2, ..., F_n) = \sum_{i=0}^{K-1} (c_i \prod_{j=1}^{n} F_j^{l_{ij}}) \Big/ \sum_{i=0}^{P-1} (b_i \prod_{j=1}^{n} F_j^{m_{ij}})$$

Otherwise, an impulse function is added to each numerator polynomial term to allow selection of expression or viseme.

$$I(F_1,F_2,...,F_n) = \sum_{i=0}^{K-1} \delta(F_{s_i} - a_i)(c_i \prod_{j=1}^{n} F_j^{\,l_{ij}}) \Big/ \sum_{i=0}^{P-1} (b_i \prod_{j=1}^{n} F_j^{\,m_{ij}})$$

In both equations, K and P are the numbers of polynomial products, c_i and b_i are the coefficient of the *i*th product. l_{ij} and m_{ij} are the power of F_j in the *i*th product. An impulse function equals 1 when $F_{s_i} = a_i$, otherwise, equals 0. F_{s_i} can only be viseme_select1, viseme_select2, expression_select1, and expression_select2. a_i is an integer that ranges from 0 to 6 when F_{s_i} is expression_select1 or expression_select2, ranges 0 to 14 when F_{s_i} is viseme_select1 or viseme_select2. The encoder shall send an interpolation function table which contains K, P, a_i, s_i, c_i, b_i, l_{ij}, m_{ij} to the terminal.

To aid in the explanation below, it is assumed that there are N different sets of FAPs with index 1 to N, and that each set has n_i, i=1,..,N parameters. It is also assumed that there are L directed links in the FIG and that each link points from the FAP set with index P_i to the FAP set with index C_i, for i = 1, .. , L

The FAPs field shall contain a list of FAP-indices specifying which animation parameters form sets of FAPs. Each set of FAP indices is terminated by –1. There shall be a total of N + n_1 + n_2 + … + n_N numbers in this field, with N of them being –1. FAP#1 to FAP#68 are of indices 1 to 68. Fields of the **Viseme** FAP (FAP#1), namely, **viseme_select1**, **viseme_select2**, **viseme_blend**, are of indices from 69 to 71. Fields of the **Expression** FAP (FAP#2), namely, **expression_select1**, **expression_select2**, **expression_intensity1**, **expression_intensity2** are of indices from 72 to 75. When the parent node contains a **Viseme** FAP, three indices, 69, 70, 71, shall be included in the node (but not index 1). When a parent node contains an **Expression** FAP, four indices, 72,73,74,75, shall be included in the node (but not index 2).

The **graph** field shall contain a list of pairs of integers, specifying a directed links between sets of FAPs. The integers refer to the indices of the sets specified in the **FAPs** field, and thus range from 1 to N. When more than one direct link terminates at the same set, that is, when the second value in the pair is repeated, the links have precedence determined by their order in this field. This field shall have a total of 2L numbers, corresponding to the directed links between the parents and children in the FIG.

The **numeratorTerms** field shall be a list containing the number of terms in the polynomials of the numerators of the rational functions used to interpolae parameter values. Each element in the list corresponds to *K* in equation 1 above). Each link i (that is, the *i*th integer pair) in the **graph** field must have n_{Ci} values specified, one for each child FAP. The order in the **numeratorTerms** list shall correspond to the order of the links in the **graph** field and the order that the child FAP appears in the **FAPs** field. There shall be n_{C1} + n_{C2} + … + n_{CL} numbers in this field.

The **denominatorTerms** field shall contain a list of the number of terms in the polynomials of the denominator of the rational functions controlling the parameter value. Each element in the list corresponds to *P* in equation 1. Each link i (that is, the *i*th integer pair) in the **graph** field must have n_{Ci} values specified, one for each child FAP. The order in the **denominatorTerms** list corresponds to the order of the links in the **graph** field and the order that the child FAP appears in the **FAPs** field. There shall be n_{C1} + n_{C2} + … + n_{CL} numbers in this field.

The **numeratorImpulse** field shall contain a list of impulse functions in the numerator of the rational function for links with the **Viseme** or **Expression** FAP in parent node. This list corresponds to the $\delta(F_{s_i} - a_i)$. Each entry in the list is (s_i, a_i).

The **numeratorExp** field shall contain a list of exponents of the polynomial terms in the numerator of the rational function controlling the parameter value. This list corresponds to l_{ij}. For each child FAP in each link i, n_{Pi}*K values need to be specified. The order in the **numeratorExp** list shall correspond to the order of the links in the **graph** field and the order that the child FAP appears in the **FAPs** field.

NOTE — K may be different for each child FAP.

The **denominatorExp** field shall contain a list of exponents of the polynomial terms of the denominator of the rational function controlling the parameter value. This list corresponds to m_{ij}. For each child FAP in each link i, n_{Pi}*P values need to be specified. The order in the **denominatorExp** list shall correspond to the order of the links in the **graph** field and the order that the child FAP appears in the **FAPs** field.

NOTE — P may be different for each child FAP.

The **numeratorCoefs** field shall contain a list of coefficients of the polynomial terms of the numerator of the rational function controlling the parameter value. This list corresponds to c_i. The list shall have *K* terms for each child parameter that appears in a link in the FIG, with the order in **numeratorCoefs** corresponding to the order in **graph** and **FAPs**.

NOTE — K is dependent on the polynomial, and is not a fixed constant.

The **denominatorCoefs** field shall contain a list of coefficients of the polynomial terms in the numerator of the rational function controlling the parameter value. This list corresponds to b_i. The list shall have *P* terms for each child parameter that appears in a link in the FIG, with the order in **denominatorCoefs** corresponding to the order in **graph** and **FAPs**.

NOTE — P is dependent on the polynomial, and is not a fixed constant.

EXAMPLE — Suppose a FIG contains four nodes and 2 links. Node 1 contains FAP#3, FAP#3, FAP#5. Node 2 contains FAP#6, FAP#7. Node 3 contains an expression FAP, which means contains FAP#72, FAP#73, FAP#74, and FAP#75. Node 4 contains FAP#12 and FAP#17. Two links are from node 1 to node 2, and from node 3 to node 4. For the first link, the interpolation functions are

$$F_6 = (F_3 + 2F_4 + 3F_5 + 4F_3F_4^2)/(5F_5 + 6F_3F_4F_5)$$

$$F_7 = F_4 .$$

For the second link, the interpolation functions are

$$F_{12} = \delta(F_{72} - 6)(0.6F_{74}) + \delta(F_{73} - 6)(0.6F_{75})$$

$$F_{17} = \delta(F_{72} - 6)(-1.5F_{74}) + \delta(F_{73} - 6)(-1.5F_{75}) .$$

The second link simply says that when the expression is surprise (FAP#72=6 or FAP#73=6), for FAP#12, the value is 0.6 times of expression intensity FAP#74 or FAP#75; for FAP#17, the value is –1.5 tims of FAP#74 or FAP#75.

After the FIT node given below, we explain each field separately.

```
FIT {
  FAPs            [ 3 4 5 -1 6 7 –1 72 73 74 75 –1 12 17 -1]
  graph           [ 1 2 3 4]
  numeratorTerms  [ 4 1 2 2 ]
  denominatorTerms  [2 1 1 1]
  numeratorExp    [1 0 0   0 1 0   0 0 1  1 2 0  0 1 0
                       0 0 1 0  0 0 0 1  0 0 1 0  0 0 0 1 ]
  denominatorExp  [ 0 0 1  1 1 1  0 0 0
                       0 0 0 0  0 0 0 0 ]
  numeratorImpulse  [ 72 6  73 6  72 6  73 6 ]
  numeratorCoefs  [1 2 3 4   1   0.6 0.6  -1.5 –1.5 ]
  denominatorCoefs  [5 6 1 1 1 ]
}
```

`FAPs [ 3 4 5 -1 6 7 –1 72 73 74 75 –1 12 17 -1]`
Four sets of FAPs are defined, the first with FAPs number 3, 4, and 5, the second with FAPs number 6 and 7, the third with FAPs number 72, 73, 74, 75, and the fourth with FAPs number 12, 17.

`graph  [ 1 2 3 4]`
The first set is made to be the parent of the second set, so that FAPs number 6 and 7 will be determined by FAPs 3, 4, and 5. Also, the third set is made to be the parent of the fourth set, so that FAPs number 12 and 17 will be determined by FAPs 72, 73, 74, and 75.

`numeratorTerms [ 4 1 2 2]`
The rational functions that define F6 and F7 are selected to have 4 and 1 terms in their numerator, respectively. Also, the rational functions that define F12 and F17 are selected to have 2 and 2 terms in their numerator, respectively.

`denominatorTerms [ 2 1 1 1]`
The rational functions that define F6 and F7 are selected to have 2 and 1 terms in their denominator, respectively. Also, the rational functions that define F12 and F17 are selected to both have 1 term in their denominator.

`numeratorExp [ 1 0 0   0 1 0   0 0 1  1 2 0     0 1 0   0 0 1 0  0 0 0 1  0 0 1 0  0 0 0 1]`
The numerator selected for the rational function defining F6 is F3 + 2F4 + 3 F5 + 4F3F42. There are 3 parent FAPs, and 4 terms, leading to 12 exponents for this rational function. For F7, the numerator is just F4, so there are three exponents only (one for each FAP). Values for F12 and F17 are derived in the same way.

`denominatorExp [ 0 0 1  1 1 1    0 0 0   0 0 0 0  0 0 0 0]`
The denominator selected for the rational function defining F6 is 5F5+ 6F3F4F5 , so there are 3 parent FAPs and 2 terms and hence, 6 exponents for this rational function. For F7, the denominator is just 1, so there are three exponents only (one for each FAP). Values for F12 and F17 are derived in the same way.

`numeratorImpulse [72 6  73 6  72 6  73 6]`

For the second link, all four numerator polynomial terms contain impulse function $\delta(F_{72} - 6)$ or $\delta(F_{73} - 6)$.

`numeratorCoefs [ 1 2 3 4  1   0.6 0.6  -1.5 -1.5]`
There is one coefficient for each term in the numerator of each rational function.

`denominatorCoefs [ 5 6   1   1 1]`
There is one coefficient for each term in the denominator of each rational function.

9.4.2.43 Fog

9.4.2.43.1 Node interface

Fog {

exposedField	SFColor	**color**	1 1 1
exposedField	SFString	**fogType**	"LINEAR"
exposedField	SFFloat	**visibilityRange**	0.0
eventIn	SFBool	**set_bind**	
eventOut	SFBool	**isBound**	

}

NOTE — For the binary encoding of this node see Annex H.1.43.

9.4.2.43.2 Functionality and semantics

The semantics of the **Fog** node are specified in ISO/IEC 14772-1:1998, subclause 6.19 [10].

9.4.2.44 FontStyle

9.4.2.44.1 Node interface

FontStyle {

field	MFString	**family**	["SERIF"]
field	SFBool	**horizontal**	TRUE
field	MFString	**justify**	["BEGIN"]
field	SFString	**language**	""
field	SFBool	**leftToRight**	TRUE
field	SFFloat	**size**	1.0

```
    field         SFFloat     spacing             1.0
    field         SFString    style               "PLAIN"
    field         SFBool      topToBottom         TRUE
}
```

NOTE — For the binary encoding of this node see Annex H.1.44.

9.4.2.44.2 Functionality and semantics

The semantics of the **FontStyle** node are specified in ISO/IEC 14772-1:1998, subclause 6.20 [10].

9.4.2.45 Form

9.4.2.45.1 Node interface

```
Form {
    eventIn       MFNode      addChildren
    eventIn       MFNode      removeChildren
    exposedField  MFNode      children            []
    exposedField  SFVec2f     size                -1, -1
    exposedField  MFInt32     groups              []
    exposedField  MFInt32     constraints         []
    exposedField  MFInt32     groupsIndex         []
}
```

NOTE — For the binary encoding of this node see Annex H.1.45.

9.4.2.45.2 Functionality and semantics

The **Form** node specifies the placement of its children according to relative alignment and distribution constraints. Distribution spreads objects regularly, with an equal spacing between them.

The **children** field shall specify a list of nodes that are to be arranged. The children's position is implicit and order is important.

The **size** field specifies the width and height of the layout frame.

The **groups** field specifies the list of groups of objects on which the constraints can be applied. The children of the **Form** node are numbered from 1 to n, 0 being reserved for a reference to the form itself. A group is a list of child indices, terminated by a -1.

The **constraints** and the **groupsIndex** fields specify the list of constraints. One constraint is constituted by a constraint type from the **constraints** field, coupled with a set of group indices terminated by a –1 contained in the **groupsIndex** field. There shall be as many strings in **constraints** as there are –1-terminated sets in **groupsIndex**. The n-th constraint string shall be applied to the n-th set in the **groupsIndex** field.

Constraints belong to two categories: alignment and distribution constraints.

Components referred to in the tables below are components whose indices appear in the list following the constraint type. When rank is mentioned, it refers to the rank in that list.

The semantics of the **<s>**, when present in the name of a constraint, is the following. It shall be a number, integer when the scene uses pixel metrics, and float otherwise, which specifies the space mentioned in the semantics of the constraint.

Table 31 - Alignment Constraints

Alignment Constraints	**Type Index**	**Effect**
AL: Align Left edges	“AL”	The xmin of constrained components becomes equal to the xmin of the left-most component.
AH: Align centers Horizontally	“AH”	The (xmin+xmax)/2 of constrained components becomes equal to the (xmin+xmax)/2 of the group of constrained components as computed before this constraint is applied.
AR: Align Right edges	“AR”	The xmax of constrained components becomes equal to the xmax of the right-most component.
AT: Align Top edges	“AT”	The ymax of all constrained components becomes equal to the ymax of the top-most component.
AV: Align centers Vertically	“AV”	The (ymin+ymax)/2 of constrained components becomes equal to the (ymin+ymax)/2 of the group of constrained components as computed before this constraint is applied.
AB: Align Bottom edges	“AB”	The ymin of constrained components becomes equal to the ymin of the bottom-most component.
ALspace: Align Left edges by specified space	“AL <s>”	The xmin of the second and following components become equal to the xmin of the first component plus the specified space.
ARspace: Align Right edges by specified space	“AR <s>”	The xmax of the second and following components becomes equal to the xmax of the first component minus the specified space.
ATspace: Align Top edges by specified space	“AT <s>”	The ymax of the second and following components becomes equal to the ymax of the first component minus the specified space.
ABspace: Align Bottom edges by specified space	“AB <s>”	The ymin of the second and following components become equal to the ymin of the first component plus the specified space.

The purpose of distribution constraints is to specify the space between components, by making such pairwise gaps equal either to a given value or to the effect of filling available space.

Table 32 - Distribution Constraints

Distribution Constraints	**Type Index**	**Effect**
SH: Spread Horizontally	“SH”	The differences between the xmin of each component and the xmax of the previous one all become equal. The first and the last component shall be constrained horizontally already.
SHin: Spread Horizontally in container	“SHin”	The differences between the xmin of each component and the xmax of the previous one all become equal. References are the edges of the layout.
SHspace: Spread Horizontally by specified space	“SH <s>”	The difference between the xmin of each component and the xmax of the previous one all become equal to the specified space. The first component is not moved.
SV: Spread Vertically	“SV”	The differences between the ymin of each component and the ymax of the previous one all become equal. The first and the last component shall be constrained vertically already.
SVin: Spread Vertically in container	“SVin”	The differences between the ymin of each component and the ymax of the previous one all become equal. References are the edges of the layout.
SVspace: Spread Vertically by specified space	“SV <s>”	The difference between the ymin of each component and the ymax of the previous one all become equal to the specified space. The first component is not moved.

All objects start at the center of the **Form**. The constraints are then applied in sequence.

EXAMPLE — Laying out five 2D objects.

```
Shape {
   Geometry2D Rectangle { size 50 55 } // draw the Form’s frame.
   VisualProps use VPSRect
}

Transform2D {
   translation 10 10 {
      children [
               Form {
            children [
               Shape2D { use OBJ1 }
               Shape2D { use OBJ2 }
               Shape2D { use OBJ3 }
               Shape2D { use OBJ4 }
               Shape2D { use OBJ5 }
            ]
            size 50 55
groups [ 1 -1 2 -1 3 -1 4 -1 5 -1 1 3 -1]
constraints [“SH” “SV” “AR” “AB” “AB 6”
                          “AB 7” “AL 7” “AT –2” “AR –2”]
               groupsIndex  [6 -1 1 -1 0 2 -1 0 2 -1 0 3 -1
                               0 4 -1 0 4 -1 0 5 -1 0 5 -1]
         }
      ]
   }
}
```

The above **constraints** specify the following operations:

— spread group 6 (objects 1 and 3) horizontally in container (object 0)

— spread group 1 (object 1) vertically in container

— align the right edges of groups 0 (container) and 2 (object 2)

— align the bottom edges of the container and group 2 (object 2)

— align the bottom edges of the container and group 3 (object 3) with spacing of size 6

— align the bottom edges of the container and group 4 (object 4) with spacing of size 7

— align the left edges of the container and group 4 (object 4) with spacing of size 7

— align the top edges of the container and group 5 (object 5) with spacing size of -2

— align the right edges of the container and group 5 (object 5) with spacing size of -2

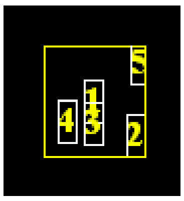


Figure 21 - Visual result of the Form node example

9.4.2.46 Group

9.4.2.46.1 Node interface

Group {

eventIn	MFNode	**addChildren**	
eventIn	MFNode	**removeChildren**	
exposedField	MFNode	**children**	[]

}

NOTE — For the binary encoding of this node see Annex H.1.46.

9.4.2.46.2 Functionality and semantics

The semantics of the **Group** node are specified in ISO/IEC 14772-1:1998, subclause 6.21 [10]. ISO/IEC 14496-1 does not support the bounding box parameters (**bboxCenter** and **bboxSize**).

Where multiple sub-graphs containing audio content (i.e. **Sound** nodes) occur as children of a **Group** node, the sounds shall be combined as described in 9.4.2.82.

9.4.2.47 ImageTexture

9.4.2.47.1 Node interface

ImageTexture {

exposedField	MFString	**url**	[]
field	SFBool	**repeatS**	TRUE
field	SFBool	**repeatT**	TRUE

}

NOTE — For the binary encoding of this node see Annex H.1.47.

9.4.2.47.2 Functionality and semantics

The semantics of the **ImageTexture** node are specified in ISO/IEC 14772-1:1998, subclause 6.22 [10].

The **url** field specifies the data source to be used (see 9.2.2.7.1).

9.4.2.48 IndexedFaceSet

9.4.2.48.1 Node interface

IndexedFaceSet {

eventIn	MFInt32	**set_colorIndex**	
eventIn	MFInt32	**set_coordIndex**	
eventIn	MFInt32	**set_normalIndex**	
eventIn	MFInt32	**set_texCoordIndex**	
exposedField	SFNode	**color**	NULL
exposedField	SFNode	**coord**	NULL
exposedField	SFNode	**normal**	NULL
exposedField	SFNode	**texCoord**	NULL
field	SFBool	**ccw**	TRUE
field	MFInt32	**colorIndex**	[]
field	SFBool	**colorPerVertex**	TRUE
field	SFBool	**convex**	TRUE
field	MFInt32	**coordIndex**	[]
field	SFFloat	**creaseAngle**	0.0
field	MFInt32	**normalIndex**	[]
field	SFBool	**normalPerVertex**	TRUE
field	SFBool	**solid**	TRUE
field	MFInt32	**texCoordIndex**	[]

}

NOTE — For the binary encoding of this node see Annex H.1.48.

9.4.2.48.2 Functionality and semantics

The semantics of the **IndexedFaceSet** node are specified in ISO/IEC 14772-1:1998, subclause 6.23 [10]. Some restrictions on these semantics are described below.

The **IndexedFaceSet** node represents a 3D polygon mesh formed by constructing faces (polygons) from points specified in the **coord** field. If the **coordIndex** field is not NULL, **IndexedFaceSet** uses the indices in its **coordIndex** field to specify the polygonal faces by connecting together points from the **coord** field. An index of -1 shall indicate that the current face has ended and the next one begins. The last face may be followed by a -1. **IndexedFaceSet** shall be specified in the local coordinate system and shall be affected by parent transformations.

The **coord** field specifies the vertices of the face set and is specified by **Coordinate** node.

If the **coordIndex** field is not NULL, the indices of the **coordIndex** field shall be used to specify the faces by connecting together points from the **coord** field. An index of -1 shall indicate that the current face has ended and the next one begins. The last face may be followed by a -1.

If the **coordIndex** field is NULL, the vertices of the **coord** field are laid out in their respective order to specify one face.

If the **color** field is NULL and there is a **Material** node defined for the **Appearance** affecting this **IndexedFaceSet**, then the **emissiveColor** of the **Material** node shall be used to draw the faces.

9.4.2.49 IndexedFaceSet2D

9.4.2.49.1 Node interface

```
IndexedFaceSet2D {
   eventIn        MFInt32    set_colorIndex
   eventIn        MFInt32    set_coordIndex
   eventIn        MFInt32    set_texCoordIndex
   exposedField   SFNode     color                NULL
   exposedField   SFNode     coord                NULL
   exposedField   SFNode     texCoord             NULL
   field          MFInt32    colorIndex           []
   field          SFBool     colorPerVertex       TRUE
   field          SFBool     convex               TRUE
   field          MFInt32    coordIndex           []
   field          MFInt32    texCoordIndex        []
}
```

NOTE — For the binary encoding of this node see Annex H.1.49.

9.4.2.49.2 Functionality and semantics

The **IndexedFaceSet2D** node is the 2D equivalent of the **IndexedFaceSet** node as defined in 9.4.2.48. The **IndexedFaceSet2D** node represents a 2D shape formed by constructing 2D faces (polygons) from 2D vertices (points) specified in the **coord** field. The **coord** field contains a **Coordinate2D** node that defines the 2D vertices, referenced by the **coordIndex** field. The faces of an **IndexedFaceSet2D** node shall not overlap each other.

The detailed semantics are identical to those for the **IndexedFaceSet** node (see 9.4.2.48), restricted to the 2D case, and with the additional differences described here.

If the **texCoord** field is NULL, a default texture coordinate mapping is calculated using the local 2D coordinate system bounding box of the 2D shape, as follows. The X dimension of the bounding box defines the S coordinates, and the Y dimension defines the T coordinates. The value of the S coordinate ranges from 0 to 1, from the left end of the bounding box to the right end. The value of the T coordinate ranges from 0 to 1, from the lower end of the bounding box to the top end. Figure 22 illustrates the default texture mapping coordinates for a simple **IndexedFaceSet2D** shape consisting of a single polygonal face.

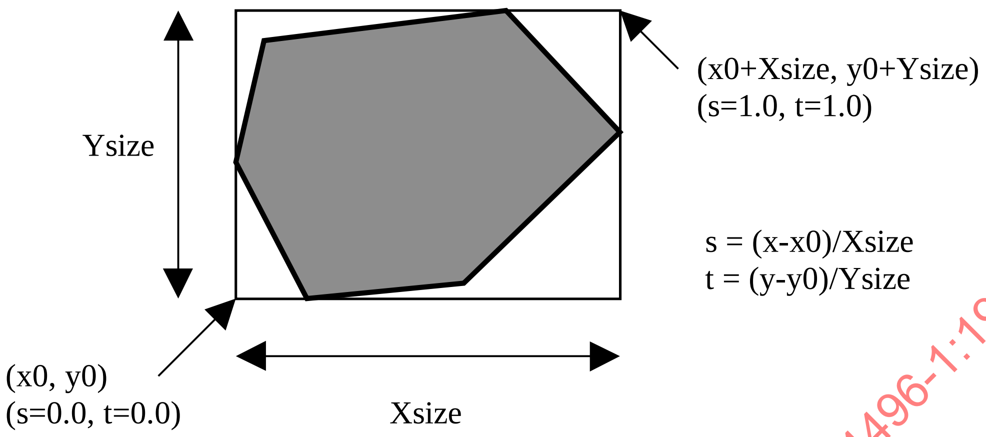


Figure 22 - IndexedFaceSet2D **default texture mapping coordinates for a simple shape**

9.4.2.50 IndexedLineSet

9.4.2.50.1 Node interface

```
IndexedLineSet {
    eventIn         MFInt32     set_colorIndex
    eventIn         MFInt32     set_coordIndex
    exposedField    SFNode      color                NULL
    exposedField    SFNode      coord                NULL
    field           MFInt32     colorIndex           []
    field           SFBool      colorPerVertex       TRUE
    field           MFInt32     coordIndex           []
}
```

NOTE — For the binary encoding of this node see Annex H.1.50.

9.4.2.50.2 Functionality and semantics

The semantics of the **IndexedLineSet** node are specified in ISO/IEC 14772-1:1998, subclause 6.24 [10].

9.4.2.51 IndexedLineSet2D

9.4.2.51.1 Node interface

```
IndexedLineSet2D {
    eventIn         MFInt32     set_colorIndex
    eventIn         MFInt32     set_coordIndex
    exposedField    SFNode      color                NULL
    exposedField    SFNode      coord                NULL
    field           MFInt32     colorIndex           []
    field           SFBool      colorPerVertex       TRUE
    field           MFInt32     coordIndex           []
}
```

NOTE — For the binary encoding of this node see Annex H.1.51.

9.4.2.51.2 Functionality and semantics

The **IndexedLineSet2D** node specifies a collection of lines or polygons.

The **coord** field shall list the vertices of the lines. When **coordIndex** is empty, the order of vertices shall be assumed to be sequential in the **coord** field. Otherwise, the **coordIndex** field determines the ordering of the vertices, with an index of -1 representing an end to the current polyline.

If the **color** field is not NULL, it shall contain a **Color** node, and the colors are applied to the line(s) as with the **IndexedLineSet** node (see 9.4.2.50).

9.4.2.52 Inline

9.4.2.52.1 Node interface

```
Inline {
    exposedField    MFString        url                     []
}
```

NOTE — For the binary encoding of this node see Annex H.1.52.

9.4.2.52.2 Functionality and semantics

The semantics of the **Inline** node are specified in ISO/IEC 14772-1:1998, subclause 6.25 [10]. ISO/IEC 14496-1 does not support the bounding box parameters (**bboxCenter** and **bboxSize**).

The **url** field specifies the data source to be used (see 9.2.2.7.1). The external source must contain a valid BIFS scene, and may include BIFS-Commands and BIFS-Anim frames

9.4.2.53 Layer2D

9.4.2.53.1 Node interface

```
Layer2D {
    eventIn         MFNode          addChildren
    eventIn         MFNode          removeChildren
    exposedField    MFNode          children                NULL
    exposedField    SFVec2f         size                    -1, -1
    exposedField    SFNode          background              NULL
    exposedField    SFNode          viewport                NULL
}
```

NOTE — For the binary encoding of this node see Annex H.1.53.

9.4.2.53.2 Functionality and semantics

The **Layer2D** node is a transparent rendering rectangle region on the screen where a 2D scene is drawn. The rectangle always faces the viewer of the scene. **Layer2D** and **Layer3D** nodes enable the composition of multiple 2D and 3D scenes (see Figure 23).

EXAMPLE — This allows users to have 2D interfaces to a 2D scene, or 3D interfaces to a 2D scene, or to view a 3D scene from different viewpoints in the same scene.

The **addChildren** eventIn specifies a list of 2D nodes that shall be added to the **Layer2D's children** field.

The **removeChildren** eventIn specifies a list of 2D nodes that shall be removed from the **Layer2D's children** field.

The **children** field may contain any 2D children nodes that define a 2D scene. Layer nodes are considered to be 2D objects within the scene. The layering of the 2D and 3D layers is specified by any relevant transformations in the scene graph. The **Layer2D** node is composed with its center at the origin of the local coordinate system and shall not be present in 3D contexts (see 9.2.2.1).

The **size** parameter shall be a floating point number that expresses the width and height of the layer in the units of the local coordinate system. In case of a layer at the root of the hierarchy, the size is expressed in terms of the default 2D coordinate system (see 9.2.2.2). A size of -1 in either direction, means that the **Layer2D** node is not specified in size in that direction, and that the size is adjusted to the size of the parent layer, or the global rendering area dimension if the layer is on the top of the hierarchy. In the case where a 2D scene or object is shared between several **Layer2D** nodes, the behaviours are defined exactly as for objects that are multiply referenced using the DEF/USE mechanism. A sensor triggers an event whenever the sensor is triggered in any of the **Layer2D** in which it is contained. The behaviors triggered by the shared sensors as well as other behaviors that apply on objects shared between several layers apply on all layers containing these objects.

A **Layer2D** stores the stack of bindable children nodes that can affect the children scene of the layer. All relevant bindable children nodes have a corresponding exposedField in the **Layer2D** node. During presentation, these fields take the value of the currently bound bindable children node for the scene that is a child of the **Layer2D** node. Initially, the bound bindable children node is the corresponding field value of the **Layer2D** node if it is defined. If the field is undefined, the first bindable children node defined in the child scene will be bound. When the binding mechanism of the bindable children node is used (**set_bind** field set to TRUE), all the parent layers containing this node set the corresponding field to the current bound node value. It is therefore possible to share scenes across layers, and to have different bound nodes active, or to trigger a change of bindable children node for all layers containing a given bindable children node. For 2D scenes, the **background** field specifies the bound **Background2D** node. The **viewport** field is reserved for future extensions for 2D scenes.

All the 2D objects contained in a single **Layer2D** node form a single composed object. This composed object is considered by other elements of the scene to be a single object. In other words, if a **Layer2D** node, A, is the parent of two objects, B and C, layered one on top of the other, it will not be possible to insert a new object, D, between B and C unless D is added as a child of A.

Layers are transparent to user input, which means that if two layers are overlapping at a given location on the screen, a user input will affect both layers, regardless of which is drawn on top of the other. For instance, if two buttons placed in two different layers are overlapping, the click of the user at the location of the topmost button will also affect the button contained in the layer behind. Authors should carefully design behaviors in the overlapping layers.

EXAMPLE — In the following example, the same scene is used in two different **Layer2D** nodes. However, one scene is initially viewed with background b1, the other with background b2. When the user clicks on the button1 object, all layers are set with background b3.

```
OrderedGroup{
   children [
      Transform2D { # A set of transforms to translate and scale the layer
         ...
         children [
            Layer2D {
               background DEF b1 Background2D {…}
                  # It is possible to define the bindable children node directly in
                  # the corresponding field
               children [
                  DEF MYSCENE Transform2D {
                     children [
                        DEF b3 Background2D {…} # A shared background
                        DEF TS TouchSensor{}
                        DEF button1 Shape{..}   # The button 1
                                    # The objects of my scene
                     ]
                  }
               ]
            }
         ]
      }
      Transform2D {
                  # Another set of transforms to translate and scale the layer
         children [
            Layer2D {
                children [
                DEF b2 Background2D{…} # It is possible to define the bindable
                                       # children node in the children field.
                                       # b2 is initially bound sicne it is the
                                       # first background 2D in the children
                                       # field OF the parent Layer2d
                Transform2D USE MYSCENE
                ]
              }
            ]
          }
        ]
}

ROUTE TS.isActive TO b3.set_bind
```

9.4.2.54 Layer3D

9.4.2.54.1 Node interface

Layer3D {

eventIn	MFNode	**addChildren**	
eventIn	MFNode	**removeChildren**	
exposedField	MFNode	**children**	NULL
exposedField	SFVec2f	**size**	-1, -1
exposedField	SFNode	**background**	NULL
exposedField	SFNode	**fog**	NULL
exposedField	SFNode	**navigationInfo**	NULL
exposedField	SFNode	**viewpoint**	NULL

}

NOTE — For the binary encoding of this node see Annex H.1.54.

9.4.2.54.2 Functionality and semantics

The **Layer3D** node is a transparent, rectangular rendering region where a 3D scene is drawn. The **Layer3D** node may be composed in the same manner as any other 2D node. It represents a rectangular region on the screen facing the viewer. The basic **Layer3D** semantics are identical to those for **Layer2D** (see 9.4.2.53) but with 3D (rather than 2D) children. In general, **Layer3D** nodes shall not be present in 3D co-ordinate systems. The permitted exception to this in when a **Layer3D** node is the "top" node that begins a 3D scene or context (see 9.2.2.1).

The following fields specify bindable children nodes for **Layer3D**:

— **background** for **Background** nodes

— **fog** for **Fog** nodes

— **navigationInfo** for **NavigationInfo** nodes

— **viewpoint** for **Viewpoint** nodes

The **viewpoint** field can be used to allow the viewing of the same scene with several viewpoints.

NOTE — The rule for transparency to behaviors is also true for navigation in **Layer3D**. Authors should carefully design the various **Layer3D** nodes in a given scene to take account of navigation. Overlapping several **Layer3D** with navigation turned on may trigger strange navigation effects which are difficult to control by the user. Unless it is a feature of the content, navigation can be easily turned off using the **NavigationInfo** type field, or **Layer3D's** can be designed not to be superimposed.

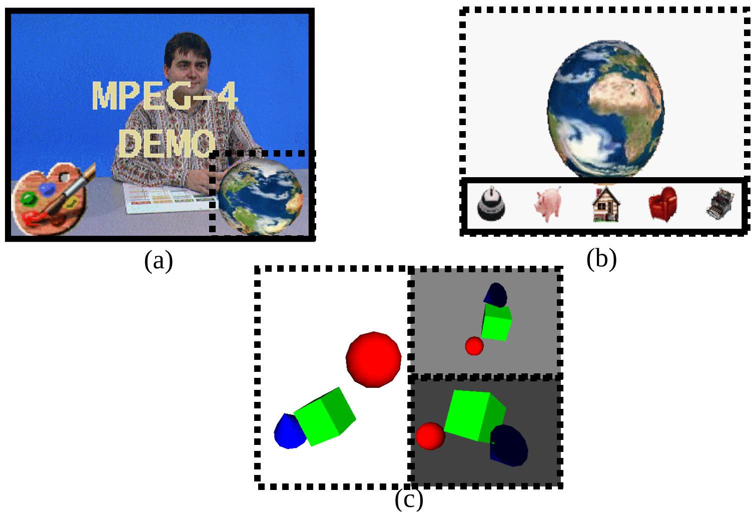


Figure 23 - Three Layer2D and Layer3D examples composed in a 2D space.

Layer2D's are indicated by a continuous line; **Layer3D's** by a dashed line. Image (a) shows a **Layer3D** containing a 3D view of the earth on top of a **Layer2D** composed of a video, a logo and a text. Image (b) shows a **Layer3D** of the earth with a **Layer2D** containing various icons on top. Image (c) shows 3 views of a 3D scene with 3 non-overlapping **Layer3D**.

9.4.2.55 Layout

9.4.2.55.1 Node interface

```
Layout {
   eventIn        MFNode      addChildren
   eventIn        MFNode      removeChildren
   exposedField   MFNode      children              []
   exposedField   SFBool      wrap                  FALSE
   exposedField   SFVec2f     size                  -1, -1
   exposedField   SFBool      horizontal            TRUE
   exposedField   MFString    justify               ["BEGIN"]
   exposedField   SFBool      leftToRight           TRUE
   exposedField   SFBool      topToBottom           TRUE
   exposedField   SFFloat     spacing               1.0
   exposedField   SFBool      smoothScroll          FALSE
   exposedField   SFBool      loop                  FALSE
   exposedField   SFBool      scrollVertical        TRUE
   exposedField   SFFloat     scrollRate            0.0
}
```

NOTE — For the binary encoding of this node see Annex H.1.55.

9.4.2.55.2 Functionality and semantics

The **Layout** node specifies the placement (layout) of its children in various alignment modes as specified. For text children, this is by their **fontStyle** fields, and for non-text children by the fields **horizontal**, **justify**, **leftToRight**, **topToBottom** and **spacing** present in this node. It also provides the functionality of scrolling its children horizontally or vertically.

The **children** field shall specify a list of nodes that are to be arranged. Note that the children's position is implicit and that order is important.

The **wrap** field specifies whether children are allowed to wrap to the next row (or column in vertical alignment cases) after the edge of the layout frame is reached. If **wrap** is set to TRUE, children that would be positioned across or past the frame boundary are wrapped (vertically or horizontally) to the next row or column. If **wrap** is set to FALSE, children are placed in a single row or column that is clipped if it is larger than the layout.

When **wrap** is TRUE, if text objects larger than the layout frame need to be placed, these texts shall be broken down into pieces that are smaller than the layout. The preferred places for breaking text are spaces, tabs, hyphens, carriage returns and line feeds. When there is no such character in the texts to be broken, the texts shall be broken at the last character that is entirely placed in the layout frame.

The **size** field specifies the width and height of the layout frame.

The **horizontal**, **justify**, **leftToRight**, **topToBottom** and **spacing** fields have the same meaning as in the **FontStyle** node (see 9.4.2.44).

The **scrollRate** field specifies the scroll rate in meters per second. When **scrollRate** is zero, then there is no scrolling and the remaining scroll-related fields are ignored.

The **smoothScroll** field selects between smooth and line-by-line/character-by-character scrolling of children. When TRUE, smooth scroll is applied.

The **loop** field specifies continuous looping of children when set to TRUE. When **loop** is FALSE, child nodes that have scrolled out of the scroll layout frame will be deleted. When **loop** is TRUE, then the set of children scrolls continuously, wrapping around when they have scrolled out of the layout area. If the set of children is smaller than the layout area, some empty space will be scrolled with the children. If the set of children is bigger than the layout area, then only some of the children will be displayed at any point in time. When **scrollVertical** is TRUE and **loop** is TRUE and **scrollRate** is negative (top-to-bottom scrolling), then the bottom-most object will reappear on top of the layout frame as soon as the top-most object has scrolled entirely into the layout frame.

The **scrollVertical** field specifies whether the scrolling is done vertically or horizontally. When set to TRUE, the scrolling rate shall be interpreted as a vertical scrolling rate and a positive rate shall be interpreted as scrolling towards the top. When set to FALSE, the scrolling rate shall be interpreted as a horizontal scrolling rate and a positive rate shall mean scrolling to the right.

Objects are placed one by one, in the order they are given in the children list. Text objects are placed according to the **horizontal**, **justify**, **leftToRight**, **topToBottom** and **spacing** fields of their **FontStyle** node. Other objects are placed according to the same fields of the **Layout** node. The reference point for the placement of an object is the reference point as left by the placement of the previous object in the list.

In the case of vertical alignment, objects may be placed with respect to their top, bottom, center or baseline. The baseline of non-text objects is the same as their bottom.

Spacing shall be coherent only within sequences of objects with the same orientation (same value of **horizontal** field). The notions of top edge, bottom edge, base line, vertical center, left edge, right edge, horizontal center, line height and row width shall have a single meaning over coherent sequences of objects. This means that over a sequence of objects where **horizontal** is TRUE, **topToBottom** is TRUE and **spacing** has the same value, then the vertical size of the lines is computed as follows:

— maxAscent is the maximum of the ascent on all text objects.

— maxDescent is the maximum of the descent on all text objects.

— maxHeight is the maximum height of non-text objects.

If the minor mode in the **justify** field of the layout is FIRST (baseline alignment), then the non-text objects shall be aligned on the baseline, which means the vertical size of the line is:

$$\text{size} = \max(\text{maxAscent}, \text{maxHeight}) + \text{maxDescent}$$

If the minor mode in the justify field of the layout is any other value, then the non-text objects shall be aligned with respect to the top, bottom or center, which means the size of the line is:

size = max(maxAscent+maxDescent, maxHeight)

The first line is placed with its top edge flush to the top edge of the layout; the base line is placed maxAscent units lower, and the bottom edge is placed maxDescent units lower. The center line is in the middle, between the top and bottom edges. The top edges of subsequent lines are placed at regular intervals of value spacing × size.

The other cases can be inferred from the above description. When the orientation is vertical, then the baseline, ascent and descent are not useful for the computation of the width of the rows. All objects only have a width. Column size is the maximum width over all objects.

EXAMPLE —

If **wrap** is FALSE:

a) If **horizontal** is TRUE, then objects are placed in a single line. The layout direction is given by the **leftToRight** field. Horizontal alignment in the row is done according to the first argument in **justify** (major mode = flush left, flush right, centered), and vertical alignment is done according to the second argument in **justify** (minor mode = flush top, flush bottom, flush baseline, centered). The **topToBottom** field is meaningless in this configuration.

b) If ***horizontal*** is FALSE, then objects are placed in a single column. The layout direction is given by the ***topToBottom*** field. Vertical alignment in the column is done according to the first argument in ***justify*** (major mode), and horizontal alignment is done according to the second argument in justify (minor mode).

If **wrap** is TRUE:

a) If ***horizontal*** is TRUE, then objects are placed in multiple lines. The layout direction is given by the ***leftToRight*** field. The wrapping direction is given by the ***topToBottom*** field. Horizontal alignment in the lines is done according to the first argument in ***justify*** (major mode), and vertical alignment is done according to the second argument in ***justify*** (minor mode).

b) If ***horizontal*** is FALSE, then objects are placed in multiple column. The layout direction is given by the ***topToBottom*** field. The wrapping direction is given by the ***leftToRight*** field. Vertical alignment in the columns is done according to the first argument in ***justify*** (major mode), and horizontal alignment is done according to the second argument in ***justify*** (minor mode).

If **scrollRate** is zero, then the **Layout** is static and positions change only when children are modified.

If **scrollRate** is non-zero, then the position of the children is updated according to the values of **scrollVertical**, **scrollRate**, **smoothScroll** and **loop**.

If **scrollVertical** is TRUE, then if ***scrollRate*** is positive, then the scrolling direction is left-to-right, and vice-versa.

If **scrollVertical** is FALSE, then if ***scrollRate*** is positive, then the scrolling direction is bottom-to-top, and vice-versa.

9.4.2.56 LineProperties

9.4.2.56.1 Node interface

```
LineProperties {
    exposedField    SFColor      lineColor      0, 0, 0
    exposedField    SFInt32      lineStyle      0
    exposedField    SFFloat      width          1.0
}
```

NOTE — For the binary encoding of this node see Annex H.1.56.

9.4.2.56.2 Functionality and semantics

The **LineProperties** node specifies line parameters used in 2D and 3D rendering.

The **lineColor** field specifies the color with which to draw the lines and outlines of 2D geometries.

The **lineStyle** field shall contain the line style type to apply to lines. The allowed values are:

Table 33 - lineStyle **description**

lineStyle	Description
0	solid
1	dash
2	dot
3	dash-dot
4	dash-dash-dot
5	dash-dot-dot

The terminal shall draw each line style in a manner that is distiguishable from each other line style.

The **width** field determines the width, in the local coordinate system, of rendered lines. The apparent width depends on the local transformation.

The cap and join style to be used are as follows. The wide lines should end with a square form flush with the end of the lines. The join style is described in Figure 24.

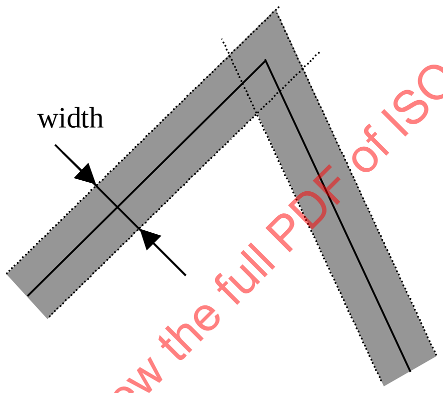


Figure 24 - Cap and join style for LineProperties

9.4.2.57 ListeningPoint

9.4.2.57.1 Node interface

```
ListeningPoint {
    eventIn        SFBool       set_bind
    exposedField   SFBool       jump            TRUE
    exposedField   SFRotation   orientation     0, 0, 1, 0
    exposedField   SFVec3f      position        0, 0, 10
    field          SFString     description     ""
    eventOut       SFTime       bindTime
    eventOut       SFBool       isBound
}
```

NOTE — For the binary encoding of this node see Annex H.1.57.

9.4.2.57.2 Functionality and semantics

The **ListeningPoint** node specifies the reference position and orientation for spatial audio presentation. If there is no **ListeningPoint** given in a scene, the apparent listener position is slaved to the active **ViewPoint**.

The semantics are identical to those of the **Viewpoint** node (see 9.4.2.97).

9.4.2.58 LOD

9.4.2.58.1 Node interface

```
LOD {
    exposedField    MFNode      level               []
    field           SFVec3f     center              0, 0, 0
    field           MFFloat     range               []
}
```

NOTE — For the binary encoding of this node see Annex H.1.58.

9.4.2.58.2 Functionality and semantics

The semantics of the **LOD** node are specified in ISO/IEC 14772-1:1998, subclause 6.26 [10].

9.4.2.59 Material

9.4.2.59.1 Node interface

```
Material {
    exposedField    SFFloat     ambientIntensity    0.2
    exposedField    SFColor     diffuseColor        0.8, 0.8, 0.8
    exposedField    SFColor     emissiveColor       0, 0, 0
    exposedField    SFFloat     shininess           0.2
    exposedField    SFColor     specularColor       0, 0, 0
    exposedField    SFFloat     transparency        0.0
}
```

NOTE — For the binary encoding of this node see Annex H.1.59.

9.4.2.59.2 Functionality and semantics

The semantics of the **Material** node are specified in ISO/IEC 14772-1:1998, subclause 6.27 [10].

9.4.2.60 Material2D

9.4.2.60.1 Node interface

```
Material2D {
    exposedField    SFColor     emissiveColor       0.8, 0.8, 0.8
    exposedField    SFBool      filled              FALSE
    exposedField    SFNode      lineProps           NULL
    exposedField    SFFloat     transparency        0.0
}
```

NOTE — For the binary encoding of this node see Annex H.1.60.

9.4.2.60.2 Functionality and semantics

The **Material2D** node specifies the characteristics of a rendered 2D **Shape**. Material2D shall be used as the material field of an Appearance node in certain circumstances (see 9.4.2.3.2)

The **emissiveColor** field specifies the color of the 2D **Shape**.

The **filled** field specifies whether rendered nodes are filled or drawn using lines. This field affects **IndexedFaceSet2D**, **Circle** and **Rectangle** nodes.

The **lineProps** field contains information about line rendering in the form of a **LineProperties** node. If the field is null the line properties take on a default behaviour identical to the default settings of the **LineProperties** node (see 9.4.2.56) for more information.

The **transparency** field specifies the transparency of the 2D **Shape**.

9.4.2.61 MovieTexture

9.4.2.61.1 Node interface

```
MovieTexture {
    exposedField   SFBool      loop                FALSE
    exposedField   SFFloat     speed               1.0
    exposedField   SFTime      startTime           0
    exposedField   SFTime      stopTime            0
    exposedField   MFString    url                 []
    field          SFBool      repeatS             TRUE
    field          SFBool      repeatT             TRUE
    eventOut       SFTime      duration_changed
    eventOut       SFBool      isActive
}
```

NOTE — For the binary encoding of this node see Annex H.1.61.

9.4.2.61.2 Functionality and semantics

The **loop**, **startTime**, and **stopTime** exposedFields and the **isActive** eventOut, and their effects on the **MovieTexture** node, are described in 9.2.1.6.1.

The **speed** exposedField controls playback speed. It does not affect the delivery of the stream attached to the **MovieTexture** node. For streaming media, value of **speed** other than 1 shall be ignored.

A **MovieTexture** shall display frame or VOP 0 if **speed** is 0. For positive values of **speed**, the frame or VOP that an active **MovieTexture** will display at time *now* corresponds to the frame or VOP at movie time (i.e., in the movie's local time base with frame or VOP 0 at time 0, at speed = 1):

fmod (now - **startTime**, duration/**speed**)

If **speed** is negative, then the frame or VOP to display is the frame or VOP at movie time:

duration + fmod(now - **startTime**, duration/**speed**).

A **MovieTexture** node is inactive before **startTime** is reached. If **speed** is non-negative, then the first VOP shall be used as texture, if it is already available. If **speed** is negative, then the last VOP shall be used as texture, if it is already available.

When a **MovieTexture** becomes inactive, the VOP corresponding to the time at which the **MovieTexture** became inactive shall persist as the texture. The **speed** exposedField indicates how fast the movie shall be played. A speed of 2 indicates the movie plays twice as fast. Note that the **duration_changed** eventOut is not affected by the **speed** exposedField. **set_speed** events shall be ignored while the movie is playing.

The **url** field specifies the data source to be used (see 9.2.2.7.1).

9.4.2.62 NavigationInfo

9.4.2.62.1 Node interface

```
NavigationInfo {
    eventIn        SFBool      set_bind
    exposedField   MFFloat     avatarSize          [0.25, 1.6, 0.75]
    exposedField   SFBool      headlight           TRUE
    exposedField   SFFloat     speed               1.0
    exposedField   MFString    type                ["WALK", "ANY"]
    exposedField    SFFloat     visibilityLimit     0.0
    eventOut        SFBool      isBound
}
```

NOTE — For the binary encoding of this node see Annex H.1.62.

9.4.2.62.2 Functionality and semantics

The semantics of **NavigationInfo** are specified in ISO/IEC 14772-1:1998, subclause 6.29 [10].

9.4.2.63 Normal

9.4.2.63.1 Node interface

```
Normal {
    exposedField    MFVec3f     vector              []
}
```

NOTE — For the binary encoding of this node see Annex H.1.63.

9.4.2.63.2 Functionality and semantics

The semantics of the **Normal** node are specified in ISO/IEC 14772-1:1998, subclause 6.30 [10].

9.4.2.64 NormalInterpolator

9.4.2.64.1 Node interface

```
NormalInterpolator {
    eventIn         SFFloat     set_fraction
    exposedField    MFFloat     key                 []
    exposedField    MFVec3f     keyValue            []
    eventOut        MFVec3f     value_changed
}
```

NOTE — For the binary encoding of this node see Annex H.1.64.

9.4.2.64.2 Functionality and semantics

The semantics of the **NormalInterpolator** node are specified in ISO/IEC 14772-1:1998, subclause 6.31 [10].

9.4.2.65 OrderedGroup

9.4.2.65.1 Node interface

```
OrderedGroup {
    eventIn         MFNode      addChildren
    eventIn         MFNode      removeChildren
    exposedField    MFNode      children            []
    exposedField    MFFloat     order               []
}
```

NOTE — For the binary encoding of this node see Annex H.1.65.

9.4.2.65.2 Functionality and semantics

The **OrderedGroup** node controls the visual layering order of its children. When used as a child of a **Layer2D** node, it allows the control of which shapes obscure others. When used as a child of a **Layer3D** node, it allows content creators to specify the rendering order of elements of the scene that have identical z values. This allows conflicts between coplanar or close polygons to be resolved.

The **addChildren** eventIn specifies a list of objects that shall be added to the **OrderedGroup** node.

The **removeChildren** eventIn specifies a list of objects that shall be removed from the **OrderedGroup** node.

The **children** field is the current list of objects contained in the **OrderedGroup** node.

When the **order** field is empty (the default) children are layered in order, first child to last child, with the last child being rendered last. If the **order** field contains values, one value is assigned to each child. Entries in the **order** field array match the child in the corresponding element of the **children** field array. The child with the lowest order value is rendered before all others. The remaining children are rendered in increasing order. The child corresponding to the highest **order** value is rendered last.

Since 2D shapes have no z value, this is the sole determinant of the visual ordering of the shapes. However, when the **OrderedGroup** node is used with 3D shapes, its ordering mechanism shall be used in place of the natural z order of the shapes themselves. The resultant image shall show the shape with the highest **order** value on top, regardless of its z value. However, the resultant z-buffer contains a z value corresponding to the shape closest to the viewer at that pixel. The **order** shall be used to specify which geometry should be drawn first, to avoid conflicts between coplanar or close polygons.

NOTE — Content authors must use this functionality carefully since, depending on the **Viewpoint**, 3D shapes behind a given object in the natural z order may appear in front of this object.

9.4.2.66 OrientationInterpolator

9.4.2.66.1 Node interface

```
OrientationInterpolator {
    eventIn         SFFloat         set_fraction
    exposedField    MFFloat         key                 []
    exposedField    MFRotation      keyValue            []
    eventOut        SFRotation      value_changed
}
```

NOTE — For the binary encoding of this node see Annex H.1.66.

9.4.2.66.2 Functionality and semantics

The semantics of the **OrientationInterpolator** node are specified in ISO/IEC 14772-1:1998, subclause 6.32 [10].

9.4.2.67 PixelTexture

9.4.2.67.1 Node interface

```
PixelTexture {
    exposedField    SFImage         image               0 0 0
    field           SFBool          repeatS             TRUE
    field           SFBool          repeatT             TRUE
}
```

NOTE — For the binary encoding of this node see Annex H.1.67.

9.4.2.67.2 Functionality and semantics

The semantics of the **PixelTexture** node are specified in ISO/IEC 14772-1:1998, subclause 6.33 [10].

9.4.2.68 PlaneSensor

9.4.2.68.1 Node interface

```
PlaneSensor {
    exposedField   SFBool     autoOffset          TRUE
    exposedField   SFBool     enabled             TRUE
    exposedField   SFVec2f    maxPosition         -1 -1
    exposedField   SFVec2f    minPosition         0 0
    exposedField   SFVecf3f   offset              0 0 0
    eventOut       SFBool     isActive
    eventOut       SFVec3f    trackPoint_changed
    eventOut       SFVec3f    translation_changed
}
```

9.4.2.68.2 Fnctionality and semantics

The semantics of the **PlaneSensor** node are specified in ISO/IEC 14772-1:1998, subclause 6.34 [10].

9.4.2.69 PlaneSensor2D

9.4.2.69.1 Node interface

```
PlaneSensor2D {
    exposedField   SFBool     autoOffset          TRUE
    exposedField   SFBool     enabled             TRUE
    exposedField   SFVec2f    maxPosition         0, 0
    exposedField   SFVec2f    minPosition         0, 0
    exposedField   SFVec2f    offset              0, 0
    eventOut       SFBool     isActive
    eventOut       SFVec2f    trackPoint_changed
    eventOut       SFVec2f    translation_changed
}
```

NOTE — For the binary encoding of this node see Annex H.1.68.

9.4.2.69.2 Functionality and semantics

This sensor detects pointer device dragging and enables the dragging of objects on the 2D rendering plane.

The semantics of **PlaneSensor2D** are a restricted case for 2D of the semantics for the **PlaneSensor** node (see 9.4.2.68).

9.4.2.70 PointLight

9.4.2.70.1 Node interface

```
PointLight {
    exposedField   SFFloat    ambientIntensity    0.0
    exposedField   SFVec3f    attenuation         1, 0, 0
    exposedField   SFColor    color               1, 1, 1
    exposedField   SFFloat    intensity           1.0
    exposedField   SFVec3f    location            0, 0, 0
    exposedField   SFBool     on                  TRUE
    exposedField   SFFloat    radius              100.0
}
```

NOTE — For the binary encoding of this node see Annex H.1.69.

9.4.2.70.2 Functionality and semantics

The semantics of the **PointLight** node are specified in ISO/IEC 14772-1:1998, subclause 6.35 [10].

9.4.2.71 PointSet

9.4.2.71.1 Node interface

```
PointSet {
    exposedField    SFNode      color       NULL
    exposedField    SFNode      coord       NULL
}
```

NOTE — For the binary encoding of this node see Annex H.1.70.

9.4.2.71.2 Functionality and semantics

The semantics of the **PointSet** node are specified in ISO/IEC 14772-1:1998, subclause 6.36 [10].

9.4.2.72 PointSet2D

9.4.2.72.1 Node interface

```
PointSet2D {
    exposedField    SFNode      color       NULL
    exposedField    SFNode      coord       NULL
}
```

NOTE — For the binary encoding of this node see Annex H.1.71.

9.4.2.72.2 Functionality and semantics

This is a 2D equivalent of the **PointSet** node (see 9.4.2.71), with semantics that are the 2D restriction of that node.

9.4.2.73 PositionInterpolator

9.4.2.73.1 Node interface

```
PositionInterpolator {
    eventIn         SFFloat     set_fraction
    exposedField    MFFloat     key             []
    exposedField    MFVec3f     keyValue        []
    eventOut        SFVec3f     value_changed
}
```

NOTE — For the binary encoding of this node see Annex H.1.72.

9.4.2.73.2 Functionality and semantics

The semantics of the **PositionInterpolator** node are specified in ISO/IEC 14772-1:1998, subclause 6.37 [10].

9.4.2.74 PositionInterpolator2D

9.4.2.74.1 Node interface

```
PositionInterpolator2D {
    eventIn         SFFloat     set_fraction
    exposedField    MFFloat     key             []
    exposedField   MFVec2f     keyValue           []
    eventOut       SFVec2f     value_changed
}
```

NOTE — For the binary encoding of this node see Annex H.1.73.

9.4.2.74.2 Functionality and semantics

This is a 2D equivalent of the **PositionInterpolator** node (see 9.4.2.73) with semantics that are the 2D restriction of that node.

9.4.2.75 ProximitySensor

9.4.2.75.1 Node interface

```
ProximitySensor {
    exposedField   SFVec3f     center             0, 0, 0
    exposedField   SFVec3f     size               0, 0, 0
    exposedField   SFBool      enabled            TRUE
    eventOut       SFBool      isActive
    eventOut       SFVec3f     position_changed
    eventOut       SFRotation  orientation_changed
    eventOut       SFTime      enterTime
    eventOut       SFTime      exitTime
}
```

NOTE — For the binary encoding of this node see Annex H.1.74.

9.4.2.75.2 Functionality and semantics

The semantics of the **ProximitySensor** node are specified in ISO/IEC 14772-1:1998, subclause 6.38 [10].

9.4.2.76 ProximitySensor2D

9.4.2.76.1 Node interface

```
ProximitySensor2D {
    exposedField   SFVec2f     center             0, 0
    exposedField   SFVec2f     size               0, 0
    exposedField   SFBool      enabled            TRUE
    eventOut       SFBool      isActive
    eventOut       SFVec2f     position_changed
    eventOut       SFFloat     orientation_changed
    eventOut       SFTime      enterTime
    eventOut       SFTime      exitTime
}
```

NOTE — For the binary encoding of this node see Annex H.1.75.

9.4.2.76.2 Functionality and semantics

This is the 2D equivalent of the **ProximitySensor** node (see 9.4.2.75) with semantics that are the 2D restriction of the that node.

9.4.2.77 QuantizationParameter

9.4.2.77.1 Node interface

```
QuantizationParameter {
    field          SFBool      isLocal            FALSE
    field          SFBool      position3DQuant    FALSE
    field      SFVec3f     position3DMin            -∞, -∞, -∞
    field      SFVec3f     position3DMax            +∞, +∞, +∞
    field      SFInt32     position3DNbBits         16
    field      SFBool      position2DQuant          FALSE
    field      SFVec2f     position2DMin            -∞, -∞
    field      SFVec2f     position2DMax            +∞, +∞
    field      SFInt32     position2DNbBits         16
    field      SFBool      drawOrderQuant           TRUE
    field      SFVec3f     drawOrderMin             -∞
    field      SFVec3f     drawOrderMax             +∞
    field      SFInt32     drawOrderNbBits          8
    field      SFBool      colorQuant               TRUE
    field      SFFloat     colorMin                 0.0
    field      SFFloat     colorMax                 1.0
    field      SFInt32     colorNbBits              8
    field      SFBool      textureCoordinateQuant   TRUE
    field      SFFloat     textureCoordinateMin     0.0
    field      SFFloat     textureCoordinateMax     1.0
    field      SFInt32     textureCoordinateNbBits  16
    field      SFBool      angleQuant               TRUE
    field      SFFloat     angleMin                 0.0
    field      SFFloat     angleMax                 2¶
    field      SFInt32     angleNbBits              16
    field      SFBool      scaleQuant               FALSE
    field      SFFloat     scaleMin                 0.0
    field      SFFloat     scaleMax                 +∞
    field      SFInt32     scaleNbBits              8
    field      SFBool      keyQuant                 TRUE
    field      SFFloat     keyMin                   0.0
    field      SFFloat     keyMax                   1.0
    field      SFInt32     keyNbBits                8
    field      SFBool      normalQuant              TRUE
    field      SFInt32     normalNbBits             8
    field      SFBool      sizeQuant                FALSE
    field      SFFloat     sizeMin                  0.0
    field      SFFloat     sizeMax                  +∞
    field      SFInt32     sizeNbBits               8
    field      SFBool      useEfficientCoding       FALSE
}
```

NOTE — For the binary encoding of this node see Annex H.1.76.

9.4.2.77.2 Functionality and semantics

The **QuantizationParameter** node describes the quantization values to be applied on single fields of numerical types. For each of identified categories of fields, a minimal and maximal value is given as well as a number of bits to represent the given class of fields. Additionally, it is possible to set the **isLocal** field to apply the quantization only to the node following the **QuantizationParameter** node. The use of a node structure for declaring the quantization parameters allows the application of the DEF and USE mechanisms that enable reuse of the **QuantizationParameter** node. Also, it enables the parsing of this node in the same manner as any other scene information.

The **QuantizationParameter** node may only appear as a child of a grouping node. When a **QuantizationParameter** node appears in the scene graph, the quantization is set to TRUE, and will apply to subsequent nodes as follows:

If the **isLocal** boolean is set to FALSE, the quantization applies to all siblings following the **QuanitzationParameter** node, and thus to all their children as well.

If the **isLocal** boolean is set to TRUE, the quantization only applies to the following sibling node in the children list of the parent node. If no sibling is following the **QuantizationParameter** node declaration, the node has no effect.

In all cases, the quantization is applied only in the scope of a single BIFS command. That is, if a command in the same access unit, or in another access unit inserts a node in a context in which the quantization was active, no quantization will be applied, except if a new **QuantizationParameter** node is defined in this new command.

The information contained in the **QuantizationParameter** node fields applies within the context of the node scope as follows. For each category of fields, a boolean sets the quantization on or off, the minimal and maximal values are set, as well as the number of bits for the quantization. This information, combined with the node coding table, enables the relevant information to quantize the fields to be obtained. The quantization parameters are applied as explained in 9.3.3.

If the **useEfficientCoding** boolean is set to FALSE, the encoding of floats shall be performed using 32 bits, according to IEEE Std 754-1985 [12].

If the **useEfficientCoding** boolean is set to TRUE, the encoding of floats shall use the syntax described in 9.3.7.11. The scope of the use of the efficient coding is the same as that of the **QuantizationParameter** node. This means that the values of the fields of the current **QuantizationParameter** node are not sent in the efficient coding mode unless the context is within the scope of a previously sent **QuantizationParameter** whose **useEfficientCoding** bit was set to true.

9.4.2.78 Rectangle

9.4.2.78.1 Node interface

```
Rectangle {
    exposedField    SFVec2f    size              2, 2
}
```

NOTE — For the binary encoding of this node see Annex H.1.77.

9.4.2.78.2 Functionality and semantics

This node specifies a rectangle. The **size** field specifies the horizontal and vertical size of the rendered rectangle.

9.4.2.79 ScalarInterpolator

9.4.2.79.1 Node interface

```
ScalarInterpolator {
    eventIn         SFFloat    set_fraction
    exposedField    MFFloat    key               []
    exposedField    MFFloat    keyValue          []
    eventOut        SFFloat    value_changed
}
```

NOTE — For the binary encoding of this node see Annex H.1.78.

9.4.2.79.2 Functionality and semantics

The semantics of the **ScalarInterpolator** node are specified in ISO/IEC 14772-1:1998, subclause 6.39 [10].

9.4.2.80 Script

9.4.2.80.1 Node interface

```
Script {
    exposedField    MFString   url               []
    field           SFBool     directOutput      FALSE
```

```
   field        SFBool       mustEvaluate          FALSE
   Any number of the following may then follow:
   eventIn      eventType    eventName
   field        fieldType    fieldName             initialValue
   eventOut     eventType    eventName
}
```

NOTE — For the binary encoding of this node see Annex H.1.79.

9.4.2.80.2 Functionality and semantics

The **Script** node is used to describe behaviour in a programmtic way in a scene. Script nodes typically

— signify a change or user action

— receive events from other nodes

— contain a program module that performs some computation

— effect change somewhere else in the scene by sending events

Each **Script** node has associated programming language code, referenced by the **url** field, that is executed to carry out the **Script** node's function. That code is referred to as the "script" in the rest of this description.

9.4.2.80.2.1 Detailed Semantics

The semantics of this node are as defined in ISO/IEC 14772-1:1998, subclause 6.40 [10], with the following exception. The interface functions `CreateVRMLFromString()` and `CreateVRMLFromURL()` are not supported. The terminal shall support JavaScript.

EXAMPLE — The following scene contains two spheres that exchange colors when they are clicked with the mouse. The script is used to hold the current color state (in the variable num). The script variables `color1` and `color2` are used to hold the colors that are flipped back and forth between the two spheres. The script variable `color` is used to hold the last color state of the first sphere, and this color is routed to the second sphere. The first sphere color is set directly in the script.

```
Group {
   children [
      Viewpoint {
         fieldOfView 0.785398
      }
      DirectionalLight {
         color 1 1 1
      }
      Shape {
         geometry Sphere { radius 0.5}      # first sphere…
         appearance Appearance {
            material DEF COLOR Material {diffuseColor 1 0 0}
         }
      }
      Transform {
         translation -2 0 0
         children [
            Shape {
               geometry Sphere { radius 1.0} #second sphere…
               appearance Appearance {
               material DEF COLOR2 Material {diffuseColor 1 1 1}
               }
            }
            DEF TS TouchSensor{} #clicking on the 2nd sphere will activate the script
         ]
      }
      DEF SC Script {
         eventIn SFBool touch
         field SFNode node USE COLOR
         field SFColor color1 0 1 0 # constant color for sphere
         field SFColor color2 0 0 1 # same as above
         field SFInt32 num 1  # holds the current color state
         eventOut SFColor color # holds the last color in COLOR
         url "javascript:
            function touch (value, tp) {
               color = node.diffuseColor;
               if (num==1) {
                  node.diffuseColor = color1;
                  num = 2;
               } else {
                  node.diffuseColor = color2;
                  num = 1;
               }
            }
         "
      }
   ]
}
ROUTE TS.isActive TO SC.touch     # activates the script when sensor is touched
ROUTE SC.color TO COLOR2.diffuseColor # routes the last color of COLOR to COLOR2
```

9.4.2.81 Shape

9.4.2.81.1 Node interface

Shape {

exposedField	SFNode	**appearance**	NULL
exposedField	SFNode	**geometry**	NULL

}

NOTE — For the binary encoding of this node see Annex H.1.80.

9.4.2.81.2 Functionality and semantics

The semantics of the **Shape** node are specified in ISO/IEC 14772-1:1998, subclause 6.41 [10].

9.4.2.82 Sound

9.4.2.82.1 Node interface

Sound {

exposedField	SFVec3f	**direction**	0, 0, 1
exposedField	SFFloat	**intensity**	1.0
exposedField	SFVec3f	**location**	0, 0, 0
exposedField	SFFloat	**maxBack**	10.0
exposedField	SFFloat	**maxFront**	10.0
exposedField	SFFloat	**minBack**	1.0
exposedField	SFFloat	**minFront**	1.0
exposedField	SFFloat	**priority**	0.0
exposedField	SFNode	**source**	NULL
field	SFBool	**spatialize**	TRUE

}

NOTE — For the binary encoding of this node see Annex H.1.81.

9.4.2.82.2 Functionality and semantics

The **Sound** node is used to attach sound to a scene, thereby giving it spatial qualities and relating it to the visual content of the scene.

The **Sound** node relates an audio BIFS sub-graph to the rest of an audio-visual scene. By using this node, sound may be attached to a group, and spatialized or moved around as appropriate for the spatial transforms above the node. By using the functionality of the audio BIFS nodes, sounds in an audio scene dscribed using ISO/IEC 14496-1 may be filtered and mixed before being spatially composited into the scene.

The semantics of this node are as defined in ISO/IEC 14472-1:1997, subclause 6.42, with the following exceptions and additions.

The **source** field allows the connection of an audio sub-graph containing the sound.

The **spatialize** field determines whether the **Sound** shall be spatialized. If this flag is set, the sound shall be presented spatially according to the local coordinate system and current **listeningPoint**, so that it apparently comes from a source located at the **location** point, facing in the direction given by **direction**. The exact manner of spatialization is implementation-dependant, but implementators are encouraged to provide the maximum sophistication possible depending on terminal resources.

If there are multiple channels of sound output from the child sound, they may or may not be spatialized, according to the **phaseGroup** properties of the child, as follows. Any individual channels, that is, channels not phase-related to other channels, are summed linearly and then spatialized. Any phase-grouped channels are not spatialized, but passed through this node unchanged. The sound presented in the scene is thus a single spatialized sound, represented by the sum of the individual channels, plus an “ambient” sound represented by mapping all the remaining channels into the presentation system as described in 9.2.2.13.2.2.

If the **spatialize** field is not set, the audio channels from the child are passed through unchanged, and the sound presented in the scene due to this node is an “ambient” sound represented by mapping all the audio channels output by the child into the presentation system as described in 9.2.2.13.2.2.

As with the visual objects in the scene, the **Sound** node may be included as a child or descendant of any of the grouping or transform nodes. For each of these nodes, the sound semantics are as follows.

Affine transformations presented in the grouping and transform nodes affect the apparent spatialization position of spatialized sound. They have no effect on “ambient” sounds.

If a particular grouping or transform node has multiple **Sound** nodes as descendants, then they are combined for presentation as follows. Each of the **Sound** nodes may be producing a spatialized sound, a multichannel ambient sound, or both. For all of the spatialized sounds in descendant nodes, the sounds are linearly combined through simple summation from presentation. For multichannel ambient sounds, the sounds are linearly combined channel-by-channel for presentation.

EXAMPLE — **Sound** node S1 generates a spatialized sound s1 and five channels of multichannel ambient sound a1[1-5]. **Sound** node S2 generates a spatialized sound s2 and two channels of multichannel ambient sound a2[1-2]. S1 and S2 are grouped under a single **Group** node. The resulting sound is the superposition of the spatialized sound s1, the spatialized sound s2, and the five-channel ambient multichannel sound represented by a3[1-5], where

a3[1] = a1[1] + a2[1]

a3[2] = a1[2] + a2[2]

a3[3] = a1[3]

a3[4] = a1[4]

a3[5] = a1[5]

9.4.2.83 Sound2D

9.4.2.83.1 Node interface

```
Sound2D {
    exposedField    SFFloat     intensity           1.0
    exposedField    SFVec2f     location            0,0
    exposedField   SFNode     source                   NULL
    field          SFBool     spatialize               TRUE
}
```

NOTE — For the binary encoding of this node see Annex H.1.82.

9.4.2.83.2 Functionality and semantics

The **Sound2D** node relates an audio BIFS sub-graph to the other parts of a 2D audio-visual scene. It shall not be used in 3D contexts (see 9.2.2.1). By using this node, sound may be attached to a group of visual nodes. By using the functionality of the audio BIFS nodes, sounds in an audio scene may be filtered and mixed before being spatially composed into the scene.

The **intensity** field adjusts the loudness of the sound. Its value ranges from 0.0 to 1.0, and this value specifies a factor that is used during the playback of the sound.

The **location** field specifies the location of the sound in the 2D scene.

The **source** field connects the audio source to the **Sound2D** node.

The **spatialize** field specifies whether the sound shall be spatialized on the 2D screen. If this flag is set, the sound shall be spatialized with the maximum sophistication possible. The 2D sound is spatialized assuming a distance of one meter between the user and a 2D scene of size 2m x 1.5m, giving the minimum and maximum azimuth angles of –45° and +45°, and the minimum and maximum elevation angles of -37° and +37 °.

The same rules for multichannel audio spatialization apply to the **Sound2D** node as to the **Sound** (3D) node (see 9.4.2.82). Using the **phaseGroup** flag in the **AudioSource** node it is possible to determine whether the channels of the source sound contain important phase relations, and that spatialization at the terminal should not be performed.

As with the visual objects in the scene (and for the **Sound** node), the **Sound2D** node may be included as a child or descendant of any of the grouping or transform nodes. For each of these nodes, the sound semantics are as follows.

Affine transformations presented in the grouping and transform nodes affect the apparent spatialization position of spatialized sound.

If a transform node has multiple **Sound2D** nodes as descendants, then they are combined for presentation as described in 9.4.2.82. If **Sound** and **Sound2D** nodes are both used in a scene, all shall be treated the same way according to these semantics.

9.4.2.84 Sphere

9.4.2.84.1 Node interface

```
Sphere {
    field          SFFloat    Radius                   1.0
}
```

NOTE — For the binary encoding of this node see Annex H.1.83.

9.4.2.84.2 Functionality and semantics

The semantics of the **Sphere** node are specified in ISO/IEC 14772-1:1998, subclause 6.43 [10].

9.4.2.85 SphereSensor

9.4.2.85.1 Node interface

SphereSensor {

exposedField	SFBool	**autoOffset**	TRUE
exposedField	SFBool	**enabled**	TRUE
exposedField	SFRotation	**offset**	0 1 0 0
eventOut	SFBool	**isActive**	
eventOut	SFRotation	**rotation_changed**	
eventOut	SFVec3f	**trackPoint_changed**	

}

NOTE — For the binary encoding of this node see Annex H.1.84.

9.4.2.85.2 Functionality and semantics

The semantics of the **SphereSensor** node are specified in ISO/IEC 14772-1:1998, subclause 6.44 [10].

9.4.2.86 SpotLight

9.4.2.86.1 Node interface

SpotLight {

exposedField	SFFloat	**ambientIntensity**	0.0
exposedField	SFVec3f	**attenuation**	1, 0, 0
exposedField	SFFloat	**beamWidth**	1.5708
exposedField	SFColor	**color**	1, 1, 1
exposedField	SFFloat	**cutOffAngle**	0.785398
exposedField	SFVec3f	**direction**	0, 0, -1
exposedField	SFFloat	**intensity**	1.0
exposedField	SFVec3f	**location**	0, 0, 0
exposedField	SFBool	**on**	TRUE
exposedField	SFFloat	**radius**	100.0

}

NOTE — For the binary encoding of this node see Annex H.1.85.

9.4.2.86.2 Functionality and semantics

The semantics of the **SpotLight** node are specified in ISO/IEC 14772-1:1998, subclause 6.45 [10].

9.4.2.87 Switch

9.4.2.87.1 Node interface

Switch {

exposedField	MFNode	**choice**	[]
exposedField	SFInt32	**whichChoice**	-1

}

NOTE — For the binary encoding of this node see Annex H.1.86.

9.4.2.87.2 Functionality and semantics

The semantics of the **Switch** node are specified in ISO/IEC 14772-1:1998, subclause 6.46 [10], with the following restrictions.

If some of the child sub-graphs contain audio content (i.e., the subgraphs contain **Sound** nodes), the child sounds are switched on and off according to the value of the **whichChoice** field. That is, only sound that corresponds to **Sound** nodes in the **whichChoice'th** subgraph of this node are played. The others are muted.

9.4.2.88 TermCap

9.4.2.88.1 Node interface

```
TermCap {
    eventIn     SFTime      evaluate
    field       SFInt32     capability        0
    eventOut    SFInt32     value
}
```

NOTE — For the binary encoding of this node see Annex H.1.87.

9.4.2.88.2 Functionality and semantics

The **TermCap** node is used to query the resources of the terminal. By ROUTEing the result to a **Switch** node, simple adaptive content may be authored using BIFS.

When this node is instantiated, the value of the **capability** field shall be examined by the system and the **value** eventOut generated to indicate the associated system capability. The **value** eventOut is updated and generated whenever an **evaluate** eventIn is received.

The **capability** field specifies a terminal resource to query. The semantics of the **value** field vary depending on the value of this field. The capabilities which may be queried are:

Table 34 - Semantics of value**, dependent on** capability

capability	Semantics of value
0	frame rate
1	color depth
2	screen size
3	graphics hardware
32	audio output format
33	maximum audio sampling rate
34	spatial audio capability
64	CPU load
65	memory load

The exact semantics differ depending on the value of the **capability** field, as follows.

capability: 0 (frame rate)

For this value of **capability**, the current rendering frame rate is measured. The exact method of measurement not specified.

Table 35 - Semantics of value **for** capability**=0**

value	Semantics
0	unknown or can't determine
1	less than 5 fps
2	5-10 fps
3	10-20 fps
4	20-40 fps
5	more than 40 fps

For the breakpoint between overlapping values between each range (i.e. 5, 10, 20, and 40), the higher value of **value** shall be used (ie, 2, 3, 4, and 5 respectively). This applies to each of the subsequent **capability-value** tables as well.

capability: 1 (color depth)

For this value of **capability**, the color depth of the rendering terminal is measured. At the time this node is instantiated, the **value** field is set to indicate the color depth as follows:

Table 36 - Semantics of value **for** capability**=1**

value	**Semantics**
0	unknown or can't determine
1	1 bit/pixel
2	grayscale
3	color, 3-12 bit/pixel
4	color, 12-24 bit/pixel
5	color, more than 24 bit/pixel

capability: 2 (screen size)

For this value of **capability**, the window size (in horizontal lines) of the output window of the rendering terminal is measured:

Table 37 - Semantics of value **for** capability**=2**

value	**Semantics**
0	unknown or can't determine
1	less than 200 lines
2	200-400 lines
3	400-800 lines
4	800-1600 lines
5	1600 or more lines

capability: 3 (graphics hardware)

For this value of **capability**, the available of graphics acceleration hardware of the rendering terminal is measured. At the time this node is instantiated, the **value** field is set to indicate the available graphics hardware:

Table 38 - Semantics of value **for** capability**=3**

value	**Semantics**
0	unknown or can't determine
1	no acceleration
2	matrix multiplication
3	matrix multiplication + texture mapping (less than 1M memory)
4	matrix multiplication + texture mapping (less than 4M memory)
5	matrix multiplication + texture mapping (more than 4M memory)

capability: 32 (audio output format)

For this value of **capability**, the audio output format (speaker configuration) of the rendering terminal is measured. At the time this node is instantiated, the **value** field is set to indicate the audio output format.

Table 39 - Semantics of value **for** capability**=32**

value	**Semantics**
0	unknown or can't determine
1	mono
2	stereo speakers
3	stereo headphones
4	five-channel surround
5	more than five speakers

capability: 33 (maximum audio sampling rate)

For this value of **capability**, the maximum audio output sampling rate of the rendering terminal is measured. At the time this node is instantiated, the **value** field is set to indicate the maximum audio output sampling rate.

Table 40 - Semantics of value **for** capability**=33**

value	**Semantics**
0	unknown or can't determine
1	less than 16000 Hz
2	16000-32000 Hz
3	32000-44100 Hz
4	44100-48000 Hz
5	48000 Hz or more

capability: 34 (spatial audio capability)

For this value of **capability**, the spatial audio capability of the rendering terminal is measured. At the time this node is instantiated, the **value** field is set to indicate the spatial audio capability.

Table 41 - Semantics of value **for** capability**=34**

value	**Semantics**
0	unknown or can't determine
1	no spatial audio
2	panning only
3	azimuth only
4	full 3-D spatial audio

capability: 64 (CPU load)

For this value of **capability**, the CPU load of the rendering terminal is measured. The exact method of measurement is not specified. The value of the **value** eventOut indicates the available CPU resources as a percentage of the maximum available; that is, if all of the CPU cycles are being consumed, and no extra calculation can be performed without compromising real-time performance, the indicated value is 100%; if twice as much calculation as currently being done can be so performed, the indicated value is 50%.

Table 42 - Semantics of value **for** capability**=64**

value	**Semantics**
0	unknown or can't determine
1	less than 20% loaded
2	20-40% loaded
3	40-60% loaded
4	60-80% loaded
5	80-100% loaded

capability: 65 (RAM available)

For this value of **capability**, the available memory of the rendering terminal is measured. The exact method of measurement is not specified.

Table 43 - Semantics of value **for** capability**=65**

value	**Semantics**
0	unknown or can't determine
1	less than 100 KB free
2	100 KB – 500 KB free
3	500 KB – 2 MB free
4	2 MB – 8 MB free
5	8 MB – 32 MB free
6	32 MB – 200 MB free
7	more than 200 MB free

9.4.2.89 Text

9.4.2.89.1 Node interface

```
Text {
    exposedField    MFString    string        []
    exposedField    MFFloat     length        []
    exposedField    SFNode      fontStyle     NULL
    exposedField    SFFloat     maxExtent     0.0
}
```

NOTE — For the binary encoding of this node see Annex H.1.88.

9.4.2.89.2 Functionality and semantics

The semantics of the **Text** node are specified in ISO/IEC 14772-1:1998, subclause 6.47 [10].

9.4.2.90 TextureCoordinate

9.4.2.90.1 Node interface

```
TextureCoordinate {
    exposedField    MFVec2f     point         []
}
```

NOTE — For the binary encoding of this node see Annex H.1.89.

9.4.2.90.2 Functionality and semantics

The semantics of the **TextureCoordinate** node are specified in ISO/IEC 14772-1:1998, subclause 6.48 [10].

9.4.2.91 TextureTransform

9.4.2.91.1 Node interface

```
TextureTransform {
   exposedField   SFVec2f    center            0, 0
   exposedField   SFFloat    rotation          0.0
   exposedField   SFVec2f    scale             1, 1
   exposedField   SFVec2f    translation       0, 0
}
```

NOTE — For the binary encoding of this node see Annex H.1.90.

9.4.2.91.2 Functionality and semantics

The semantics of the **TextureTransform** node are specified in ISO/IEC 14772-1:1998, subclause 6.49 [10].

9.4.2.92 TimeSensor

9.4.2.92.1 Node interface

```
TimeSensor {
   exposedField   SFTime     cycleInterval     1
   exposedField   SFBool     enabled           TRUE
   exposedField   SFBool     loop              FALSE
   exposedField   SFTime     startTime         0
   exposedField   SFTime     stopTime          0
   eventOut       SFTime     cycleTime
   eventOut       SFFloat    fraction_changed
   eventOut       SFBool     isActive
   eventOut       SFTime     time
}
```

NOTE — For the binary encoding of this node see Annex H.1.91.

9.4.2.92.2 Functionality and semantics

The semantics of the **TimeSensor** node are specified in ISO/IEC 14772-1:1998, subclause 6.50 [10].

9.4.2.93 TouchSensor

9.4.2.93.1 Node interface

```
TouchSensor {
   exposedField   SFBool     enabled           TRUE
   eventOut       SFVec3f    hitNormal_changed
   eventOut       SFVec3f    hitPoint_changed
   eventOut       SFVec2f    hitTexCoord_changed
   eventOut       SFBool     isActive
   eventOut       SFBool     isOver
   eventOut       SFTime     touchTime
}
```

NOTE — For the binary encoding of this node see Annex H.1.92.

9.4.2.93.2 Functionality and semantics

The semantics of the **TouchSensor** node are specified in ISO/IEC 14772-1:1998, subclause 6.51 [10].

9.4.2.94 Transform

9.4.2.94.1 Node interface

```
Transform {
    eventIn         MFNode        addChildren
    eventIn         MFNode        removeChildren
    exposedField    SFVec3f       center                0, 0, 0
    exposedField    MFNode        children              []
    exposedField    SFRotation    rotation              0, 0, 1, 0
    exposedField    SFVec3f       scale                 1, 1, 1
    exposedField    SFRotation    scaleOrientation      0, 0, 1, 0
    exposedField    SFVec3f       translation           0, 0, 0
}
```

NOTE — For the binary encoding of this node see Annex H.1.93.

9.4.2.94.2 Functionality and semantics

The semantics of the **Transform** node are specified in ISO/IEC 14772-1:1998, subclause 6.52 [10]. ISO/IEC 14496-1 does not support the bounding box parameters (**bboxCenter** and **bboxSize**).

If some of the child subgraphs contain audio content (i.e., the subgraphs contain **Sound** nodes), the child sounds are transformed and mixed as follows.

If each of the child sounds is a spatially presented sound, the **Transform** node applies to the local coordinate system of the **Sound** nodes to alter the apparent spatial location and direction. If the children are not spatially presented but have equal numbers of channels, the **Transform** node has no effect on the childrens' sounds. After any such transformation, the combination of sounds is performed as described in 9.4.2.82.

If the children are not spatially presented but have equal numbers of channels, the **Transform** node has no effect on the childrens' sounds. The child sounds are summed equally to produce the audio output at this node.

If some children are spatially presented and some not, or all children do not have equal numbers of channels, the semantics are not defined.

9.4.2.95 Transform2D

9.4.2.95.1 Node interface

```
Transform2D {
    eventIn         MFNode        addChildren
    eventIn         MFNode        removeChildren
    exposedField    SFVec2f       center                0, 0
    exposedField    MFNode        children              []
    exposedField    SFFloat       rotationAngle         0.0
    exposedField    SFVec2f       scale                 1, 1
    exposedField    SFFloat       scaleOrientation      0.0
    exposedField    SFVec2f       translation           0, 0
}
```

NOTE — For the binary encoding of this node see Annex H.1.94.

9.4.2.95.2 Functionality and semantics

The **Transform2D** node allows the translation, rotation and scaling of its 2D children objects.

The **rotation** field specifies a rotation of the child objects, in radians, which occurs about the point specified by **center**.

The **scale** field specifies a 2D scaling of the child objects. The scaling operation takes place following a rotation of the 2D coordinate system that is specified, in radians, by the **scaleOrientation** field. The rotation of the co-ordinate system is notional and purely for the purpose of applying the scaling and is undone before any further actions are performed. No permanent rotation of the co-ordinate system is implied.

The **translation** field specifies a 2D vector which translates the child objects.

The scaling, rotation and translation are applied in the following order: scale, rotate, translate.

The **children** field contains a list of zero or more children nodes which are grouped by the **Transform2D** node.

The **addChildren** and **removeChildren** eventIns are used to add or remove child nodes from the **children** field of the node. Children are added to the end of the list of children and special note should be taken of the implications of this for implicit drawing orders.

If some of the child subgraphs contain audio content (i.e., the subgraphs contain **Sound** nodes), the child sounds are transformed and mixed as follows.

If each of the child sounds is a spatially presented sound, the **Transform** node applies to the local coordinate system of the **Sound** nodes to alter the apparent spatial location and direction. If the children are not spatially presented but have equal numbers of channels, the **Transform** node has no effect on the childrens' sounds. After any such transformation, the combination of sounds is performed as described in 9.4.2.82.

If the children are not spatially presented but have equal numbers of channels, the **Transform** node has no effect on the children's sounds. The child sounds are summed equally to produce the audio output at this node.

If some children are spatially presented and some not, or all children do not have equal numbers of channels, the semantics are not defined.

9.4.2.96 Valuator

9.4.2.96.1 Node interface

```
Valuator {
   eventIn      SFBool       inSFBool
   eventIn      SFColor      inSFColor
   eventIn      MFColor      inMFColor
   eventIn      SFFloat      inSFFloat
   eventIn      MFFloat      inMFFloat
   eventIn      SFInt32      inSFInt32
   eventIn      MFInt32      inMFInt32
   eventIn      SFRotation   inSFRotation
   eventIn      MFRotation   inMFRotation
   eventIn      SFString     inSFString
   eventIn      MFString     inMFString
   eventIn      SFTime       inSFTime
   eventIn      SFVec2f      inSFVec2f
   eventIn      MFVec2f      inMFVec2f
   eventIn      SFVec3f      inSFVec3f
   eventIn      MFVec3f      inMFVec3f
   eventOut     SFBool       outSFBool
   eventOut     SFColor      outSFColor
   eventOut     MFColor      outMFColor
   eventOut     SFFloat      outSFFloat
   eventOut     MFFloat      outMFFloat
   eventOut     SFInt32      outSFInt32
   eventOut     MFInt32      outMFInt32
   eventOut     SFRotation   outSFRotation
   eventOut     MFRotation   outMFRotation
   eventOut     SFString     outSFString
```

```
    eventOut       MFString    outMFString
    eventOut       SFTime      outSFTime
    eventOut       SFVec2f     outSFVec2f
    eventOut       MFVec2f     outMFVec2f
    eventOut       SFVec3f     outSFVec3f
    eventOut       MFVec3f     outMFVec3f
    exposedField   SFFloat     factor1             1.0
    exposedField   SFFloat     factor2             1.0
    exposedField   SFFloat     factor3             1.0
    exposedField   SFFloat     factor4             1.0
    exposedField   SFFloat     offset1             0.0
    exposedField   SFFloat     offset2             0.0
    exposedField   SFFloat     offset3             0.0
    exposedField   SFFloat     offset4             0.0
    exposedField   SFBool      sum                 FALSE
}
```

NOTE — For the binary encoding of this node see Annex H.1.95.

9.4.2.96.2 Functionality and semantics

A **Valuator** node can receive an event of any type, and on reception of such an event, will trigger eventOuts of many different types. Upon reception of an event on any of its eventIns, on each eventOut connected to a ROUTE an event will be generated. The value of this event is governed by the equation below. This node serves as a simple type casting method.

Each output value is dependent on the input value with the following relationship:

$$\text{output value} = \text{factor} * x + \text{offset}$$

In the above equation, factor is one of the exposedField values and offset is one of the eventOut values specified in the node inteface. All values specified in the above equation are floating point values.

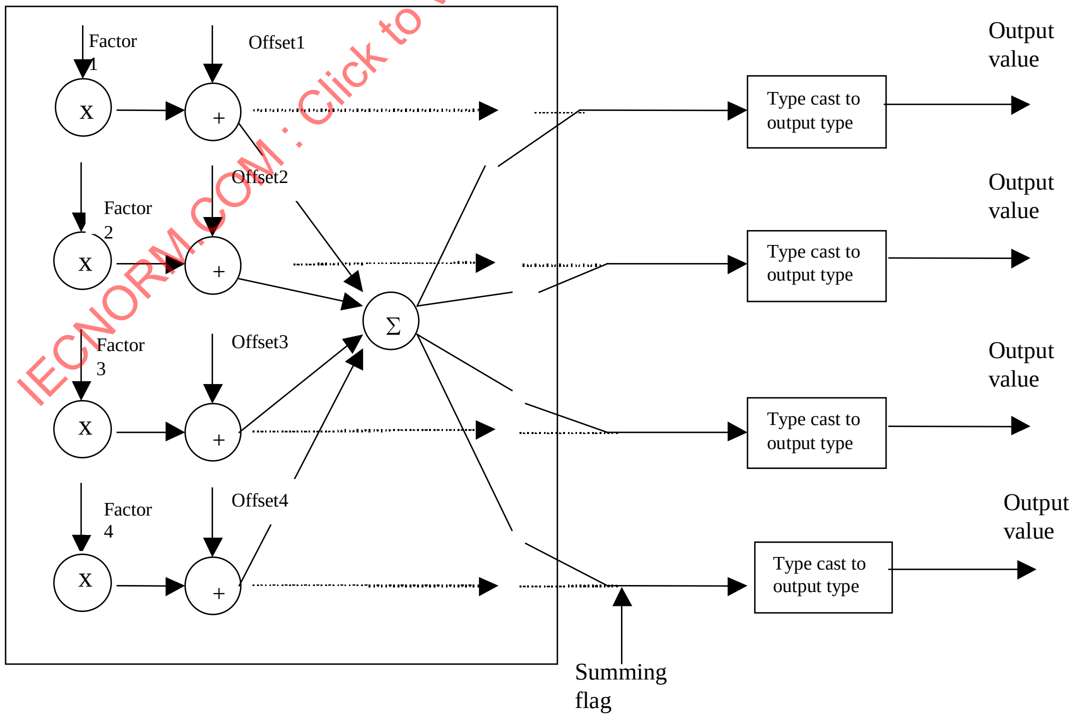


Figure 25 - Valuator functionaliy

Referring to the above figure, there are input paths each catering to an input value. Depending on the data type, there may be one to four input values. For example the SFRotation will require four input paths but the SFInt32 will only require the first input path. Each input path will operate identically.

Table 44 - Simple typecasting conversion from other data types to float.

From	**Conversion to float**
integer	Direct conversion. (1 to 1.0)
Boolean	true – 1.0 false – 0.0
double	Truncate to 32-bit precision

Table 45 - Simple typecasting conversion from float to other data types.

To	**Conversion from float**
integer	Truncate floating point. eg (1.11 to 1)
Boolean	0.0 to False Any other values to true
double	Direct conversion

Each input value is converted to a floating-point value using a simple typecasting rule as illustrated in Table 45. After conversion, the values are multiplied by the corresponding factor value and added to the corresponding offset value. Depending on whether the summer is enabled, either the summed value or the individual values are presented at the output.

Depending on the output data type required, the corresponding number of output values are retrieved and converted to the output types according to Table 44.

In the event that the input value is of a multi-valued type and the output is of a single value type, the first value of the multi-valued input is used.

EXAMPLE — The **Valuator** node can be seen as an event type adapter. One use of this node is the modification of the SFInt32 **whichChoice** field of a **Switch** node by an event. There is no interpolator or sensor node with a **SFInt32** eventOut. Thus, if a two-state button is described with a **Switch** containing the description of each state in choices 0 and 1. The triggering event of any type can be routed to a **Valuator** node whose **SFInt32** field is routed to the **whichChoice** field of the **Switch**.

9.4.2.97 Viewpoint

9.4.2.97.1 Node interface

```
Viewpoint {
    eventIn        SFBool        set_bind
    exposedField   SFFloat       fieldOfView          0.785398
    exposedField   SFBool        jump                 TRUE
    exposedField   SFRotation    orientation          0, 0, 1, 0
    exposedField   SFVec3f       position             0, 0, 10
    field          SFString      description          ""
    eventOut       SFTime        bindTime
    eventOut       SFBool        isBound
}
```

NOTE — For the binary encoding of this node see Annex H.1.96.

9.4.2.97.2 Functionality and semantics

The semantics of the **Viewpoint** node are specified in ISO/IEC 14772-1:1998, subclause 6.53 [10].

9.4.2.98 Viseme

9.4.2.98.1 Node interface

```
Viseme {
    field           SFInt32         viseme_select1          0
    field           SFInt32         viseme_select2          0
    field           SFInt32         viseme_blend            0
    field           SFBool          viseme_def              FALSE
}
```

NOTE — For the binary encoding of this node see Annex H.1.97.

9.4.2.98.2 Functionality and semantics

The **Viseme** node defines a blend of two visemes from a standard set of 14 visemes as defined in ISO/IEC 14496-2, Annex C, Table C-5.

The **viseme_select1** field specifies viseme 1.

The **viseme_select2** field specifies viseme 2.

The **viseme_blend** field specifies the blend of the two visemes.

If **viseme_def** is TRUE, the current FAPs shall be used to define a viseme and store it.

9.4.2.99 VisibilitySensor

9.4.2.99.1 Node interface

```
VisibilitySensor {
    exposedField    SFVec3f         center                  0 0 0
    exposedField    SFBool          enabled                 TRUE
    exposedField    SFVec3f         size                    0 0 0
    eventOut        SFTime          enterTime
    eventOut        SFTime          exitTime
    eventOut        SFBool          isActive
}
```

NOTE — For the binary encoding of this node see Annex H.1.98.

9.4.2.99.2 Functionality and semantics

The semantics of the **VisibilitySensor** node are specified in ISO/IEC 14772-1:1998, subclause 6.54 [10].

9.4.2.100 WorldInfo

9.4.2.100.1 Node interface

```
WorldInfo {
    field           MFString        info                    []
    field           SFString        title                   ""
}
```

NOTE — For the binary encoding of this node see Annex H.1.99.

9.4.2.100.2 Functionality and semantics

The semantics of the **WorldInfo** node are specified in ISO/IEC 14772-1:1998, subclause 6.55 [10].

10 Synchronization of Elementary Streams

10.1 Introduction

This subclause defines the tools to maintain temporal synchronisation within and among elementary streams. The conceptual elements that are required for this purpose, namely time stamps and clock reference information, have already been introduced in clause 7. The syntax and semantics to convey these elements to a receiving terminal are embodied in the sync layer, specified in 10.2. This syntax is configurable to adapt to the needs of different types of elementary streams. The required configuration information is specified in 10.2.3.

On the sync layer, an elementary stream is mapped into a sequence of packets, called an SL-packetized stream (SPS). Packetization information has to be exchanged between the entity that generates an elementary stream and the sync layer. This relation is best described by a conceptual interface between both layers, termed the elementary stream interface (ESI). The ESI is a reference point that need not be accessible in an implementation. It is described in 10.3.

SL-packetized streams are conveyed through a delivery mechanism that is outside the scope of ISO/IEC 14496-1. This delivery mechanism is only described in terms of the DMIF Application Interface (DAI) whose semantics are specified in ISO/IEC 14496-6. It specifies the information that needs to be exchanged between the sync layer and the delivery mechanism. The basic data transport feature that this delivery mechanism shall provide is the framing of the data packets generated by the sync layer. The DAI is a reference point that need not be accessible in an implementation. The required properties of the DAI are described in 10.4.

The items specified in this clause are depicted in Figure 26 below.

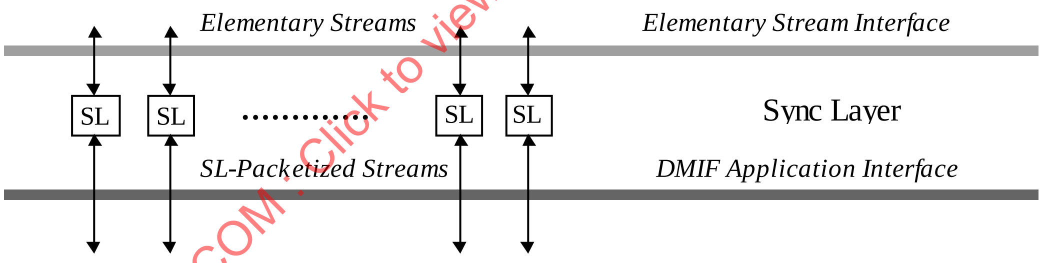


Figure 26 - The sync layer

10.2 Sync Layer

10.2.1 Overview

The sync layer (SL) specifies a syntax for the packetization of elementary streams into access units or parts thereof. Such a packet is called SL packet. The sequence of SL packets resulting from one elementary stream is called an SL-packetized stream (SPS). Access units are the only semantic entities at this layer that need to be preserved from end to end. Their content is opaque. Access units are used as the basic unit for synchronisation.

An SL packet consists of an SL packet header and an SL packet payload. The SL packet header provides means for continuity checking in case of data loss and carries the coded representation of the time stamps and associated information. The detailed semantics of the time stamps are specified in 7.3 that defines the timing aspects of the Systems Decoder Model. The SL packet header is configurable as specified in 10.2.3. The SL packet header itself is specified in 10.2.4.

An SL packet does not contain an indication of its length. Therefore, SL packets must be framed by a suitable lower layer protocol using, e.g., the FlexMux tool specified in 11.2. Consequently, an SL-packetized stream is not a self-contained data stream that can be stored or decoded without such framing.

An SL-packetized stream does not provide identification of the ES_ID associated to the elementary stream (see 8.6.4) in the SL packet header. This association must be conveyed through a stream map table using the appropriate signalling means of the delivery mechanism.

10.2.2 SL Packet Specification

10.2.2.1 Syntax

```
class SL_Packet (SLConfigDescriptor SL) {
   aligned(8) SL_PacketHeader slPacketHeader(SL);
   aligned(8) SL_PacketPayload slPacketPayload;
}
```

10.2.2.2 Semantics

In order to properly parse an `SL_Packet`, it is required that the `SLConfigDescriptor` for the elementary stream to which the `SL_Packet` belongs is known, since the `SLConfigDescriptor` conveys the configuration of the syntax of the SL packet header.

`slPacketHeader` – an `SL_PacketHeader` element as specified in 10.2.4.

`slPacketPayload` – an `SL_PacketPayload` that contains an opaque payload.

10.2.3 SL Packet Header Configuration

10.2.3.1 Syntax

```
class SLConfigDescriptor extends BaseDescriptor : bit(8) tag=SLConfigDescrTag {
   bit(8) predefined;
   if (predefined==0) {
      bit(1) useAccessUnitStartFlag;
      bit(1) useAccessUnitEndFlag;
      bit(1) useRandomAccessPointFlag;
      bit(1) hasRandomAccessUnitsOnlyFlag;
      bit(1) usePaddingFlag;
      bit(1) useTimeStampsFlag;
      bit(1) useIdleFlag;
      bit(1) durationFlag;
      bit(32) timeStampResolution;
      bit(32) OCRResolution;
      bit(8) timeStampLength; // must be ≤ 64
      bit(8) OCRLength;    // must be ≤ 64
      bit(8) AU_Length;    // must be ≤ 32
      bit(8) instantBitrateLength;
      bit(4) degradationPriorityLength;
      bit(5) AU_seqNumLength; // must be ≤ 16
      bit(5) packetSeqNumLength; // must be ≤ 16
      bit(2) reserved=0b11;
      if (durationFlag) {
         bit(32) timeScale;
         bit(16) accessUnitDuration;
         bit(16) compositionUnitDuration;
      }
      if (!useTimeStampsFlag) {
         bit(timeStampLength) startDecodingTimeStamp;
         bit(timeStampLength) startCompositionTimeStamp;
      }
   }
   aligned(8) bit(1) OCRstreamFlag;
   const bit(7) reserved=0b1111.111;
   if (OCRstreamFlag)
      bit(16) OCR_ES_Id;
}
```

10.2.3.2 Semantics

The SL packet header may be configured according to the needs of each individual elementary stream. Parameters that can be selected include the presence, resolution and accuracy of time stamps and clock references. This flexibility allows, for example, a low bitrate elementary stream to incur very little overhead on SL packet headers.

For each elementary stream the configuration is conveyed in an `SLConfigDescriptor`, which is part of the associated `ES_Descriptor` within an object descriptor.

The configurable parameters in the SL packet header can be divided in two classes: those that apply to each SL packet (e.g. OCR, sequenceNumber) and those that are strictly related to access units (e.g. time stamps, accessUnitLength, instantBitrate, degradationPriority).

`predefined` – allows to default the values from a set of predefined parameter sets as detailed below.

NOTE — This table will be updated by amendments to ISO/IEC 14496 to include predefined configurations as required by future profiles.

Table 46 - Overview of predefined `SLConfigDescriptor` values

`Predefined` field value	**Description**
0x00	Custom
0x01	null SL packet header
0x02 - 0xFF	Reserved for ISO use

Table 47 – Detailed predefined SLConfigDescriptor values

`predefined` field value	**0x01**
`useAccessUnitStartFlag`	0
`useAccessUnitEndFlag`	0
`useRandomAccessPointFlag`	0
`usePaddingFlag`	0
`useTimeStampsFlag`	0
`useIdleFlag`	0
`durationFlag`	-
`timeStampResolution`	-
`OCRResolution`	-
`timeStampLength`	-
`OCRlength`	-
`AU_length`	0
`instantBitrateLength`	-
`degradationPriorityLength`	0
`AU_seqNumLength`	0
`packetSeqNumLength`	0
`timeScale`	-
`accessUnitDuration`	-
`compositionUnitDuration`	-
`startDecodingTimeStamp`	-
`startCompositionTimeStamp`	-

`useAccessUnitStartFlag` – indicates that the `accessUnitStartFlag` is present in each SL packet header of this elementary stream.

useAccessUnitEndFlag – indicates that the accessUnitEndFlag is present in each SL packet header of this elementary stream.

If neither useAccessUnitStartFlag nor useAccessUnitEndFlag are set this implies that each SL packet corresponds to a complete access unit.

useRandomAccessPointFlag – indicates that the RandomAccessPointFlag is present in each SL packet header of this elementary stream.

hasRandomAccessUnitsOnlyFlag – indicates that each SL packet corresponds to a random access point. In that case the randomAccessPointFlag need not be used.

usePaddingFlag – indicates that the paddingFlag is present in each SL packet header of this elementary stream.

useTimeStampsFlag – indicates that time stamps are used for synchronisation of this elementary stream. They are conveyed in the SL packet headers. Otherwise, the parameters accessUnitRate, compositionUnitRate, startDecodingTimeStamp and startCompositionTimeStamp conveyed in this SL packet header configuration shall be used for synchronisation.

useIdleFlag – indicates that idleFlag is used in this elementary stream.

durationFlag – indicates that the constant duration of access units and composition units for this elementary stream is subsequently signaled.

timeStampResolution – is the resolution of the time stamps in clock ticks per second.

OCRResolution – is the resolution of the object time base in cycles per second.

timeStampLength – is the length of the time stamp fields in SL packet headers. timeStampLength shall take values between zero and 64 bit.

OCRlength – is the length of the objectClockReference field in SL packet headers. A length of zero indicates that no objectClockReferences are present in this elementary stream. If OCRstreamFlag is set, OCRLength shall be zero. Else OCRlength shall take values between zero and 64 bit.

AU_Length – is the length of the accessUnitLength fields in SL packet headers for this elementary stream. AU_Length shall take values between zero and 32 bit.

instantBitrateLength – is the length of the instantBitrate field in SL packet headers for this elementary stream.

degradationPriorityLength – is the length of the degradationPriority field in SL packet headers for this elementary stream.

AU_seqNumLength – is the length of the AU_sequenceNumber field in SL packet headers for this elementary stream.

packetSeqNumLength – is the length of the packetSequenceNumber field in SL packet headers for this elementary stream.

timeScale – used to express the duration of access units and composition units. One second is evenly divided in timeScale parts.

accessUnitDuration – the duration of an access unit is accessUnitDuration * 1/timeScale seconds.

compositionUnitDuration – the duration of a composition unit is compositionUnitDuration * 1/timeScale seconds.

startDecodingTimeStamp – conveys the time at which the first access unit of this elementary stream shall be decoded. It is conveyed in the resolution specified by timeStampResolution.

startCompositionTimeStamp – conveys the time at which the composition unit corresponding to the first access unit of this elementary stream shall be decoded. It is conveyed in the resolution specified by timeStampResolution.

OCRstreamFlag – indicates that an OCR_ES_ID syntax element will follow.

OCR_ES_ID – indicates the ES_ID of the elementary stream within the name scope (see 8.7.2.4) from which the time base for this elementary stream is derived.

10.2.4 SL Packet Header Specification

10.2.4.1 Syntax

```
aligned(8) class SL_PacketHeader (SLConfigDescriptor SL) {
   if (SL.useAccessUnitStartFlag)
      bit(1) accessUnitStartFlag;
   if (SL.useAccessUnitEndFlag)
      bit(1) accessUnitEndFlag;
   if (SL.OCRLength>0)
      bit(1) OCRflag;
   if (SL.useIdleFlag)
      bit(1) idleFlag;
   if (SL.usePaddingFlag)
      bit(1) paddingFlag;
   if (paddingFlag)
      bit(3) paddingBits;

   if (!idleFlag && (!paddingFlag || paddingBits!=0)) {
      if (SL.packetSeqNumLength>0)
         bit(SL.packetSeqNumLength) packetSequenceNumber;
      if (OCRflag)
         bit(SL.OCRLength) objectClockReference;

      if (accessUnitStartFlag) {
         if (SL.useRandomAccessPointFlag)
            bit(1) randomAccessPointFlag;
         bit(SL.AU_seqNumLength) AU_sequenceNumber;
         if (SL.useTimeStampsFlag) {
            bit(1) decodingTimeStampFlag;
            bit(1) compositionTimeStampFlag;
         }
         if (SL.instantBitrateLength>0)
            bit(1) instantBitrateFlag;
         if (decodingTimeStampFlag)
            bit(SL.timeStampLength) decodingTimeStamp;
         if (compositionTimeStampFlag)
            bit(SL.timeStampLength) compositionTimeStamp;
         if (SL.AU_Length > 0)
            bit(SL.AU_Length)    accessUnitLength;
         if (instantBitrateFlag)
            bit(SL.instantBitrateLength)      instantBitrate;
         if (SL.degradationPriorityLength>0)
               bit(SL.degradationPriorityLength) degradationPriority;
      }
   }
}
```

10.2.4.2 Semantics

accessUnitStartFlag – when set to one indicates that an access unit starts in this SL packet. If this syntax element is omitted from the SL packet header configuration its default value is known from the previous SL packet with the following rule:

accessUnitStartFlag = (previous-SL packet has accessUnitEndFlag==1) ? 1 : 0.

accessUnitEndFlag – when set to one indicates that an access unit ends in this SL packet. If this syntax element is omitted from the SL packet header configuration its default value is only known after reception of the subsequent SL packet with the following rule:

accessUnitEndFlag = (subsequent-SL packet has accessUnitStartFlag==1) ? 1 : 0.

If neither AccessUnitStartFlag nor AccessUnitEndFlag are configured into the SL packet header this implies that each SL packet corresponds to a single access unit, hence both accessUnitStartFlag = accessUnitEndFlag = 1.

NOTE — When the SL packet header is configured to use accessUnitStartFlag but neither accessUnitEndFlag nore accessUnitLength, it is not guaranteed that the terminal can determine the end of an access unit before the subsequent one is received.

OCRflag – when set to one indicates that an objectClockReference will follow. The default value for OCRflag is zero.

idleFlag – indicates that this elementary stream will be idle (i.e., not produce data) for an undetermined period of time. This flag may be used by the decoder to discriminate between deliberate and erroneous absence of subsequent SL packets.

paddingFlag – indicates the presence of padding in this SL packet. The default value for paddingFlag is zero.

paddingBits – indicate the mode of padding to be used in this SL packet. The default value for paddingBits is zero.

If paddingFlag is set and paddingBits is zero, this indicates that the subsequent payload of this SL packet consists of padding bytes only. accessUnitStartFlag, randomAccessPointFlag and OCRflag shall not be set if paddingFlag is set and paddingBits is zero.

If paddingFlag is set and paddingBits is greater than zero, this indicates that the payload of this SL packet is followed by paddingBits of zero stuffing bits for byte alignment of the payload.

packetSequenceNumber – if present, it shall be continuously incremented for each SL packet as a modulo counter. A discontinuity at the decoder corresponds to one or more missing SL packets. In that case, an error shall be signalled to the sync layer user. If this syntax element is omitted from the SL packet header configuration, continuity checking by the sync layer cannot be performed for this elementary stream.

Duplication of SL packets: elementary streams that have a sequenceNumber field in their SL packet headers may use duplication of SL packets for error resilience. The duplicated SL packet(s) shall immediately follow the original. The packetSequenceNumber of duplicated SL packets shall have the same value and each byte of the original SL packet shall be duplicated, with the exception of an objectClockReference field, if present, which shall encode a valid value for the duplicated SL packet.

objectClockReference – contains an Object Clock Reference time stamp. The OTB time value t is reconstructed from this OCR time stamp according to the following formula:

$$t = (\texttt{objectClockReference}/\texttt{SL.OCRResolution}) + k*(2^{\texttt{SL.OCRLength}}/\texttt{SL.OCRResolution})$$

where k is the number of times that the objectClockReference counter has wrapped around.

objectClockReference is only present in the SL packet header if OCRflag is set.

NOTE — It is possible to convey just an OCR value and no payload within an SL packet.

The following is the semantics of the syntax elements that are only present at the start of an access unit when explicitly signaled by accessUnitStartFlag in the bitstream:

randomAccessPointFlag – when set to one indicates that random access to the content of this elementary stream is possible here. randomAccessPointFlag shall only be set if accessUnitStartFlag is set. If this syntax element is omitted from the SL packet header configuration, its default value is the value of SLConfigDescriptor.hasRandomAccessUnitsOnlyFlag for this elementary stream.

AU_sequenceNumber – if present, it shall be continuously incremented for each access unit as a modulo counter. A discontinuity at the decoder corresponds to one or more missing access units. In that case, an error shall be signalled to the sync layer user. If this syntax element is omitted from the SL packet header configuration, access unit continuity checking by the sync layer cannot be performed for this elementary stream.

Duplication of access units: elementary streams that have a AU_sequenceNumber field in their SL packet headers may use duplication of access units. The duplicated access unit(s) shall immediately follow the original. The AU_sequenceNumber of such access units shall have the same value and each byte of the original one or more SL packets shall be duplicated, with the exception of an objectClockReference field, if present, which shall encode a valid value for the duplicated access unit.

decodingTimeStampFlag – indicates that a decoding time stamp is present in this packet.

compositionTimeStampFlag – indicates that a composition time stamp is present in this packet.

accessUnitLengthFlag – indicates that the length of this access unit is present in this packet.

instantBitrateFlag – indicates that an instantBitrate is present in this packet.

decodingTimeStamp – is a decoding time stamp as configured in the associated SLConfigDescriptor. The decoding time td of this access unit is reconstructed from this decoding time stamp according to the formula:

$$td = (\mathtt{decodingTimeStamp}/\mathtt{SL.timeStampResolution} + k * 2^{\mathtt{SL.timeStampLength}}/\mathtt{SL.timeStampResolution}$$

where k is the number of times that the decodingTimeStamp counter has wrapped around.

compositionTimeStamp – is a composition time stamp as configured in the associated SLConfigDescriptor. The composition time tc of the first composition unit resulting from this access unit is reconstructed from this composition time stamp according to the formula:

$$td = (\mathtt{compositionTimeStamp}/\mathtt{SL.timeStampResolution} + k * 2^{\mathtt{SL.timeStampLength}}/\mathtt{SL.timeStampResolution}$$

where k is the number of times that the compositionTimeStamp counter has wrapped around.

accessUnitLength – is the length of the access unit in bytes. If this syntax element is not present or has the value zero, the length of the access unit is unknown.

instantBitrate – is the instantaneous bit rate of this elementary stream until the next instantBitrate field is found.

degradationPriority – indicates the importance of the payload of this access unit. The streamPriority defines the base priority of an ES. degradationPriority defines a decrease in priority for this access unit relative to the base priority. The priority for this access unit is given by:

AccessUnitPriority = streamPriority – degradationPriority

degradationPriority remains at this value until its next occurrence. This indication is used for graceful degradation by the decoder of this elementary stream. The relative amount of complexity degradation among access units of different elementary streams increases as AccessUnitPriority decreases.

10.2.5 Clock Reference Stream

An elementary stream of `streamType` = ClockReferenceStream may be declared by means of the object descriptor. It is used for the sole purpose of conveying Object Clock Reference time stamps. Multiple elementary streams in a name scope may make reference to such a ClockReferenceStream by means of the `OCR_ES_ID` syntax element in the `SLConfigDescriptor` to avoid redundant transmission of Clock Reference information.

On the sync layer a ClockReferenceStream is realized by configuring the SL packet header syntax for this SL-packetized stream such that only OCR values of the required `OCRresolution` and `OCRlength` are present in the SL packet header.

There shall not be any SL packet payload present in an SL-packetized stream of `streamType` = ClockReferenceStream.

A ClockReferenceStream shall set the `hasRandomAccessUnitsOnlyFlag` to one.

The following indicates recommended values for the `SLConfigDescriptor` of a Clock Reference Stream:

Table 48 – SLConfigDescriptor parameter values for a ClockReferenceStream

`useAccessUnitStartFlag`	0
`useAccessUnitEndFlag`	0
`useRandomAccessPointFlag`	0
`usePaddingFlag`	0
`useTimeStampsFlag`	0
`useIdleFlag`	0
`durationFlag`	0
`timeStampResolution`	0
`timeStampLength`	0
`AU_length`	0
`degradationPriorityLength`	0
`AU_seqNumLength`	0

10.2.6 Restrictions for elementary streams sharing the same object time base

While it is possible to share an object time base between multiple elementary streams through `OCR_ES_ID`, a number of restrictions for the access to and processing of these elementary streams exist as follows:

1. When several elementary streams share a single object time base, the elementary streams without embedded object clock reference information shall not be used by the player, even if accessible, until the elementary stream carrying the object clock reference information becomes accessible (see 8.7.3 for the stream access procedure).

2. If an elementary stream without embedded object clock reference information is made available to the terminal at a later point in time than the elementary stream carrying the object clock reference information, it shall be delivered in synchronization with the other stream(s). Note that this implies that such a stream might not start playing from its beginning, depending on the current value of the object time base.

3. When an elementary stream carrying object clock reference information becomes unavailable or is otherwise manipulated in its delivery (e.g., paused), all other elementary streams which use the same object time base shall follow this behavior, i.e., become unavailable or be manipulated in the same way.

4. When an elementary stream without embedded object clock reference information becomes unavailable this has no influence on the other elementary streams that share the same object time base.

10.2.7 Usage of configuration options for object clock reference and time stamp values

10.2.7.1 Resolution of ambiguity in object time base recovery

Due to the limited length of `objectClockReference` values these time stamps may be ambiguous. The OTB time value can be reconstructed each time an `objectClockReference` is transmitted in the headers of an SL packet according to the following formula:

$t_{OTB\_reconstructed}=(\texttt{objectClockReference/SL.OCRResolution})+k*(2^{\texttt{SL.OCRLength}}\texttt{/SL.OCRResolution})$

with k being an integer value denoting the number of wrap-arounds. The resulting time base $t_{OTB\_reconstructed}$ is measured in seconds.

When the first `objectClockReference` for an elementary stream is acquired, the value k shall be set to one. For each subsequent occurence of `objectClockReference` the value k is estimated as follows:

The terminal shall implement a mechanism to estimate the value of the object time base for any time instant.

Each time an `objectClockReference` is received, the current estimated value of the OTB $t_{OTB\_estimated}$ shall be sampled. Then, $t_{OTB\_rec}(k)$ is evaluated for different values of k. The value k that minimizes the term $| t_{OTB\_estimated} - t_{OTB\_rec}(k)|$ shall be assumed to yield the correct value of $t_{OTB\_reconstructed}$. This value may be used as new input to the object time base estimation mechanism.

The application shall ensure that this procedure yields an unambiguous value of k by selecting an appropriate length and resolution of the `objectClockReference` element and a sufficiently high frequency of insertion of `objectClockReference` values in the elementary stream. The choices for these values depend on the delivery jitter for SL packets as well as the anticipated maximum drift between the clocks of the transmitting and receiving terminal.

10.2.7.2 Resolution of ambiguity in time stamp recovery

Due to the limited length of `decodingTimeStamp` and `compositionTimeStamp` values these time stamps may become ambiguous according to the following formula:

$t_{ts}(m)=(\texttt{TimeStamp/SL.timeStampResolution})+m*(2^{\texttt{SL.timeStampLength}}\texttt{/SL.timeStampResolution})$

with `TimeStamp` being either a `decodingTimeStamp` or a `compositionTimeStamp` and m being an integer value denoting the number of wrap-arounds.

The correct value $t_{timestamp}$ of the time stamp can be estimated as follows:

Each time a `TimeStamp` is received, the current estimated value of the OTB $t_{OTB\_estimated}$ shall be sampled. $t_{ts}(m)$ is evaluated for different values of m. The value m that minimizes the term $| t_{OTB\_estimated} – t_{ts}(m)|$ shall be assumed to yield the correct value of $t_{timestamp}$.

The application may choose, separately for every individual elementary stream, the length and resolution of time stamps so as to match its requirements on unambiguous positioning of time events. This choice depends on the maximum time that an SL packet with a `TimeStamp` may be sent prior to the point in time indicated by the `TimeStamp` as well as the required precision of temporal positioning.

10.2.7.3 Usage considerations for object clock references and time stamps

The time line of an object time base allows to discriminate two time instants separated by more than 1/`SL.OCRResolution`. `OCRResolution` should be chosen sufficiently high to match the accuracy needed by the application to synchronize a set of elementary streams.

The decoding and composition time stamp allow to discriminate two time instants separated by more than 1/`SL.timeStampResolution`. `timeStampResolution` should be chosen sufficiently high to match the accuracy needed by the application in terms of positioning of access units for a given elementary stream.

A `TimeStampResolution` higher than the `OCRResolution` will not achieve better discrimination between events. If `TimeStampResolution` is lower than the `OCRResolution`, events for this specific stream cannot be positioned with the maximum precision possible with this given `OCRResolution`.

The parameter `OCRLength` is signaled in the SL header configuration. $2^{\texttt{SL.OCRLength}}$/`SL.OCRResolution` is the time interval covered by the `objectClockReference` counter before it wraps around. `OCRLength` should be chosen sufficiently high to match the application needs for unambiguous positioning of time events from a set of elementary streams.

When an application knows the value k defined in 10.2.7.1, the OTB time line is unambiguous for any time value. When the application cannot reconstruct the k factor, as for example in any application that permits random access without additional side information, the OTB time line is ambiguous modulo $2^{\texttt{SL.OCRLength}}$/`SL.OCRResolution`. Therefore, any time stamp refering to this OTB is ambiguous. Therefore, any time stamp refering to this OTB is ambiguous. It may, however, be considered unambiguous within an application environment through knowledge about the maximum expected delivery jitter and constraints on the time by which an access unit can be sent prior to its decoding time.

Note that elementary streams that choose the time interval $2^{\texttt{SL.timeStampLength}}$/`SL.timeStampResolution` higher than $2^{\texttt{SL.OCRLength}}$/`SL.OCRResolution` can still only position time events unambiguously in the smaller of the two intervals.

In cases, where k and m can not be estimated correctly, the buffer model may be violated, resulting in unpredictable performance of the decoder.

EXAMPLE — Let's assume an application that wants to synchronize elementary streams with a precision of 1 ms. `OCRResolution` should be chosen equal to or higher than 1000 (the time between two successive ticks of the OCR is then equal to 1ms). Let's assume `OCRResolution=2000`.

The application assumes a drift between the STB and the OTB of 0.1% (i.e. 1ms every second). The clocks need therefore to be adjusted at least every second (i.e. in the worst case, the clocks will have drifted 1ms which is the precision constraint). Let's assume that `objectClockReference` are sent every 1s.

The application wants to have an unambiguous OTB time line of 24h without need to reconstruct the k factor. The `OCRLength` is therefore chosen accordingly such that $2^{\texttt{SL.OCRLength}}$/`SL.OCRResolution=24h`.

Let's assume now that the application wants to synchronize events within a single elementary stream with a precision of 10 ms. `TimeStampResolution` should be chosen equal to or higher than 100 (the time between two successive ticks of the `TimeStamp` is then equal to 10ms). Let's assume `TimeStampResolution=200`.

The application wants to be able to send access units at maximum 1 minute ahead of their decoding or composition time. The `timeStampLength` is therefore chosen as

$2^{\texttt{SL.timeStampLength}}$/`SL.timeStampResolution = 2` minutes.

10.3 Elementary Stream Interface (Informative)

The elementary stream interface (ESI) is a conceptual interface that specifies which data need to be exchanged between the entity that generates an elementary stream and the sync layer. Communication between the coding and sync layers cannot only include compressed media, but requires additional information such as time codes, length of access units, etc.

An implementation of ISO/IEC 14496-1, however, does not have to implement the elementary stream interface. It is possible to integrate parsing of the SL-packetized stream and media data decompression in one decoder entity. Note that even in this case the decoder receives a sequence of packets at its input through the DMIF Application Interface (see 10.4) rather than a data stream.

The interface to receive elementary stream data from the sync layer has a number of parameters that reflect the side information that has been retrieved while parsing the incoming SL-packetized stream:

ESI.receiveData (*ESdata, dataLength, idleFlag, objectClockReference, decodingTimeStamp, compositionTimeStamp, accessUnitStartFlag, randomAccessFlag, accessUnitEndFlag, accessUnitLength, degradationPriority, errorStatus*)

ESdata - a number of *dataLength* data bytes for this elementary stream

dataLength - the length in byte of *ESdata*

idleFlag – if set to one it indicates that this elementary stream will not produce further data for an undetermined period of time.

objectClockReference – contains a reading of the object time base valid for the point in time when the first byte of ESdata enters the decoder buffer.

decodingTimeStamp - the decoding time for the access unit to which this *ESdata* belongs

compositionTimeStamp - the composition time for the access unit to which this *ESdata* belongs

accessUnitStartFlag - indicates that the first byte of *ESdata* is the start of an access unit

randomAccessFlag - indicates that the first byte of *ESdata* is the start of an access unit allowing for random access

accessUnitEndFlag - indicates that the last byte of *ESdata* is the end of an access unit

accessUnitLength - the length of the access unit to which this Esdata belongs in byte

degradationPriority - indicates the degradation priority for this access unit

errorStatus - indicates whether *ESdata* is error free, possibly erroneous or whether data has been lost preceding the current *ESdata* bytes

A similar interface to send elementary stream data to the sync layer requires the following parameters that will subsequently be encoded on the sync layer:

ESI.sendData (*ESdata, dataLength, idleFlag, objectClockReference, decodingTimeStamp, compositionTimeStamp, accessUnitStartFlag, randomAccessFlag, accessUnitEndFlag, accessUnitLength, degradationPriority*)

ESdata - a number of *dataLength* data bytes for this elementary stream

dataLength - the length in byte of *ESdata*

idleFlag – if set to one it indicates that this elementary stream will not produce further data for an undetermined period of time.

objectClockReference – contains a reading of the object time base valid for the point in time when the first byte of ESdata enters the decoder buffer.

decodingTimeStamp - the decoding time for the access unit to which this *ESdata* belongs

compositionTimeStamp - the composition time for the access unit to which this *ESdata* belongs

accessUnitStartFlag - indicates that the first byte of *ESdata* is the start of an access unit

randomAccessFlag - indicates that the first byte of *ESdata* is the start of an access unit allowing for random access

accessUnitEndFlag - indicates that the last byte of *ESdata* is the end of an access unit

accessUnitLength - the length of the access unit to which this Esdata belongs in byte

degradationPriority - indicates the degradation priority for this access unit

10.4 DMIF Application Interface

The DMIF Application Interface is a conceptual interface that specifies which data need to be exchanged between the sync layer and the delivery mechanism. Communication between the sync layer and the delivery mechanism includes SL-packetized data as well as additional information to convey the length of each SL packet.

An implementation of ISO/IEC 14496-1 does not have to expose the DMIF Application Interface. A terminal compliant with ISO/IEC 14496-1, however, shall have the functionality described by the DAI to be able to receive the SL packets that constitute an SL-packetized stream. Specifically, the delivery mechanism below the sync layer shall supply a method to frame or otherwise encode the length of the SL packets transported through it.

The DMIF Application Interface specified in ISO/IEC 14496-6 embodies a superset of the required data delivery functionality. The DAI has data primitives to receive and send data, which include indication of the data size. With this interface, each invocation of a DA_Data or a DA_DataCallback shall transfer one SL packet between the sync layer and the delivery mechanism below.

11 Multiplexing of Elementary Streams

11.1 Introduction

Elementary stream data encapsulated in SL-packetized streams are sent/received through the DMIF Application Interface, as specified in clause 10. Multiplexing procedures and the architecture of the delivery protocol layers are outside the scope of ISO/IEC 14496-1. However, care has been taken to define the sync layer syntax and semantics such that SL-packetized streams can be easily embedded in various transport protocol stacks.

The analysis of existing transport protocol stacks has shown that, for stacks with fixed length packets (e.g., MPEG-2 Transport Stream) or with high multiplexing overhead (e.g., RTP/UDP/IP), it may be advantageous to have a generic, low complexity multiplexing tool that allows interleaving of data with low overhead and low delay. This is particularly important for low bit rate applications. Such a multiplex tool is specified in this clause. Its use is optional.

11.2 FlexMux Tool

11.2.1 Overview

The FlexMux tool is a flexible multiplexer that accommodates interleaving of SL-packetized streams with varying instantaneous bit rate. The basic data entity of the FlexMux is a FlexMux packet, which has a variable length. One or more SL packets are embedded in a FlexMux packet as specified in detail in the remainder of this clause. The FlexMux tool provides identification of SL packets originating from different elementary streams by means of FlexMux Channel numbers. Each SL-packetized stream is mapped into one FlexMux Channel. FlexMux packets with data from different SL-packetized streams can therefore be arbitrarily interleaved. The sequence of FlexMux packets that are interleaved into one stream are called a FlexMux Stream.

A FlexMux Stream retrieved from storage or transmission can be parsed as a single data stream without the need for any side information. However, the FlexMux requires framing of FlexMux packets by the underlying layer for random access or error recovery. There is no requirement to frame each individual FlexMux packet. The FlexMux also requires reliable error detection by the underlying layer. This design has been chosen acknowledging the fact that framing and error detection mechanisms are in many cases provided by the transport protocol stack below the FlexMux.

Two different modes of operation of the FlexMux providing different features and complexity are defined. They are called Simple Mode and MuxCode Mode. A FlexMux Stream may contain an arbitrary mixture of FlexMux packets using either Simple Mode or MuxCode Mode. The syntax and semantics of both modes are specified below.

11.2.2 Simple Mode

In the simple mode one SL packet is encapsulated in one FlexMux packet and tagged by an `index` which is equal to the FlexMux Channel number as indicated in Figure 27. This mode does not require any configuration or maintenance of state by the receiving terminal.

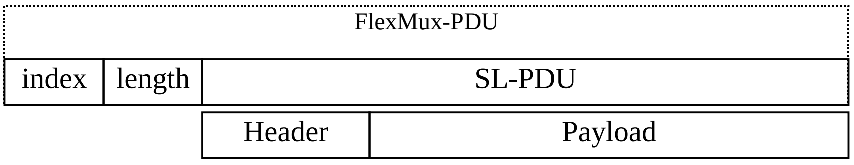


Figure 27 - Structure of FlexMux packet in simple mode

11.2.3 MuxCode mode

In the MuxCode mode one or more SL packets are encapsulated in one FlexMux packet as indicated in Figure 28. This mode requires configuration and maintenance of state by the receiving terminal. The configuration describes how FlexMux packets are shared between multiple SL packets. In this mode the `index` value is used to dereference configuration information that defines the allocation of the FlexMux packet payload to different FlexMux Channels.

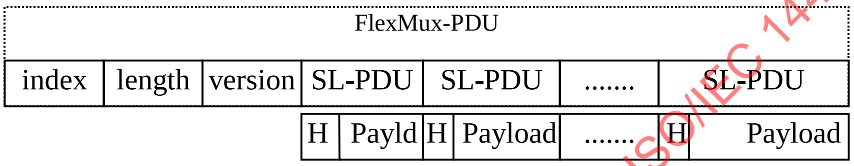


Figure 28 - Structure of FlexMux packet in MuxCode mode

11.2.4 FlexMux packet specification

11.2.4.1 Syntax

```
class FlexMuxPacket {
   unsigned int(8) index;
   bit(8) length;
   if (index>239) {
      bit(4) version;
      const bit(4) reserved=0b1111;
      multiple_SL_Packet mPayload;
   } else {
      SL_Packet sPayload;
   }
}
```

11.2.4.2 Semantics

The two modes of the FlexMux, Simple Mode and MuxCode Mode are distinguished by the value of `index` as specified below.

`index` – if `index` is smaller than 240 then

FlexMux Channel = `index`

This range of values corresponds to the Simple Mode. If `index` has a value in the range 240 to 255 (inclusive), then the MuxCode Mode is used and a `MuxCode` is referenced as

`MuxCode` = `index` - 240

`MuxCode` is used to associate the `payload` to FlexMux Channels as described in 11.2.3.

NOTE — Although the number of FlexMux Channels is limited to 256, the use of multiple FlexMux streams allows virtually any number of elementary streams to be provided to the terminal.

`length` – the length of the FlexMux packet `payload` in bytes. This is equal to the length of the single encapsulated SL packet in Simple Mode and to the total length of the multiple encapsulated SL packets in MuxCode Mode.

`version` – indicates the current version of the `MuxCodeTableEntry` referenced by `MuxCode`. `Version` is used for error resilience purposes. If this version does not match the version of the referenced `MuxCodeTableEntry` that has most recently been received, the FlexMux packet cannot be parsed. The implementation is free to either wait until the required version of `MuxCodeTableEntry` becomes available or to discard the FlexMux packet.

`sPayload` – a single SL packet (Simple Mode)

`mPayload` – one or more SL packets (MuxCode Mode)

11.2.4.3 Configuration for MuxCode Mode

11.2.4.3.1 Syntax

```
aligned(8) class MuxCodeTableEntry {
   int    i, k;
   bit(8) length;
   bit(4) MuxCode;
   bit(4) version;
   bit(8) substructureCount;
   for (i=0; i<substructureCount; i++) {
      bit(5) slotCount;
      bit(3) repetitionCount;
      for (k=0; k<slotCount; k++){
         bit(8) flexMuxChannel[[i]][[k]];
         bit(8) numberOfBytes[[i]][[k]];
      }
   }
}
```

11.2.4.3.2 Semantics

The configuration for MuxCode Mode is signaled by `MuxCodeTableEntry` messages. The transport of the `MuxCodeTableEntry` shall be defined during the design of the transport protocol stack that makes use of the FlexMux tool. Part 6 of this International Standard defines a method to convey this information using the DN_TransmuxConfig primitive.

The basic requirement for the transport of the configuration information is that data arrives reliably in a timely manner. However, no specific performance bounds are required for this control channel since version numbers allow to detect FlexMux packets that cannot currently be decoded and, hence, trigger suitable action in the receiving terminal.

`length` – the length in bytes of the remainder of the `MuxCodeTableEntry` following the `length` element.

`MuxCode` – the number through which this MuxCode table entry is referenced.

`version` – indicates the version of the `MuxCodeTableEntry`. Only the latest received version of a `MuxCodeTableEntry` is valid.

`substructureCount` – the number of substructures of this `MuxCodeTableEntry.`

`slotCount` – the number of slots with data from different FlexMux Channels that are described by this substructure.

`repetitionCount` – indicates how often this substructure is to be repeated. A `repetitionCount` zero indicates that this substructure is to be repeated infinitely. `repetitionCount` zero is only permitted in the last substructure of a MuxCodeTableEntry.

`flexMuxChannel[i][k]` – the FlexMux Channel to which the data in this slot belongs.

`numberOfBytes[i][k]` – the number of data bytes in this slot associated to `flexMuxChannel[i][k]`. This number of bytes corresponds to one SL packet.

11.2.5 Usage of MuxCode Mode

The `MuxCodeTableEntry` describes how a FlexMux packet is partitioned into slots that carry data from different FlexMux Channels. This is used as a template for parsing FlexMux packets. If a FlexMux packet is longer than the template, parsing shall resume from the beginning of the template. If a FlexMux packet is shorter than the template, the remainder of the template is ignored.

Note that the usage of MuxCode mode may not be efficient if SL packets for a given elementary stream do not have a constant length. Given the overhead for an update of the associated MuxCodeTableEntry, usage of simple mode might be more efficient.

Note further that data for a single FlexMux channel may be conveyed through an arbitrary sequence of FlexMux packets with both simple mode and MuxCode mode.

EXAMPLE —

In this example we assume the presence of three substructures. Each one has a different slot count as well as repetition count. The exact parameters are as follows:

`substructureCount`	= 3
`slotCount`[i]	= 2, 3, 2 (for the corresponding substructure)
`repetitionCount`[i]	= 3, 2, 1 (for the corresponding substructure)

We further assume that each slot configures channel number FMC*n* (`flexMuxChannel`) with a number of bytes Bytes*n* (`numberOfBytes`). This configuration would result in a splitting of the FlexMux packet payload to:

FMC1 (Bytes1), FMC2 (Bytes2)	repeated 3 times, then
FMC3 (Bytes3), FMC4 (Bytes4), FMC5 (Bytes5)	repeated 2 times, then
FMC6 (Bytes6), FMC7 (Bytes7)	repeated once

The layout of the corresponding FlexMux packet would be as shown in Figure 29.

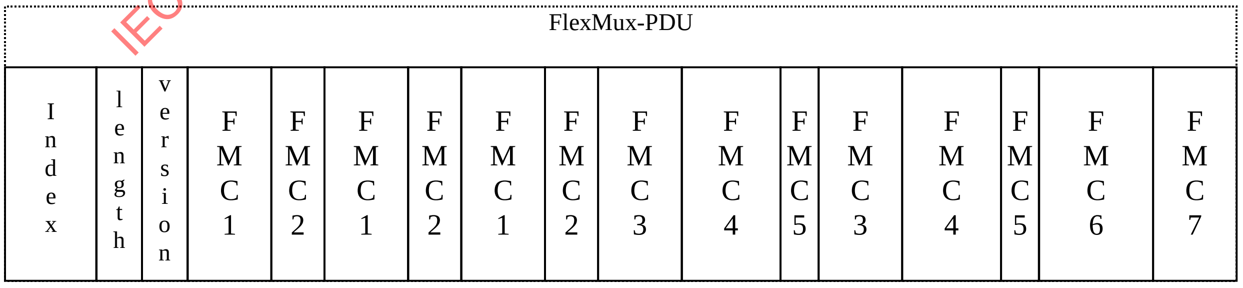


Figure 29 - Example for a FlexMux packet in MuxCode mode

12 Syntactic Description Language

12.1 Introduction

This subclause describes the mechanism with which bitstream syntax is documented in ISO/IEC 14496. This mechanism is based on a Syntactic Description Language (SDL), documented here in the form of syntactic description rules. It directly extends the C-like syntax used in ISO/IEC 11172:1993 and ISO/IEC 13818:1996 into a well-defined framework that lends itself to object-oriented data representations. In particular, SDL assumes an object-oriented underlying framework in which bitstream units consist of “classes.” This framework is based on the typing system of the C++ and Java programming languages. SDL extends the typing system by providing facilities for defining bitstream-level quantities, and how they should be parsed.

The elementary constructs are described first, followed by the composite syntactic constructs, and arithmetic and logical expressions. Finally, syntactic control flow and built-in functions are addressed. Syntactic flow control is needed to take into account context-sensitive data. Several examples are used to clarify the structure.

12.2 Elementary Data Types

The SDL uses the following elementary data types:

1. Constant-length direct representation bit fields or Fixed Length Codes — FLCs. These describe the encoded value exactly as it is to be used by the appropriate decoding process.

2. Variable length direct representation bit fields, or parametric FLCs. These are FLCs for which the actual length is determined by the context of the bitstream (e.g., the value of another parameter).

3. Constant-length indirect representation bit fields. These require an extra lookup into an appropriate table or variable to obtain the desired value or set of values.

4. Variable-length indirect representation bit fields (e.g., Huffman codes).

These elementary data types are described in more detail in the clauses to follow immediately.

All quantities shall be represented in the bitstream with the most significant byte first, and also with the most significant bit first.

12.2.1 Constant-Length Direct Representation Bit Fields

Constant-length direct representation bit fields shall be represented as:

Rule E.1: Elementary Data Types
[**aligned**] ***type***[**(***length***)**] *element_name* [= *value*]; // C++-style comments allowed

The ***type*** may be any of the following: **int** for signed integer, **unsigned int** for unsigned integer, **double** for floating point, and **bit** for raw binary data. The *length* attribute indicates the length of the element in bits, as it is actually stored in the bitstream. Note that a data ***type*** equal to **double** shall only use 32 or 64 bit lengths. The *value* attribute shall be present only when the value is fixed (e.g., start codes or object IDs), and it may also indicate a range of values (i.e., ‘0x01..0xAF’). The ***type*** and the optional *length* attributes are always present, except if the data is non-parsable, i.e., it is not included in the bitstream. The keyword **aligned** indicates that the data is aligned on a byte boundary. As an example, a start code would be represented as:

```
aligned bit(32) picture_start_code=0x00000100;
```

An optional numeric modifier, as in **aligned(**32**)**, may be used to signify alignment on other than byte boundary. Allowed values are 8, 16, 32, 64, and 128. Any skipped bits due to alignment shall have the value ‘0’. An entity such as temporal reference would be represented as:

```
unsigned int(5) temporal_reference;
```

where **`unsigned int(`***5***`)`** indicates that the element shall be interpreted as a 5-bit unsigned integer. By default, data shall be represented with the most significant bit first, and the most significant byte first.

The value of parsable variables with declarations that fall outside the flow of declarations (see 12.6) shall be set to 0.

Constants shall be defined using the keyword **`const`**.

EXAMPLE —

```
const int SOME_VALUE=255; // non-parsable constant
const bit(3) BIT_PATTERN=1;  // this is equivalent to the bit string "001"
```

To designate binary values, the **`0b`** prefix shall be used, similar to the **`0x`** prefix for hexadecimal numbers. A period ('.') may be optionally placed after every four digits for readability. Hence 0x0F is equivalent to 0b0000.1111.

In several instances, it may be desirable to examine the immediately following bits in the bitstream, without actually consuming these bits. To support this behavior, a '**`*`**' character shall be placed after the parse size parentheses to modify the parse size semantics.

Rule E.2: Look-ahead parsing
[**`aligned`**] ***type*** **`(`** *length* **`)*`** *element_name*;

For example, the value of next 32 bits in the bitstream can be checked to be an unsigned integer without advancing the current position in the bitstream using the following representation:

```
aligned unsigned int (32)* next_code;
```

12.2.2 Variable Length Direct Representation Bit Fields

This case is covered by Rule E.1, by allowing the *length* attribute to be a variable included in the bitstream, a non-parsable variable, or an expression involving such variables.

EXAMPLE —

```
unsigned int(3) precision;
int(precision) DC;
```

12.2.3 Constant-Length Indirect Representation Bit Fields

Indirect representation indicates that the actual value of the element at hand is indirectly specified by the bitstream through the use of a table or map. In other words, the value extracted from the bitstream is an index to a table from which the final desired value is extracted. This indirection may be expressed by defining the map itself:

Rule E.3: Maps
`map` *MapName* **`(`** *output_type* **`) {`**
  index, **`{`** *value_1*, … *value_M* **`}`**,
  …
`}`

These tables are used to translate or map bits from the bitstream into a set of one or more values. The input type of a **`map`** (the *index* specified in the first column) shall always be **`bit`**. The *output_type* entry shall be either a predefined type or a defined class (classes are defined in 12.3.1). The **`map`** is defined as a set of pairs of such indices and values. Keys are binary string constants while values are *output_type* constants. Values shall be specified as aggregates surrounded by curly braces, similar to C or C++ structures.

EXAMPLE —

```
class YUVblocks {// classes are fully defined later on
   int Yblocks;
   int Ublocks;
   int Vblocks;
}

// a table that relates the chroma format with the number of blocks
// per signal component
map blocks_per_component (YUVblocks) {
   0b00, {4, 1, 1}, // 4:2:0
   0b01, {4, 2, 2}, // 4:2:2
   0b10, {4, 4, 4}  // 4:4:4
}
```

The next rule describes the use of such a **map**.

Rule E.4: Mapped Data Types
type **(** *MapName* **)** *name*;

The ***type*** of the variable shall be identical to the ***type*** returned from the **map**.

EXAMPLE —

```
YUVblocks(blocks_per_component) chroma_format;
```

Using the above declaration, a particular value of the **map** may be accessed using the construct: `chroma_format.Ublocks`.

12.2.4 Variable Length Indirect Representation Bit Fields

For a variable length element utilizing a Huffman or variable length code table, an identical specification to the fixed length case shall be used:

```
class val {
   unsigned int foo;
   int bar;
}

map sample_vlc_map (val)  {
   0b0000.001,     {0, 5},
   0b0000.0001, {1, -14}
}
```

The only difference is that the indices of the **map** are now of variable length. The variable-length codewords are (as before) binary strings, expressed by default in '0b' or '0x' format, optionally using the period ('.') every four digits for readability.

Very often, variable length code tables are partially defined. Due to the large number of possible entries, it may be inefficient to keep using variable length codewords for all possible values. This necessitates the use of escape codes, that signal the subsequent use of a fixed-length (or even variable length) representation. To allow for such exceptions, parsable type declarations are allowed for **map** values.

EXAMPLE — This example uses the class type 'val' as defined above.

```
map sample_map_with_esc (val)  {
   0b0000.001,     {0, 5},
   0b0000.0001, {1, -14},
   0b0000.0000.1,  {5, int(32)},
   0b0000.0000.0,  {0, -20}
}
```

When the codeword 0b0000.0000.1 is encountered in the bitstream, then the value '5' is assigned to the first element (`val.foo`). The following 32 bits are parsed and assigned as the value of the second element (`val.bar`). Note that, in case more than one element utilizes a parsable type declaration, the order is significant and is the order in which elements are parsed. In addition, the type within the **map** declaration shall match the type used in the class declaration associated with the **map**'s return type.

12.3 Composite Data Types

12.3.1 Classes

Classes are the mechanism with which definitions of composite types or objects is performed. Their definition is as follows.

Rule C.1: Classes

[**aligned**] [**abstract**] [**expandable**[**(** *maxClassSize* **)**]] **class** *object_name* [**extends** *parent_class*] [:
  bit(*length* **)** [*id_name*=] *object_id* | *id_range*] **{**
  [*element*; …] // zero or more elements
}

The different elements within the curly braces are the definitions of the elementary bitstream components discussed in 12.2 or control flow elements that will be discussed in a subsequent subclause.

The optional keyword **extends** specifies that the **class** is "derived" from another **class**. Derivation implies that all information present in the base **class** is also present in the derived **class**, and that, in the bitstream, all such information *precedes* any additional bitstream syntax declarations specified in the new **class**.

The optional attribute *id_name* allows to assign an *object_id*, and, if present, is the key demultiplexing entity which allows differentiation between base and derived objects. It is also possible to have a range of possible values: the *id_range* is specified as *start_id* .. *end_id*, inclusive of both bounds.

If the attribute id_name is used, a derived **class** may appear at any point in the bitstream where its base **class** is specified in the syntax. This allows to express polymorphism in the SDL syntax description. The actual **class** to be parsed is determined as follows:

- The base **class** declaration shall assign a constant value or range of values to *object_id*.
- Each derived **class** declaration shall assign a constant value or ranges of values to *object_id*. This value or set of values shall correspond to legal *object_id* value(s) for the base **class**.

NOTE 1 — Derivation of classes is possible even when object_ids are not used. However, in that case derived classes may not replace their base **class** in the bitstream.

NOTE 2 — Derived classes may use the same *object_id* value as the base **class**. In that case classes can only be discriminated through context information.

EXAMPLE —

```
class slice: aligned bit(32) slice_start_code=0x00000101 .. 0x000001AF {
   // here we get vertical_size_extension, if present
   if (scalable_mode==DATA_PARTITIONING) {
      unsigned int(7) priority_breakpoint;
   }
   …
}

class foo {
   int(3) a;
   ...
}
```

```
class bar extends foo {
   int(5) b; // this b is preceded by the 3 bits of a
   int(10) c;
   ...
}
```

The order of declaration of the bitstream components is important: it is the same order in which the elements appear in the bitstream. In the above examples, `bar.b` immediately precedes `bar.c` in the bitstream.

Objects may also be encapsulated within other objects. In this case, the *element* in Rule C.1 is an object itself.

12.3.2 Abstract Classes

When the **abstract** keyword is used in the **class** declaration, it indicates that only derived classes of this **class** shall be present in the bitstream. This implies that the derived classes may use the entire range of IDs available. The declaration of the abstract **class** requires a declaration of an ID, with the value 0.

EXAMPLE —

```
abstract class Foo : bit(1) id=0 { // the value 0 is not really used
   ...
}

// derived classes are free to use the entire range of IDs
class Foo0 extends Foo : bit(1) id=0 {
   ...
}

class Foo1 extends Foo : bit(1) id=1 {
   ...
}

class Example {
   Foo f;  // can only be Foo0 or Foo1, not Foo
}
```

12.3.3 Expandable classes

When the **expandable** keyword is used in the **class** declaration, it indicates that the **class** may contain implicit arrays or undefined trailing data, called the "expansion". In this case the **class** encodes its own size in bytes explicitly. This may be used for classes that require future compatible extension or that may include private data. A legacy device is able to decode an expandable **class** up to the last parsable variable that has been defined for a given revision of this **class**. Using the size information, the parser shall skip the **class** data following the last known syntax element. Anywhere in the syntax where a set of expandable classes with *object_id* is expected it is permissible to intersperse expandable classes with unknown *object_id* values. These classes shall be skipped, using the size information.

The size encoding precedes any parsable variables of the **class**. If the **class** has an *object_id*, the encoding of the *object_id* precedes the size encoding. The size information shall not include the number of bytes needed for the size and the *object_id* encoding. Instances of expandable classes shall always have a size corresponding to an integer number of bytes. The size information is accessible within the class as class instance variable `sizeOfInstance`.

If the **expandable** keyword has a *maxClassSize* attribute, then this indicates the maximum permissible size of this **class** in bytes, including any expansion.

The length encoding is itself defined in SDL as follows:

```
int sizeOfInstance = 0;
bit(1) nextByte;
bit(7) sizeOfInstance;
while(nextByte) {
   bit(1) nextByte;
   bit(7) sizeByte;
   sizeOfInstance = sizeOfInstance<<7 | sizeByte;
}
```

12.3.4 Parameter types

A parameter type defines a **class** with parameters. This is to address cases where the data structure of the **class** depends on variables of one or more other objects. Since SDL follows a declarative approach, references to other objects, in such cases, cannot be performed directly (none is instantiated). Parameter types provide placeholders for such references, in the same way as the arguments in a C function declaration. The syntax of a **class** definition with parameters is as follows.

Rule C.2: Class Parameter Types

[**aligned**] [**abstract**] **class** *object_name* [**(***parameter list***)**] [**extends** *parent_class*]
[: **bit(***length***)** [*id_name*=] *object_id* | *id_range*] **{**
[*element*; …] // zero or more elements
}

The parameter list is a list of ***type*** names and variable name pairs separated by commas. Any element of the bitstream, or value derived from the bitstream with a variable-length codeword, or a constant, can be passed as a parameter.

A **class** that uses parameter types is dependent on the objects in its parameter list, whether **class** objects or simple variables. When instantiating such a **class** into an object, the parameters have to be instantiated objects of their corresponding classes or types.

EXAMPLE —

```
class A {
   // class body
   ...
   unsigned int(4) format;
}

class B (A a, int i) {     // B uses parameter types
   unsigned int(i) bar;
   ...
   if( a.format == SOME_FORMAT ) {
      ...
   }
   ...
}

class C {
   int(2) i;
   A a;
   B foo( a, I); // instantiated parameters are required
}
```

12.3.5 Arrays

Arrays are defined in a similar way as in C/C++, i.e., using square brackets. Their length, however, can depend on run-time parameters such as other bitstream values or expressions that involve such values. The array declaration is applicable to both elementary as well as composite objects.

Rule A.1: Arrays

typespec *name* **[***length***]**;

`typespec` is a ***`type`*** specification (including bitstream representation information, e.g. '**`int(`**`2`**`)`**'). The attribute *name* is the name of the array, and *length* is its length.

EXAMPLE —

```
unsigned int(4) a[5];
int(10) b;
int(2) c[b];
```

Here 'a' is an array of 5 elements, each of which is represented using 4 bits in the bitstream and interpreted as an unsigned integer. In the case of 'c', its length depends on the actual value of 'b'. Multi-dimensional arrays are allowed as well. The parsing order from the bitstream corresponds to scanning the array by incrementing first the right-most index of the array, then the second, and so on .

12.3.6 Partial Arrays

In several situations, it is desirable to load the values of an array one by one, in order to check, for example, a terminating or other condition. For this purpose, an extended array declaration is allowed in which individual elements of the array may be accessed.

Rule A.2: Partial Arrays

`typespec` *name***`[[`***index***`]]`**;

Here *index* is the element of the array that is defined. Several such partial definitions may be given, but they shall all agree on the ***`type`*** specification. This notation is also valid for multidimensional arrays.

EXAMPLE —

```
int(4) a[[3]][[5]];
```

indicates the element a(5, 3) of the array (the element in the 6th row and the 4th column), while

```
int(4) a[3][[5]];
```

indicates the entire sixth column of the array, and

```
int(4) a[[3]][5];
```

indicates the entire fourth row of the array, with a length of 5 elements.

NOTE — **a[***5***]** means that the array has five elements, whereas **a[[***5***]]** implies that there are at least six.

12.3.7 Implicit Arrays

When a series of polymorphic classes is present in the bitstream, it may be represented as an array of the same type as that of the base **`class`**. Let us assume that a set of polymorphic classes is defined, derived from the base **`class`** Foo (may or may not be abstract):

```
class Foo : int(16) id = 0 {
   ...
}
```

For an array of such objects, it is possible to implicitly determine the length by examining the validity of the **`class`** ID. Objects are inserted in the array as long as the ID can be properly resolved to one of the IDs defined in the base (if not abstract) or its derived classes. This behavior is indicated by an array declaration without a length specification.

EXAMPLE 1 —

```
class Example {
   Foo f[];   // length implicitly obtained via ID resolution
}
```

To limit the minimum and maximum length of the array, a range specification may be inserted in the specification of the length.

EXAMPLE 2 —

```
class Example {
   Foo f[1 .. 255];   // at least 1, at most 255 elements
}
```

In this example, ‘f’ may have at least 1 and at most 255 elements.

12.4 Arithmetic and Logical Expressions

All standard arithmetic and logical operators of C++ are allowed, including their precedence rules.

12.5 Non-Parsable Variables

In order to accommodate complex syntactic constructs, in which context information cannot be directly obtained from the bitstream but only as a result of a non-trivial computation, non-parsable variables are allowed. These are strictly of local scope to the **class** they are defined in. They may be used in expressions and conditions in the same way as bitstream-level variables. In the following example, the number of non-zero elements of an array is computed.

```
unsigned int(6) size;
int(4) array[size];
…
int i; // this is a temporary, non-parsable variable
for (i=0, n=0; i<size; i++) {
   if (array[[i]]!=0)
      n++;
}

int(3) coefficients[n];
// read as many coefficients as there are non-zero elements in array
```

12.6 Syntactic Flow Control

The syntactic flow control provides constructs that allow conditional parsing, depending on context, as well as repetitive parsing. The familiar C/C++ if-then-else construct is used for testing conditions. Similarly to C/C++, zero corresponds to false, and non-zero corresponds to true.

Rule FC.1: Flow Control Using If-Then-Else

```
if (condition) {
   …
} [else if (condition) {
   …
}] [else {
   …
}]
```

EXAMPLE 1 —

```
class conditional_object {
   unsigned int(3) foo;
   bit(1) bar_flag;
   if (bar_flag) {
      unsigned int(8) bar;
   }
   unsigned int(32) more_foo;
}
```

Here the presence of the entity ‘bar’ is determined by the ‘bar_flag’.

EXAMPLE 2 —

```
class conditional_object {
   unsigned int(3) foo;
   bit(1) bar_flag;
   if (bar_flag) {
      unsigned int(8) bar;
   } else {
      unsigned int(some_vlc_table) bar;
   }
   unsigned int(32) more_foo;
}
```

Here we allow two different representations for ‘bar’, depending on the value of ‘bar_flag’. We could equally well have another entity instead of the second version (the variable length one) of ‘bar’ (another object, or another variable). Note that the use of a flag necessitates its declaration before the conditional is encountered. Also, if a variable appears twice (as in the example above), the types shall be identical.

In order to facilitate cascades of if-then-else constructs, the ‘switch’ statement is also allowed.

Rule FC.2: Flow Control Using Switch

```
switch (condition) {
      [case label1: …]
      [default:]
}
```

The same category of context-sensitive objects also includes iterative definitions of objects. These simply imply the repetitive use of the same syntax to parse the bitstream, until some condition is met (it is the conditional repetition that implies context, but fixed repetitions are obviously treated the same way). The familiar structures of ‘for’, ‘while’, and ‘do’ loops can be used for this purpose.

Rule FC.3: Flow Control Using For

```
for  (expression1; expression2; expression3) {
      …
}
```

expression1 is executed prior to starting the repetitions. Then *expression2* is evaluated, and if it is non-zero (true) the declarations within the braces are executed, followed by the execution of *expression3*. The process repeats until *expression2* evaluates to zero (false).

Note that it is not allowed to include a variable declaration in *expression1* (in contrast to C++).

Rule FC.4: Flow Control Using Do

```
do {
      …
} while (condition);
```

Here the block of statements is executed until *condition* evaluates to false. Note that the block will be executed at least once.

Rule FC.5: Flow Control Using While

```
while (condition) {
        …
}
```

The block is executed zero or more times, as long as *condition* evalutes to non-zero (true).

12.7 Built-In Operators

The following built-in operators are defined.

Rule O.1: lengthof() Operator

```
lengthof(variable)
```

This operator returns the length, in bits, of the quantity contained in parentheses. The length is the number of bits that was most recently used to parse the quantity at hand. A return value of 0 means that no bits were parsed for this variable.

12.8 Scoping Rules

All parsable variables have class scope, i.e., they are available as class member variables.

For non-parsable variables, the usual C++/Java scoping rules are followed (a new scope is introduced by curly braces: '{' and '}'). In particular, only variables declared in class scope are considered class member variables, and are thus available in objects of that particular type.

13 Profiles

13.1 Introduction

This clause defines profiles and levels for the usage of the tools defined in this part of ISO/IEC 14496. Each profile at a given level constitutes a subset of ISO/IEC 14496-1 to which system manufacturers and content creators can claim conformance in order to ensure interoperability.

The object descriptor profiles (OD profiles) specify the allowed configurations of the object descriptor tool and the sync layer tool. The scene graph profiles specify the allowed scene graph elements of the BIFS tool. The graphics profiles specify the graphics elements of the BIFS tool that are allowed.

Profile definitions, by themselves, are not sufficient to provide a full characterization of a receiving terminal's capabilities and the resources needed for a presentation. For this reason, levels are defined within each profile. Levels constrain the values of parameters in a given profile in order to specify an upper complexity bound.

13.2 OD Profile Definitions

13.2.1 Overview

The object descriptor profiles (OD profiles) specify the configurations of the object descriptor tool and the sync layer tool that are allowed. The object descriptor tool provides a structure for all descriptive information. The sync layer tool provides the syntax to convey, among others, timing information for elementary streams. object descriptor profiles are used, in particular, to reduce the amount of asynchronous operations as well as the amount of permanent storage.

13.2.2 OD Profiles Tools

The following tools are available to construct OD profiles:

— Object descriptor (OD) tool as defined in 8.5.

— Sync layer (SL) tool as defined in 10.2.

— Object content information (OCI) tool as defined in 8.4.

— Intellectual property management and protection (IPMP) tool as defined in 8.3.

13.2.3 OD Profiles

The OD profiles are defined in the following table. Currently, only one profile is defined, comprising all the tools. No additional profiles are foreseen at the moment, but the possibility of adding Profiles through amendments is left open.

Table 49 - OD Profiles

	OD Profiles
OD Tools	**Core**
SL	X
OD	X
OCI	X
IPMP	X

Decoders that claim compliance to a given profile shall implement all the tools with an 'X' entry for that profile.

13.2.4 OD Profiles@Levels

13.2.4.1 Levels for the Core Profile

No levels are defined yet for the OD Core profile. Future definition of Levels is anticipated; this will happen by means of an amendment to this part of the standard.

13.3 Scene Graph Profile Definitions

13.3.1 Overview

The scene graph profiles specify the scene graph elements of the BIFS tool that are allowed. These elements provide the means to describe the spatio-temporal locations, the hierarchical dependencies as well as the behaviors of audio-visual objects in a scene. Profiling of scene graph elements of the BIFS tool serves to restrict the memory requirements and computational complexities of scene graph traversal and processing of specified behaviors during the composition and rendering processes.

13.3.2 Scene Graph Profiles Tools

The following tools are available to construct the definitions for scene graph profiles:

— BIFS nodes related to scene description as defined in Table 50.

— BIFS commands and BIFS animation as defined in 9.3.6 and 9.3.8, respectively.

— BIFS ROUTES as defined in 9.3.7.45.

13.3.3 Scene Graph Profiles

The following table defines the scene graph profiles:

Table 50 - Scene graph profiles

	Scene Graph Profiles			
Scene Graph Tools	**Audio**	**Simple 2D**	**Complete 2D**	**Complete**
Anchor			X	X
AudioBuffer	X		X	X
AudioDelay	X		X	X
AudioFX	X		X	X
AudioMix	X		X	X
AudioSwitch	X		X	X
Billboard				X
Collision				X
Composite2DTexture			X	X
Composite3DTexture				X
Form			X	X
Group	X	X	X	X
Inline			X	X
Layer2D			X	X
Layer3D				X
Layout			X	X
ListeningPoint			X	X
LOD				X
NavigationInfo				X
OrderedGroup		X	X	X
QuantizationParameter			X	X
Sound				X
Sound2D	X	X	X	X
Switch			X	X
Transform				X
Transform2D		X	X	X
Viewpoint				X
WorldInfo			X	X
Node Update			X	X
Route Update			X	X
Scene Update	X	X	X	X
AnimationStream			X	X
Script			?	X
ColorInterpolator			X	X
Conditional			X	X
CoordinateInterpolator2D			X	X
CoordinateInterpolator				X
CylinderSensor				X
DiscSensor			X	X
NormalInterpolator				X
OrientationInterpolator				X
PlaneSensor2D			X	X
PlaneSensor				X
PositionInterpolator				X
PositionInterpolator2D			X	X
ProximitySensor				X
ProximitySensor2D			X	X
ROUTE			X	X

ScalarInterpolator			X	X
SphereSensor				X
TermCap			X	X
TimeSensor			X	X
TouchSensor			X	X
VisibilitySensor				X
Valuator			X	X

Decoders that claim compliance to a given profile shall implement all the tools with an ‘X’ entry for that profile.

13.3.3.1 BIFS nodes for audio objects

The presence of AudioClip and AudioSource nodes in BIFS scene graph depends on the selected Audio profile. The following table describes what nodes are allowed in the BIFS scene graph depending on the Audio profile.

Table 51 - BIFS nodes for audio objects

Audio Profiles	Allowed Audio Object Nodes
Main	AudioClip, AudioSource
Scalable	AudioClip, AudioSource
Speech	AudioClip, AudioSource
Low Rate Synthesis	AudioClip, AudioSource

13.3.3.2 BIFS nodes for visual objects

The presence of ImageTexture, Background2D, Background, MovieTexture, Face, Expression, FAP, FDP, FIT, FaceDefMesh, FaceDefTable, FaceDefTransform, Viseme nodes in a BIFS scene graph depends on the selected Visual profile. The following table describes what nodes are allowed in the BIFS scene graph depending on the choice of the Visual profile.

Table 52 - BIFS nodes for visual objects

Visual Profiles	Allowed visual object nodes
Simple	ImageTexture, Background2D, Background, MovieTexture
Simple Scalable	ImageTexture, Background2D, Background, MovieTexture
Core	ImageTexture, Background2D, Background, MovieTexture
Main	ImageTexture, Background2D, Background, MovieTexture
Simple Scalable	ImageTexture, Background2D, Background, MovieTexture
N-Bit	ImageTexture, Background2D, Background, MovieTexture
Hybrid	ImageTexture, Background2D, Background, MovieTexture, Face, Expression, FAP, FDP, FIT, FaceDefMesh, FaceDefTable, FaceDefTransform, Viseme
Basic Animated Texture	ImageTexture, Background2D, Background, Face, Expression, FAP, FDP, FIT, FaceDefMesh, FaceDefTable, FaceDefTransform, Viseme
Scaleable Texture	ImageTexture, Background2D, Background
Simple Face	Face, Expression, FAP, FDP, FIT, FaceDefMesh, FaceDefTable, FaceDefTransform, Viseme

If the terminal complies with a 2D graphics profile only, the terminal may choose to ignore the contents of the FDP, FIT, FaceDefMesh, FaceDefTable, FaceDefTransform nodes.

13.3.4 Scene Graph Profiles@Levels

13.3.4.1 Levels for the Audio Scene Graph Profile

13.3.4.1.1 Functionalities provided

The Audio scene graph profile provides for a set of BIFS scene graph elements for usage in audio only applications. The Audio scene graph profile supports applications like broadcast radio.

13.3.4.1.2 Levels

No levels are yet defined for the Audio scene graph profile. Future definition of Levels is anticipated; this will happen by means of an amendment to this part of the standard.

13.3.4.2 Levels for the Simple 2D Scene Graph Profile

13.3.4.2.1 Functionalities provided

The Simple 2D scene graph profile provides for only those BIFS scene graph elements necessary to place one or more audio-visual objects in a scene. The Simple 2D scene graph profile allows presentation of audio-visual content with potential update of the complete scene but no interaction capabilities. The Simple 2D scene graph profile supports applications like broadcast television.

13.3.4.2.2 Level 1

The following restrictions apply for the Simple 2D scene graph profile at Level 1:

Table 53 - Restrictions for Simple 2D scene graph profile at Level 1

Transform2D	
Field name	
addChildren	Ignored
removeChildren	Ignored
children	X.
center	Ignored
rotationAngle	0
scale	1, 1
scaleOrientation	0
translation	X

X = allowed;
else: default value	

The metric shall be the pixel metrics. BIFSConfig.isPixel=1.

A cascade of Transform2D nodes is not allowed. Children nodes of a Transform2D node shall not be Transform2D nodes. Only one initial update to convey the complete scene graph is allowed.

13.3.4.3 Levels for the Complete 2D Scene Graph Profile

13.3.4.3.1 Functionalities provided

The Complete 2D scene graph profile provides for all the 2D scene description elements of the BIFS tool. It supports features such as 2D transformations and alpha blending. The Complete 2D scene graph profile enables 2D applications that require extensive and customized interactivity.