INTERNATIONAL STANDARD

ISO/IEC 23000-13

Second edition
2017-11

Information technology - Multimedia application format (MPEG-A) —

Part 13: Augmented reality application format

Technologies de l'information - Format des applications multimedias —

Partie 13: Format pour les Applications de Realité Augmentée

Reference number
ISO/IEC 23000-13:2017(E)

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

Contents

Page

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. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: www.iso.org/iso/foreword.html.

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

This second edition cancels and replaces the first edition (ISO/IEC 23000-13:2014), which has been technically revised.

It also incorporates the Amendment ISO/IEC 23000-13:2014/Amd. 1:2015.

A list of all parts in the ISO/IEC 23000 series can be found on the ISO website.

Introduction

Augmented Reality (AR) applications refer to a view of a real-world environment (RWE), whose elements are augmented by content, such as graphics or sound, in a computer driven process. Augmented Reality Application Format (ARAF) is a collection of a subset of the ISO/IEC 14496-11 Scene Description and Application Engine standard, combined with other relevant MPEG standards (e.g. ISO/IEC 23005, MPEG-V), designed to enable the consumption of 2D/3D multimedia content. Consequently, this document focuses not on client or server procedures, but on the data formats used to provide an augmented reality presentation.

Information technology - Multimedia application format (MPEG-A) —

Part 13:
Augmented reality application format

1 Scope

This document specifies the following:

— scene description elements for representing AR content;

— mechanisms to connect to local and remote sensors and actuators;

— mechanisms to integrated compressed media (image, audio, video, graphics);

— mechanisms to connect to remote resources such as maps and compressed media.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 10646-1:2012, *Information technology — Universal multiple-octet coded character set (UCS) — Part 1: Architecture and basic multilingual plane*

ISO/IEC 14496-1:2010 + Amd. 2:2014, *Information technology — Coding of audio-visual objects — Part 1: Systems*

ISO/IEC 14496-3:2009, *Information technology — Coding of audio-visual objects — Part 3: Audio*

ISO/IEC 14496-11:2015, *Information technology — Coding of audio-visual objects — Part 11: Scene description and application engine*

ISO/IEC 14496-16:2011, *Information technology — Coding of audio-visual objects — Part 16: Animation Framework eXtension (AFX)*

ISO/IEC 14772-1:1997, *Information technology — Computer graphics and image processing — The Virtual Reality Modeling Language — Part 1: Functional specification and UTF-8 encoding*

ISO/IEC 23005-5, *Information technology — Media context and control — Part 5: Data formats for interaction devices*

3 Terms, definitions, and abbreviated terms

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 23000-1 and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at http://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

3.1.1
ARAF browser
augmented reality application format compliant browser

3.1.2
MAR scene
textual result of the MAREC creation, played by an *ARAF browser* (3.1.1)

Note 1 to entry: The result is a MAR experience.

3.1.3
MAR experience
act of playing the ARAF scene using an *ARAF browser* (3.1.1)

Note 1 to entry: The ARAF browser interprets the ARAF scene and presents the result on the end-user's device.

3.1.4
content creator
creator of the media files that are being used within the *MAR experience* (3.1.3)

Note 1 to entry: The media files can be 2D and/or 3D graphics, images, videos and/or sounds.

3.1.5
end-user device
smartphone or mobile device used by an end-user to play a *MAR scene* (3.1.2)

Note 1 to entry: The device shall have an ARAF browser installed.

3.1.6
processing server
server that offers at least one required processing functionality for a *MAR experience* (3.1.3) and it is capable of communicating with an *ARAF browser* (3.1.1)

3.1.7
target resource
target image or target image descriptor

Note 1 to entry: The target image represents the image that shall be detected and recognized by a recognition library. The target image descriptor is represented by the visual descriptors extracted from a target image. The target resources may be specified by the MAREC or they can be already stored in databases on remote servers.

3.1.8
prerecorded video
prerecorded 2D video whose location is specified by MAREC

Note 1 to entry: The video file can be stored locally (on the device where the MAR experience is played) or remotely (anywhere else on the web). The recognition process shall be performed on the frames (still images) composing the video.

3.1.9
live video camera (stream)
live 2D video camera feed

Note 1 to entry: The URL of the camera providing the real time capture is specified by the MAREC. The URL can point to one of the cameras of the device where the MAR experience is played or to any other camera that can provide a live video stream and the ARAF browser can connect to.

3.1.10
image recognition library
library that is able to recognize *target resources* (3.1.7) in a video

Note 1 to entry: The library can run locally (implemented in the ARAF browser) or remotely (on a processing server). The result of an image recognition library is an array of indexes of the recognized target resources.

3.1.11
image recognition and tracking library
library that is able to recognize and track *target resources* (3.1.7) in a video

Note 1 to entry: The library can run locally (implemented in the ARAF browser) or remotely (on a processing server). The result of a recognition and tracking library is an array of indexes of the recognized target resources and their pose matrixes. Each recognized target resource shall have a pose matrix associated or a default value if the corresponding pose matrix could not be computed.

3.1.12
augmentation resource
media objects that are used in the augmentation of the *MAR experience* (3.1.3)

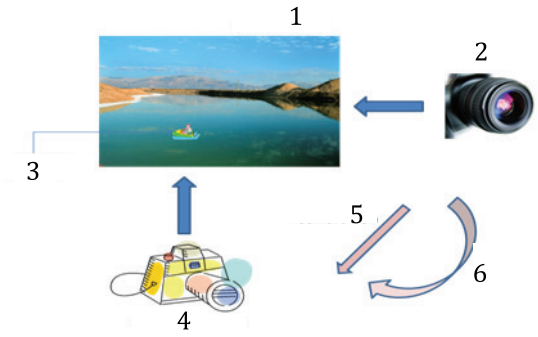
Note 1 to entry: A valid augmentation resource can be a 2D/3D grapic element, an image, a video, a sound or a BIFS scene. The augmentation resources can be stored locally in the MAR Scene or remotely anywhere on the Web, as long as the ARAF browser is capable of accesing their locations. In this case, a URL pointing to the augmentation resource is stored in the MAR scene.

3.2 Abbreviated terms

AR	Augmented Reality
ARAF	Augmented Reality Application Format
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
MAR	Mixed and Augmented Reality
MARE	Mixed and Augmented Reality Experience
MAREC	Mixed and Augmented Reality Experience Creator
PROTO	A PROTOtype is a mechanism used to group together scene graph elements in order to implement one or several specific functionalities.
RTR	Recognized Target Resource

4 ARAF principle and context

Augmented Reality (AR) applications refer to a view of a real-world environment whose elements are augmented by content, such as graphics or sound, in a computer driven process. Figure 1 illustrates two real and virtual cameras and the composition of a real image and graphics objects. Annex C describes several application scenarios for augmented reality.



Key

1 real picture
2 real camera
3 graphic object
4 virtual camera
5 calibration
6 position and orientation

Figure 1 — Simplified illustration of the AR principle

The Augmented Reality Application Format (ARAF) is an extension of a subset of the MPEG-4 part 11 Scene Description and Application Engine standard, combined with other relevant MPEG standards (MPEG-4, MPEG-V), designed to enable the consumption of 2D/3D multimedia content as depicted in Figure 2.

An ARAF, available as a file or stream, is interpreted by a device, called **ARAF device**. The nodes of the ARAF scene point to different sources of multimedia content such as 2D/3D image, 2D/3D audio, 2D/3D video, 2D/3D graphics and sensor/sensory information sources/sinks that are either remote or/and local.

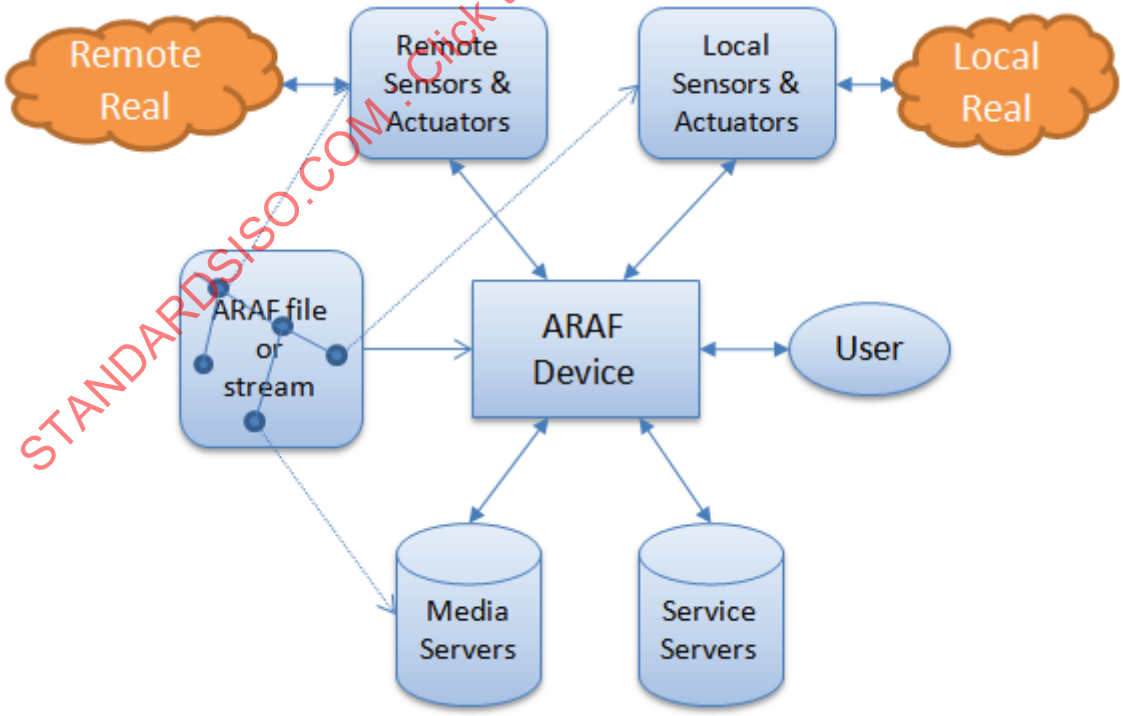


Figure 2 — The ARAF context

5 ARAF scene description

5.1 General

To describe the multimedia scene, ARAF is based on ISO/IEC 14496-11 which at its turn is based on ISO/IEC 14772-1 (VRML97). About two hundred nodes are standardized in MPEG-4 BIFS and VRML, allowing various kinds of scenes to be constructed. ARAF is referring to a subset of MPEG-4 BIFS nodes and external prototypes as defined in ISO/IEC 14496-11:2015, Annex E for scene description as presented in Table 1.

Table 1 — ARAF nodes and prototypes

Category	Sub-category	Node, prototypes/elements name in MPEG-4 BIFS/XMT	Type
Elementary media	Audio	AudioSource	Node
		Sound	Node
		Sound2D	Node
	Image and video	ImageTexture	Node
		MovieTexture	Node
	Textual information	FontStyle	Node
		Text	Node
	Graphics	Appearance	Node
		Color	Node
		LineProperties	Node
		LinearGradient	Node
		Material	Node
		Material2D	Node
		Rectangle	Node
		Shape	Node
		SBVCAnimationV2	Node
		SBBone	Node
		SBSegment	Node
		SBSite	Node
		SBSkinnedModel	Node
		MorphShape	Node
		Coordinate	Node
		TextureCoordinate	Node
		Normal	Node
		IndexedFaceSet	Node
		IndexedLineSet	Node
Programming information		Script	Node

Table 1 *(continued)*

Category	Sub-category	Node, prototypes/elements name in MPEG-4 BIFS/XMT	Type
User interactivity		InputSensor	Node
		OutputActuator	Node
		SphereSensor	Node
		TimeSensor	Node
		TouchSensor	Node
		MediaSensor	Node
		PlaneSensor	Node
		Background	Node
Scene related) information (spatial and temporal relationships		Background2D	Node
		CameraCalibration	PROTO
		Group	Node
		Inline	Node
		Layer2D	Node
		Layer3D	Node
		Layout	Node
		NavigationInfo	Node
		OrderedGroup	Node
		LocImg	PROTO
		RemImgProxy	PROTO
		RemImgServer	PROTO
		RemImgComp	PROTO
		LocAud	PROTO
		RemAud	PROTO
		Switch	Node
		Transform	Node
		Transform2D	Node
		Viewpoint	Node
		Viewport	Node
		Form	Node
Dynamic and animated scene		OrientationInterpolator	Node
		ScalarInterpolator	Node
		CoordinateInterpolator	Node
		ColorInterpolator	Node
		PositionInterpolator	Node
		Valuator	Node
Communication and compression		BitWrapper	Node
		MediaControl	Node
	Maps	Map	PROTO
		MapOverlay	PROTO
		MapMarker	PROTO
		MapPlayer	PROTO
Terminal		TermCap	Node

All the above listed elements are specified in MPEG-4 Part 11. However, to facilitate the implementation of ARAF content, the current document contains their XML syntax as well as the semantics and functionality.

MPEG-4 Part 11 describes a scene with a hierarchical structure that can be represented as a graph. Nodes of the graph build up various types of objects, such as audio video, image, graphic, text, etc. Furthermore, to ensure the flexibility, a new, user-defined type of node derived from a parent one can also be defined on demand by using the *Proto* method.

In general, nodes expose a set of parameters, through which aspects of their appearance and behavior can be controlled. By setting these values, scene designers have a tool to force a scene-reconstruction at clients' terminals to adhere to their intention in a predefined manner. In more complicated scenario, the structure of BIFS nodes is not necessarily static; nodes can be added or removed from the scene graph arbitrarily.

Certain types of nodes called *sensors*, such as TimeSensor, TouchSensor, can interact with users and generate appropriate triggers, which are transmitted to others nodes by routing mechanism, causing changes in state of these receiving nodes. They are bases for the dynamic behavior of a multimedia content supported by MPEG-4.

The maximum flexibility in the programmable feature of MPEG-4 scene is carried out with the *Script* node. By routing mechanism to Event In *valueIn* attribute of Script node, the associated function (defined in its URL attribute) with the same name Event In *valueIn* () will be triggered. The behavior of this function is user-defined, i.e. scene-designer can freely process some computations, and then sets the values for every Event Out *valueOut* attribute, which consecutively affect the states of other nodes linked to them.

Direct manipulation of nodes' states is also possible in MPEG-4 Part 11: the Field *field* attribute can refer to any node in the scene; through this link, all attributes of the contacted node will be exposed to direct setting and modifying operators within the *Script* node. The syntax of the language used to implement the function of Script node is ECMAScript (see ISO/IEC 16262).

ARAF supports the definition and reusability of complex objects by using the MPEG-4 PROTO mechanism. The PROTO statement creates its own nodes by defining a configurable object prototype; it can integrate any other node from the scene graph.

ARAF makes extensive use of EXTERNPROTO mechanism which are nodes whose syntax is identified by URNs as given in ISO/IEC 14496-11:2915, Annex E, while their names are only informative and for convenience can be changed by the content creator in the declaration step.

Table 1 indicates the MPEG-4 Part 11 nodes that are included in ARAF. For each node, it is specified in the version of this document when it was published. Further, the XML syntax as well as the semantics and functionality of these elements are described.

5.1.1 Elementary media

5.1.1.1 Audio

The following audio related nodes are used in ARAF: AudioSource, Sound, Sound2D.

5.1.1.1.1 AudioSource

5.1.1.1.1.1 XSD description

```
<complexType name="AudioSourceType">
 <all>
   <element ref="xmta:IS" minOccurs="0"/>
   <element name="children" form="qualified" minOccurs="0">
     <complexType>
       <group ref="xmta:SFAudioNodeType" minOccurs="0" maxOccurs="unbounded"/>
     </complexType>
```

```
    </element>
  </all>
  <attribute name="url" type="xmta:MFUrl" use="optional"/>
  <attribute name="pitch" type="xmta:SFFloat" use="optional" default="1"/>
  <attribute name="speed" type="xmta:SFFloat" use="optional" default="1"/>
  <attribute name="startTime" type="xmta:SFTime" use="optional" default="0"/>
  <attribute name="stopTime" type="xmta:SFTime" use="optional" default="0"/>
  <attribute name="numChan" type="xmta:SFInt32" use="optional" default="1"/>
  <attribute name="phaseGroup" type="xmta:MFInt32" use="optional"/>
  <attributeGroup ref="xmta:DefUseGroup"/>
 </complexType>
 <element name="AudioSource" type="xmta:AudioSourceType"/>
```

5.1.1.1.1.2 Functionality and semantics

As defined in ISO/IEC 14496-11:2015, 7.2.2.15.

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 or AdvancedAudioBuffer data to be used as sound samples within a structured audio decoding process. Only AudioBuffer and AdvancedAudioBuffer 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, the parametric speech (HVXC) and the parametric audio (HILN) 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 getttune() core opcode in the structured audio decoder (see ISO/IEC 14496-3:2009, Clause 5). 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 14496-3:2009, Clause 5), the parametric speech (HVXC) and the parametric audio (HILN) decoder. 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 14496-3:2009, 5.7.3.3.6, list item 8 describes the use of this field to control the structured audio decoder. To adjust the speed of other decoder types, use the AudioFX node with an appropriate effects orchestra (see ISO/IEC 14496-3:2009, 5.9.14.4).

The startTime and stopTime exposedFields and their effects on the AudioSource node are described in ISO/IEC 14496-11:2015, 7.1.1.1.6.2. The numChan field describes how many channels of audio are in the decoded bitstream.

5.1.1.1.2 Sound

5.1.1.1.2.1 XSD description

```
<complexType name="SoundType">
 <all>
    <element ref="xmta:IS" minOccurs="0"/>
    <element name="source" form="qualified" minOccurs="0">
        <complexType>
          <group ref="xmta:SFAudioNodeType" minOccurs="0"/>
        </complexType>
    </element>
 </all>
 <attribute name="direction" type="xmta:SFVec3f" use="optional" default="0 0 1"/>
 <attribute name="intensity" type="xmta:SFFloat" use="optional" default="1"/>
 <attribute name="location" type="xmta:SFVec3f" use="optional" default="0 0 0"/>
 <attribute name="maxBack" type="xmta:SFFloat" use="optional" default="10"/>
 <attribute name="maxFront" type="xmta:SFFloat" use="optional" default="10"/>
 <attribute name="minBack" type="xmta:SFFloat" use="optional" default="1"/>
```



```
    <attribute name="minFront" type="xmta:SFFloat" use="optional" default="1"/>
    <attribute name="priority" type="xmta:SFFloat" use="optional" default="0"/>
    <attribute name="spatialize" type="xmta:SFBool" use="optional" default="true"/>
    <attributeGroup ref="xmta:DefUseGroup"/>
  </complexType>
  <element name="Sound" type="xmta:SoundType"/>
```

5.1.1.1.2.2 Functionality and semantics

As defined in ISO/IEC 14496-11:2015, 7.2.2.116.

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 described using ISO/IEC 14496-11 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, 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 ISO/IEC 14496-11:2015, 7.1.1.2.13.2.2. 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 ISO/IEC 14496-11:2015, 7.1.1.2.13.2.2.

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 apparant 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.

5.1.1.1.3 Sound2D

5.1.1.1.3.1 XSD description

```
<complexType name="Sound2DType">
 <all>
   <element ref="xmta:IS" minOccurs="0"/>
   <element name="source" form="qualified" minOccurs="0">
     <complexType>
        <group ref="xmta:SFAudioNodeType" minOccurs="0"/>
     </complexType>
   </element>
 </all>
 <attribute name="intensity" type="xmta:SFFloat" use="optional" default="1"/>
 <attribute name="location" type="xmta:SFVec2f" use="optional" default="0 0"/>
 <attribute name="spatialize" type="xmta:SFBool" use="optional" default="true"/>
 <attributeGroup ref="xmta:DefUseGroup"/>
```

```
</complexType>
<element name="Sound2D" type="xmta:Sound2DType"/>
```

5.1.1.1.3.2 Functionality and semantics

As defined in ISO/IEC 14496-11:2015, 7.2.2.117.

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. 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 1 m between the user and a 2D scene of size 2 m × 1,5 m, 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. 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. If Sound and Sound2D nodes are both used in a scene, all shall be treated the same way according to these semantics.

5.1.1.2 Image and video

The following image and video related nodes are used in ARAF: ImageTexture, MovieTexture.

5.1.1.2.1 ImageTexture

5.1.1.2.1.1 XSD description

```
<complexType name="ImageTextureType">
 <all>
   <element ref="xmta:IS" minOccurs="0"/>
 </all>
 <attribute name="url" type="xmta:MFUrl" use="optional"/>
 <attribute name="repeatS" type="xmta:SFBool" use="optional" default="true"/>
 <attribute name="repeatT" type="xmta:SFBool" use="optional" default="true"/>
 <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="ImageTexture" type="xmta:ImageTextureType"/>
```

5.1.1.2.1.2 Functionality and semantics

As defined in ISO/IEC 14772-1:1997, 6.22.

The ImageTexture node defines a texture map by specifying an image file and general parameters for mapping to geometry. Texture maps are defined in a 2D coordinate system (s, t) that ranges from [0.0, 1.0] in both directions. The bottom edge of the image corresponds to the S-axis of the texture map, and left edge of the image corresponds to the T-axis of the texture map. The lower-left pixel of the image corresponds to s=0, t=0, and the top-right pixel of the image corresponds to s=1, t=1.

The texture is read from the URL specified by the url field. When the url field contains no values ([]), texturing is disabled. Browsers shall support the JPEG and PNG image file formats. In addition, browsers may support other image formats (e.g. CGM) which can be rendered into a 2D image. Support for the GIF format is also recommended (including transparency).

The repeatS and repeatT fields specify how the texture wraps in the S and T directions. If repeatS is TRUE (the default), the texture map is repeated outside the [0.0, 1.0] texture coordinate range in the S direction so that it fills the shape. If repeatS is FALSE, the texture coordinates are clamped in the S direction to lie within the [0.0, 1.0] range. The repeatT field is analogous to the repeatS field.

5.1.1.2.2 MovieTexture

5.1.1.2.2.1 XSD description

```
<complexType name="MovieTextureType">
 <all>
    <element ref="xmta:IS" minOccurs="0"/>
 </all>
 <attribute name="loop" type="xmta:SFBool" use="optional" default="false"/>
 <attribute name="speed" type="xmta:SFFloat" use="optional" default="1"/>
 <attribute name="startTime" type="xmta:SFTime" use="optional" default="0"/>
 <attribute name="stopTime" type="xmta:SFTime" use="optional" default="0"/>
 <attribute name="url" type="xmta:MFUrl" use="optional"/>
 <attribute name="repeatS" type="xmta:SFBool" use="optional" default="true"/>
 <attribute name="repeatT" type="xmta:SFBool" use="optional" default="true"/>
 <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="MovieTexture" type="xmta:MovieTextureType"/>
```

5.1.1.2.2.2 Functionality and semantics

As defined in ISO/IEC 14496-11:2015, 7.2.2.86.

The loop, startTime, and stopTime exposedFields and the isActive eventOut, and their effects on the MovieTexture node, are described in ISO/IEC 14496-11:2015, 7.1.1.1.6.2. 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.

5.1.1.3 Textual information

The following text related nodes are used in ARAF: FontStyle, Text.

5.1.1.3.1 FontStyle

5.1.1.3.1.1 XSD description

```
<complexType name="FontStyleType">
 <all>
   <element ref="xmta:IS" minOccurs="0"/>
```

```
        </all>
        <attribute name="family" type="xmta:MFString" use="optional"
default="&amp;quot;SERIF&amp;quot;"/>
        <attribute name="horizontal" type="xmta:SFBool" use="optional" default="true"/>
        <attribute name="justify" type="xmta:MFString" use="optional"
default="&amp;quot;BEGIN&amp;quot;"/>
        <attribute name="language" type="xmta:SFString" use="optional"
default="&amp;quot;&amp;quot;"/>
        <attribute name="leftToRight" type="xmta:SFBool" use="optional" default="true"/>
        <attribute name="size" type="xmta:SFFloat" use="optional" default="1"/>
        <attribute name="spacing" type="xmta:SFFloat" use="optional" default="1"/>
        <attribute name="style" type="xmta:SFString" use="optional"
default="&amp;quot;PLAIN&amp;quot;"/>
        <attribute name="topToBottom" type="xmta:SFBool" use="optional" default="true"/>
        <attributeGroup ref="xmta:DefUseGroup"/>
      </complexType>
      <element name="FontStyle" type="xmta:FontStyleType"/>
```

5.1.1.3.1.2 Functionality and semantics

As defined in ISO/IEC 14772-1:1997, 6.20.

The FontStyle node defines the size, family, and style used for Text nodes, as well as the direction of the text strings and any language-specific rendering techniques used for non-English text.

The size field specifies the nominal height, in the local coordinate system of the Text node, of glyphs rendered and determines the spacing of adjacent lines of text. Values of the size field shall be greater than zero.

The spacing field determines the line spacing between adjacent lines of text. The distance between the baseline of each line of text is (spacing × size) in the appropriate direction (depending on other fields described below). Values of the spacing field shall be non-negative.

Font attributes are defined with the family and style fields. The browser shall map the specified font attributes to an appropriate available font as described below.

The family field contains a case-sensitive MFString value that specifies a sequence of font family names in preference order. The browser shall search the MFString value for the first font family name matching a supported font family. If none of the string values matches a supported font family, the default font family "SERIF" shall be used. All browsers shall support at least "SERIF" (the default) for a serif font such as Times Roman; "SANS" for a sans-serif font such as Helvetica; and "TYPEWRITER" for a fixed-pitch font such as Courier. An empty family value is identical to ["SERIF"].

The style field specifies a case-sensitive SFString value that may be "PLAIN" (the default) for default plain type; "BOLD" for boldface type; "ITALIC" for italic type; or "BOLDITALIC" for bold and italic type. An empty style value ("") is identical to "PLAIN".

The horizontal, leftToRight, and topToBottom fields indicate the direction of the text. The horizontal field indicates whether the text advances horizontally in its major direction (horizontal = TRUE, the default) or vertically in its major direction (horizontal = FALSE). The leftToRight and topToBottom fields indicate direction of text advance in the major (characters within a single string) and minor (successive strings) axes of layout. Which field is used for the major direction and which is used for the minor direction is determined by the horizontal field.

For horizontal text (horizontal = TRUE), characters on each line of text advance in the positive X direction if leftToRight is TRUE or in the negative X direction if leftToRight is FALSE. Characters are advanced according to their natural advance width. Each line of characters is advanced in the negative Y direction if topToBottom is TRUE or in the positive Y direction if topToBottom is FALSE. Lines are advanced by the amount of size × spacing.

For vertical text (horizontal = FALSE), characters on each line of text advance in the negativeY direction if topToBottom is TRUE or in the positive Y direction if topToBottom is FALSE. Characters are advanced according to their natural advance height. Each line of characters is advanced in the positive X direction

if leftToRight is TRUE or in the negative X direction if leftToRight is FALSE. Lines are advanced by the amount of size × spacing.

The justify field determines alignment of the above text layout relative to the origin of the object coordinate system. The justify field is an MFString which can contain 2 values. The first value specifies alignment along the major axis and the second value specifies alignment along the minor axis, as determined by the horizontal field. An empty justify value ("") is equivalent to the default value. If the second string, minor alignment, is not specified, minor alignment defaults to the value "FIRST". Thus, justify values of "", "BEGIN", and ["BEGIN" "FIRST"] are equivalent.

The major alignment is along the X-axis when horizontal is TRUE and along the Y-axis when horizontal is FALSE. The minor alignment is along the Y-axis when horizontal is TRUE and along the X-axis when horizontal is FALSE. The possible values for each enumerant of the justify field are "FIRST", "BEGIN", "MIDDLE", and "END". For major alignment, each line of text is positioned individually according to the major alignment enumerant. For minor alignment, the block of text representing all lines together is positioned according to the minor alignment enumerant.

The language field specifies the context of the language for the text string. Due to the multilingual nature of the ISO/IEC 10646-1, the language field is needed to provide a proper language attribute of the text string. The format is based on RFC 1766: language[_territory].

The value for the language tag is based on ISO 639 (e.g. “zh” for Chinese, “jp” for Japanese, and “sc” for Swedish.) The territory tag is based on ISO 3166 country codes (e.g. “TW” for Taiwan and “CN” for China for the “zh” Chinese language tag). If the language field is empty (""), local language bindings are used.

The semantics of the FontStyle node are the ones specified above (ISO/IEC 14772-1:1997, 6.20), with the exception that the field types are exposedField and the semantics of the size and spacing fields are as follows.

The size field defines the size of the EM box of a font (The EM is a relative measure of the height of the glyphs in a font defined in a device- and resolution-independent font design units). This value corresponds to the distance between two adjacent baselines of unadjusted text, set in a particular font. The value of the size field is conveyed using the same metric units that are used for a scene description. If a scene uses pixel-based metrics, the value of the size field is specified in pixels, otherwise, it specifies the size in meters.

The spacing field defines the distance between two adjacent lines of text as the product of size and spacing. Special fonts provided in a font data stream can be accessed using the following syntax:

“OD:<odid>;FSID:<fsid>”, where:

— <odid> is the numeric value of the objectDescriptorID of the associated font data stream,

— <fsid> is the numeric value of the requested font subset as conveyed by fontSubsetID within the associated font data stream.

5.1.1.3.2 Text

5.1.1.3.2.1 XSD description

```
<complexType name="TextType">
 <all>
    <element ref="xmta:IS" minOccurs="0"/>
    <element name="fontStyle" form="qualified" minOccurs="0">
      <complexType>
         <group ref="xmta:SFFontStyleNodeType" minOccurs="0"/>
      </complexType>
    </element>
</all>
<attribute name="string" type="xmta:MFString" use="optional"/>
<attribute name="length" type="xmta:MFFloat" use="optional"/>
<attribute name="maxExtent" type="xmta:SFFloat" use="optional" default="0"/>
```

```
    <attributeGroup ref="xmta:DefUseGroup"/>
  </complexType>
  <element name="Text" type="xmta:TextType"/>
```

5.1.1.3.2.2 Functionality and semantics

As defined in ISO/IEC 14772-1:1997, 6.47.

The Text node specifies a two-sided, flat text string object positioned in the Z=0 plane of the local coordinate system based on values defined in the fontStyle field (see ISO/IEC 14772-1:1997, 6.20, FontStyle). Text nodes may contain multiple text strings specified using the UTF-8 encoding as specified by ISO 10646-1 (see 2.[UTF8]). The text strings are stored in the order in which the text mode characters are to be produced as defined by the parameters in the FontStyle node.

The text strings are contained in the string field. The fontStyle field contains one FontStyle node that specifies the font size, font family and style, direction of the text strings, and any specific language rendering techniques used for the text.

The maxExtent field limits and compresses all of the text strings if the length of the maximum string is longer than the maximum extent, as measured in the local coordinate system. If the text string with the maximum length is shorter than the maxExtent, then there is no compressing. The maximum extent is measured horizontally for horizontal text (FontStyle node: horizontal=TRUE) and vertically for vertical text (FontStyle node: horizontal=FALSE). The maxExtent field shall be greater than or equal to zero.

The length field contains an MFFloat value that specifies the length of each text string in the local coordinate system. If the string is too short, it is stretched (either by scaling the text or by adding space between the characters). If the string is too long, it is compressed (either by scaling the text or by subtracting space between the characters). If a length value is missing (for example, if there are four strings but only three length values), the missing values are considered to be 0. The length field shall be greater than or equal to zero.

Specifying a value of 0 for both the maxExtent and length fields indicates that the string may be any length.

— **ISO 10646-1 Character encodings**

Characters in ISO 10646 (see 2.[UTF8]) are encoded in multiple octets. Code space is divided into four units, as follows:

```
+-------------+-------------+-------------+-------------+
| Group-octet | Plane-octet | Row-octet | Cell-octet |
+-------------+-------------+-------------+-------------+
```

ISO 10646-1 allows two basic forms for characters:

a) UCS-2 (Universal Coded Character Set-2). This form is also known as the Basic Multilingual Plane (BMP). Characters are encoded in the lower two octets (row and cell).

b) UCS-4 (Universal Coded Character Set-4). Characters are encoded in the full four octets.

In addition, three transformation formats (UCS Transformation Format or UTF) are accepted: UTF-7, UTF-8, and UTF-16. Each represents the nature of the transformation: 7-bit, 8-bit, or 16-bit. UTF-7 and UTF-16 are referenced in 2.[UTF8].

UTF-8 maintains transparency for all ASCII code values (0...127). It allows ASCII text (0x0..0x7F) to appear without any changes and encodes all characters from 0x80.. 0x7FFFFFFF into a series of six or fewer bytes.

If the most significant bit of the first character is 0, the remaining seven bits are interpreted as an ASCII character. Otherwise, the number of leading 1 bits indicates the number of bytes following. There is always a zero bit between the count bits and any data.

The first byte is one of the following. The X indicates bits available to encode the character:

```
0XXXXXXX only one byte   0..0x7F (ASCII)
110XXXXX two bytes       Maximum character value is 0x7FF
```



```
1110XXXX three bytes     Maximum character value is 0xFFFF
11110XXX four bytes      Maximum character value is 0x1FFFFF
111110XX five bytes     Maximum character value is 0x3FFFFFF
1111110X six bytes       Maximum character value is 0x7FFFFFFF
```

All following bytes have the format 10XXXXXX.

As a two byte example, the symbol for a register trade mark is ® or 174 in ISO Latin-1 (see ISO/IEC 8859-1). It is encoded as 0x00AE in UCS-2 of ISO 10646. In UTF-8, it has the following two byte encoding: 0xC2, 0xAE.

— **Appearance**

Textures are applied to text as follows. The texture origin is at the origin of the first string, as determined by the justification. The texture is scaled equally in both S and T dimensions, with the font height representing 1 unit. S increases to the right, and T increases up.

ISO/IEC 14772-1:1997, 4.14 has details on VRML lighting equations and how Appearance, Material and textures interact with lighting.

The Text node does not participate in collision detection.

5.1.1.4 Graphics

The following graphics related nodes are used in ARAF: Appearance, Color, LineProperties, LinearGradient, Material, Material2D, Rectangle, Shape, SBVCAnimationV2, SBBone, SBSegment, SBSite, SBSkinnedModel, MorphShape, Coordinate, TextureCoordinate, Normal, IndexedFaceSet, IndexedLineSet.

5.1.1.4.1 Appearance

5.1.1.4.1.1 XSD Description

```
<complexType name="AppearanceType">
 <all>
    <element ref="xmta:IS" minOccurs="0"/>
    <element name="material" form="qualified" minOccurs="0">
       <complexType>
          <group ref="xmta:SFMaterialNodeType" minOccurs="0"/>
       </complexType>
    </element>
    <element name="texture" form="qualified" minOccurs="0">
      <complexType>
          <group ref="xmta:SFTextureNodeType" minOccurs="0"/>
      </complexType>
    </element>
    <element name="textureTransform" form="qualified" minOccurs="0">
       <complexType>
            <group ref="xmta:SFTextureTransformNodeType" minOccurs="0"/>
      </complexType>
    </element>
  </all>
  <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="Appearance" type="xmta:AppearanceType"/>
```

5.1.1.4.1.2 Functionality and semantics

As defined in ISO/IEC 14772-1:1997, 6.3.

The Appearance node specifies the visual properties of geometry. The value for each of the fields in this node may be NULL. However, if the field is non-NULL, it shall contain one node of the appropriate type.

The material field, if specified, shall contain a Material node. If the material field is NULL or unspecified, lighting is off (all lights are ignored during rendering of the object that references this Appearance) and the unlit object colour is (1, 1, 1). Details of the VRML lighting model are in ISO/IEC 14772-1:1997, 4.14.

The texture field, if specified, shall contain one of the various types of texture nodes (ImageTexture, MovieTexture, or PixelTexture). If the texture node is NULL or the texture field is unspecified, the object that references this Appearance is not textured.

The textureTransform field, if specified, shall contain a TextureTransform node. If the textureTransform is NULL or unspecified, the textureTransform field has no effect.

Additional specification: ISO/IEC 14496-11:2015, 7.2.2.6.2.

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, Text.

Inside a Shape node in a 2D context, if no Appearance and therefore no Material2D is defined, the default values and behavior of the Material2D node shall be used. In a 3D context, the default behavior is specified in ISO/IEC 14772-1 (the object is unlit and has color 1 1 1).

5.1.1.4.2 Color

5.1.1.4.2.1 XSD description

```
<complexType name="ColorType">
 <all>
   <element ref="xmta:IS" minOccurs="0"/>
 </all>
 <attribute name="color" type="xmta:MFColor" use="optional"/>
 <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="Color" type="xmta:ColorType"/>
```

5.1.1.4.2.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.9.

This node defines a set of RGB colours to be used in the fields of another node.

Color nodes are only used to specify multiple colours for a single geometric shape, such as colours for the faces or vertices of an IndexedFaceSet. A Material node is used to specify the overall material parameters of lit geometry. If both a Material node and a Color node are specified for a geometric shape, the colours shall replace the diffuse component of the material.

RGB or RGBA textures take precedence over colours; specifying both an RGB or RGBA texture and a Color node for geometric shape will result in the Color node being ignored.

5.1.1.4.3 LineProperties

5.1.1.4.3.1 XSD description

```
<complexType name="LinePropertiesType">
 <all>
    <element ref="xmta:IS" minOccurs="0"/>
 </all>
 <attribute name="lineColor" type="xmta:SFColor" use="optional" default="0 0 0"/>
 <attribute name="lineStyle" type="xmta:SFInt32" use="optional" default="0"/>
 <attribute name="width" type="xmta:SFFloat" use="optional" default="1"/>
```



```
    <attributeGroup ref="xmta:DefUseGroup"/>
  </complexType>
  <element name="LineProperties" type="xmta:LinePropertiesType"/>
```

5.1.1.4.3.2 Functionality and semantics

As defined in ISO/IEC 14496-11:2015, 7.2.2.75.2.

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 described in Table 2.

Table 2 — LineProperties: line styles

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 width is not subject to 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.

5.1.1.4.4 LinearGradient

5.1.1.4.4.1 XSD description

```
  <complexType name="LinearGradientType">
   <all>
     <element ref="xmta:IS" minOccurs="0"/>
     <element name="transform" form="qualified" minOccurs="0">
       <complexType>
          <group ref="xmta:SF3DNodeType" minOccurs="0"/>
       </complexType>
     </element>
   </all>
   <attribute name="endPoint" type="xmta:SFVec2f" use="optional" default="1 0"/>
   <attribute name="key" type="xmta:MFFloat" use="optional"/>
   <attribute name="keyValue" type="xmta:MFColor" use="optional"/>
   <attribute name="opacity" type="xmta:MFFloat" use="optional" default="1"/>
   <attribute name="spreadMethod" type="xmta:SFInt32" use="optional" default="0"/>
   <attribute name="startPoint" type="xmta:SFVec2f" use="optional" default="0 0"/>
   <attributeGroup ref="xmta:DefUseGroup"/>
  </complexType>
  <element name="LinearGradient" type="xmta:LinearGradientType"/>
```

5.1.1.4.4.2 Functionality and semantics

As defined in ISO/IEC 14496-11:2015, 7.2.2.76.2.

The LinearGradient node is a texture ode that generates a texture procedurally. startPoint and endPoint define the gradient vector, in percentage of the bounds of the object.

Key represents a location along the gradient vector, expressed in percentage of its length. At each location, an RGB color is specified in keyValue. Opacity for each color value can be specified. By default,

colors are 100 % opaque. One value of opacity can be specified meaning all color values have the same opacity, else an opacity shall be specified for each color value.

Transform is an optional parameter that defines how the coordinate system of the gradient can be transformed from the gradient coordinate system onto the target coordinate system. By default, the gradient coordinate system is the same as the object it is applied to. This allows effects such as skewing the gradient.

Only a 2D Transformation node (e.g. Transform2D, TransformMatrix2D) can be present here.

spreadMethod can be pad (0), reflect (1), or repeat (2). It indicates what happens if the gradient starts or ends inside the bounds of the object. Pad means that the last color is used, reflect says to reflect the gradient pattern start-to-end, end-to-start, … repeatedly until the target object is filled, and repeat says to repeat the gradient pattern start-to-end, start-to-end, … until the target object is filled.

Opacity for each color value can be specified. By default, colors are 100 % opaque. One value of opacity can be specified, meaning all color values have the same opacity, else an opacity shall be specified for each color value.

5.1.1.4.5 Material

5.1.1.4.5.1 XSD description

```
        <complexType name="MaterialType">
         <all>
           <element ref="xmta:IS" minOccurs="0"/>
         </all>
         <attribute name="ambientIntensity" type="xmta:SFFloat" use="optional"
default="0.2"/>
         <attribute name="diffuseColor" type="xmta:SFColor" use="optional" default="0.8
0.8 0.8"/>
         <attribute name="emissiveColor" type="xmta:SFColor" use="optional" default="0 0
0"/>
         <attribute name="shininess" type="xmta:SFFloat" use="optional" default="0.2"/>
         <attribute name="specularColor" type="xmta:SFColor" use="optional" default="0 0
0"/>
         <attribute name="transparency" type="xmta:SFFloat" use="optional" default="0"/>
         <attributeGroup ref="xmta:DefUseGroup"/>
       </complexType>
       <element name="Material" type="xmta:MaterialType"/>
```

5.1.1.4.5.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.27.

The Material node specifies surface material properties for associated geometry nodes and is used by the VRML lighting equations during rendering. All of the fields in the Material node range from 0.0 to 1.0.

The fields in the Material node determine how light reflects off an object to create colour.

a) The *ambientIntensity* field specifies how much ambient light from light sources this surface shall reflect. Ambient light is omnidirectional and depends only on the number of light sources, not their positions with respect to the surface. Ambient colour is calculated as *ambientIntensity* × *diffuseColor*.

b) The *diffuseColor* field reflects all VRML light sources depending on the angle of the surface with respect to the light source. The more directly the surface faces the light, the more diffuse light reflects.

c) The *emissiveColor* field models "glowing" objects. This can be useful for displaying pre-lit models (where the light energy of the room is computed explicitly), or for displaying scientific data.

d) The *specularColor* and *shininess* fields determine the specular highlights (e.g. the shiny spots on an apple). When the angle from the light to the surface is close to the angle from the surface to the

viewer, the *specularColor* is added to the diffuse and ambient colour calculations. Lower shininess values produce soft glows, while higher values result in sharper, smaller highlights.

e) The *transparency* field specifies how "clear" an object is, with 1.0 being completely transparent, and 0.0 completely opaque.

5.1.1.4.6 Material2D

5.1.1.4.6.1 XSD description

```
<complexType name="Material2DType">
 <all>
    <element ref="xmta:IS" minOccurs="0"/>
    <element name="lineProps" form="qualified" minOccurs="0">
       <complexType>
         <group ref="xmta:SFLinePropertiesNodeType" minOccurs="0"/>
       </complexType>
    </element>
 </all>
 <attribute name="emissiveColor" type="xmta:SFColor" use="optional" default="0.8
0.8 0.8"/>
 <attribute name="filled" type="xmta:SFBool" use="optional" default="false"/>
 <attribute name="transparency" type="xmta:SFFloat" use="optional" default="0"/>
 <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="Material2D" type="xmta:Material2DType"/>
```

5.1.1.4.6.2 Functionality and semantics

As defined in ISO/IEC 14496-11:2015, 7.2.2.80.2.

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 ISO/IEC 14496-11:2015, 7.2.2.6.2). The **emissiveColor** field specifies the color of the 2D **Shape**. If the shape is not filled, the interior is not drawn. The **filled** field specifies whether rendered nodes are filled or drawn using lines. This field affects **IndexedFaceSet2D**, **Circle** and **Rectangle** nodes. If the rendered node is not filled, the line shall be drawn centered on the rendered node outline. That means that half the line will fall inside the rendered node and the other half outside.

The **lineProps** field contains information about line rendering in the form of a **LineProperties** node. When **filled** is true, if **lineProps** is null, no outline is drawn; if **lineProps** is non-null, an outline is drawn using **lineProps** information. When **filled** is false and **lineProps** is null, an outline is drawn with default width (1), default style (solid) and as line color the emissive color of the Material2D. When **filled** is false and **lineProps** is defined, line color, width and style are taken from the **lineProps** node. See ISO/IEC 14496-11:2015, 7.2.2.75 for more information on **LineProperties**.

The **transparency** field specifies the transparency of the 2D **Shape** and applies both to the filled interior as well as to the outline when drawn. The part of the line which lies outside of the geometry shall not be sensitive to pointer activity. When mapping texture onto a geometry and an outline is to be drawn, the texture shall first mapped onto the geometry, where the geometry dimensions are those without an outline. Then after the geometry is textured the outline shall be drawn.

5.1.1.4.7 Rectangle

5.1.1.4.7.1 XSD description

```
<complexType name="RectangleType">
<all>
   <element ref="xmta:IS" minOccurs="0"/>
</all>
<attribute name="size" type="xmta:SFVec2f" use="optional" default="2 2"/>
<attributeGroup ref="xmta:DefUseGroup"/>
```

```
</complexType>
<element name="Rectangle" type="xmta:RectangleType"/>
```

5.1.1.4.7.2 Functionality and semantics

As defined in ISO/IEC 14496-11:2015, 7.2.2.110.2.

This node specifies a rectangle centered at (0.0) in the local coordinate system. The **size** field specifies the horizontal and vertical size of the rendered rectangle.

5.1.1.4.8 Shape

5.1.1.4.8.1 XSD description

```
<complexType name="ShapeType">
<all>
  <element ref="xmta:IS" minOccurs="0"/>
  <element name="appearance" form="qualified" minOccurs="0">
     <complexType>
       <group ref="xmta:SFAppearanceNodeType" minOccurs="0"/>
     </complexType>
  </element>
  <element name="geometry" form="qualified" minOccurs="0">
     <complexType>
       <group ref="xmta:SFGeometryNodeType" minOccurs="0"/>
     </complexType>
  </element>
 </all>
 <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="Shape" type="xmta:ShapeType"/>
```

5.1.1.4.8.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.41.

The Shape node has two fields, *appearance* and *geometry,* which are used to create rendered objects in the world. The *appearance* field contains an Appearance node that specifies the visual attributes (e.g. material and texture) to be applied to the geometry. The *geometry* field contains a geometry node. The specified geometry node is rendered with the specified appearance nodes applied.

ISO/IEC 14772-1:1997, 4.14 contains details of the VRML lighting model and the interaction between Appearance nodes and geometry nodes.

If the *geometry* field is NULL, the object is not drawn.

5.1.1.4.9 SBVCAnimationV2

5.1.1.4.9.1 XSD description

```
<complexType name="SBVCAnimationV2Type">
 <all>
    <element ref="xmta:IS" minOccurs="0"/>
    <element name="virtualCharacters" form="qualified" minOccurs="0">
       <complexType>
          <group ref="xmta:SF3DNodeType" minOccurs="0" maxOccurs="unbounded"/>
       </complexType>
    </element>
 </all>
 <attribute name="activeUrlIndex" type="xmta:MFInt32" use="optional"/>
 <attribute name="loop" type="xmta:SFBool" use="optional" default="false"/>
 <attribute name="speed" type="xmta:SFFloat" use="optional" default="1"/>
 <attribute name="startTime" type="xmta:SFTime" use="optional" default="0"/>
 <attribute name="stopTime" type="xmta:SFTime" use="optional" default="0"/>
 <attribute name="transitionTime" type="xmta:SFFloat" use="optional" default="0"/>
 <attribute name="url" type="xmta:MFUrl" use="optional"/>
```

```
    <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="SBVCAnimationV2" type="xmta:SBVCAnimationV2Type"/>
```

5.1.1.4.9.2 Functionality and semantics

As specified in ISO/IEC 14496-16:2011, 4.7.1.7.

This node is an extension of the SBVCAnimation node and the added functionality consists in streaming control and animation data collection. The BBA stream can be controlled as a elementary media stream, and can be used in connection with the MediaControl node.

The **virtualCharacters** field specifies a list of SBSkinnedModel nodes. The length of the list can be 1 or greater.

The **url** field refers to the BBA stream which contains encoded animation data related to the SBSkinnedModel nodes from the virtualCharacters list and is used for outband bitstreams. The animation will be extracted from the first element of the animation URL list and if the case when it is not available the following element will be used.

The **loop**, **startTime**, and **stopTime** exposedFields and the **isActive** eventOut, and their effects on the SBVCAnimationV2 node, are similar with the ones described by VRML specifications (ISO/IEC 14772-1) for AudioClip, MovieTexture, and TimeSensor nodes and are described as follows.

The values of the exposedFields are used to determine when the node becomes active or inactive.

The SBVCAnimationV2 node can execute for 0 or more cycles. A cycle is defined by field data within the node. If, at the end of a cycle, the value of **loop** is FALSE, execution is terminated. Conversely, if **loop** is TRUE at the end of a cycle, a time-dependent node continues execution into the next cycle. A time-dependent node with **loop** TRUE at the end of every cycle continues cycling forever if **startTime** >= **stopTime**, or until **stopTime** if **startTime** < **stopTime**.

The SBVCAnimationV2 node generates an **isActive** TRUE event when it becomes active and generates an **isActive** FALSE event when it becomes inactive. These are the only times at which an **isActive** event is generated. In particular, **isActive** events are not sent at each tick of a simulation.

The SBVCAnimationV2 node is inactive until its **startTime** is reached. When time *now* becomes greater than or equal to **startTime**, an **isActive** TRUE event is generated and the SBVCAnimationV2 node becomes active (*now* refers to the time at which the player is simulating and displaying the virtual world). When a SBVCAnimationV2 node is read from a mp4 file and the ROUTEs specified within the mp4 file have been established, the node should determine if it is active and, if so, generate an **isActive** TRUE event and begin generating any other necessary events. However, if a SBVCAnimationV2 node would have become inactive at any time before the reading of the mp4 file, no events are generated upon the completion of the read.

An active SBVCAnimationV2 node will become inactive when **stopTime** is reached if **stopTime** > **startTime**. The value of **stopTime** is ignored if **stopTime** <= **startTime**. Also, an active SBVCAnimationV2 node will become inactive at the end of the current cycle if **loop** is FALSE. If an active SBVCAnimationV2 node receives a *set_loop* FALSE event, execution continues until the end of the current cycle or until **stopTime** (if **stopTime** > **startTime**), whichever occurs first. The termination at the end of cycle can be overridden by a subsequent *set_loop* TRUE event.

Any *set_startTime* events to an active SBVCAnimationV2 node are ignored. Any *set_stopTime* event where **stopTime** <= **startTime** sent to an active SBVCAnimationV2 node is also ignored. A *set_stopTime* event where **startTime** < **stopTime** <= *now* sent to an active SBVCAnimationV2 node results in events being generated as if **stopTime** has just been reached. That is, final events, including an **isActive** FALSE, are generated and the node becomes inactive. The **stopTime**_*changed* event will have the *set_stopTime* value.

A SBVCAnimationV2 node may be restarted while it is active by sending a *set_stopTime* event equal to the current time (which will cause the node to become inactive) and a *set_startTime* event, setting it to the current time or any time in the future. These events will have the same time stamp and should be processed as *set_stopTime,* then *set_startTime* to produce the correct behaviour.

The **speed** exposedField controls playback speed. It does not affect the delivery of the stream attached to the **SBVCAnimationV2** node. For streaming media, value of **speed** other than 1 shall be ignored.

A **SBVCAnimationV2** shall display first frame if **speed** is 0. For positive values of **speed**, the frame that an active **SBVCAnimationV2** will display at time *now* corresponds to the frame at animation time (i.e. in the animation's local time base with frame 0 at time 0, at speed = 1):

fmod (now - **startTime**, duration/**speed**)

If **speed** is negative, then the frame to display is the frame at animation time:

duration + fmod(now - **startTime**, duration/**speed**).

When a **SBVCAnimationV2** becomes inactive, the frame corresponding to the time at which the **SBVCAnimationV2** became inactive shall persist. The **speed** exposedField indicates how fast the movie shall be played. A speed of 2 indicates the animation 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 animation is playing.

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.

activeUrlIndex allows to select or to combine specific animation resource referred in the url[] field. When this field is instantiated, the behavior of the url[] field changes from the alternative selection into a combined selection. In the case of alternative mode, if the first resource in the **url[]** field is not available, the second one will be used, and so on. In the combined mode, the following cases can occur.

a) **activeUrlIndex** has one field: the resource from **url[]** that has this index is used for animation. When the **activeUrlIndex** is updated a transition between to the old animation (frame) and the new one is performed. The transition use linear interpolation for translation, center and scale and SLERP for spherical data as rotation and scaleOrientation. The time of transition is specified by using the transitionTime field.

b) activeUrlIndex has several fields: a composition between the two resources is performed by the terminal: for the bones that are common in two or more resources a mean procedure has to be applied. The mean is computed by using linear interpolation for translation, center and scale and SLERP for spherical data as rotation and scaleOrientation.

In all the cases, when a transition between two animation resources is needed, when the transitionTime is not zero, a interpolation shall be performed by the player. The transitionTime is specified in milliseconds.

5.1.1.4.10 SBBone

5.1.1.4.10.1 XSD Description

```
<complexType name="SBBoneType">
  <all>
    <element ref="xmta:IS" minOccurs="0"/>
    <element name="children" form="qualified" minOccurs="0">
      <complexType>
        <group ref="xmta:SF3DNodeType" minOccurs="0" maxOccurs="unbounded"/>
      </complexType>
    </element>
</all>
<attribute name="boneID" type="xmta:SFInt32" use="optional" default="0"/>
<attribute name="center" type="xmta:SFVec3f" use="optional" default="0 0 0"/>
<attribute name="endpoint" type="xmta:SFVec3f" use="optional" default="0 0 1"/>
<attribute name="falloff" type="xmta:SFInt32" use="optional" default="1"/>
<attribute name="ikChainPosition" type="xmta:SFInt32" use="optional" default="0"/>
<attribute name="ikPitchLimit" type="xmta:MFFloat" use="optional"/>
<attribute name="ikRollLimit" type="xmta:MFFloat" use="optional"/>
```

```
        <attribute name="ikTxLimit" type="xmta:MFFloat" use="optional"/>
        <attribute name="ikTyLimit" type="xmta:MFFloat" use="optional"/>
        <attribute name="ikTzLimit" type="xmta:MFFloat" use="optional"/>
        <attribute name="ikYawLimit" type="xmta:MFFloat" use="optional"/>
        <attribute name="rotation" type="xmta:SFRotation" use="optional" default="0 0 1
0"/>
        <attribute name="rotationOrder" type="xmta:SFInt32" use="optional" default="0"/>
        <attribute name="scale" type="xmta:SFVec3f" use="optional" default="1 1 1"/>
        <attribute name="scaleOrientation" type="xmta:SFRotation" use="optional"
default="0 0 1 0"/>
        <attribute name="sectionInner" type="xmta:MFFloat" use="optional"/>
        <attribute name="sectionOuter" type="xmta:MFFloat" use="optional"/>
        <attribute name="sectionPosition" type="xmta:MFFloat" use="optional"/>
        <attribute name="skinCoordIndex" type="xmta:MFInt32" use="optional"/>
        <attribute name="skinCoordWeight" type="xmta:MFFloat" use="optional"/>
        <attribute name="translation" type="xmta:SFVec3f" use="optional" default="0 0 0"/>
        <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="SBBone" type="xmta:SBBoneType"/>
```

5.1.1.4.10.2 Functionality and semantics

As specified in ISO/IEC 14496-1:2002, 6.8.1.1.2.

SBBone node specifies data related to one bone from the skeleton.

The **boneID** field is a unique identifier which allows that the bone to be addressed at animation run-time.

The **center** field specifies a translation offset from the origin of the local coordinate system.

The **translation** field specifies a translation to the bone coordinate system.

The **rotation** field specifies a rotation of the bone coordinate system.

The **scale** field specifies a non-uniform scale of the bone coordinate system. **scale** values shall be greater than zero.

The **scaleOrientation** specifies a rotation of the bone coordinate system before the scale (to specify scales in arbitrary orientations). The **scaleOrientation** applies only to the scale operation.

The possible geometric 3D transformation consists of (in order): 1) (possibly) non-uniform scale about an arbitrary point, 2) a rotation about an arbitrary point and axis and 3) a translation.

The **rotationOrder** field specifies the rotation order when deals with the decomposition of the rotation in respect with system coordinate axes.

Two ways of specifying the influence region of the bone are allowed:

The **skinCoordIndex** field contains a list of indices of all skin vertices affected by the current bone. Mostly, the skin influence region of bone will contain vertices from the 3D neighborhood of the bone, but special cases of influence are also accepted.

The **skinCoordWeight** field contains a list of weights (one per vertex listed in **skinCoordIndex**) that measures the contribution of the current bone to the vertex in question. The length **skinCoordIndex** is equal with the length of **skinCoordWeight**. The sum of all **skinCoordWeight** related to the same vertex shall be 1.

The **endpoint** field specifies the bone 3D end point and is used to compute the bone length.

The **sectionInner** field is a list of inner influence region radii for different sections.

The **sectionOuter** field is a list of outer influence region radii for different sections.

The **sectionPosition** field is a list of positions of all the sections defined by the designer.

The **falloff** field specifies the function between the amplitude affectedness and distance: –1 for x^3, 0 for x^2, 1 for x, 2 for sinx, 3 for $x^{1/2}$ and 4 for $x^{1/3}$.

The two schemes can be used independently or in combination, in which case the individual vertex weights take precedence.

The **ikChainPosition** field specifies the position of the bone in the kinematics chain. If the bone is the root of the IK chain then **ikChainPosition**=1. In this case, when applying IK scheme, only the orientation of the bone is changed. If the bone is last in the kinematics chain **ikChainPosition**=2. In this case, the animation stream has to include the desired position of the bone (X, Y and Z world coordinates). If **ikChainPosition**=3 the bone belongs to the IK chain but is not the first or the last one in the chain. In this case, position and orientation of the bone are computed by the IK procedure. Finally, if the bone does not belong to any IK chain (**ikChainPosition**=0), it is necessary to transmit the bone local transformation in order to animate the bone. If an animation stream contains motion information about a bone which has **ikChainPosition** 1, this information will be ignored. If an animation stream contains motion information about a bone which has **ikChainPosition** 3, this means that the animation producer wants to ensure the orientation of the bone and the IK solver will use this value as a constrain.

The **ikYawLimit** field consists in a pair of min/max values which limit the bone rotation with respect to the X axis.

The **ikPitchLimit** field consists in a pair of min/max values which limit the bone rotation with respect to the Y axis.

The **ikRollLimit** field consists in a pair of min/max values which limit the bone rotation with respect to the Z axis.

The **ikTxLimit** field consists in a pair of min/max values which limit the bone translation in the X direction.

The **ikTyLimit** field consists in a pair of min/max values which limit the bone translation in the Y direction.

The **ikTzLimit** field consists in a pair of min/max values which limit the bone translation in the Z direction.

The **SBBone** node is used as a building block to describe the hierarchy of the articulated model by attaching one or more child objects. The **children** field has the same semantic as used in BIFS; the absolute geometric transformation of any child of a bone is obtained through a composition with the bone-parent transformation.

5.1.1.4.11 SBSegment

5.1.1.4.11.1 XSD description

```
        <complexType name="SBSegmentType">
          <all>
            <element ref="xmta:IS" minOccurs="0"/>
            <element name="children" form="qualified" minOccurs="0">
               <complexType>
                 <group ref="xmta:SF3DNodeType" minOccurs="0" maxOccurs="unbounded"/>
               </complexType>
            </element>
          </all>
          <attribute name="centerOfMass" type="xmta:SFVec3f" use="optional" default="0 0
0"/>
          <attribute name="mass" type="xmta:SFFloat" use="optional" default="0"/>
          <attribute name="momentsOfInertia" type="xmta:MFVec3f" use="optional" default="0
0 0 0 0 0 0 0 0"/>
          <attribute name="name" type="xmta:SFString" use="optional"
default="&amp;quot;&amp;quot;"/>
          <attributeGroup ref="xmta:DefUseGroup"/>
```

```
    </complexType>
    <element name="SBSegment" type="xmta:SBSegmentType"/>
```

5.1.1.4.11.2 Functionality and semantics

As specified in ISO/IEC 14496-1:2002, 6.8.1.2.2.

The **name** field shall be present, so that the **SBSegment** can be identified at runtime. Each **SBSegment** should have a DEF name that matches the **name** field for that Segment, but with a distinguishing prefix in front of it.

The **mass** field is the total mass of the segment.

The **centerOfMass** field is the location within the segment of its center of mass. Note that a zero value was chosen for the mass in order to give a clear indication that no mass value is available.

The **momentsOfInertia** field contains the moment of inertia matrix. The first three elements are the first row of the 3x3 matrix, the next three elements are the second row, and the final three elements are the third row.

The **children** field can be any object attached at this level of the skeleton, including a **SBSkinnedModel**.

An **SBSegment** node is a grouping node especially introduced to address two issues.

a) The first one is to the requirement to separate different parts from the skinned model into deformation-independent parts. Between two deformation-independent parts the geometrical transformation of one of them do not imply skin deformations on the other. This is essential for run-time animation optimization. The **SBSegment** node may contain as a child an **SBSkinnedModel** node (see the **SBSkinnedModel** node description below). Portions of the model which are not part of the seamless mesh can be attached to the skeleton hierarchy by using an **SBSegment** node.

b) The second deals with the requirement to attach standalone 3D objects at different parts of the skeleton hierarchy. For example, a ring can be attached to a finger; the ring geometry and attributes are defined outside of skinned model but the ring will have the same local geometrical transformation as the attached bone.

5.1.1.4.12 SBSite

5.1.1.4.12.1 XSD description

```
        <complexType name="SBSiteType">
          <all>
             <element ref="xmta:IS" minOccurs="0"/>
             <element name="children" form="qualified" minOccurs="0">
                <complexType>
                   <group ref="xmta:SF3DNodeType" minOccurs="0" maxOccurs="unbounded"/>
                </complexType>
             </element>
          </all>
          <attribute name="center" type="xmta:SFVec3f" use="optional" default="0 0 0"/>
          <attribute name="name" type="xmta:SFString" use="optional"
default="&amp;quot;&amp;quot;"/>
          <attribute name="rotation" type="xmta:SFRotation" use="optional" default="0 0 1
0"/>
          <attribute name="scale" type="xmta:SFVec3f" use="optional" default="1 1 1"/>
          <attribute name="scaleOrientation" type="xmta:SFRotation" use="optional"
default="0 0 1 0"/>
          <attribute name="translation" type="xmta:SFVec3f" use="optional" default="0 0
0"/>
          <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="SBSite" type="xmta:SBSiteType"/>
```

5.1.1.4.12.2 Functionality and semantics

As specified in ISO/IEC 14496-1:2002, 6.8.1.3.2.

The **center** field specifies a translation offset and can be used to compute a bone length. The **rotation** field specifies a rotation of the coordinate system of the **SBSite** node.

The **scale** field specifies a non-uniform scale of the **SBSite** node coordinate system and the **scale** values shall be greater than zero.

The **scaleOrientation** specifies a rotation of the coordinate system of the **SBSite** node before the scale, thus allowing a scale at an arbitrary orientation. The **scaleOrientation** applies only to the scale operation.

The **translation** field specifies a translation of the coordinate system of the **SBSite** node.

The **children** field is used to store any object that can be attached to the **SBSegment** node.

The **SBSite** node can be used for three purposes. The first is to define an "end effector", i.e. a location which can be used by an inverse kinematics system. The second is to define an attachment point for accessories such as clothing. The third is to define a location for a virtual camera in the reference frame of a **SBSegment** node.

SBSite nodes are stored within the **children** field of an **SBSegment** node. The **SBSite** node is a specialized grouping node that defines a coordinate system for nodes in its **children** field that is relative to the coordinate systems of its parent node. The reason a **SBSite** node is considered a specialized grouping node is that it can only be defined as a child of a **SBSegment** node.

5.1.1.4.13 SBSkinnedModel

5.1.1.4.13.1 XSD description

```
          <complexType name="SBSkinnedModelType">
           <all>
             <element ref="xmta:IS" minOccurs="0"/>
             <element name="bones" form="qualified" minOccurs="0">
               <complexType>
                  <group ref="xmta:SFSBBoneNodeType" minOccurs="0" maxOccurs="unbounded"/>
               </complexType>
             </element>
             <element name="muscles" form="qualified" minOccurs="0">
               <complexType>
                  <group ref="xmta:SFSBMuscleNodeType" minOccurs="0"
maxOccurs="unbounded"/>
               </complexType>
             </element>
             <element name="segments" form="qualified" minOccurs="0">
               <complexType>
                  <group ref="xmta:SFSBSegmentNodeType" minOccurs="0"
maxOccurs="unbounded"/>
               </complexType>
             </element>
             <element name="sites" form="qualified" minOccurs="0">
                <complexType>
                   <group ref="xmta:SFSBSiteNodeType" minOccurs="0" maxOccurs="unbounded"/>
                </complexType>
             </element>
             <element name="skeleton" form="qualified" minOccurs="0">
                <complexType>
                   <group ref="xmta:SF3DNodeType" minOccurs="0" maxOccurs="unbounded"/>
                </complexType>
             </element>
             <element name="skin" form="qualified" minOccurs="0">
                <complexType>
                   <group ref="xmta:SF3DNodeType" minOccurs="0" maxOccurs="unbounded"/>
                </complexType>
```

```
            </element>
            <element name="skinCoord" form="qualified" minOccurs="0">
               <complexType>
                 <group ref="xmta:SFCoordinateNodeType" minOccurs="0"/>
               </complexType>
            </element>
            <element name="skinNormal" form="qualified" minOccurs="0">
               <complexType>
                 <group ref="xmta:SFNormalNodeType" minOccurs="0"/>
               </complexType>
            </element>
            <element name="weighsComputationSkinCoord" form="qualified" minOccurs="0">
               <complexType>
                 <group ref="xmta:SF3DNodeType" minOccurs="0"/>
               </complexType>
            </element>
          </all>
          <attribute name="center" type="xmta:SFVec3f" use="optional" default="0 0 0"/>
          <attribute name="name" type="xmta:SFString" use="optional"
default="&amp;quot;&amp;quot;"/>
          <attribute name="rotation" type="xmta:SFRotation" use="optional" default="0 0 1
0"/>
          <attribute name="scale" type="xmta:SFVec3f" use="optional" default="1 1 1"/>
          <attribute name="scaleOrientation" type="xmta:SFRotation" use="optional"
default="0 0 1 0"/>
          <attribute name="translation" type="xmta:SFVec3f" use="optional" default="0 0
0"/>
          <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="SBSkinnedModel" type="xmta:SBSkinnedModelType"/>
```

5.1.1.4.13.2 Functionality and semantics

As specified in ISO/IEC 14496-16:2011, 4.5.2.1.2.

The **SBSkinnedModel** node is the top of the hierarchy of Skin&Bones related nodes and contains the definition parameters for the entire seamless model or of a seamless part of the model.

The **name** field specify the name of the skinned model allowing easily identification at the animation run-time.

The **center** field specifies a translation offset from the origin of the local coordinate system.

The **translation** field specifies a translation of the coordinate system.

The **rotation** field specifies a rotation of the coordinate system

The **scale** field specifies a non-uniform scale of the coordinate system. **scale** values shall be greater than zero.

The **scaleOrientation** specifies a rotation of the coordinate system before the scale (to specify scales in arbitrary orientations). The **scaleOrientation** applies only to the scale operation.

The **skinCoord** contains the 3d coordinates of all vertices of the seamless model.

The **skin** consists of a collection of shapes that share the same **skinCoord**. This mechanism allows considering the model as a continuous mesh and, in the same time, to attach different attributes (like color, texture) to different parts of the model.

The **skeleton** field specifies the root of the bones hierarchy.

The **bones** fields consist in the lists of all bones previously defined as **SBBone** node.

The **segments** fields consist in the lists of all bones previously defined as **SBSegment** node.

The **sites** fields consist in the lists of all bones previously defined as **SBSites** node. The **muscles** fields consist in the lists of all bones previously defined as **SBMuscle** node.

The **weighsComputationSkinCoord** field describes a specific static position of the skinned model. In many cases the static position of the articulated model defined by **skinCoord** and **skin** fields is not appropriate to compute the influence region of a bone. In this case the **weighsComputationSkinCoord** field allows specifying the skinned model vertices in a more appropriate static posture. This posture will be used just during the initialization stage and ignored during the animation. All the skeleton transformations are related to the posture defined by **skinCoord** field.

5.1.1.4.14 MorphShape

5.1.1.4.14.1 XSD description

```
<complexType name="MorphShapeType">
 <all>
   <element ref="xmta:IS" minOccurs="0"/>
   <element name="baseShape" form="qualified" minOccurs="0">
      <complexType>
         <group ref="xmta:SF3DNodeType" minOccurs="0"/>
      </complexType>
   </element>
   <element name="targetShapes" form="qualified" minOccurs="0">
      <complexType>
         <group ref="xmta:SF3DNodeType" minOccurs="0" maxOccurs="unbounded"/>
      </complexType>
   </element>
 </all>
 <attribute name="morphID" type="xmta:SFInt32" use="optional" default="0"/>
 <attribute name="weights" type="xmta:MFFloat" use="optional"/>
 <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="MorphShape" type="xmta:MorphShapeType"/>
```

5.1.1.4.14.2 Functionality and semantics

As specified in ISO/IEC 14496-16:2011, 4.3.6.

morphID: a unique identifier between 0 and 1023 which allows that the morph to be addressed at animation run-time.

baseShape: a Shape node that represent the base mesh. The geometry field of the baseShape can be any geometry supported by ISOIEC 14496 (e.g. IndexedFaceSet, IndexedLineSet, SolidRep).

targetShapes: a vector of Shapes nodes representing the shape of the target meshes. The tool used for definig an appearance and a geometry of a target shape shall be the same as the tool used for defining the appearance and the geometry of the base shape (e.g. if the baseShape is defined by using IndexedFaceSet, all the target shapes shall be defined by using IndexedFaceSet).

weights: a vector of integers of the same size as the **targetShapes**. The morphed shape is obtained according to Formula (1):

$$M = B + \sum_{i=1}^{n} (T_i - B) * w_i \qquad (1)$$

where

M is the morphed shape;

B is the base shape;

T_i is the target shape i;

W_i is the weight of the T_i.

The morphing is performed for all the components of the Shape (Appearance and Geometry) that have different values in the base shape and the target shapes (e.g. if the base shape and the target shapes are definined by using IndexedFaceSet and the *coord* field contains different values in the base shape and in the target geometries, the *coord* component of the morph shape is obtained by using previous equation applied to the *coord* field. Note that the size of the *coord* field shall be the same for the base shapes and the target shapes).

If the shapes (base and targets) are defined by using IndexedFaceSet, a tipical decoder should support morphing of the following geometry components: *coord*, *normals*, *color*, *texCoord*.

5.1.1.4.15 Coordinate

5.1.1.4.15.1 XSD description

```
    <complexType name="CoordinateType">
     <all>
       <element ref="xmta:IS" minOccurs="0"/>
     </all>
     <attribute name="point" type="xmta:MFVec3f" use="optional"/>
     <attributeGroup ref="xmta:DefUseGroup"/>
  </complexType>
  <element name="Coordinate" type="xmta:CoordinateType"/>
```

5.1.1.4.15.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.12.

This node defines a set of 3D coordinates to be used in the *coord* field of vertex-based geometry nodes including IndexedFaceSet, IndexedLineSet, and PointSet.

5.1.1.4.16 TextureCoordinate

5.1.1.4.16.1 XSD description

```
    <complexType name="TextureCoordinateType">
     <all>
       <element ref="xmta:IS" minOccurs="0"/>
     </all>
     <attribute name="point" type="xmta:MFVec2f" use="optional"/>
     <attributeGroup ref="xmta:DefUseGroup"/>
  </complexType>
  <element name="TextureCoordinate" type="xmta:TextureCoordinateType"/>
```

5.1.1.4.16.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.48.

The TextureCoordinate node specifies a set of 2D texture coordinates used by vertex-based geometry nodes (e.g. IndexedFaceSet and ElevationGrid) to map textures to vertices. Textures are two dimensional colour functions that, given an (*s*, *t*) coordinate, return a colour value *colour*(*s*, *t*) Texture map values (ImageTexture, MovieTexture, and PixelTexture) range from [0.0, 1.0] along the S-axis and T-axis. However, TextureCoordinate values, specified by the *point* field, may be in the range (−∞,∞). Texture coordinates identify a location (and thus a colour value) in the texture map. The horizontal coordinate *s* is specified first, followed by the vertical coordinate *t*.

If the texture map is repeated in a given direction (S-axis or T-axis), a texture coordinate C (*s* or *t*) is mapped into a texture map that has N pixels in the given direction as follows:

Texture map location = [C - floor(C)] × N

If the texture map is not repeated, the texture coordinates are clamped to the 0.0 to 1.0 range as follows:

```
    Texture map location = N,      if C > 1,0,
                         = 0,0,    if C < 0,0,
                         = C × N,  if 0,0 <= C <= 1,0.
```

5.1.1.4.17 Normal

5.1.1.4.17.1 XSD description

```
        <complexType name="NormalType">
         <all>
           <element ref="xmta:IS" minOccurs="0"/>
         </all>
         <attribute name="vector" type="xmta:MFVec3f" use="optional"/>
         <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="Normal" type="xmta:NormalType"/>
```

5.1.1.4.17.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.30.

This node defines a set of 3D surface normal vectors to be used in the *vector* field of some geometry nodes (e.g. IndexedFaceSet and ElevationGrid). This node contains one multiple-valued field that contains the normal vectors. Normals shall be of unit length.

5.1.1.4.18 IndexedFaceSet

5.1.1.4.18.1 XSD description

```
        <complexType name="IndexedFaceSetType">
         <all>
            <element ref="xmta:IS" minOccurs="0"/>
            <element name="color" form="qualified" minOccurs="0">
              <complexType>
                <group ref="xmta:SFColorNodeType" minOccurs="0"/>
              </complexType>
            </element>
            <element name="coord" form="qualified" minOccurs="0">
              <complexType>
                <group ref="xmta:SFCoordinateNodeType" minOccurs="0"/>
              </complexType>
            </element>
            <element name="normal" form="qualified" minOccurs="0">
              <complexType>
                <group ref="xmta:SFNormalNodeType" minOccurs="0"/>
              </complexType>
            </element>
            <element name="texCoord" form="qualified" minOccurs="0">
              <complexType>
                <group ref="xmta:SFTextureCoordinateNodeType" minOccurs="0"/>
              </complexType>
            </element>
         </all>
         <attribute name="ccw" type="xmta:SFBool" use="optional" default="true"/>
         <attribute name="colorIndex" type="xmta:MFInt32" use="optional"/>
         <attribute name="colorPerVertex" type="xmta:SFBool" use="optional"
default="true"/>
         <attribute name="convex" type="xmta:SFBool" use="optional" default="true"/>
         <attribute name="coordIndex" type="xmta:MFInt32" use="optional"/>
         <attribute name="creaseAngle" type="xmta:SFFloat" use="optional" default="0"/>
         <attribute name="normalIndex" type="xmta:MFInt32" use="optional"/>
         <attribute name="normalPerVertex" type="xmta:SFBool" use="optional"
default="true"/>
         <attribute name="solid" type="xmta:SFBool" use="optional" default="true"/>
         <attribute name="texCoordIndex" type="xmta:MFInt32" use="optional"/>
         <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="IndexedFaceSet" type="xmta:IndexedFaceSetType"/>
```

5.1.1.4.18.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.23.

The IndexedFaceSet node represents a 3D shape formed by constructing faces (polygons) from vertices listed in the *coord* field. The coord field contains a Coordinate node that defines the 3D vertices referenced by the *coordIndex* field. IndexedFaceSet uses the indices in its *coordIndex* field to specify the polygonal faces by indexing into the coordinates in the Coordinate node. An index of "–1" indicates that the current face has ended and the next one begins. The last face may be (but does not have to be) followed by a "–1" index. If the greatest index in the *coordIndex* field is N, the Coordinate node shall contain N+1 coordinates (indexed as 0 to N). Each face of the IndexedFaceSet shall have:

a) at least three non-coincident vertices;

b) vertices that define a planar polygon;

c) vertices that define a non-self-intersecting polygon.

Otherwise, the results are undefined.

The IndexedFaceSet node is specified in the local coordinate system and is affected by the transformations of its ancestors.

Descriptions of the *coord*, *normal*, and *texCoord* fields are provided in the Coordinate, Normal, and TextureCoordinate nodes, respectively.

Details on lighting equations and the interaction between *color* field, *normal* field, textures, materials, and geometries are provided in ISO/IEC 14772-1:1997, 4.14.

If the *color* field is not NULL, it shall contain a Color node whose colours are applied to the vertices or faces of the IndexedFaceSet as follows:

— If *colorPerVertex* is FALSE, colours are applied to each face, as follows.

  — If the *colorIndex* field is not empty, then one colour is used for each face of the IndexedFaceSet. There shall be at least as many indices in the *colorIndex* field as there are faces in the IndexedFaceSet. If the greatest index in the *colorIndex* field is N, then there shall be N+1 colours in the Color node. The *colorIndex* field shall not contain any negative entries.

  — If the *colorIndex* field is empty, then the colours in the Color node are applied to each face of the IndexedFaceSet in order. There shall be at least as many colours in the Color node as there are faces.

— If *colorPerVertex* is TRUE, colours are applied to each vertex, as follows.

  — If the *colorIndex* field is not empty, then colours are applied to each vertex of the IndexedFaceSet in exactly the same manner that the *coordIndex* field is used to choose coordinates for each vertex from the Coordinate node. The *colorIndex* field shall contain at least as many indices as the *coordIndex* field, and shall contain end-of-face markers (–1) in exactly the same places as the *coordIndex* field. If the greatest index in the *colorIndex* field is N, then there shall be N+1 colours in the Color node.

  — If the *colorIndex* field is empty, then the *coordIndex* field is used to choose colours from the Color node. If the greatest index in the *coordIndex* field is N, then there shall be N+1 colours in the Color node.

If the *color* field is NULL, the geometry shall be rendered normally using the Material and texture defined in the Appearance node (see ISO/IEC 14772-1:1997, 4.14 for details).

If the *normal* field is not NULL, it shall contain a Normal node whose normals are applied to the vertices or faces of the IndexedFaceSet in a manner exactly equivalent to that described above for applying colours to vertices/faces (where *normalPerVertex* corresponds to *colorPerVertex* and *normalIndex*

corresponds to *colorIndex*). If the *normal* field is NULL, the browser shall automatically generate normals, using *creaseAngle* to determine if and how normals are smoothed across shared vertices (see ISO/IEC 14772-1:1997, 4.6.3.5, Crease angle field).

If the *texCoord* field is not NULL, it shall contain a TextureCoordinate node. The texture coordinates in that node are applied to the vertices of the IndexedFaceSet as follows.

— If the *texCoordIndex* field is not empty, then it is used to choose texture coordinates for each vertex of the IndexedFaceSet in exactly the same manner that the *coordIndex* field is used to choose coordinates for each vertex from the Coordinate node. The *texCoordIndex* field shall contain at least as many indices as the *coordIndex* field, and shall contain end-of-face markers (–1) in exactly the same places as the *coordIndex* field. If the greatest index in the *texCoordIndex* field is N, then there shall be N+1 texture coordinates in the TextureCoordinate node.

— If the *texCoordIndex* field is empty, then the *coordIndex* array is used to choose texture coordinates from the TextureCoordinate node. If the greatest index in the *coordIndex* field is N, then there shall be N+1 texture coordinates in the TextureCoordinate node.

If the *texCoord* field is NULL, a default texture coordinate mapping is calculated using the local coordinate system bounding box of the shape. The longest dimension of the bounding box defines the S coordinates, and the next longest defines the T coordinates. If two or all three dimensions of the bounding box are equal, ties shall be broken by choosing the X, Y, or Z dimension in that order of preference. The value of the S coordinate ranges from 0 to 1, from one end of the bounding box to the other. The T coordinate ranges between 0 and the ratio of the second greatest dimension of the bounding box to the greatest dimension.

Some restrictions specified in ISO/IEC 14496-11:2015, 7.2.2.66.2.

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.

In order to use 3D Mesh Coding (3DMC) with the **IndexedFaceSet** node, the **use3DMeshCoding** flag in BIFSv2Config should be set to TRUE, as described in ISO/IEC 14496-11:2015, 8.5.3.2. This will require every IndexedFaceSet node in that elementary stream to be coded with 3DMC. Note that 3DMC does not support the use of DEF and USE within the fields of **IndexedFaceSet**. Also, an empty **IndexedFaceSet** should not be included in a stream where use3DmeshCoding flag is set to TRUE. A scene with both 3DMC coded and BIFS coded **IndexedFaceSet** nodes can be created by sending the compressed and uncompressed nodes in separate streams.This can be done with an **Inline** node or by sending separate elementary streams in the same object descriptor. The latter approach has the advantage of keeping the nodes in the same name space, see the example in ISO/IEC 14496-11:2015, 7.8.

5.1.1.4.19 IndexedLineSet

5.1.1.4.19.1 XSD description

```
<complexType name="IndexedLineSetType">
 <all>
   <element ref="xmta:IS" minOccurs="0"/>
```

```
            <element name="color" form="qualified" minOccurs="0">
               <complexType>
                 <group ref="xmta:SFColorNodeType" minOccurs="0" />
               </complexType>
             </element>
             <element name="coord" form="qualified" minOccurs="0">
               <complexType>
                 <group ref="xmta:SFCoordinateNodeType" minOccurs="0" />
               </complexType>
             </element>
           </all>
           <attribute name="colorIndex" type="xmta:MFInt32" use="optional"/>
           <attribute name="colorPerVertex" type="xmta:SFBool" use="optional" default="true"
/>
           <attribute name="coordIndex" type="xmta:MFInt32" use="optional"/>
           <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="IndexedLineSet" type="xmta:IndexedLineSetType"/>
```

5.1.1.4.19.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.24.

The IndexedLineSet node represents a 3D geometry formed by constructing polylines from 3D vertices specified in the *coord* field. IndexedLineSet uses the indices in its *coordIndex* field to specify the polylines by connecting vertices from the *coord* field. An index of "–1" indicates that the current polyline has ended and the next one begins. The last polyline may be (but does not have to be) followed by a "–1". IndexedLineSet is specified in the local coordinate system and is affected by the transformations of its ancestors.

The *coord* field specifies the 3D vertices of the line set and contains a Coordinate node.

Lines are not lit, are not texture-mapped, and do not participate in collision detection. The width of lines is implementation dependent and each line segment is solid (i.e. not dashed).

If the *color* field is not NULL, it shall contain a Color node. The colours are applied to the line(s) as follows:

a) If *colorPerVertex* is FALSE.

   1) If the *colorIndex* field is not empty, one colour is used for each polyline of the IndexedLineSet. There shall be at least as many indices in the *colorIndex* field as there are polylines in the IndexedLineSet. If the greatest index in the *colorIndex* field is N, there shall be N+1 colours in the Color node. The *colorIndex* field shall not contain any negative entries.

   2) If the *colorIndex* field is empty, the colours from the Color node are applied to each polyline of the IndexedLineSet in order. There shall be at least as many colours in the Color node as there are polylines.

b) If *colorPerVertex* is TRUE.

   — If the *colorIndex* field is not empty, colours are applied to each vertex of the IndexedLineSet in exactly the same manner that the *coordIndex* field is used to supply coordinates for each vertex from the Coordinate node. The *colorIndex* field shall contain at least as many indices as the *coordIndex* field and shall contain end-of-polyline markers (–1) in exactly the same places as the *coordIndex* field. If the greatest index in the *colorIndex* field is N, there shall be N+1 colours in the Color node.

   — If the *colorIndex* field is empty, the *coordIndex* field is used to choose colours from the Color node. If the greatest index in the *coordIndex* field is N, there shall be N+1 colours in the Color node.

If the *color* field is NULL and there is a Material defined for the Appearance affecting this IndexedLineSet, the *emissiveColor* of the Material shall be used to draw the lines. Details on lighting equations as they affect IndexedLineSet nodes are described in ISO/IEC 14772-1:1997, 4.14.

5.1.2 Programming information

The following programming related node is used in ARAF: Script.

5.1.2.1 Script

5.1.2.1.1 XSD description

```
<complexType name="ScriptType">
 <sequence>
    <element ref="xmta:field" minOccurs="0" maxOccurs="unbounded"/>
    <element ref="xmta:IS" minOccurs="0" maxOccurs="unbounded"/>
 </sequence>
 <attribute name="url" type="xmta:MFScript" use="optional"/>
 <attribute name="directOutput" type="xmta:SFBool" use="optional" default="false"/>
 <attribute name="mustEvaluate" type="xmta:SFBool" use="optional" default="false"/>
 <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="Script" type="xmta:ScriptType"/>
```

5.1.2.1.2 Functionality and semantics

As defined in ISO/IEC 14772-1:1997, 6.40.

The Script node is used to program behaviour 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, and

— 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. Details on the *url* field can be found in ISO/IEC 14772-1:1997, 4.5.

Browsers are not required to support any specific language. Detailed information on scripting languages is described in ISO/IEC 14772-1:1997, 4.12 supporting a scripting language for which a language binding is specified shall adhere to that language binding.

Sometime before a script receives the first event it shall be initialized (any language-dependent or user-defined initialize() is performed). The script is able to receive and process events that are sent to it. Each event that can be received shall be declared in the Script node using the same syntax as is used in a prototype definition:

```
eventIn type name
```

The *type* can be any of the standard VRML fields (as defined in ISO/IEC 14772-1:1997, Clause 5). *Name* shall be an identifier that is unique for this Script node.

The Script node is able to generate events in response to the incoming events. Each event that may be generated shall be declared in the Script node using the following syntax:

```
eventOut type name
```

With the exception of the *url* field, exposedFields are not allowed in Script nodes.

If the Script node's *mustEvaluate* field is FALSE, the browser may delay sending input events to the script until its outputs are needed by the browser. If the *mustEvaluate* field is TRUE, the browser shall send input events to the script as soon as possible, regardless of whether the outputs are needed. The *mustEvaluate* field shall be set to TRUE only if the Script node has effects that are not known to the browser (such as sending information across the network). Otherwise, poor performance may result.

Once the script has access to a VRML node (via an SFNode or MFNode value either in one of the Script node's fields or passed in as an eventIn), the script is able to read the contents of that node's exposed fields. If the Script node's *directOutput* field is TRUE, the script may also send events directly to any node to which it has access, and may dynamically establish or break routes. If *directOutput* is FALSE (the default), the script may only affect the rest of the world via events sent through its eventOuts. The results are undefined if *directOutput* is FALSE and the script sends events directly to a node to which it has access.

A script is able to communicate directly with the VRML browser to get information such as the current time and the current world URL. This is strictly defined by the API for the specific scripting language being used.

The location of the Script node in the scene graph has no affect on its operation. For example, if a parent of a Script node is a Switch node with whichChoice set to "–1" (i.e. ignore its children), the Script node continues to operate as specified (i.e. it receives and sends events).

5.1.3 User interactivity

5.1.3.1 General

The following user interactivity related nodes are used in ARAF: InputSensor, SphereSensor, TimeSensor, TouchSensor, MediaSensor, PlaneSensor.

5.1.3.2 InputSensor

5.1.3.2.1 XSD description

```
    <complexType name="InputSensorType">
     <all>
       <element ref="xmta:IS" minOccurs="0"/>
       <element ref="xmta:buffer" minOccurs="0"/>
     </all>
     <attribute name="enabled" type="xmta:SFBool" use="optional" default="true"/>
     <attribute name="url" type="xmta:MFUrl" use="optional" default="&amp;quot;&amp;quot;"/>
     <attributeGroup ref="xmta:DefUseGroup"/>
  </complexType>
  <element name="InputSensor" type="xmta:InputSensorType"/>
```

5.1.3.2.2 Functionality and semantics

As defined in ISO/IEC 14496-11:2015, 7.2.2.71.2.

The **InputSensor** node is used to add entry points for user inputs into a BIFS scene. It allows user events to trigger updates of the value of a field or the value of an element of a multiple field of an existing node.

Input devices are modelled as devices that generate frames of user input data. A device data frame (DDF) consists in a *list of values of any of the allowed types for node fields*. Values from DDFs are used to update the scene. For example, the DDF definition for a simple mouse is:

MouseDataFrame *[*

SFVec2f cursorPosition

SFBool singleButtonDown

]

The encoding of the DDF is implementation-dependent. Devices may send only complete DDF or sometimes subsets of DDF as well. The **buffer** field is a buffered bit string which contains a list of BIFS-Commands in the form of a CommandFrame (see ISO/IEC 14496-11, 8.6.2). Allowed BIFS-Commands are the following: FieldReplacement (see ISO/IEC 14496-11, 8.6.21), IndexedValueReplacement (see ISO/IEC 14496-11, 8.6.22) and NodeDeletion with a NULL node argument (see ISO/IEC 14496-11,

8.7.3.2). The **buffer** shall contain a number of BIFSCommands that matches the number of fields in the DDF definition for the attached device. The type of the field replaced by the nth command in the **buffer** shall match the type of the nth field in the DDF definition.

The **url** field specifies the data source to be used (see ISO/IEC 14496-11, 7.1.1.2.7.1). The **url** field shall point to a stream of type UserInteractionStream, which "access units" are DDFs. When the **enabled** is set to TRUE, upon reception of a DDF, each value (in the order of the DDF definition) is placed in the corresponding replace command according to the DDF definition, and then the replace command is executed. These updates are not time-stamped; they are executed at the time of the event, assuming a zero-decoding time. It is not required that all the replace commands be executed when the **buffer** is executed. Each replace command in the **buffer** can be independently triggered depending on the data present in the current DDF. Moreover, the presence in the **buffer** field of a NodeDeletion command at the nth position indicates that the value of the DDF corresponding to the nth field of the DDF definition shall be ignored.

The **eventTime** eventOut carrying the current time is generated after a DDF has been processed.

5.1.3.3 OutputActuator

5.1.3.3.1 XSD description

```
<ProtoDeclare name="OutputActuator" locations="org:mpeg:outputactuator">
    <field name="enabled" type="Boolean" vrml97Hint="exposedField" booleanValue="TRUE"/>
    <field name="url" type="Strings" vrml97Hint="exposedField" stringArrayValue= ""/>
    <!—Any number of eventIn fields!-->
</ProtoDeclare>
```

5.1.3.3.2 BIFS Textual description

```
EXTERNPROTO OutputActuator [
    eventIn         SFBool      activate
    exposedField    SFBool      enabled     TRUE
    exposedField    MFString    url         []
    Any number of the following may then follow:
    eventIn         eventType   DDFEventName
]""org:mpeg:outputActuator"
```

5.1.3.3.3 Functionality and semantics

The **OutputActuator** proto is used to communicate between the scene and the MPEG-V actuator. How to map these commands to the physical device is out of the scope of this document. It should be noted that the device interprets the command and produces the effect immediately when the command is received. The proto definition of **OutputActuator** is described below.

The **OutputActuator** PROTO can receive variable number of events that in turn generate Device Data Frames (**DDFs**) that are sent to the actuator. Each eventIn corresponds to one field in the DDF and has the same type. When **activate** eventIn is received, the DDF is assembled and sent to the device.

When declaring an **OutputActuator** in a **BIFS** scene, the eventIn fields shall be placed in their order of appearance in the associated DDF, after all other fields are declared. The **activate** eventIn shall be declared first in the extern proto declaration.

The **url** field specifies the device to be controled.

Only if the **enabled** field is TRUE then the DDFs are generated.

The **mandatory** input events of the OutputActuator interface are as described in Table 3.

Table 3 — OuputActuator: input events

Actuator type	Input events, in the given order	Input event meaning
Light	SFFloat, SFColor	intensity, color
Vibration	SFFloat	intensity
Tactile	MFFloat	intensity
Flash	SFFloat, SFColor	intensity, color
Heating	SFFloat	intensity
Cooling	SFFloat	intensity
Wind	SFFloat	intensity
Sprayer	SFFloat, SFInt32	intensity, sprayingType
Scent	SFFloat, SFint32	intensity, scent
Fog	SFFloat	intensity
Rigid Body Motion	MFVec3f, MFVec3f, MFVec3f, MFVec3f, MFVec3f, MFVec3f	direction, speed, acceleration, angle, angleSpeed, angleAcceleration
Kinesthetic	MFVec3f, MFVec3f, MFVec3f, MFVec3f	position, orientation, force, torque

In the following example, two actuators of different types (Light and Vibration) are presented. In this case, the EXTERNPROTO OutputActuator is declared twice as follows:

```
EXTERNPROTO LightActuator [
eventIn          SFBool     activate
exposedField     SFBool     enabled     TRUE
exposedField     MFString   url         [hw://lightDevice1]
eventIn          SFFloat    intensity
eventIn          SFColor    color
]"org:mpeg:outputactuator"

EXTERNPROTO VibrationActuator [
eventIn          SFBool     activate
exposedField     SFBool     enabled     TRUE
exposedField     MFString   url         [hw://vibrationDevice1]
eventIn          SFFloat    intensity
]"org:mpeg:outputactuator"

This could be instantiated in the scene as follows:

DEF SI_LIGHT ScalarInterpolator {
    Key          [0 0.5 1]
    keyValues    [0.1 0.2 0.3]
}

DEF SI_VIBRATION ScalarInterpolator {
    Key          [0 0.5 1]
    keyValues    [0.1 0.2 0.3]
}

DEF LIGHT_1 LightActuator {}
DEF LIGHT_2 LightActuator {
    url          [hw://lightDevice2]
}

DEF VIBRATION_1 VibrationActuator {}
DEF VIBRATION_2 VibrationActuator {
    enabled       FALSE
    url           [hw://vibrationDevice2]
}

ROUTE SI_LIGHT.value_changed TO LIGHT_1.intensity
ROUTE SI_LIGHT.value_changed TO LIGHT_2.intensity

ROUTE SI_VIBRATION.value_changed TO VIBRATION_1.intensity
ROUTE SI_VIBRATION.value_changed TO VIBRATION_2.intensity
```

5.1.3.4 SphereSensor

5.1.3.4.1 XSD description

```
<complexType name="SphereSensorType">
 <all>
   <element ref="xmta:IS" minOccurs="0"/>
 </all>
 <attribute name="autoOffset" type="xmta:SFBool" use="optional" default="true"/>
 <attribute name="enabled" type="xmta:SFBool" use="optional" default="true"/>
 <attribute name="offset" type="xmta:SFRotation" use="optional" default="0 1 0 0"/>
 <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="SphereSensor" type="xmta:SphereSensorType"/>
```

5.1.3.4.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.44.

The SphereSensor node maps pointing device motion into spherical rotation about the origin of the local coordinate system. The SphereSensor node uses the descendent geometry of its parent node to determine whether it is liable to generate events.

The *enabled* exposed field enables and disables the SphereSensor node. If *enabled* is TRUE, the sensor reacts appropriately to user events. If *enabled* is FALSE, the sensor does not track user input or send events. If *enabled* receives a FALSE event and *isActive* is TRUE, the sensor becomes disabled and deactivated, and outputs an *isActive* FALSE event. If *enabled* receives a TRUE event, the sensor is enabled and ready for user activation.

The SphereSensor node generates events when the pointing device is activated while the pointer is indicating any descendent geometry nodes of the sensor's parent group. See ISO/IEC 14772-1:1997, 4.6.7.5, for details on using the pointing device to activate the SphereSensor.

Upon activation of the pointing device (e.g. mouse button down) over the sensor's geometry, an *isActive* TRUE event is sent. The vector defined by the initial point of intersection on the SphereSensor's geometry and the local origin determines the radius of the sphere that is used to map subsequent pointing device motion while dragging. The virtual sphere defined by this radius and the local origin at the time of activation is used to interpret subsequent pointing device motion and is not affected by any changes to the sensor's coordinate system while the sensor is active. For each position of the bearing, a *rotation_changed* event is sent which corresponds to the sum of the relative rotation from the original intersection point plus the *offset* value. *trackPoint_changed* events reflect the unclamped drag position on the surface of this sphere. When the pointing device is deactivated and *autoOffset* is TRUE, *offset* is set to the last *rotation_changed* value and an *offset_changed* event is generated. See ISO/IEC 14772-1:1997, 4.6.7.4 for more details.

When the sensor generates an *isActive* TRUE event, it grabs all further motion events from the pointing device until it is released and generates an *isActive* FALSE event (other pointing-device sensors shall not generate events during this time). Motion of the pointing device while *isActive* is TRUE is termed a "drag". If a 2D pointing device is in use, *isActive* events will typically reflect the state of the primary button associated with the device (i.e. *isActive* is TRUE when the primary button is pressed and FALSE when it is released). If a 3D pointing device (e.g. wand) is in use, *isActive* events will typically reflect whether the pointer is within (or in contact with) the sensor's geometry.

While the pointing device is activated, *trackPoint_changed* and *rotation_changed* events are output. *trackPoint_changed* events represent the unclamped intersection points on the surface of the invisible sphere. If the pointing device is dragged off the sphere while activated, browsers may interpret this in a variety of ways (e.g. clamp all values to the sphere or continue to rotate as the point is dragged away from the sphere). Each movement of the pointing device while *isActive* is TRUE generates *trackPoint_changed* and *rotation_changed* events.

5.1.3.5 TimeSensor

5.1.3.5.1 XSD description

```
<complexType name="TimeSensorType">
 <all>
   <element ref="xmta:IS" minOccurs="0"/>
 </all>
 <attribute name="cycleInterval" type="xmta:SFTime" use="optional" default="1"/>
 <attribute name="enabled" type="xmta:SFBool" use="optional" default="true"/>
 <attribute name="loop" type="xmta:SFBool" use="optional" default="false"/>
 <attribute name="startTime" type="xmta:SFTime" use="optional" default="0"/>
 <attribute name="stopTime" type="xmta:SFTime" use="optional" default="0"/>
 <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="TimeSensor" type="xmta:TimeSensorType"/>
```

5.1.3.5.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.50.

TimeSensor nodes generate events as time passes. TimeSensor nodes can be used for many purposes including:

— driving continuous simulations and animations;

— controlling periodic activities (e.g. one per minute);

— initiating single occurrence events such as an alarm clock.

The TimeSensor node contains two discrete eventOuts: *isActive* and *cycleTime*. The *isActive* eventOut sends TRUE when the TimeSensor node begins running, and FALSE when it stops running. The *cycleTime* eventOut sends a time event at *startTime* and at the beginning of each new cycle (useful for synchronization with other time-based objects). The remaining eventOuts generate continuous events. The *fraction_changed* eventOut, an SFFloat in the closed interval [0.1], sends the completed fraction of the current cycle. The *time* eventOut sends the absolute time for a given *simulation tick*.

If the *enabled* exposedField is TRUE, the TimeSensor node is enabled and may be running. If a *set_enabled* FALSE event is received while the TimeSensor node is running, the sensor performs the following actions:

— evaluates and sends all relevant outputs;

— sends a FALSE value for *isActive*;

— disables itself.

Events on the exposedFields of the TimeSensor node (e.g. *set_startTime)* are processed and their corresponding eventOuts (e.g. *startTime_changed)* are sent regardless of the state of the *enabled* field. The remaining discussion assumes *enabled* is TRUE.

The *loop, startTime,* and *stopTime* exposedFields and the *isActive* eventOut and their effects on the TimeSensor node are discussed in detail in ISO/IEC 14772:1997, 4.6.9. The "cycle" of a TimeSensor node lasts for *cycleInterval* seconds. The value of *cycleInterval* shall be greater than zero.

A *cycleTime* eventOut can be used for synchronization purposes such as sound with animation. The value of a *cycleTime* eventOut will be equal to the time at the beginning of the current cycle. A *cycleTime* eventOut is generated at the beginning of every cycle, including the cycle starting at *startTime*. The first *cycleTime* eventOut for a TimeSensor node can be used as an alarm (single pulse at a specified time).

When a TimeSensor node becomes active, it generates an *isActive* = TRUE event and begins generating *time, fraction_changed,* and *cycleTime* events which may be routed to other nodes to drive animation or simulated behaviours. The behaviour at read time is described below. The *time* event sends the absolute

time for a given tick of the TimeSensor node (time fields and events represent the number of seconds since midnight GMT January 1, 1970).

fraction_changed events output a floating point value in the closed interval [0, 1]. At *startTime* the value of *fraction_changed* is 0. After *startTime,* the value of *fraction_changed* in any cycle will progress through the range [0.0, 1.0]. At *startTime* + × *cycleInterval,* for N = 1, 2, ..., that is, at the end of every cycle, the value of *fraction_changed* is 1.

Let *now* represent the time at the current simulation tick. Then the *time* and *fraction_changed* eventOuts can then be computed as:

```
time = now
temp = (now - startTime) / cycleInterval
f    = fractionalPart(temp)
if (f == 0.0 && now > startTime) fraction_changed = 1.0
else fraction_changed = f
```

where fractionalPart(x) is a function that returns the fractional part, (that is, the digits to the right of the decimal point), of a nonnegative floating point number.

A TimeSensor node can be set up to be active at read time by specifying *loop* TRUE (not the default) and *stopTime* less than or equal to *startTime* (satisfied by the default values). The *time* events output absolute times for each tick of the TimeSensor node simulation. The *time* events shall start at the first simulation tick greater than or equal to *startTime. time* events end at *stopTime* or at *startTime* + × *cycleInterval* for some positive integer value of *N*, or loop forever depending on the values of the other fields. An active TimeSensor node shall stop at the first simulation tick when *now* >= *stopTime* > *startTime*.

No guarantees are made with respect to how often a TimeSensor node generates time events, but a TimeSensor node shall generate events at least at every simulation tick. TimeSensor nodes are guaranteed to generate final *time* and *fraction_changed* events. If loop is FALSE at the end of the *N*th cycleInterval and was TRUE at *startTime + M* × *cycleInterval* for all *0 < M < N*, the final *time* event will be generated with a value of (*startTime + N* × *cycleInterval*) or *stopTime (*if *stopTime* > *startTime),* whichever value is less. If *loop* is TRUE at the completion of every cycle, the final event is generated as evaluated at *stopTime* (if *stopTime* > *startTime)* or never.

An active TimeSensor node ignores *set_cycleInterval* and *set_startTime* events. An active TimeSensor node also ignores *set_stopTime* events for *set_stopTime* less than or equal to *startTime*. For example, if a *set_startTime* event is received while a TimeSensor node is active, that *set_startTime* event is ignored (the *startTime* field is not changed, and a *startTime_changed* eventOut is not generated). If an active TimeSensor node receives a *set_stopTime* event that is less than the current time, and greater than *startTime*, it behaves as if the *stopTime* requested is the current time and sends the final events based on the current time (note that *stopTime* is set as specified in the eventIn).

A TimeSensor read from a VRML file shall generate *isActive* TRUE, *time* and *fraction_changed* events if the sensor is enabled and all conditions for a TimeSensor to be active are met.

5.1.3.6 TouchSensor

5.1.3.6.1 XSD description

```
<complexType name="TouchSensorType">
 <all>
   <element ref="xmta:IS" minOccurs="0"/>
 </all>
 <attribute name="enabled" type="xmta:SFBool" use="optional" default="true"/>
 <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="TouchSensor" type="xmta:TouchSensorType"/>
```

5.1.3.6.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.51.

A TouchSensor node tracks the location and state of the pointing device and detects when the user points at geometry contained by the TouchSensor node's parent group. A TouchSensor node can be enabled or disabled by sending it an *enabled* event with a value of TRUE or FALSE. If the TouchSensor node is disabled, it does not track user input or send events.

The TouchSensor generates events when the pointing device points toward any geometry nodes that are descendants of the TouchSensor's parent group. See ISO/IEC 14772-1:1997, 4.6.7.5, for more details on using the pointing device to activate the TouchSensor.

The *isOver* eventOut reflects the state of the pointing device with regard to whether it is pointing towards the TouchSensor node's geometry or not. When the pointing device changes state from a position such that its bearing does not intersect any of the TouchSensor node's geometry to one in which it does intersect geometry, an *isOver* TRUE event is generated. When the pointing device moves from a position such that its bearing intersects geometry to one in which it no longer intersects the geometry, or some other geometry is obstructing the TouchSensor node's geometry, an *isOver* FALSE event is generated. These events are generated only when the pointing device has moved and changed "over" state. Events are not generated if the geometry itself is animating and moving underneath the pointing device.

As the user moves the bearing over the TouchSensor node's geometry, the point of intersection (if any) between the bearing and the geometry is determined. Each movement of the pointing device, while *isOver* is TRUE, generates *hitPoint_changed*, *hitNormal_changed* and *hitTexCoord_changed* events. *hitPoint_changed* events contain the 3D point on the surface of the underlying geometry, given in the TouchSensor node's coordinate system. *hitNormal_changed* events contain the surface normal vector at the *hitPoint*. *hitTexCoord_changed* events contain the texture coordinates of that surface at the *hitPoint*. The values of *hitTexCoord_changed* and *hitNormal_changed* events are computed as appropriate for the associated shape.

If *isOver* is TRUE, the user may activate the pointing device to cause the TouchSensor node to generate *isActive* events (e.g. by pressing the primary mouse button). When the TouchSensor node generates an *isActive* TRUE event, it grabs all further motion events from the pointing device until it is released and generates an *isActive* FALSE event (other pointing-device sensors will not generate events during this time). Motion of the pointing device while *isActive* is TRUE is termed a "drag." If a 2D pointing device is in use, *isActive* events reflect the state of the primary button associated with the device (i.e. *isActive* is TRUE when the primary button is pressed and FALSE when it is released). If a 3D pointing device is in use, *isActive* events will typically reflect whether the pointing device is within (or in contact with) the TouchSensor node's geometry.

The eventOut field *touchTime* is generated when all three of the following conditions are true.

— The pointing device was pointing towards the geometry when it was initially activated (*isActive* is TRUE).

— The pointing device is currently pointing towards the geometry (*isOver* is TRUE).

— The pointing device is deactivated (*isActive* FALSE event is also generated).

In a 2D context, there are restrictions on the SFVec3f eventOuts:

— hitNormal_changed always returns [0.0, 0.0, 1.0];

— hitPoint_changed always has 0.0 as Z coordinate.

5.1.3.7 MediaSensor

5.1.3.7.1 XSD description

```
<complexType name="MediaSensorType">
 <all>
   <element ref="xmta:IS" minOccurs="0"/>
 </all>
```

```
        <attribute name="url" type="xmta:MFUrl" use="optional"/>
        <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="MediaSensor" type="xmta:MediaSensorType"/>
```

5.1.3.7.2 Functionality and semantics

As defined in ISO/IEC 14496-11, 7.2.2.71.2.

The **MediaSensor** node monitors the availability and presentation status of one or more stream objects.

The **url** field identifies a list of stream objects monitored by the **MediaSensor** node. All the stream objects in the **url** field shall belong to the same media stream. A stream object is considered to be available when any of its composition units is available in the composition buffer and is due for composition at that time. A stream object is considered to be no longer available when it is paused or stopped. A stream object is considered to "become available" when it "is available" for the first time. When there are several monitored stream objects available at the same time, the fields in the **MediaSensor** convey information about the stream object that became available last. If the stream that last became available becomes inactive, the **MediaSensor** node shall convey information about the first active stream in its **url** field. The **isActive** event sends a TRUE value each time one of the monitored stream objects referred by the **url** field becomes available, and a FALSE value when all of them become not available. Whenever a new composition unit is due for composition, a **mediaCurrentTime** event is sent and indicates the media time of that composition unit within the stream object.

The **streamObjectStartTime** event conveys the start of the stream object within a stream, relative to media time zero of the whole stream. The **mediaDuration** event conveys the duration of the stream object in seconds. It is set to –1 if this duration is unknown. The **info** event conveys information about the stream object that is currently monitored. Its first element identifies the stream object using the same syntax as in the **url** field.

The **streamObjectStartTime, mediaDuration** and **info** events are triggered when any stream object in the **url** field becomes available.

5.1.3.8 PlaneSensor

5.1.3.8.1 XSD description

```
<complexType name="PlaneSensorType">
    <all>
       <element ref="xmta:IS" minOccurs="0"/>
    </all>
    <attribute name="autoOffset" type="xmta:SFBool" use="optional" default="true"
/>
    <attribute name="enabled" type="xmta:SFBool" use="optional" default="true"
/>
    <attribute name="maxPosition" type="xmta:SFVec2f" use="optional" default="-1 -1"
/>
    <attribute name="minPosition" type="xmta:SFVec2f" use="optional" default="0 0"
/>
    <attribute name="offset" type="xmta:SFVec3f" use="optional" default="0 0 0"
/>
    <attributeGroup ref="xmta:DefUseGroup"/>
  </complexType>
  <element name="PlaneSensor" type="xmta:PlaneSensorType"/>
```

5.1.3.8.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.34.

The PlaneSensor node maps pointing device motion into two-dimensional translation in a plane parallel to the Z=0 plane of the local coordinate system. The PlaneSensor node uses the descendent geometry of its parent node to determine whether it is liable to generate events.

The *enabled* exposedField enables and disables the PlaneSensor. If *enabled* is TRUE, the sensor reacts appropriately to user events. If *enabled* is FALSE, the sensor does not track user input or send events. If *enabled* receives a FALSE event and *isActive* is TRUE, the sensor becomes disabled and deactivated, and outputs an *isActive* FALSE event. If *enabled* receives a TRUE event, the sensor is enabled and made ready for user activation.

The PlaneSensor node generates events when the pointing device is activated while the pointer is indicating any descendent geometry nodes of the sensor's parent group. See ISO/IEC 14772-1:1997, 4.6.7.5 for details on using the pointing device to activate the PlaneSensor.

Upon activation of the pointing device (e.g. mouse button down) while indicating the sensor's geometry, an *isActive* TRUE event is sent. Pointer motion is mapped into relative translation in the *tracking plane*, (a plane parallel to the sensor's local Z=0 plane and coincident with the initial point of intersection). For each subsequent movement of the bearing, a *translation_changed* event is output which corresponds to the sum of the relative translation from the original intersection point to the intersection point of the new bearing in the plane plus the *offset* value. The sign of the translation is defined by the Z=0 plane of the sensor's coordinate system. *trackPoint_changed* events reflect the unclamped drag position on the surface of this plane. When the pointing device is deactivated and *autoOffset* is TRUE, *offset* is set to the last *translation_changed* value and an *offset_changed* event is generated. More details are provided in See ISO/IEC 14772-1:1997, 4.6.7.4.

When the sensor generates an *isActive* TRUE event, it grabs all further motion events from the pointing device until it is deactivated and generates an *isActive* FALSE event. Other pointing-device sensors shall not generate events during this time. Motion of the pointing device while *isActive* is TRUE is referred to as a "drag." If a 2D pointing device is in use, *isActive* events typically reflect the state of the primary button associated with the device (i.e. *isActive* is TRUE when the primary button is pressed, and is FALSE when it is released). If a 3D pointing device (e.g. wand) is in use, *isActive* events typically reflect whether the pointer is within or in contact with the sensor's geometry.

minPosition and *maxPosition* may be set to clamp *translation_changed* events to a range of values as measured from the origin of the Z=0 plane. If the X or Y component of *minPosition* is greater than the corresponding component of *maxPosition*, *translation_changed* events are not clamped in that dimension. If the X or Y component of *minPosition* is equal to the corresponding component of *maxPosition*, that component is constrained to the given value. This technique provides a way to implement a line sensor that maps dragging motion into a translation in one dimension.

While the pointing device is activated and moved, *trackPoint_changed* and *translation_changed* events are sent. *trackPoint_changed* events represent the unclamped intersection points on the surface of the tracking plane. If the pointing device is dragged off of the tracking plane while activated (e.g. above horizon line), browsers may interpret this in variety ways (e.g. clamp all values to the horizon). Each movement of the pointing device, while *isActive* is TRUE, generates *trackPoint_changed* and *translation_changed* events.

For further information about this behaviour, information can be found in ISO/IEC 14772-1:1997, 4.6.7.3, ISO/IEC 14772-1:1997, 4.6.7.4, and ISO/IEC 14772-1:1997, 4.6.7.5.

5.1.4 Scene related information (spatial and temporal relationships)

5.1.4.1 General

The following scene related nodes are used in ARAF: ARContent, Background, Background2D, CameraCalibration, Group, Inline, Layer2D, Layer3D, Layout, NavigationInfo, OrderedGroup, LocImg, RemImgProxy, RemImgServer, RemImgComp, LocAud, RemAud, Switch, Transform, Transform2D, Viewpoint, Viewport, Form.

5.1.4.2 Background

5.1.4.2.1 XSD description

```
<complexType name="BackgroundType">
  <all>
     <element ref="xmta:IS" minOccurs="0"/>
  </all>
  <attribute name="groundAngle" type="xmta:MFFloat" use="optional"/>
  <attribute name="groundColor" type="xmta:MFColor" use="optional"/>
  <attribute name="backUrl" type="xmta:MFUrl" use="optional"/>
  <attribute name="bottomUrl" type="xmta:MFUrl" use="optional"/>
  <attribute name="frontUrl" type="xmta:MFUrl" use="optional"/>
  <attribute name="leftUrl" type="xmta:MFUrl" use="optional"/>
  <attribute name="rightUrl" type="xmta:MFUrl" use="optional"/>
  <attribute name="topUrl" type="xmta:MFUrl" use="optional"/>
  <attribute name="skyAngle" type="xmta:MFFloat" use="optional"/>
  <attribute name="skyColor" type="xmta:MFColor" use="optional" default="0 0 0"/>
  <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="Background" type="xmta:BackgroundType"/>
```

5.1.4.2.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.5.

The Background node is used to specify a colour backdrop that simulates ground and sky, as well as a background texture, or *panorama*, that is placed behind all geometry in the scene and in front of the ground and sky. Background nodes are specified in the local coordinate system and are affected by the accumulated rotation of their ancestors as described below.

Background nodes are bindable nodes as described in ISO/IEC 14772-1:1997, 4.6.10. There exists a Background stack, in which the top-most Background on the stack is the currently active Background. To move a Background to the top of the stack, a TRUE value is sent to the *set_bind* eventIn. Once active, the Background is then bound to the browsers view. A FALSE value sent to *set_bind* removes the Background from the stack and unbinds it from the browser's view. More detail on the bind stack is described in ISO/IEC 14772-1:1997, 4.6.10.

The backdrop is conceptually a partial sphere (the ground) enclosed inside of a full sphere (the sky) in the local coordinate system with the viewer placed at the centre of the spheres. Both spheres have infinite radius and each is painted with concentric circles of interpolated colour perpendicular to the local Y-axis of the sphere. The Background node is subject to the accumulated rotations of its ancestors' transformations. Scaling and translation transformations are ignored. The sky sphere is always slightly farther away from the viewer than the ground partial sphere causing the ground to appear in front of the sky where they overlap.

The *skyColor* field specifies the colour of the sky at various angles on the sky sphere. The first value of the *skyColor* field specifies the colour of the sky at 0.0 radians representing the zenith (i.e. straight up from the viewer). The *skyAngle* field specifies the angles from the zenith in which concentric circles of colour appear. The zenith of the sphere is implicitly defined to be 0.0 radians, the natural horizon is at $\pi/2$ radians, and the nadir (i.e. straight down from the viewer) is at π radians. *skyAngle* is restricted to non-decreasing values in the range [0.0, π]. There shall be one more *skyColor* value than there are *skyAngle* values. The first colour value is the colour at the zenith, which is not specified in the *skyAngle* field. If the last *skyAngle* is less than *pi*, then the colour band between the last *skyAngle* and the nadir is clamped to the last *skyColor*. The sky colour is linearly interpolated between the specified *skyColor* values.

The *groundColor* field specifies the colour of the ground at the various angles on the ground partial sphere. The first value of the *groundColor* field specifies the colour of the ground at 0.0 radians representing the nadir (i.e. straight down from the user). The *groundAngle* field specifies the angles from the nadir that the concentric circles of colour appear. The nadir of the sphere is implicitly defined at 0.0 radians. *groundAngle* is restricted to non-decreasing values in the range [0.0, $\pi/2$]. There shall be one more *groundColor* value than there are *groundAngle* values. The first colour value is for the nadir which is not specified in the *groundAngle* field. If the last *groundAngle* is less than $\pi/2$, the region between the

last *groundAngle* and the equator is non-existant. The ground colour is linearly interpolated between the specified *groundColor* values.

The *backUrl*, *bottomUrl*, *frontUrl*, *leftUrl*, *rightUrl*, and *topUrl* fields specify a set of images that define a background panorama between the ground/sky backdrop and the scene's geometry. The panorama consists of six images, each of which is mapped onto a face of an infinitely large cube contained within the backdrop spheres and centred in the local coordinate system. The images are applied individually to each face of the cube. On the front, back, right, and left faces of the cube, when viewed from the origin looking down the negative Z-axis with the Y-axis as the view up direction, each image is mapped onto the corresponding face with the same orientation as if the image were displayed normally in 2D (*backUrl* to back face, *frontUrl* to front face, *leftUrl* to left face, and *rightUrl* to right face*)*. On the top face of the cube, when viewed from the origin looking along the +Y-axis with the +Z-axis as the view up direction, the *topUrl* image is mapped onto the face with the same orientation as if the image were displayed normally in 2D. On the bottom face of the box, when viewed from the origin along the negative Y-axis with the negative Z-axis as the view up direction, the *bottomUrl* image is mapped onto the face with the same orientation as if the image were displayed normally in 2D.

5.1.4.3 Background2D

5.1.4.3.1 XSD description

```
<complexType name="Background2DType">
 <all>
   <element ref="xmta:IS" minOccurs="0"/>
 </all>
 <attribute name="backColor" type="xmta:SFColor" use="optional" default="0 0 0"/>
 <attribute name="url" type="xmta:MFUrl" use="optional"/>
 <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="Background2D" type="xmta:Background2DType"/>
```

5.1.4.3.2 Functionality and semantics

As defined in ISO/IEC 14496-11:2015, 7.2.2.18.2.

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 ISO/IEC 14496-11:2015, 7.1.1.2.7.1). The **backColor** field specifies a colour to be used as the background. This is not a geometry node. The top-left corner of the image is mapped to the top-left corner of the **Layer2D** and the right-bottom corner of the image is stretched to the right-bottom corner of the **Layer2D**, regardless of the current transformation. Scaling and/or rotation do not have any effect on this node. The background image will always exactly fill the entire **Layer2D**, regardless of **Layer2D** size, without tiling or cropping.

When a **Background2D** node is included in a 3D context, that is in a **Group**, **Layer3D**, or **CompositeTexture3D** node, then it shall be rendered behind all other geometries and be scaled to fit in the enclosing frame. For Group node, this frame is the whole scene. For **Layer3D** and **CompositeTexture3D** the background image is scaled to fit in the frame of the node.

5.1.4.4 CameraCalibration

5.1.4.4.1 XSD description

```
<ProtoDeclare name="CameraCalibration" locations="org:mpeg:CameraCalibration">
  <field name="source" type="Strings" vrml97Hint="exposedField" stringArrayValue=""/>
```

```
  <field name="enabled" type="Boolean" vrml97Hint="exposedField" booleanValue="false"/>
  <field name="startTime" type="Time" vrml97Hint="exposedField" timeValue="0"/>
  <field name="timeBetweenSnapshots" type="Time" vrml97Hint="exposedField" timeValue="4"/>
  <field name="snapshotsCount" type="Integer" vrml97Hint="exposedField" intValue="10"/>
  <field name="boardSize" type="Vector2" vrml97Hint="exposedField" vector2Value="8 5"/>
  <field name="onStatus" type="Integer" vrml97Hint="eventOut"/>
</ProtoDeclare>
```

5.1.4.4.2 BIFS Textual description

```
EXTERNPROTO CameraCalibration[
   exposedField  MFString  source                []
   exposedField  SFBool    enabled               FALSE
   exposedField  SFTime    startTime             0
   exposedField  SFTime    timeBetweenSnapshots  4
   exposedField  SFInt32   snapshotCount         6
   exposedField  SFVec2f   boardSize             8 5
   eventOut      SFInt32   onStatus
] "org:mpeg:CameraCalibration"
```

5.1.4.4.3 Functionality and semantics

The exposed field **source** specifies the URL for the camera for which the calibration is performed.

The exposed field **enabled** specifies whether the calibration algorithm is executed.

The exposed field **startTime** specifies at which scene time the calibration algorithm should start running.

The exposed field **timeBetweenSnapshots** specifies the time between each taken snapshot in seconds.

The exposed field **snapshotCount** specifies the number of snapshots that will be taken during the calibration procedure.

The exposed field **boardSize** specifies the number of cross points on the chessboard that is used for calibration.

The eventOut field **onStatus** outputs the current status of the calibration process as defined below

— 1: a snapshot was taken

— 2: calibration was succesful

— –1: calibration was unsuccesful

5.1.4.5 Group

5.1.4.5.1 XSD description

```
    <complexType name="GroupType">
     <all>
       <element ref="xmta:IS" minOccurs="0"/>
       <element name="children" form="qualified" minOccurs="0">
           <complexType>
             <group ref="xmta:SF3DNodeType" minOccurs="0" maxOccurs="unbounded"/>
           </complexType>
       </element>
     </all>
     <attributeGroup ref="xmta:DefUseGroup"/>
   </complexType>
   <element name="Group" type="xmta:GroupType"/>
```

5.1.4.5.2 Functionality and semantics

The semantics of the **Group** node are specified in ISO/IEC 14772-1:1997, 6.21. 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 ISO/IEC 14772-1:1997, 7.2.2.116.

As specified in ISO/IEC 14772-1:1997, 6.21, a Group node contains children nodes without introducing a new transformation. It is equivalent to a Transform node containing an identity transform.

More details on the *children*, *addChildren*, and *removeChildren* fields and eventIns can be found in ISO/IEC 14772-1:1997, 4.6.5.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Group node's children. This is a hint that may be used for optimization purposes. The results are undefined if the specified bounding box is smaller than the actual bounding box of the children at any time. A default *bboxSize* value, (–1, –1, –1), implies that the bounding box is not specified and, if needed, is calculated by the browser. A description of the *bboxCenter* and *bboxSize* fields is contained in ISO/IEC 14772-1:1997, 4.6.4.

5.1.4.6 Inline

5.1.4.6.1 XSD description

```
<complexType name="InlineType">
  <all>
    <element ref="xmta:IS" minOccurs="0"/>
  </all>
  <attribute name="url" type="xmta:MFUrl" use="optional"/>
  <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="Inline" type="xmta:InlineType"/>
```

5.1.4.6.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.25.

The Inline node is a grouping node that reads its children data from a location in the World Wide Web. Exactly when its children are read and displayed is not defined (e.g. reading the children may be delayed until the Inline node's bounding box is visible to the viewer). The *url* field specifies the URL containing the children. An Inline node with an empty URL does nothing.

Each specified URL shall refer to a valid VRML file that contains a list of children nodes, prototypes, and routes at the top level as described in ISO/IEC 14772-1:1997, 4.6.5. The results are undefined if the URL refers to a file that is not VRML or if the VRML file contains non-children nodes at the top level.

If multiple URLs are specified, the browser may display a URL of a lower preference VRML file while it is obtaining, or if it is unable to obtain, the higher preference VRML file.

The results are undefined if the contents of the URL change after it has been loaded.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Inline node's children. This is a hint that may be used for optimization purposes. The results are undefined if the specified bounding box is smaller than the actual bounding box of the children at any time. A default *bboxSize* value, (–1, –1, –1), implies that the bounding box is not specified and if needed shall be calculated by the browser.

5.1.4.7 Layer2D

5.1.4.7.1 XSD description

```
<complexType name="Layer2DType">
 <all>
    <element ref="xmta:IS" minOccurs="0"/>
    <element name="children" form="qualified" minOccurs="0">
      <complexType>
        <group ref="xmta:SF2DNodeType" minOccurs="0" maxOccurs="unbounded"/>
      </complexType>
```

```
        </element>
        <element name="background" form="qualified" minOccurs="0">
          <complexType>
            <group ref="xmta:SFBackground2DNodeType" minOccurs="0"/>
          </complexType>
        </element>
        <element name="viewport" form="qualified" minOccurs="0">
          <complexType>
             <group ref="xmta:SFViewportNodeType" minOccurs="0"/>
          </complexType>
        </element>
      </all>
      <attribute name="size" type="xmta:SFVec2f" use="optional" default="-1 -1"/>
      <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="Layer2D" type="xmta:Layer2DType"/>
```

5.1.4.7.2 Functionality and semantics

As defined in ISO/IEC 14496-11, 7.2.2.72.2.

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.

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 ISO/IEC 14496-11, 7.1.1.2.1).

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 ISO/IEC 14496-11, 7.1.1.2.2). 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 if the background field is set to NULL. If the background field is specified, any transparent part of the background will also let user input through to lower layers.

5.1.4.8 Layer3D

5.1.4.8.1 XSD description

```
<complexType name="Layer3DType">
 <all>
    <element ref="xmta:IS" minOccurs="0"/>
    <element name="children" form="qualified" minOccurs="0">
       <complexType>
          <group ref="xmta:SF3DNodeType" minOccurs="0" maxOccurs="unbounded"/>
       </complexType>
    </element>
    <element name="background" form="qualified" minOccurs="0">
       <complexType>
          <group ref="xmta:SFBackground3DNodeType" minOccurs="0"/>
       </complexType>
    </element>
    <element name="fog" form="qualified" minOccurs="0">
       <complexType>
          <group ref="xmta:SFFogNodeType" minOccurs="0"/>
       </complexType>
    </element>
    <element name="navigationInfo" form="qualified" minOccurs="0">
       <complexType>
          <group ref="xmta:SFNavigationInfoNodeType" minOccurs="0"/>
       </complexType>
    </element>
    <element name="viewpoint" form="qualified" minOccurs="0">
       <complexType>
          <group ref="xmta:SFViewpointNodeType" minOccurs="0"/>
       </complexType>
    </element>
  </all>
  <attribute name="size" type="xmta:SFVec2f" use="optional" default="-1 -1"/>
  <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="Layer3D" type="xmta:Layer3DType"/>
```

5.1.4.8.2 Functionality and semantics

As defined in ISO/IEC 14496-11, 7.2.2.73.2.

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 ISO/IEC 14496-11, 7.2.2.72) 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 ISO/IEC 14496-11, 7.1.1.2.1).

The following fields specify bindable children nodes for **Layer3D**:

— **background** for **Background** and **Background2D** 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.

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.

5.1.4.9 Layout

5.1.4.9.1 XSD description

```
      <complexType name="LayoutType">
       <all>
          <element ref="xmta:IS" minOccurs="0"/>
          <element name="children" form="qualified" minOccurs="0">
            <complexType>
               <group ref="xmta:SF2DNodeType" minOccurs="0" maxOccurs="unbounded"/>
            </complexType>
          </element>
       </all>
       <attribute name="wrap" type="xmta:SFBool" use="optional" default="false"/>
       <attribute name="size" type="xmta:SFVec2f" use="optional" default="-1 -1"/>
       <attribute name="horizontal" type="xmta:SFBool" use="optional" default="true"/>
       <attribute name="justify" type="xmta:MFString" use="optional"
default="&amp;quot;BEGIN&amp;quot;"/>
       <attribute name="leftToRight" type="xmta:SFBool" use="optional" default="true"/>
       <attribute name="topToBottom" type="xmta:SFBool" use="optional" default="true"/>
       <attribute name="spacing" type="xmta:SFFloat" use="optional" default="1"/>
       <attribute name="smoothScroll" type="xmta:SFBool" use="optional" default="false"/>
       <attribute name="loop" type="xmta:SFBool" use="optional" default="false"/>
       <attribute name="scrollVertical" type="xmta:SFBool" use="optional" default="true"/>
       <attribute name="scrollRate" type="xmta:SFFloat" use="optional" default="0"/>
       <attribute name="scrollMode" type="xmta:SFInt32" use="optional" default="0"/>
       <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="Layout" type="xmta:LayoutType"/>
```

5.1.4.9.2 Functionality and semantics

As defined in ISO/IEC 14496-11, 7.2.2.74.2.

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 ISO/IEC 14496-11, 7.2.2.61).

The **scrollRate** field specifies the time needed in seconds to scroll the layout in the given direction. For example, a layout of 200 × 100 pixels scrolling vertically with a **scrollRate** value of 2 will translate its objects vertically of 100/2 times the simulation frame duration in seconds (eg 1,65 pixels at 30 fps). 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 topmost 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: size = max(maxAscent, maxHeight) + 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.

5.1.4.10 NavigationInfo

5.1.4.10.1 XSD description

```
      <complexType name="NavigationInfoType">
       <all>
         <element ref="xmta:IS" minOccurs="0"/>
       </all>
       <attribute name="avatarSize" type="xmta:MFFloat" use="optional" default="0.25 0.75
0.75"/>
       <attribute name="headlight" type="xmta:SFBool" use="optional" default="true"/>
```

```
        <attribute name="speed" type="xmta:SFFloat" use="optional" default="1"/>
        <attribute name="type" type="xmta:MFString" use="optional"
default="&amp;quot;WALK&amp;quot;"/>
        <attribute name="visibilityLimit" type="xmta:SFFloat" use="optional" default="0"/>
        <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="NavigationInfo" type="xmta:NavigationInfoType"/>
```

5.1.4.10.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.29.

The NavigationInfo node contains information describing the physical characteristics of the viewer's avatar and viewing model. NavigationInfo node is a bindable node (see ISO/IEC 14772-1:1997, 4.6.10). Thus, there exists a NavigationInfo node stack in which the top-most NavigationInfo node on the stack is the currently bound NavigationInfo node. The current NavigationInfo node is considered to be a child of the current Viewpoint node regardless of where it is initially located in the VRML file. Whenever the current Viewpoint nodes changes, the current NavigationInfo node shall be re-parented to it by the browser. Whenever the current NavigationInfo node changes, the new NavigationInfo node shall be re-parented to the current Viewpoint node by the browser.

If a TRUE value is sent to the *set_bind* eventIn of a NavigationInfo node, the node is pushed onto the top of the NavigationInfo node stack. When a NavigationInfo node is bound, the browser uses the fields of the NavigationInfo node to set the navigation controls of its user interface and the NavigationInfo node is conceptually re-parented under the currently bound Viewpoint node. All subsequent scaling changes to the current Viewpoint node's coordinate system automatically change aspects (see below) of the NavigationInfo node values used in the browser (e.g. scale changes to any ancestors' transformations). A FALSE value sent to *set_bind* pops the NavigationInfo node from the stack, results in an *isBound* FALSE event, and pops to the next entry in the stack which shall be re-parented to the current Viewpoint node. ISO/IEC 14772-1:1997, 4.6.10 has more details on binding stacks.

The *type* field specifies an ordered list of navigation paradigms that specify a combination of navigation types and the initial navigation type. The navigation type of the currently bound NavigationInfo node determines the user interface capabilities of the browser. For example, if the currently bound NavigationInfo node's *type* is "WALK", the browser shall present a WALK navigation user interface paradigm (see below for description of WALK). Browsers shall recognize and support at least the following navigation types: "ANY", "WALK", "EXAMINE", "FLY", and "NONE".

If "ANY" does not appear in the *type* field list of the currently bound NavigationInfo, the browser's navigation user interface shall be restricted to the recognized navigation types specified in the list. In this case, browsers shall not present a user interface that allows the navigation type to be changed to a type not specified in the list. However, if any one of the values in the *type* field are "ANY", the browser may provide any type of navigation interface, and allow the user to change the navigation type dynamically. Furthermore, the first recognized type in the list shall be the initial navigation type presented by the browser's user interface.

ANY navigation specifies that the browser may choose the navigation paradigm that best suits the content and provide a user interface to allow the user to change the navigation paradigm dynamically. The results are undefined if the currently bound NavigationInfo's *type* value is "ANY" and Viewpoint transitions (see ISO/IEC 14772-1:1997, 6.53) are triggered by the Anchor node (see ISO/IEC 14772-1:1997, 6.2) or the loadURL()scripting method (see ISO/IEC 14772-1:1997, 4.12.10).

WALK navigation is used for exploring a virtual world on foot or in a vehicle that rests on or hovers above the ground. It is strongly recommended that WALK navigation define the up vector in the +Y direction and provide some form of terrain following and gravity in order to produce a walking or driving experience. If the bound NavigationInfo's *type* is "WALK", the browser shall strictly support collision detection (see ISO/IEC 14772-1:1997, 6.8).

FLY navigation is similar to WALK except that terrain following and gravity may be disabled or ignored. There shall still be some notion of "up" however. If the bound NavigationInfo's *type* is "FLY", the browser shall strictly support collision detection (see ISO/IEC 14772-1:1997, 6.8).

EXAMINE navigation is used for viewing individual objects and often includes (but does not require) the ability to spin around the object and move the viewer closer or further away.

NONE navigation disables and removes all browser-specific navigation user interface forcing the user to navigate using only mechanisms provided in the scene, such as Anchor nodes or scripts that include loadURL().

If the NavigationInfo type is "WALK", "FLY", "EXAMINE", or "NONE" or a combination of these types (i.e. ANY" is not in the list), Viewpoint transitions (see ISO/IEC 14772-1:1997, 6.53) triggered by the Anchor node (see ISO/IEC 14772-1:1997, 6.2) or the loadURL()scripting method (see ISO/IEC 14772-1:1997, 4.12.10) shall be implemented as a jump cut from the old Viewpoint to the new Viewpoint with transition effects that shall not trigger events besides the exit and enter events caused by the jump.

Browsers may create browser-specific navigation type extensions. It is recommended that extended *type* names include a unique suffix (e.g. HELICOPTER_mydomain.com) to prevent conflicts. Viewpoint transitions (see ISO/IEC 14772-1:1997, 6.53) triggered by the Anchor node (see ISO/IEC 14772-1:1997, 6.2) or the loadURL()scripting method (see ISO/IEC 14772-1:1997, 4.12.10) are undefined for extended navigation types. If none of the types are recognized by the browser, the default "ANY" is used. These strings values are case-sensitive ("any" is not equal to "ANY").

The *speed* field specifies the rate at which the viewer travels through a scene in metres per second. Since browsers may provide mechanisms to travel faster or slower, this field specifies the default, average speed of the viewer when the NavigationInfo node is bound. If the NavigationInfo *type* is EXAMINE, *speed* shall not affect the viewer's rotational speed. Scaling in the transformation hierarchy of the currently bound Viewpoint node (see above) scales the *speed*; parent translation and rotation transformations have no effect on *speed*. Speed shall be non-negative. Zero speed indicates that the avatar's position is stationary, but its orientation and field of view may still change. If the navigation *type* is "NONE", the *speed* field has no effect.

The *avatarSize* field specifies the user's physical dimensions in the world for the purpose of collision detection and terrain following. It is a multi-value field allowing several dimensions to be specified. The first value shall be the allowable distance between the user's position and any collision geometry (as specified by a Collision node) before a collision is detected. The second shall be the height above the terrain at which the browser shall maintain the viewer. The third shall be the height of the tallest object over which the viewer can move. This allows staircases to be built with dimensions that can be ascended by viewers in all browsers. The transformation hierarchy of the currently bound Viewpoint node scales the *avatarSize*. Translations and rotations have no effect on *avatarSize*.

For purposes of terrain following, the browser maintains a notion of the *down* direction (down vector), since gravity is applied in the direction of the down vector. This down vector shall be along the negative Y-axis in the local coordinate system of the currently bound Viewpoint node (i.e. the accumulation of the Viewpoint node's ancestors' transformations, not including the Viewpoint node's *orientation* field).

Geometry beyond the visibilityLimit may not be rendered. A value of 0.0 indicates an infinite visibility limit. The *visibilityLimit* field is restricted to be greater than or equal to zero.

The *speed*, *avatarSize* and *visibilityLimit* values are all scaled by the transformation being applied to the currently bound Viewpoint node. If there is no currently bound Viewpoint node, the values are interpreted in the world coordinate system. This allows these values to be automatically adjusted when binding to a Viewpoint node that has a scaling transformation applied to it without requiring a new NavigationInfo node to be bound as well. The results are undefined if the scale applied to the Viewpoint node is non-uniform.

The *headlight* field specifies whether a browser shall turn on a headlight. A headlight is a directional light that always points in the direction the user is looking. Setting this field to TRUE allows the browser to provide a headlight, possibly with user interface controls to turn it on and off. Scenes that enlist precomputed lighting (e.g. radiosity solutions) can turn the headlight off. The headlight shall have *intensity* = 1, *color* = (1 1 1), *ambientIntensity* = 0.0, and *direction* = (0 0 −1).

It is recommended that the near clipping plane be set to one-half of the collision radius as specified in the *avatarSize* field (setting the near plane to this value prevents excessive clipping of objects just above the collision volume, and also provides a region inside the collision volume for content authors to include geometry intended to remain fixed relative to the viewer). Such geometry shall not be occluded by geometry outside of the collision volume.

5.1.4.11 OrderedGroup

5.1.4.11.1 XSD description

```
<complexType name="OrderedGroupType">
 <all>
    <element ref="xmta:IS" minOccurs="0"/>
    <element name="children" form="qualified" minOccurs="0">
      <complexType>
        <group ref="xmta:SF3DNodeType" minOccurs="0" maxOccurs="unbounded"/>
      </complexType>
     </element>
 </all>
 <attribute name="order" type="xmta:MFFloat" use="optional"/>
 <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="OrderedGroup" type="xmta:OrderedGroupType"/>
```

5.1.4.11.2 Functionality and semantics

As defined in ISO/IEC 14496-11:2015, 7.2.2.90.2.

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. If there are more children than entries in the **order** field, those children that do not have a drawing order are drawn in the order in which they appear in the **children** field, but after the ones that have an entry in the **order** field. If there are more order entries than children, the excess order entries are ignored.

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.

Content authors should 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.

5.1.4.12 Image recognition, registration, composition in ARAF

5.1.4.12.1 General

It is obvious that an AR standard should have support for image recognition and tracking given the fact that the trivial AR application uses the device's camera to retrieve and process the camera

frame and based on the result augment and present enriched content (augmentation). Functionalities implementing local or remote image recognition, tracking and composition are therefore defined in this document.

5.1.4.12.2 presents an important set of parameters to be considered in the computation of the pose matrix of a recognized target resource. These are the internal parameters of the device's camera used to capture the real world.

5.1.4.12.2 Intrinsic camera parameters (internal parameters)

An ARAF browser implementing the prototypes related to remote image recognition (and tracking) can, and it's recommended to provide the intrinsic parameters of the camera which is providing the video frames where the augmentation is performed. This set of internal camera parameters is important especially in the process of computation the pose matrix of the recognized target resource (see the related proto descriptions). In some cases, the image processing library does not need the intrinsic camera parameters, therefore this is an optional feature of an ARAF browser.

The way how an ARAF browser computes the intrinsic camera parameters is not in the scope of this document as long as the image processing library/server receives the parameters in the correct format.

For example, the ARAF browser may (and it is recommended to) have a way of storing static parameters and/or user preferences. The Browser should automatically compute the intrinsic camera parameters if possible, otherwise to notify the end-user if the values have not been computed yet on the current device and propose a way of computing them. This would be the case when the user's intervention is needed in the process of camera calibration. A camera calibration prototype is proposed in the ARAF; therefore, an ARAF browser implementing the prototype should ask the end-user to perform the calibration before running a MAR Experience.

The following parameters are the ones considered in the context of image recognition process in ARAF:

— focal length (fl);

— the principal point (cc);

— the skew coefficient (alpha_c).

An ARAF browser should send the intrinsic camera parameters to the image recognition server using a HTTP request that has the following format:

fl=fx,fy&

cc=cx,cy&

alpha_c=angle

where *fx* and *fy* represent the focal length, *cx* and *cy* is the principal point and *alpha_c* is the skew coefficient. All these values are floats.

Refer to the functionality and semantics of the image recognition prototypes in order to see how the HTTP request should be performed from the ARAF browser (client) to the image processing server.

If any of the intrinsic camera parameters is not available, the corresponding key in the HTTP request should not be transmitted (ignored).

5.1.4.12.3 Tables referred in the image recognition and tracking prototypes

Table 4 — Image recognition and tracking: communication protocols

Communication protocol name	Reference	Code
RTP	RFC 3550-2003 Real Time Transport Protocol	0
RTSP	RFC 2326-2013 version 2.0 Real Time Streaming Protocol	1
HTTP	RFC 2616-1999 Hypertext Transfer Protocol	2
DASH	ISO/IEC 23009-1:2012 Dynamic Adaptive Streaming over HTTP	3

Table 5 — Image recognition and tracking: video file formats

Video file format	Reference
Raw video data	ISO/IEC 14496-1:2010 + Amd. 2:2014
MPEG4 Visual	ISO/IEC 14496-2
MPEG4 AVC	ISO/IEC 14496-10

Table 6 — Image recognition and tracking: camera uri

Camera URI	Description
worldFacingCamera	Refers to the primary camera, usually located at the back of the device (back camera)
userFacingCamera	Refers to the secondary camera, usually located at the front of the device (front camera)

Table 7 — Image recognition and tracking: target image formats

Target image file formats	Reference	targetResourceType keyword	targetResourceType code
JPEG	ISO/IEC 10918	JPEG	0
JPEG 2000	ISO/IEC 15444	J2K	1
PNG	ISO/IEC 15948	PNG	2
RAW	ISO 12234-2	RAW	3

Table 8 — Image recognition and tracking: target image descriptors formats

Target image descriptor file formats	Description	targetResourceType keyword	targetResourceType code
standard	CDVA	CDVA	5
reserved	Proprietary descriptors	See Annex B	90-99

5.1.4.12.4 Prototypes implementations

5.1.4.12.4.1 LocImg (local image recognition registration)

The **ARAF browser** detects and recognizes the presence of **target resources** in a **video** and computes the **pose matrix** of the recognized ones. The **target resources URL** and the **video URL** where the recognition shall be performed are specified by the **MAREC**. The ARAF browser decides which **recognition library** is more appropriate for recognizing the provided target resources considering the time constraints imposed by the MAREC. The ARAF browser sends the result to the **ARAF scene** as an array of integers representing the indexes of the recognized target resources along with the pose

matrixes (optional) for each detected target resource, as described below in the **Functionality and Semantics**.

Even though in the presented schema (below) the augmentation media is presented as input along other data provided by the MAREC, the augmentation media is not used by the LocImg PROTO implementation, meaning that it is not part of any of the PROTO fields. The augmentation resources are of course required for the scene augmentation but they do not influence the recognition process.

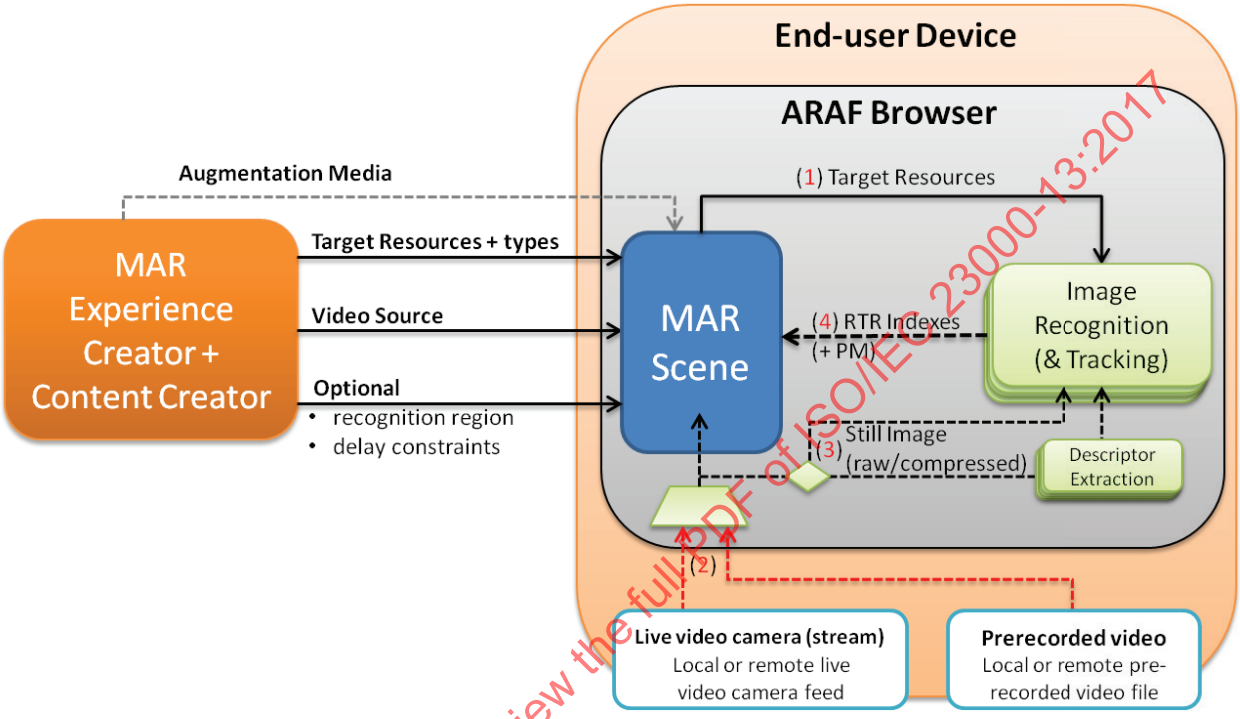


Figure 3 — Local image recognition registration

5.1.4.12.4.2 BIFS Textual description

```
EXTERNPROTO LocImg [
    exposedField SFString    videoSource           ""
    exposedField MFString    targetResources       []
    exposedField MFString    targetResourcesTypes  []
    exposedField SFBool      enabled               FALSE
    exposedField SFInt32     maximumDelay          200 #milliseconds
    exposedField SFInt32     optimalDelay          50 #milliseconds
    exposedField MFVec2f     recognitionRegion     []
    eventOut     MFInt32     onRecognition
    eventOut     MFVec3f     onTranslation
    eventOut     MFRotation  onRotation
    eventOut     SFInt32     onError
]"urg:mpeg:local_image_recongnition_registration"
```

5.1.4.12.4.3 Functionality and semantics

ARAF browser running on the end-user's device performs the recognition process of target resources in a video. The expected response is an array containing the indexes of the recognized target resources and their pose matrixes (optional) that are relative to the camera viewpoint within the video, as specified in the *targetResources* field's description. Depending on the processing library capabilities, the prototype functionality can be limited to only recognizing the target resources, without providing the pose matrixes of the recognized ones. In addition to the target resources and the video, the MAREC may also provide time constraints, indirectly affecting the frequency of the recognition (and tracking) process and the power consumption of the end user's device as described further in the **Functionality and Semantics.**

videoSource is a SFString specifying the URI/URL of the video where the recognition (and tracking) process shall be performed on. The *videoSource* can be one of the following:

a) Live 2D video camera feed:

   1) a URI to one of the cameras available on the end user's device. The possible values are specified in Table 6;

   2) a URL to an external camera providing live camera feed;

b) A URL to a prerecorded video file stored:

   — locally on the end user's device;

   — remotely on an external repository in the Web.

Based on the MAREC preferences, the video frames are sent to the recognition library every X milliseconds (the ARAF browser is in charge of computing the frequency), as long as the recognition (and tracking) process is enabled. The video frames are sent as compressed images or raw data (see Table 7 for the supported image file formats), or as descriptor files (see Table 8) depending on the chosen recognition (and tracking) library capabilities. The ARAF browser is in charge of deciding the encoding type of the video frames that shall be sent to the processing library, considering the MAREC preferences and the capabilities of the processing library.

The accepted video formats are specified in Table 5.

The accepted communication protocols are specified in Table 4. **targetResources** is an MFString where the target resources to be recognized (and tracked) within the MAR experience are specified. A URI can point to a local or remote resource file. The accepted communication protocols for the remote resources are the ones specified in Table 4. Any of the below combinations describes a valid *targetResources* assignment:

— URIs pointing to target images. The file formats specified in Table 7 are accepted.

— URIs pointing to files where target image descriptors are found. A file contains descriptors of one single target image. The file formats specified in Table 8 are accepted.

— URLs pointing to files where multiple target image descriptors are found. One file contains descriptors of multiple target images. The file formats specified in Table 8 are the supported ones.

— any combination of the cases described above can coexist in the same *targetResources* field.

The **targetResourcesTypes** field is an MFString containing an array which specifies the type of each target resource defined in the targetResources field. Each target resource shall have associated a type in order for the ARAF browser to know how to interpret the data. The possible pre-defined keywords of the *targetResourcesTypes* and their meaning are listed in Table 7 and Table 8. If the target resource is a proprietary descriptor file, and in addition to the actual image descriptors data, the proprietary recognition (and tracking) library needs some other data (for e.g. an XML file), the related content should be stored in a directory that has the same name as the resource type (for e.g. *target_resource_type_keywod/*) in order for the ARAF browser to know where the required files can be found. The names of the files within the directory have to be the same name as the corresponding descriptor file name.

enabled is a SFInt32 value indicating if the recognition (and tracking) process is enabled (running). MAREC can control the status of the recognition (and tracking) process or he can let the ARAF browser to decide whether the recognition (tracking) process should be running or not. Table 9 specifies the supported integer values of the enabled field.

Table 9 — LocImg: enabled

enabled	Description
–1	The ARAF browser decides when the recognition (and tracking) process is enabled. If not supported, the recognition process is always disabled unless a value of 0 or 1 is set by the MAREC.
0 (default)	The recognition (and tracking) process is disabled.
1	The recognition (and tracking) process is enabled.

A value of –1 specifies that the ARAF browser decides the status of the recognition (and tracking) process.

The recognition (and tracking) process is inactive while *enabled* is 0.

While *enabled* is 1, the following cases are differentiated based on the *video source*.

— *local live video camera feed*: the frames coming from the local live video camera feed are considered by the ARAF browser in the recognition (and tracking) process.

— *remote live video camera feed*: the frames coming from the remote live video camera stream are considered by the ARAF browser in the recognition (and tracking) process. Technically, the only difference between the first case and the second one is the source of the video frames. In this case, a streaming protocol should be used to fetch the remote video camera stream.

— *local prerecorded video file*: as long as *enabled* is 1, the ARAF browser plays the video file and the corresponding video frames are used in the recognition (and tracking) process. Whenever *enabled* is 0 the video play back is paused. On 1, the video starts playing from the point where it was last paused. The video play back starts from the beginning when the end of the video stream is reached and *enabled* is 1.

— *remote prerecorded video file*: idem as in the previous case except that the remote file has to be downloaded first. If a streaming protocol is being used, the ARAF browser may request (if possible) video frames whenever *enabled* is 1, as it would play back the video remotely.

MAREC should have the possibility to choose the quality of his *MAR experience* and in the same time, indirectly, the processing power consumed by the recognition (and tracking) process. MAREC can control this by setting a *maximum acceptable delay*. As described in Figure 4, a response time higher than the maximum delay indicates an unacceptable quality of the MAR experience, therefore, an ARAF browser should not present it. Any response time with a delay lower than the specified maximum delay produces a MAR experience that is at least acceptable from the point of view of the MAREC, therefore, an ARAF browser should present the MAR experience. The MAREC can also specify an optimal delay constraint informing an ARAF browser that there is no need in trying to provide recognition (and tracking) response with a higher frequency (lower delay) becuase the MAR experience has already reached the targeted quality.

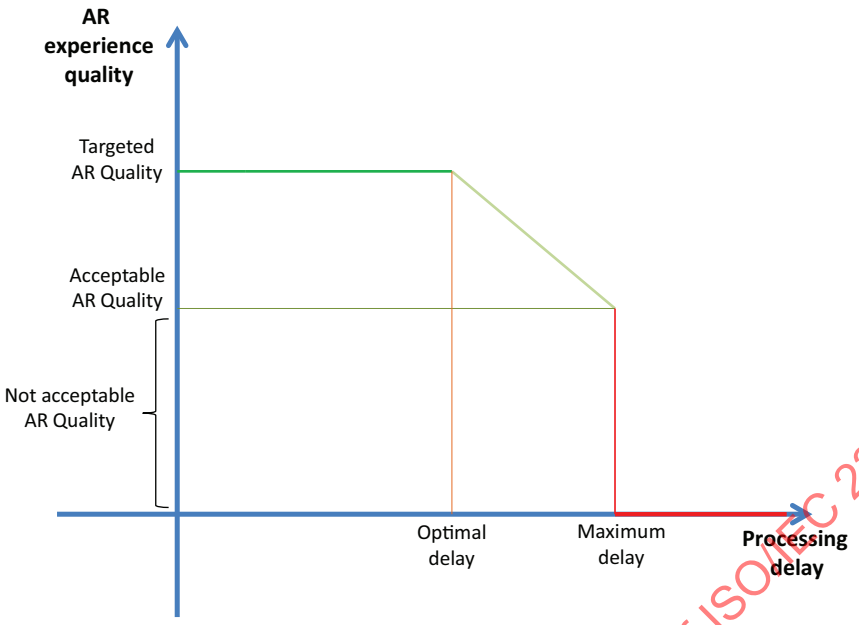


Figure 4 — LocImg: AR quality vs. processing delay

Further, the two fields implementing this functionality are presented.

maximumDelay is a SFInt32 value measured in milliseconds specifying which is the maximum acceptable delay of the recognition (and tracking) process in order for the MAR experience to be presented by an ARAF browser. The MAREC expects an answer from the recognition (and tracking) process every, at most, *maximumDelay* milliseconds.

optimalDelay is a SFInt32 value measured in milliseconds specifying which is the optimal delay of the recognition (and tracking) process. By setting this field, the MAREC suggests that there is no need in trying to provide a recognition (and tracking) response with a higher frequency (lower delay) because the MAR experience quality is the desired one.

recognitionRegion a MFVec2f field specifying two 2D points that are relative to the center of the video frame on which the recognition (and tracking) algorithm is performed. The first point indicates the lower left coordinate and the second one the upper right coordinate of a rectangle. By using this field, the MAREC suggests that only the inside area given by the rectangle has to be used in the recognition (and tracking) process, not the entire video frame. The recognition (and tracking) process can be improved by using a video frame region rather than the whole video frame but on the other hand the way how the original video frame is pre-preprocessed (e.g. cropped) may introduce delays. The ARAF browser cannot ensure that by using a recognition region the overall processing speed is improved.

onRecognition is an output event of type MFInt32 specifying the indexes of the target resources that have been recognized. An index is an integer value representing the position of a target resource in the *targetResources* array (0 indexed). The index of the first target resource is 0 and it is incremented by one for each next target resource. If a target resource is a file containing descriptors for several images, each target image descriptor within the descriptor file is assigned the next index as they were separately specified.

The following two fields are used to describe the **pose matrix** of a recognized resource. A pose matrix describes the relative position to the camera viewpoint of a recognized target resource. The pose matrix is described by one rotation and one translation vector, fields which are described below. These two fields are optional, meaning that the functionality of the prototype can be limited to only recognizing target resources, without computing their associated pose matrixes. If *onTranslation* is not used,

then implicitly *onRotation* is not used and vice-versa. If a translation is computed, then implicitly the corresponding rotation should be computed by the processing server. In other words, if the processing server is capable of computing the pose matrix, the MAREC expects that both of the fields (*onTranslation* and *onRotation*) are set.

onTranslation is an exposed MFVec3f field where the translations of the recognized target resources are stored. A SFVec3f vector specifies where the corresponding recognized target resource is relative to the camera position within the video frame where the recognition process has been performed on. The default value of a translation vector is <0,0,0>. MAREC expects a SFVec3f translation for each recognized target resource or a default value if the translation could not be computed. The MAREC considers that the n^{th} value of *onTranslation* corresponds to the target resource given by the value found on the n^{th} index of *onRecognition*. The field is optional.

onRotation is a exposed MFRotation field where the rotations of the recognized target resources are stored. A SFRotation vector specifies how the corresponding recognized target resource is rotated with respect to the camera plane within the video frame. The default value of a rotation vector is <0,0,0,0>. The MAREC expects a SFRotation vector for each recognized target resource or a default value if the rotation could not be computed. The n^{th} value of *onRotation* corresponds to the target resource given by the value found on the n^{th} index of *onRecognition*. The field is optional.

onRotation, *onTranslation* and *onRecognition* shall have the same lengths.

onError is an output event of type SFInt32.

Table 10 specifies onError possible values and their meaning.

Table 10 — LocImg: error codes

onError	Description
–1	The video source URL is invalid or not supported.
–2	At least one target resource is invalid or not supported. This error can be triggered in the cases when the ARAF browser is not able to read/access any of the target resources or the processing library does not support the format of at least one target resource.
–3	The video frame format is not supported by the recognition (and tracking) library.
–4	Unavailable recognition (and tracking) library for at least one target resource that has been specified by the MAREC.
–5	Unknown error

5.1.4.12.4.4 RemImgProxy (Remote image recognition registration proxy)

The MAREC provides a set of target resources, a video source URL and one or multiple ARAF compliant processing server URLs where recognition (and tracking) libraries are available. The ARAF browser communicates with any of the processing servers provided by the MAREC, sends the target resources and the video frames and receives the recognition (and tracking) result. The ARAF browser composes the server result in the expected format before sending it to the ARAF scene, as described in the **Functionality and Semantics** below.

Even though in Figure 5, the augmentation media is presented as input along other data provided by the MAREC, the augmentation media is not used by the ReImgProxy PROTO implementation, meaning that it is not part of any of the PROTO fields. The augmentation resources are of course required for the scene augmentation but they do not influence the recognition process.

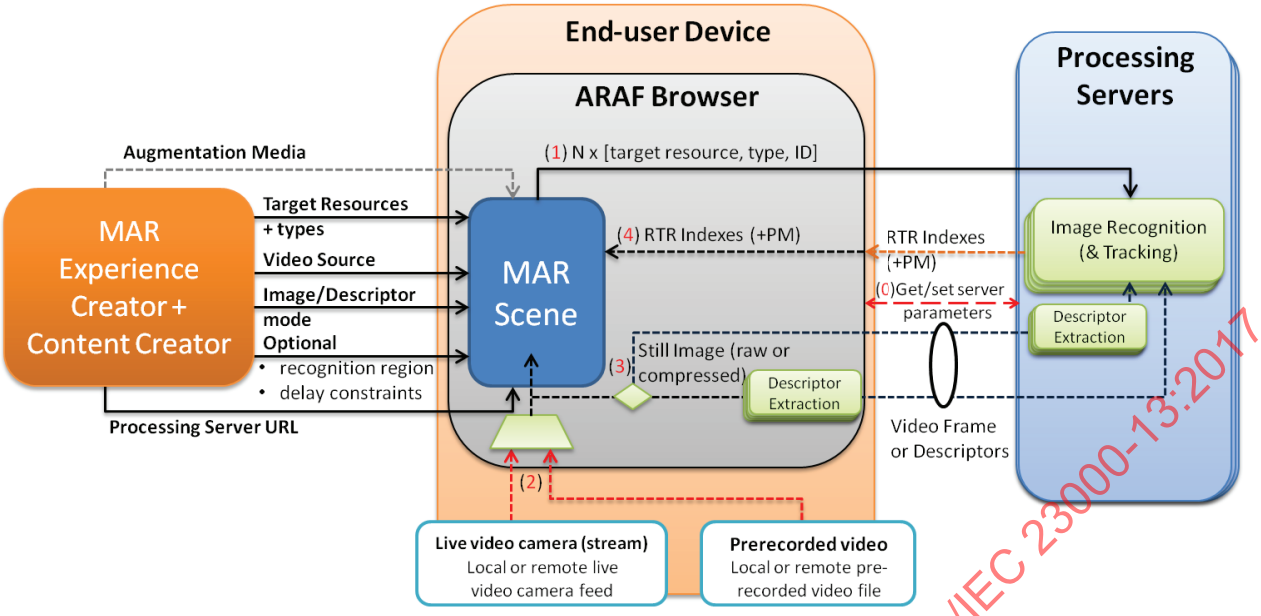


Figure 5 — Remote image recognition registration proxy

5.1.4.12.4.5 BIFS Textual Description

```
EXTERNPROTO RemImgProxy [
    exposedField SFString   videoSource             ""
    exposedField MFString   processingServerURL     []
    exposedField SFInt32    frameEncodingType       0
    exposedField SFInt32    processingType          0
    exposedField MFString   targetResources         []
    exposedField MFString   targetResourcesTypes    []
    exposedField SFBool     enabled                 FALSE
    exposedField SFInt32    maximumDelay            200 #milliseconds
    exposedField SFInt32    optimalDelay            50 #milliseconds
    exposedField MFVec2f    recognitionRegion       []
    eventOut     MFInt32    onRecognition           []
    eventOut     MFVec3f    onTranslation           []
    eventOut     MFRotation onRotation              []
    eventOut     SFInt32    onError
]"org:mpeg:remote_image_recognition_registration"
```

5.1.4.12.4.6 Functionality and semantics

The MAREC provides one or multiple processing server URLs where recognition (and tracking) libraries are available, along with the target resources to be recognized and the video source where the recognition process shall be performed on. The ARAF browser uses the provided processing servers as an external resource that is able to perform the recognition (and tracking) of the target resources. The video frames are sent to the processing server in a format that is specified by the MAREC, otherwise, the best suitable format is chosen by the ARAF browser. The recognition (and tracking) result received by the ARAF scene is an array of integers representing the indexes of the recognized target resources and optionally their pose matrixes as described in the description of the fields below.

An ARAF compliant processing server shall understand the HTTP requests presented in Table 11.

Table 11 — RemImgProxy: communication workflow

ARAF browser request	Request type	Description	Processing server response	Description
pServer/alive	GET	Get the unique key, the server parameters and capabilities	unique key 64-bit, target resource code, video frame code, server capability codes	The unique key shall be used to indentify future requests from the client. The target resource code specifies the formats of the target resources that are supported by the server. The video frame code specifies the formats of the video frame that are supported by the server. The ARAF browser decides how the video data will be encoded before is sent to the server by considering this response and the MAREC preferences (see *frameEncodingType*). The server capability code informs the ARAF browser about the type of the processing that the server is able to perform.
pServer *key&frame_format(&intrinsic camera params)*	POST	Inform the processing server about the chosen video frame format and optionally send the intrinsic camera parameters	True/false	The server reponse is True if the data is correctly received and False otherwise
pServer *key&target&type&id*	POST	Send each target resource along type code and the unique target resource id	True/false	The server reponse is True if the data is correctly received and False otherwise
pServer *key&frame*	POST	Send a new video frame to the server	identified, rotation, translation	The server reponse is a list containing recognized target resources ids and optionally their pose matrixes (rotation and translation).

Communication workflow:

a) The ARAF browser interrogates the processing server (GET */alive)* in order to detect its status and to receive the server parameters and capabilities. The server returns:

   1) a unique key that shall be transmitted by ARAF browser in future requests,

   2) the list of **codes** describing the supported target resources formats (file types or image descriptors). See Table 7 and Table 8 for the supported codes and their meaning.

   3) the list of **codes** describing the supported video frame formats (file types or image descriptors). See Table 7 and Table 8 for the supported codes and their meaning.

   4) the list of **codes** describing the sever capabilities. See Table 12 for the supported codes and their meaning.

   The server response shall be in the following format:

   key=unique key 64-bit

   &resource_code=[target resource format codes];

   &frame_code=[video frame format codes];

   &server_capability_code=[server capability codes]

Example of a possible server response:

key=2e45325f4f&resource_code=0,1&frame_code=0&server_capability_code=0,5

b) Once the key has been received, the ARAF browser knows that the processing server is ready to perform the recognition (and tracking) process. The ARAF browser decides on one video frame encoding type, based on the user preference (if specified) and on the server's response then it informs the processing server about the chosen format. The video frames of this session are sent only in the specified encoding type. In addition to the video frame format, the ARAF browser chooses one of the available capabilities to be performed by the processing server. Therefore, a capability code (see Table 12) has to be transmitted to processing server along with the unique key and the chosen video frame format. Optionally the intrinsic camera parameters (if available) are sent to the image processing server as described in 5.1.4.12.1.

The second type of request contains the target resources provided by the MAREC (one request per each target resource). Each resource shall have associated a unique ID, the one to be returned by the processing server when it is recognized, and the file format of the resource. The POST requests are as follows:

1) Send the chosen video frame format:

key=unique_key

&frame_code=the chosen encoding format code

&server_capability_code=one of the available server capabilities code

&fl=the 2d vector representing the focal length (comma separated float values, optional)

&cc=the 2d vector representing the principal point (comma separated float values, optional)

&alpha_c=the angle representing the skew coefficient (float, optional)

2) Every target resource is sent separately using a POST request:

key=unique_key

&target_resource=the target resource data

&type_code=the target resource encoding type

&id=the id of the target resource uniquely identifying the target resource in the MAR scene

The processing server returns TRUE if the data is correctly received or FALSE otherwise.

c) The ARAF browser sends a video frame to the processing server. The video frame has to be sent in the exact format that the processing server has been previously informed.

key=unique_key

&frame_data=the camera frame

The processing server's response is a list of IDs of the recognized target resources and optionally their corresponding pose matrixes or FALSE if no target resource is recognized.

identified=comma separated ids of the recognized target resources

&translation=[x,y,z]; //optional

&rotation=[x,y,z,q]; //optional

Where

— *identified* is a list of integer values separated by commas

— *translation* contains groups of 3 float values separated by semicolon. Each value is separated by comma.

— *rotation* contains groups of 4 float values separated by semicolon. Each value is separated by comma.

If the processing server does not have tracking capabilities, then the response will be composed only from the list of the recognized target resources (*identified*).

d) The loop starts over from point 3 whenever the ARAF browser has to send a new video frame data to the processing server.

videoSource is a SFString specifying the URI/URL of the video where the recognition (and tracking) process shall be performed on. The *videoSource* can be one of the following:

e) Live 2D video camera feed:

1) a URI to one of the cameras available on the end user's device. The possible values are specified in Table 6;

2) a URL to an external camera providing live camera feed;

f) A URL to a prerecorded video file stored:

— locally on the end user's device;

— remotely on an external repository in the Web.

Based on the MAREC preferences, the video frames are sent to the recognition (and tracking) library every X milliseconds (the ARAF browser is in charge of computing the frequency), as long as the recognition (and tracking) process is enabled. The video frames are sent as compressed images or raw data (see Table 7 for the supported image file formats), or as descriptor files (see Table 8) depending on the processing server capabilities. The ARAF browser is in charge of deciding the encoding type of the video frames that shall be sent to the processing server, considering the MAREC preferences and the capabilities of the server.

The accepted video formats are specified in Table 5.

The accepted communications protocols are specified in Table 4.

One or multiple codes presented in Table 12 might be sent by the processing server to the ARAF browser (see the first step described in Table 11). The purpose is to inform the ARAF browser about the processing capabilities of the server. On the other hand, the MAREC can use the dedicated prototype field (*processingType*) to express his preferences related to the processing type that he expects from the server by specifying one of the codes defined in this table. The ARAF browser finally decides what type of processing the server should perform, based on the server capability and the MAREC preferences. The server returns different responses based on the chosed processing type (see Table 11 field description).

Table 12 — RemImgProxy: server capability codes

Capability code	Description
0	**Recognition only.** The processing server is capable of performing image recognition only. This means that the server response contains information about the indexes of the recognized target resources, as defined in the description of the fields.
1	**Recognition and tracking.** The processing server is capable of performing image recognition and tracking. This means that the server response contains information about the indexes of the recognized target resources along with their pose matrixes (the computed ones), as defined in the description of the fields.

processingServerURL is a MFString used by the MAREC to specify one or multiple web addresses where ARAF compliant processing servers are available. A valid URL is one that points to a processing server that handles at least one target resource type and is able to understand ARAF browser requests, as defined in Table 11. Because a processing server can handle requests from multiple clients in the same time, a unique key is generated by the server and transmitted to the ARAF browser. The ARAF browser sends the unique generated key in each request to the processing server. This way, the processing server knows the source of the request and therefore it can perform the recognition (and tracking) process on the correct set of the target resources.

frameEncodingType field is an MFInt32 containing an array of *video frame type codes*. The MAREC has the possibility to specify the desired encoding type of the video frames that are sent to the processing server. If multiple keywords are specified by the MAREC, the ARAF browser chooses the first encoding type that matches the server capabilities. The possible pre-defined codes of the frame encoding types and their meaning are listed in Table 7 and Table 8. If the MAREC does not specify any encoding type code, the ARAF browser uses a default one. The MAREC should not specify an encoding type unless he knows that the processing server gives better results with one or another. The ARAF browser in any case interrogates the server (see the first step described in Table 11) to retrieve the supported encoding type codes and then it decides on one encoding type considering the MAREC preferences. The field is optional.

The **processingType** field is a SFInt32 value where the MAREC can specify his preferences related to the tracking capabilities of the server as follows: if the field is 1, the ARAF browser will request the processing server to provide the pose matrix of the recognized target resource. If the processing server is not capable of computing the pose matrix then the related fields (onTranslation and onRotation) will be empty, disregarding the MAREC request. By setting a 0 value on the tracking field, the MAREC suggests that the processing should not perform image tracking at all, because the result it's not going to be used in the MAR Scene. The ARAF browser informs the processing server about the MAREC preferences using a POST request as described in Table 11.

targetResources is an MFString where the target resources to be recognized (and tracked) within the MAR experience are specified. A URI can point to a local or remote resource file. The accepted communication protocols for the remote resources are the ones specified in Table 11. Any of the below combinations describes a valid *targetResources* assignment:

— URIs pointing to target images. The file formats specified in Table 7 are accepted.

— URIs pointing to files where target image descriptors are found. A file contains descriptors of one single target image. The file formats specified in Table 8 are accepted.

— URLs pointing to files where multiple target image descriptors are found. One file contains descriptors of multiple target images. The file formats specified in Table 8 are the supported ones.

— Any combination of the cases described above can coexist in the same *targetResources* field.

The **targetResourcesTypes** field is an MFString containing an array which specifies the type of each target resource defined in the targetResources field. Each target resource shall have associated a type in order for the ARAF browser to know how to interpret the data. The possible pre-defined keywords of the *targetResourcesTypes* and their meaning are listed in Table 7 and Table 8. If the target resource is a proprietary descriptor file and in addition to the actual image descriptors data the proprietary recognition (and tracking) library needs some other data (for e.g. an XML file), the related content should be stored in a directory that has the same name as the resource type (for e.g. *target_resource_type_keywod/*) in order for the ARAF browser to know where the required files can be found. The names of the files within the directory have to be the same name as the corresponding descriptor file name.

enabled is a SFInt32 value indicating if the recognition (and tracking) process is enabled (running). MAREC can control the status of the recognition (and tracking) process or he can let the ARAF browser to decide whether the recognition (tracking) process should be running or not. Table 13 specifies the supported integer values of the *enabled* field.

Table 13 — RemImgProxy: enabled

enabled	Description
–1	ARAF browser decides when the recognition (and tracking) process is enabled/disabled. If not supported, the recognition process is always disabled unless a value of 0 or 1 is set by the MAREC.
0 (default)	The recognition (and tracking) process is disabled.
1	The recognition (and tracking) process is enabled.

A value of –1 specifies that the ARAF browser decides the status of the recognition (and tracking) process.

The recognition (and tracking) process is inactive while *enabled* is 0.

While *enabled* is 1, the following case are differentiated based on the *video source*.

— *local live video camera feed*: the frames coming from the local live video camera feed are considered by the ARAF browser in the recognition (and tracking) process.

— *remote live video camera feed*: the frames coming from the remote live video camera stream are considered by the ARAF browser in the recognition (and tracking) process. Technically the only difference between the first case and the second one is the source of the video frames. In this case, a streaming protocol should be used to fetch the remote video camera stream.

— *local prerecorded video file*: as long as *enabled* is 1, the ARAF browser plays the video file and the corresponding video frames are used in the recognition (and tracking) process. Whenever *enabled* is 0 the video play back is paused. On 1, the video starts playing from the point where it was last paused. The video play back starts from the beginning when the end of the video stream is reached and *enabled* is 1.

— *remote prerecorded video file*: idem as in the previous case except that the remote file has to be downloaded first. If a streaming protocol is being used, the ARAF browser may request (if possible) video frames whenever *enabled* is 1, as it would play back the video remotely.

MAREC should have the possibility to choose the quality of his *MAR experience* and in the same time, indirectly, the processing power consumed by the recognition (and tracking) process. MAREC can control this by setting a *maximum acceptable delay*. As described in Figure 6, a response time higher than the maximum delay indicates an unacceptable quality of the MAR experience therefore an ARAF browser should not present it. Any response time with a delay lower than the specified maximum delay produces a MAR experience that is at least acceptable from the point of view of the MAREC, therefore an ARAF browser should present the MAR experience. The MAREC can also specify an optimal delay constraint informing an ARAF browser that there is no need in trying to provide recognition (and tracking) response with a higher frequency (lower delay) becuase the MAR experience has already reached the targeted quality.

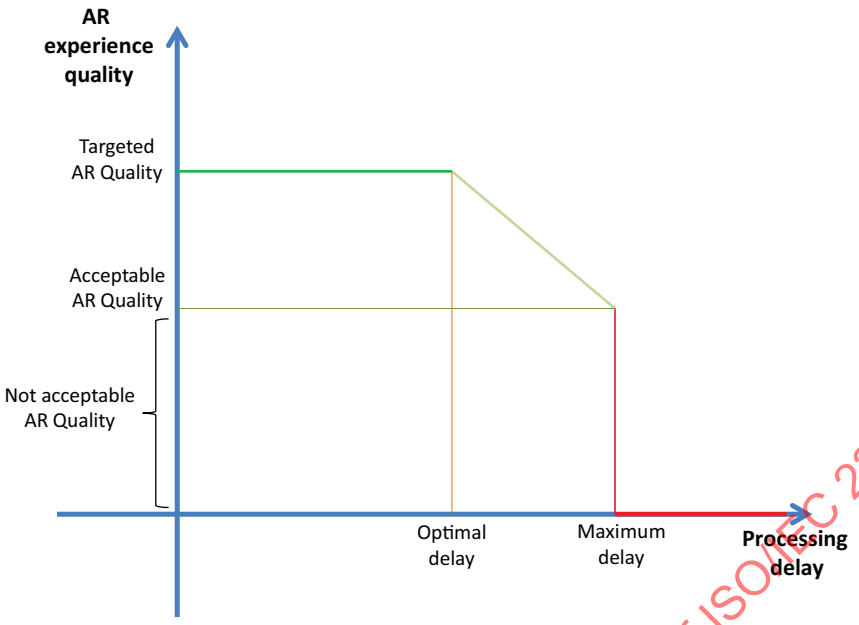


Figure 6 — RemImgProxy: AR quality vs. processing delay

Further, the two fields implementing this functionality are presented.

maximumDelay is a SFInt32 value measured in milliseconds specifying which is the maximum acceptable delay of the recognition (and tracking) process in order for the MAR experience to be presented by an ARAF browser. The MAREC expects an answer from the recognition (and tracking) process every, at most, *maximumDelay* milliseconds.

optimalDelay is a SFInt32 value measured in milliseconds specifying which is the optimal delay of the recognition (and tracking) process. By setting this field, the MAREC suggests that there is no need in trying to provide a recognition (and tracking) response with a higher frequency (lower delay) because the MAR experience quality is the desired one.

recognitionRegion a MFVec2f field specifying two 2D points that are relative to the center of the video frame on which the recognition (and tracking) algorithm is performed. The first point indicates the lower left coordinate and the second one the upper right coordinate of a rectangle. By using this field, the MAREC suggests that only the inside area given by the rectangle has to be used in the recognition (and tracking) process, not the entire video frame. The recognition (and tracking) process can be improved by using a video frame region rather than the whole video frame but on the other hand the way how the original video frame is pre-preprocessed (e.g. cropped) may introduce delays. The ARAF browser cannot ensure that by using a recognition region the overall processing speed is improved.

onRecognition is an output event of type MFInt32 specifying the indexes of the target resources that have been recognized. An index is an integer value representing the position of a target resource in the *targetResources* array (0 indexed). The index of the first target resource is 0 and it is incremented by one for each next target resource. If a target resource is a file containing descriptors for several images, each target image descriptor within the descriptor file is assigned the next index as they were separately specified.

The following two fields are used to describe the **pose matrix** of a recognized resource. A pose matrix describes the relative position to the camera viewpoint of a recognized target resource. The pose matrix is described by one rotation and one translation vector, fields which are described below. These two fields are optional, meaning that the functionality of the prototype can be limited to only recognizing target resources, without computing their associated pose matrixes. If *onTranslation* is not used,

then implicitly *onRotation* is not used and vice-versa. If a translation is computed, then implicitly the corresponding rotation should be computed by the processing server. In other words, if the processing server is capable of computing the pose matrix, the MAREC expects that both of the fields (*onTranslation* and *onRotation*) are set.

onTranslation is an exposed MFVec3f field where the translations of the recognized target resources are stored. A SFVec3f vector specifies where the corresponding recognized target resource is relative to the camera position within the video frame where the recognition process has been performed on. The default value of a translation vector is <0,0,0>. MAREC expects a SFVec3f translation for each recognized target resource or a default value if the translation could not be computed. The MAREC considers that the n^{th} value of *onTranslation* corresponds to the target resource given by the value found on the n^{th} index of *onRecognition*. The field is optional.

onRotation is a exposed MFRotation field where the rotations of the recognized target resources are stored. A SFRotation vector specifies how the corresponding recognized target resource is rotated with respect to the camera plane within the video frame. The default value of a rotation vector is <0,0,0,0>. The MAREC expects a SFRotation vector for each recognized target resource or a default value if the rotation could not be computed. The n^{th} value of *onRotation* corresponds to the target resource given by the value found on the n^{th} index of *onRecognition*. The field is optional.

onRotation, *onTranslation* and *onRecognition* shall have the same lengths.

onError is an output event of type SFInt32.

Table 14 specifies onError possible values and their meaning.

Table 14 — RemImgProxy: error codes

onError	Description
–1	The video source URL is invalid or not supported.
–2	At least one target resource is invalid or not supported. This error can be triggered in the cases when the ARAF browser is not able to read/access any of the target resources or the processing server does not support the format of at least one target resource.
–3	None of the available video frame formats are supported by the processing server. In other words the ARAF browser is not capable of sending the video frame to the processing server in one of the expected formats.
–4	Unavailable recognition (and tracking) library for at least one target resource that has been specified by the MAREC.
–5	Unknown error

5.1.4.12.4.7 RemImgServer (remote image recognition registration server)

MAREC provides a **video source** URL and one or multiple **processing server** URLs where image recognition (and tracking) libraries are available. A set of target resources (images or descriptors) and the associated augmentation resources are stored in remote databases, on the processing server or anywhere else on the Web. The ARAF browser sends video frames or video frame descriptors to the processing server, considering the MAREC preferences (if applicable) and the processing server capabilities. The server performs the recognition (and tracking) process on the received video data, searching in the set of the target resources that are stored in the database(s). Depending on the format of the video frame, the server may need to extract the descriptors before performing the recognition (and tracking) process. One of the server's responses is composed by one or multiple augmentation resource URLs that are associated with the recognized target resource and (optionally) the pose matrix of the recognized object within the video frame. The augmentation resource can be an URL to a media file (see Table 18) or a proprietary string. The proprietary string is not interpreted by the default prototype implementation therefore the MAREC has to update the default prototype implementation. The MAREC has the option to let the ARAF browser do the tracking of the recognized target resource locally. In this case the processing server shall be able to send the recognized target resource to the

ARAF browser which in turn shall be able to perform the tracking. Detailes are presented below in the **Functionality and Semantics**.

NOTE The main difference between Remote Image Recognition Registration Proxy and Remote Image Recognition Registration Server is that in the second case the Target Resources are not knwon by the MAREC but they are already stored in remote databases; therefore, the MAREC does not have to provide URIs/URLs to Target Resources.

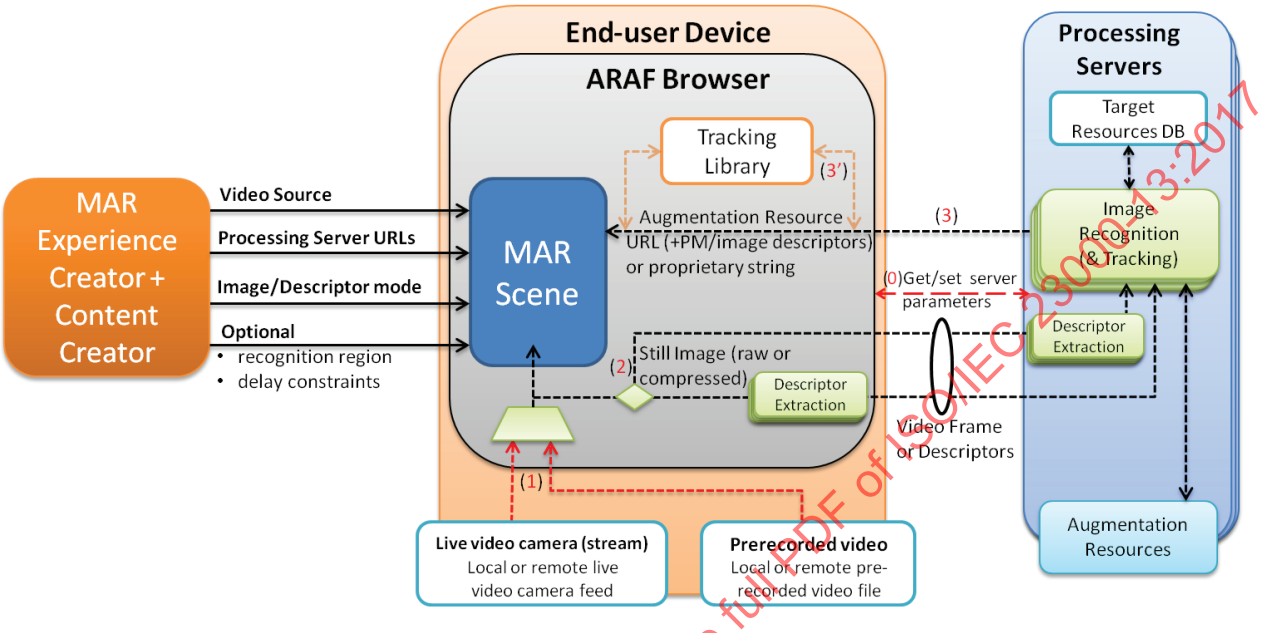


Figure 7 — Remote image recognition registration server

5.1.4.12.4.8 BIFS textual description

```
EXTERNPROTO RemImgServer [
    exposedField SFString   videoSource          ""
    exposedField SFInt32    frameEncodingType    0
    exposedField SFInt32    processingType       0
    exposedField MFString   processingServerURL  []
    exposedField SFBool     enabled              FALSE
    exposedField SFInt32    maximumDelay         200 #milliseconds
    exposedField SFInt32    optimalDelay         50 #milliseconds
    exposedField MFVec2f    recognitionRegion    []
    eventOut     MFString   augmentationMediaURL
    eventOut     MFInt32    augmentationMediaType
    eventOut     MFString   augmentationMediaString
    eventOut     MFVec3f    onTranslation
    eventOut     MFRotation onRotation
    eventOut     SFInt32    onError
]"org:mpeg:remote_image_recognition_registration_server"
```

5.1.4.12.4.9 Functionality and semantics

MAREC provides one or multiple processing server URLs where recognition (and tracking) libraries are available, along with a video source where the recognition process shall be performed and optionally the encoding code of the video frames that are sent to processing server. The ARAF browser shall use the provided processing server URLs as an external resource that is able to perform the recognition (and tracking) of the target resources that are stored in the external repositories. The recognition (and tracking) result of the processing server is an URL to an augmentation resource and the type of the media file that can be found at the given URL and optionally the pose matrix of the recognized object in the video frame. Another valid server's response is a proprietary string that is associated to the recognized object in the video frame. Considering that the processing server knows which was the last recognized target resource for a given client, a tracking algorithm can be performed remotely. In the case that the processing server is able to only perform image recognition, the ARAF browser can choose

to receive the target image descriptors from the processing server in order to perform the tracking locally. This case would be considered if only the MAREC sets the correct code in the *processingType* field (see Table 16) and the ARAF browser is capable of performing tracking of a target resource. The possible scenarious are described throughout the fields' descriptions below.

An ARAF compliant processing server shall understand the HTTP requests presented in Table 15.

Table 15 — RemImgServer: communication workflow

ARAF browser request	Request type	Description	Processing server response	Description
pServer/alive	GET	Get the unique key, the server parameters and capabilities	unique key 64-bit, video frame code, server capability code, image descriptor code	The unique key shall be used to indentify future requests from client. The video frame code specifies the formats of the video frame that are supported by the server. The ARAF browser decides how the video data will be encoded before is sent to the server by considering this response and the MAREC preferences (see *frameEncodingType*). The server capability code informs the ARAF browser about the type of the processing that the server is able to perform. The image descriptor code informs the ARAF browser about the data format of the target image descriptor that might be sent to the ARAF browser (for local tracking).
pServer *key&frame_format(&intrinsic camera params)*	POST	Inform the processing server about the chosen video frame format and optionally send the intrinsic camera parameters	True/false	The server reponse is True if the data is correctly received and False otherwise
pServer *key&frame*	POST	Send a new video frame to the server	recognized, rotation, translation	The server reponse is a URL to an augmentation resource and its pose matrix (rotation and translation) if applicable.

Communication workflow:

a) ARAF browser interrogates the Processing Server (GET */alive)* in order to detect its status and to receive the server parameters and capabilities. The server returns:

   1) a unique key that shall be transmitted by ARAF browser in future requests;

   2) the list of **codes** describing the supported video frame formats (file types or image descriptors). See Table 7 and Table 8 for the supported codes and their meaning;

   3) the list of **codes** describing the sever capabilities. See Table 16 capability codes for the supported codes and their meaning;

   4) the **code** of the image descriptor format that might be sent to the ARAF browser based on the MAREC preferences (see *processingType* field description). The ARAF browser shall know if its local tracking library is capable of interpreting the image descriptors data received from the processing. See Table 8. If applicable, the processing server sends image descriptors to the ARAF browser (for local tracking) in the specified format.

The server response shall be in the following format:

key=unique key 64-bit

&frame_code=[video frame format codes];

&server_capability_code=[server capability codes]

&image_descriptor_code=[the codes specifying supported imaget descriptor formats]

Example of a possible server response:

key=2e45325f4f&frame_code=0&server_capability_code=1&image_descriptor_code=5

b) Once the key has been received, the ARAF browser knows that the processing server is ready to perform the recognition (and tracking) process. The ARAF browser decides on one video frame encoding type, based on the user preference (if specified) and on the server's response then it informs the processing server about the chosen format. The video frames of this session are sent only in the specified encoding type. In addition to the video frame format the ARAF browser chooses one of the available capabilities to be performed by the processing server. Therefore, a capability keyword (see Table 16) has to be transmitted to processing server along with the unique key and the chosen video frame format. Optionally the intrinsic camera parameters (if available) are sent to the image processing server as described in 5.1.4.12.1. The POST request is in the following format:

key=unique_key

&frame_code=the chosen encoding format code

&server_capability_code=one of the available server capabilities code

&fl=the 2d vector representing the focal length (comma separated float values, optional)

&cc=the 2d vector representing the principal point (comma separated float values, optional)

&alpha_c=the angle representing the skew coefficient (float, optional)

The processing server returns TRUE if the data is correctly received or FALSE otherwise.

c) The ARAF browser sends a video frame to the processing server. The video frame is transmitted in the format that the processing server has been previously informed.

key=unique_key

&frame_data=the camera frame

The processing server's response can be:

— a URL to an augmentation resource that corresponds to the recognized target image and its corresponding pose matrix (if applicable) or the recognized target resource descriptors (if applicable) or FALSE if no resource is recognized.

resource=url to an augmentation resource

&descriptor=image descriptor in the previously specified format //optional

&translation=x,y,z //optional

&rotation=x,y,z,q //optional

— a proprietary string

If the processing server does not implement tracking capabilities then the response will be composed only by the augmentation resource URL, unless the MAREC specified that he prefers the ARAF browser to perform the tracking locally (see the description of the *processingType* field).

The case where the processing server returns a proprietary string is not covered by this document. The MAREC should know how the string has to be interpreted, because he is the one providing the server URL. In this case, the server response is transmitted to the MAR Scene exactly in the form as it comes.

e) The loop starts over from point 3 whenever the ARAF browser has to send a new video frame data to the processing server.

videoSource is a SFString specifying the URI/URL of the video where the recognition (and tracking) process shall be performed on. The *videoSource* can be one of the following:

f) Live 2D video camera feed

   1) a URI to one of the cameras available on the end user's device. The possible values are specified in Table 6;

   2) a URL to an external camera providing live camera feed;

g) A URL to a prerecorded video file stored

   — locally on the end user's device;

   — remotely on an external repository in the Web.

Based on the MAREC preferences, the video frames are sent to the processing server every X milliseconds (the ARAF browser is in charge of computing the frequency), as long as the recognition (and tracking) process is enabled. The video frames are sent as compressed images or raw data (see Table 7 for the supported image file formats), or as descriptor files (see Table 8) depending on the processing server capabilities. The ARAF browser is in charge of deciding the encoding type of the video frames that shall be sent to the processing server, considering the MAREC preferences and the capabilities of the server.

The accepted video formats are specified in Table 5.

The accepted communications protocols are specified in Table 4.

One or multiple codes presented in Table 16 might be sent by the processing server to the ARAF browser (see the first step described in Table 15 between the ARAF browser and the processing server). The purpose is to inform the ARAF browser about the processing capabilities of the server. On the other hand, the MAREC can use the dedicated prototype field (*processingType*) to express his preferences related to the processing type that he expects from the server by specifying one of the codes defined in this table. The ARAF browser finally decides what type of processing the server should perform, based on the server capability and the MAREC preferences. The server returns different responses based on the chosed processing type (see Table 15 and *processingType* field description).

Table 16 — RemImgServer: server capability codes

Capability code	Description
0	**Recognition only.** The processing server is capable of performing only image recognition. This means that the server response contains information about the indexes of the recognized target resources, as defined in the description of the fields.
1	**Recognition and tracking.** The processing server is capable of performing image recognition and tracking. This means that the server response contains information about the indexes of the recognized target resources along with their pose matrixes (the computed ones), as defined in the description of the fields.
2	**Recognition and image descriptors.** The processing server is capable of performing image recognition. The server response contains in addition to the recognition related information the descriptors of the recognized resource. This way the tracking can be performed by the ARAF browser locally.

processingServerURL is a MFString used by the MAREC to specify one or multiple web addresses where ARAF compliant processing servers are available. A valid URL is one that points to a processing server that handles at least one target resource type and is able to understand ARAF browser requests, as defined in Table 15. Because a processing server can handle requests from multiple clients in the same time, a unique key is generated by the server and transmitted to the ARAF browser. The ARAF browser sends the unique generated key in each request to the processing server. This way, the processing server knows the source of the request and therefore it can perform the recognition (and tracking) process on the correct set of the target resources.

frameEncodingType field is an MFInt32 containing an array of *video frame type codes*. The MAREC has the possibility to specify the desired encoding type of the video frames that are sent to the processing server. If multiple keywords are specified by the MAREC, the ARAF browser chooses the first encoding type that matches the server capabilities. The possible pre-defined codes of the frame encoding types and their meaning are listed in Table 7 and Table 8. If the MAREC does not specify any encoding type code the ARAF browser uses a default one. The MAREC should not specify an encoding type unless he knows that the processing server gives better results with one or another. The ARAF browser in any case interrogates the server (see the first step described in Table 15) to retrieve the supported encoding type codes and then it decides on one encoding type considering the MAREC preferences. The field is optional.

The **processingType** field is a SFInt32 value where the MAREC can specify his preferences related to the recognition and tracking processing performed on the server. The MAREC can choose one of the codes that are specified in Table 16. The description of the codes is found in the given table. The MAREC can only suggest what type of processing the server should perform implicitly affecting the server's response. The ARAF browser is in charge of informing the processing server what kind of job it should perform, based on the MAREC preferences and the server capabilities (Table 15 workflow explains how).

— 0: the MAREC suggests that he expects only the recognition related information from the processing server.

— 1: the MAREC suggests that he expects the recognition and tracking related information from the processing server.

— 2: the MAREC suggests that the processing server should only perform the recognition process while the tracking should be performed by the ARAF browser locally. In this case, the ARAF browser should be provided with a tracking library that is able to understand the descriptor data received from the server. If the ARAF browser is not capable of performing the tracking of the target resource, an error code should be triggered (see *onError* field description).

From the MAR scene point of view, the second and the third cases are providing the same data with the difference that in one case the pose matrix is computed remotely while in the other case the pose matrix is computed locally by the ARAF browser. The ARAF browser informs the processing server about the MAREC preferences using a POST request as described in the Table 15.

If the ARAF browser cannot perform the image tracking for any reason, an error code is sent to the MAR Scene.

enabled is a SFInt32 value indicating if the recognition (and tracking) process is enabled (running). MAREC can control the status of the recognition (and tracking) process or he can let the ARAF browser to decide whether the recognition (tracking) process should be running or not. Table 17 specifies the supported integer values of the *enabled* field.

Table 17 — RemImgServer: enabled

enabled	Description
–1	ARAF browser decides when the recognition (and tracking) process is enabled/disabled. If not supported, the recognition process is always disabled unless a value of 0 or 1 is set by the MAREC.
0 (default)	The recognition (and tracking) process is disabled.
1	The recognition (and tracking) process is enabled.

A value of –1 specifies that the ARAF browser decides the status of the recognition (and tracking) process.

The recognition (and tracking) process is inactive while *enabled* is 0.

While *enabled* is 1, the following cases are differentiated based on the *video source*:

— *local live video camera feed*: the frames coming from the local live video camera feed are considered by the ARAF browser in the recognition (and tracking) process.

— *remote live video camera feed*: the frames coming from the remote live video camera stream are considered by the ARAF browser in the recognition (and tracking) process. Technically, the only difference between the first case and the second one is the source of the video frames. In this case, a streaming protocol should be used to fetch the remote video camera stream.

— *local prerecorded video file*: as long as *enabled* is 1, the ARAF browser plays the video file and the corresponding video frames are used in the recognition (and tracking) process. Whenever *enabled* is 0 the video play back is paused. On 1, the video starts playing from the point where it was last paused. The video play back starts from the beginning when the end of the video stream is reached and *enabled* is 1.

— *remote prerecorded video file*: idem as in the previous case except that the remote file has to be downloaded first. If a streaming protocol is being used, the ARAF browser may request (if possible) video frames whenever *enabled* is 1, as it would play back the video remotely.

MAREC should have the possibility to choose the quality of his *MAR experience* and in the same time, indirectly, the processing power consumed by the recognition (and tracking) process. MAREC can control this by setting a *maximum acceptable delay*. As described in Figure 8, a response time higher than the maximum delay indicates an unacceptable quality of the MAR experience therefore an ARAF browser should not present it. Any response time with a delay lower than the specified maximum delay produces a MAR experience that is at least acceptable from the point of view of the MAREC, therefore, an ARAF browser should present the MAR experience. The MAREC can also specify an optimal delay constraint informing an ARAF browser that there is no need in trying to provide recognition (and tracking) response with a higher frequency (lower delay) becuase the MAR experience has already reached the targeted quality.

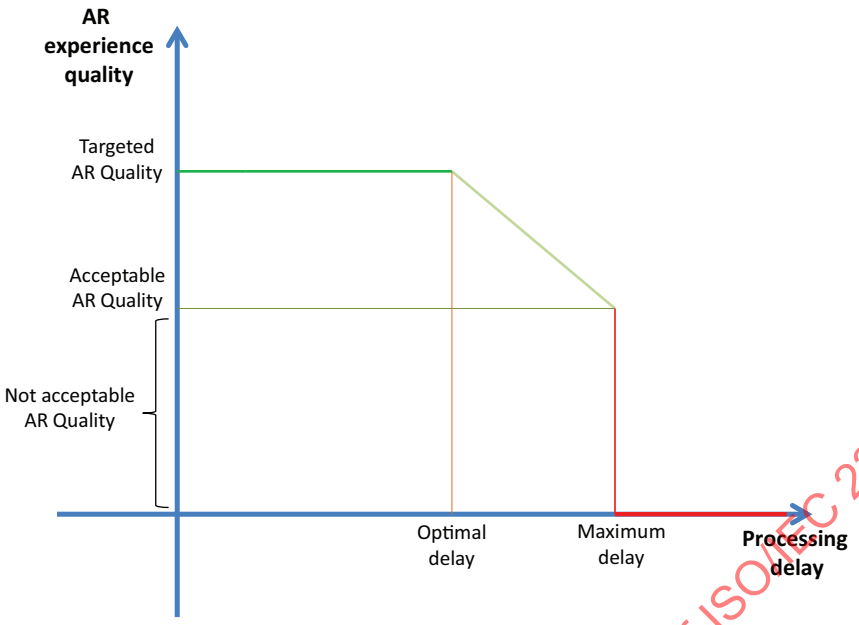


Figure 8 — RemImgServer: AR quality vs. processing delay

Further, the two fields implementing this functionality are presented.

maximumDelay is a SFInt32 value measured in milliseconds specifying which is the maximum acceptable delay of the recognition (and tracking) process in order for the MAR experience to be presented by an ARAF browser. The MAREC expects an answer from the recognition (and tracking) process every, at most, *maximumDelay* milliseconds.

optimalDelay is a SFInt32 value measured in milliseconds specifying which is the optimal delay of the recognition (and tracking) process. By setting this field, the MAREC suggests that there is no need in trying to provide a recognition (and tracking) response with a higher frequency (lower delay) because the MAR experience quality is the desired one.

recognitionRegion is a MFVec2f field specifying two 2D points that are relative to the center of the video frame on which the recognition (and tracking) algorithm is performed. The first point indicates the lower left coordinate and the second one the upper right coordinate of a rectangle. By using this field, the MAREC suggests that only the inside area given by the rectangle has to be used in the recognition (and tracking) process, not the entire video frame. The recognition (and tracking) process can be improved by using a video frame region rather than the whole video frame but on the other hand, the way how the original video frame is pre-preprocessed (e.g. cropped) may introduce delays. The ARAF browser cannot ensure that by using a recognition region, the overall processing speed is improved.

The following two fields are used to describe the **pose matrix** of a recognized resource. A pose matrix describes the relative position to the camera viewpoint of a recognized target resource. The pose matrix is described by one rotation and one translation vector, fields which are described below. These two fields are optional, meaning that the functionality of the prototype can be limited to only recognizing target resources, without computing their associated pose matrixes. If *onTranslation* is not used, then implicitly *onRotation* is not used and vice-versa. If a translation is computed, then implicitly the corresponding rotation should be computed by the processing server. In other words, if the processing server is capable of computing the pose matrix, the MAREC expects that both of the fields (*onTranslation* and *onRotation*) are set.

onTranslation is an exposed MFVec3f field where the translations of the recognized target resources are stored. A SFVec3f vector specifies where the corresponding recognized target resource is relative to the camera position within the video frame where the recognition process has been performed on. The default value of a translation vector is <0,0,0>. MAREC expects a SFVec3f translation for each recognized target resource or a default value if the translation could not be computed. The MAREC considers that the n^{th} value of *onTranslation* corresponds to the target resource given by the value found on the n^{th} index of *onRecognition*. The field is optional.

onRotation is a exposed MFRotation field where the rotations of the recognized target resources are stored. A SFRotation vector specifies how the corresponding recognized target resource is rotated with respect to the camera plane within the video frame. The default value of a rotation vector is <0,0,0,0>. The MAREC expects a SFRotation vector for each recognized target resource or a default value if the rotation could not be computed. The n^{th} value of *onRotation* corresponds to the target resource given by the value found on the n^{th} index of *onRecognition*. The field is optional.

onRotation, *onTranslation* and *onRecognition* shall have the same lengths.

augmentationMediaURL is an output event of type MFString where augmentation media URLs associated to the recognized target resource are referenced.

augmentationMediaTypes is an output event of type MFInt32 where the code of each augmentation media format referenced in the *augmentationMediaURL* field is specified. See Table 18 for the description of the supported types and their associated codes.

Table 18 — RemImgServer: augmentation media types

Augmentation resource type description	Augmentation resource keyword	Code
Image (see 5.1.4.12.3 for supported formats)	image	0
Video (see 5.1.4.12.3 for supported formats)	video	1
Audio (.wav, .mid, .mp3, .raw)	audio	2
BIFS scene (see ISO/IEC 14496-11)	2d/3d bifs scene (.mp4)	3

augmentationMediaString is a MFString field where the processing server can add textual information (e.g. labels, descriptions, etc.) associated to the recognized target resources. Because arbitrary data can be transmitted using this field, the MAREC should know the string format and therefore how to interpret the data. The server response is transmitted to the field exactly in the form as it comes. The proprietary string is not interpreted by the default prototype implementation.

onError is an output event of type SFInt32.

Table 19 specifies *onError* possible values and their meaning.

Table 19 — RemImgServer: error codes

onError	Description
–1	The video source URL is invalid or not supported.
–3	None of the available video frame formats are supported by the processing server. In other words the ARAF browser is not capable of sending the video frame to the processing server in one of the expected formats.
–5	Unknown error
–6	The ARAF browser is not capable of performing the tracking of the target resource.

5.1.4.12.4.10 RemImgComp (remote image recognition registration composition)

MAREC provides one or multiple ARAF compliant processing servers URLs and a video source URL where the recognition-tracking-composition process shall be performed. The ARAF browser sends video frames to processing server, which is detecting and recognizing target resources that are already

stored in remote database(s). The processing server is in charge of doing the composition between the video frames and the corresponding augmentation media of the recognized target resources. The composed video frames are sent back to the ARAF browser as described below in **Functionality and Semantics**.

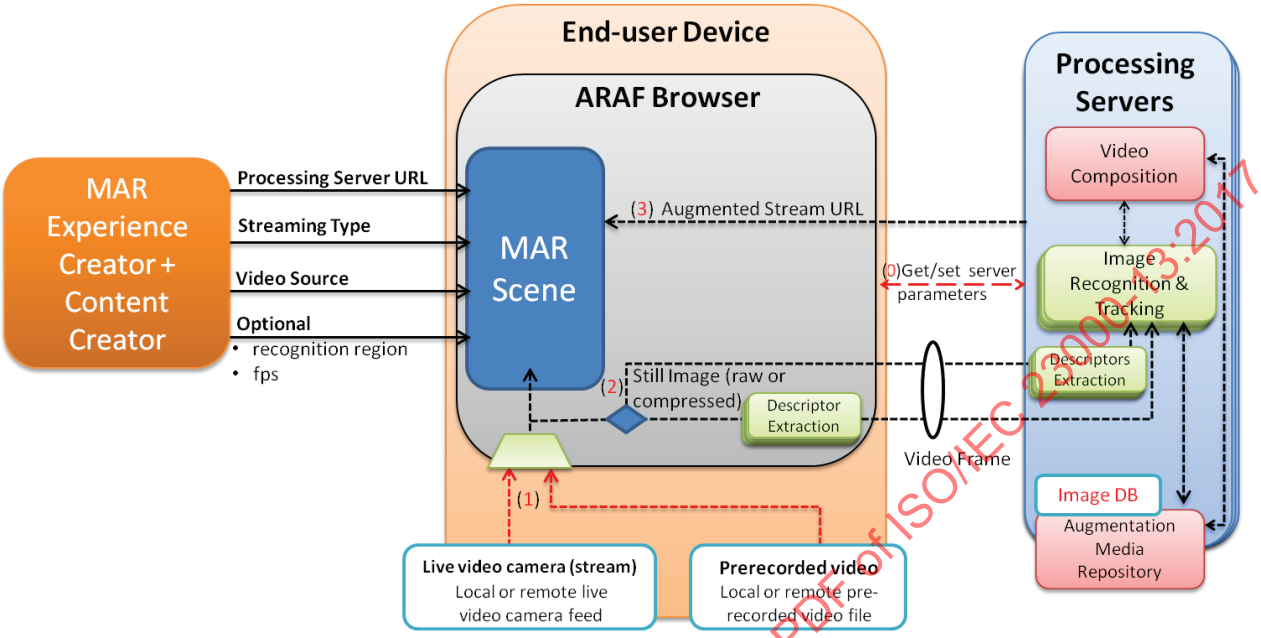


Figure 9 — Remote image recognition registration composition

5.1.4.12.4.11 BIFS Textual description

```
EXTERNPROTO RemImgComp [
    exposedField    SFString    videoSource             ""
    exposedField    SFInt32     streamingType           0
    exposedField    MFString    processingServerURL     []
    exposedField    SFBool      enabled                 FALSE
    exposedField    MFVec2f     recognitionRegion       []
    exposedField    SFInt32     fps                     -1
    exposedField    MFString    onAugmentedStream       []
    eventOut        SFInt32     onError
]"org:mpeg:remote_image_recognition_registration_composition"
```

5.1.4.12.4.12 Functionality and semantics

MAREC provides one or multiple ARAF compliant processing servers URLs and the video source URL that can be a local or remote real time capturing of a 2D camera or a locally/remotely stored, pre-recorded 2D video. The ARAF browser uses the provided processing server URLs as an external resource that is able to perform the recognition (and tracking) of the target resources that are already stored in remote databases, the registration and the composition of the video frames that are retrieved from the ARAF browser. The result of the processing server is a URL to the composed video stream where the corresponding augmentation media of the recognized target resources is overlaid.

A compliant Processing Server shall understand the HTTP requests presented in Table 20.

Table 20 — RemImgComp: communication workflow

ARAF browser request	Request type	Description	Processing server response	Description
pServer/alive	GET	Get the unique key and the server parameters	unique key 64-bit, communication protocol code	The key shall be used to indentify future requests from ARAF browser. The communication protocol code specifies the streaming communication protocols that are supported by the processing server. The ARAF browser decides which protocol is used by considering the server capabilities and the MAREC preference (see *streamingType*).
pServer *intrinsic camera params*	POST	Inform the processing server about the intrinsic camera parameters (if available)	True/False	The server reponse is True if the data is correctly received and False otherwise
pServer *key + video stream*	POST/ RTSP/ RTP/ DASH	Send the video stream to the server	Composed video stream	The server reponse is the composed video stream where the associated augmentation resources are overlaid on top of the original video.

Communication workflow:

a) ARAF browser interrogates the Processing Server (GET */alive)* in order to detect its status and to receive the server parameters. The server returns:

   1) a unique key that shall be transmitted by ARAF browser in future requests;

   2) the list of **codes** describing the supported streaming communication protocols. See Table 4 for the supported codes and their meaning.

The server response shall be in the following format:

key=unique key 64-bit

&comm_protocol_codes=[video frame format codes];

Example of a possible server response:

key=2e45325f4f&comm_protocol_codes=0

b) *[Optional step]* Once the key has been received, the ARAF browser knows that the processing server is ready to perform the recognition-tracking-composition process. If the intrinsic camera parameters are available, the ARAF browser should transmit these parameters to the server, otherwise this step may be skipped. The POST request is in the following format:

&fl=the 2d vector representing the focal length (comma separated float values, optional)

&cc=the 2d vector representing the principal point (comma separated float values, optional)

&alpha_c=the angle representing the skew coefficient (float, optional)

c) The ARAF browser decides on one communication protocol for streaming the video data, based on the user preference (if specified) and on the server's response.

The ARAF browser sends video frames to the processing server using the chosen streaming protocol. The server response is a URL pointing to the composed video stream where the associated augmentation resources are overlaid on top of the original video.

d) The loop starts over from point 3 whenever the ARAF browser has to initiate a new streaming connection with the processing server.

videoSource is a SFString specifying the URI/URL of the video where the recognition process shall be performed on. The videoSource can be one of the following:

e) Live 2D video camera feed

   1) a URI to one of the cameras available on the end user's device. The possible values are specified in Table 6;

   2) a URL to an external camera providing live camera feed;

f) A URL to a prerecorded video file stored

   — locally on the end user's device;

   — remotely on an external repository in the Web.

The accepted video formats are specified in Table 5.

The accepted communication protocols are specified in Table 4. **processingServerURL** is a MFString used by the MAREC to specify one or multiple web addresses where ARAF compliant processing servers are available. A valid URL is one that points to a processing server that is able to understand the ARAF browser requests and perform the recognition, tracking and composition of target resources as defined in the prototype description and in Table 20.

streamingType field is an MFInt32 used by the MAREC to specify the desired streaming protocol for the video data that is sent to the processing server. If multiple codes are specified by the MAREC, the ARAF browser chooses the first streaming protocol that matches the server capabilities. The possible pre-defined codes of the communication protocols and their meaning are listed in Table 4. If the MAREC does not specify any code the ARAF browser uses a default one, based on the processing server capabilities. The MAREC should not specify any code unless he knows that the processing server gives better results with one or another. The field is optional.

enabled is a SFInt32 value indicating if the recognition-tracking-composition process is enabled (running). MAREC can control the status of the process or he can let the ARAF browser to decide whether the recognition-tracking-composition process should be running or not. Table 21 specifies the supported integer values of the *enabled* field.

Table 21 — RemImgComp: enabled

enabled	Description
–1	ARAF browser decides when the recognition (and tracking) process is enabled/disabled. If not supported, the recognition process is always disabled unless a value of 0 or 1 is set by the MAREC.
0 (default)	The recognition (and tracking) process is disabled.
1	The recognition (and tracking) process is enabled.

A value of –1 specifies that the ARAF browser decides the status of the process.

The recognition-tracking-composition process is inactive while *enabled* is 0

While *enabled* is 1, the following cases are differentiated based on the *video source*.

— *local live video camera feed*: the frames coming from the local live video camera feed are considered by the ARAF browser in the recognition-tracking-composition process.

— *remote live video camera feed*: the frames coming from the remote live video camera stream are considered by the ARAF browser in the recognition-tracking-composition process. Technically, the only difference between the first case and the second one is the source of the video frames. In this case, a streaming protocol should be used to fetch the remote video camera stream.

— *local prerecorded video file*: as long as *enabled* is 1, the ARAF browser plays the video file and the corresponding video frames are used in the process. Whenever *enabled* is 0, the video play back is paused. On 1, the video starts playing from the point where it was last paused. The video play back starts from the beginning when the end of the video stream is reached and *enabled* is 1.

remote prerecorded video file: idem as in the previous case except that the remote file has to be downloaded first. If a streaming protocol is being used, the ARAF browser may request (if possible) video frames whenever *enabled* is 1, as it would play back the video remotely.

MAREC has the possibility of choosing the quality of his MAR experience by imposing a desired number of frames received per second. MAREC controls this by setting the *fps* field. The ARAF browser shall not present the composed video stream received from the processing server as long as the requirement imposed by MAREC is not fullfilled, meaning that the number of frames per second of the composed video stream is lower than the value of *fps* that has been set my the MAREC.

recognitionRegion a MFVec2f field specifying two 2D points that are relative to the center of the video frame on which the recognition (and tracking) algorithm is performed. The first point indicates the lower left coordinate and the second one the upper right coordinate of a rectangle. By using this field, the MAREC suggests that only the inside area given by the rectangle has to be used in the recognition (and tracking) process, not the entire video frame. The recognition (and tracking) process can be improved by using a video frame region rather than the whole video frame but on the other hand the way how the original video frame is pre-preprocessed (e.g. cropped) may introduce delays. The ARAF browser cannot ensure that by using a recognition region the overall processing speed is improved.

onAugmentedStream is an output event of type MFString where the URL of the composed video stream is referenced. The protocol used by the ARAF browser to fetch the augmented stream can be any of the ones specified in Table 23. If the address pointing to the augmented stream file is not supported by the ARAF browser, an error code is triggered (see Table 22).

onError is an output event of type SFInt32.

Table 22 specifies onError possible values and their meaning.

Table 22 — RemImgComp: error codes

onError	Description
–1	The video source URL is invalid or not supported.
–5	Unknown error
–6	None of the available communication protocols are supported by the processing server.
–7	The augmented stream cannot be fetched. There might be a communication protocol incompatibility or the augmented stream is not available at the given address.

5.1.4.13 Audio recognition in ARAF

5.1.4.13.1 General

The audio recognition algorithms have been significantly improved nowadays, therefore, the second edition of ARAF should take advantage of the audio data that may exist in an ARAF experience and add scene augmentation based on the audio samples being recognized by local audio recognition libraries or by remote audio recognition servers.

Among many other applications that an audio recognition service can support, some of them are mentioned.

— Second screen application: an ARAF experience synchronized with a TV programme. The augmentation media presented in the experience is chosen based on the recognized programme and the timestamp.

— Vocal commands: an ARAF experience is controlled by the end used using vocal commands

— User context: an ARAF experience changes its state or triggers augmentations when an audio sequence is recognized.

5.1.4.13.2 Tables referred in the audio recognition prototypes

Table 23 — Audio recognition: communication protocols

Communication protocol name	Reference	Code
RTP	RFC 3550-2003 Real Time Transport Protocol	0
RTSP	RFC 2326-2013 version 2.0 Real Time Streaming Protocol	1
HTTP	RFC 2616-1999 Hypertext Transfer Protocol	2
DASH	ISO/IEC 23009-1:2012 Dynamic Adaptive Streaming over HTTP	3

Table 24 — Audio recognition: augmentation media types

Augmentation resource type description	Augmentation resource keyword	Code
Image (see table Image Formats for supported formats)	image	See Image Formats codes
Video (see table Video Formats for supported formats)	video	See Video Formats codes
Audio (.wav, .mid, .mp3, .raw)	audio	See Audio formats codes
BIFS scene (see ISO/IEC 14496-11)	BIFS (.mp4)	0

Table 25 — Audio recognition: video file formats

Video file format	Reference	Code
Raw video data	ISO/IEC 14496-1:2010 + Amd. 2:2014	1
MPEG4 Visual	ISO/IEC 14496-2:2004	2
MPEG4 AVC	ISO/IEC 14496-10:2012	3
MPEG-4 HEVC	ISO/IEC 23008-2	4

Table 26 — Audio recognition: image file formats

Target image file formats	Reference	targetResourceType keyword	Code
JPEG	ISO/IEC 10918	JPEG	7
JPEG 2000	ISO/IEC 15444	J2K	8
PNG	ISO/IEC 15948	PNG	9
RAW	ISO 12234-2	RAW	10

Table 27 — Audio recognition: audio file formats

Audio format	Reference	targetResourceType keyword	Code
raw	ISO/IEC 14496-1:2010/Amd. 2:2014	RAW	13
mp3	ISO/IEC 11172-3, ISO/IEC 13818-3	MP3	14
mp4	SO/IEC 14496-14	MP4	15
aac	ISO/IEC 13818-7, ISO/IEC 14496-3	AAC	16
Audio descriptors	ISO/IEC 15938-4:2002	MPEG7	17

Table 28 — Audio recognition: MIC URI

MIC URL	Description
hw://mic	Refers to the internal microphone of the device.

5.1.4.13.3 Prototypes implementations

5.1.4.13.3.1 LocAud (local audio recognition)

A local audio recognition service is needed in those MAR Experiences where a small amount of audio data is used in the recognition process, for example, vocal commands, recognition of audio sequences in a controlled environment.

The ARAF browser recognizes the presence of microphone audio sequences in target audio resources. The target audio resource URLs and the MIC URI where the audio recognition shall be performed are specified by the MAREC. The ARAF browser decides which recognition library is more appropriate for recognizing the audio data captured by the device's microphone. The ARAF browser sends the result to the ARAF scene as an array of integers representing the indexes of the recognized target audio resources along with the timestamps for each detected target resource (if applicable), as described below in the **Functionality and Semantics**.

Even though in the presented schema (below) the augmentation media is presented as input along other data provided by the MAREC, the augmentation media is not used by the PROTO implementation, meaning that it is not part of any of the PROTO fields. The augmentation resources are of course required for the scene augmentation but they do not influence the recognition process.

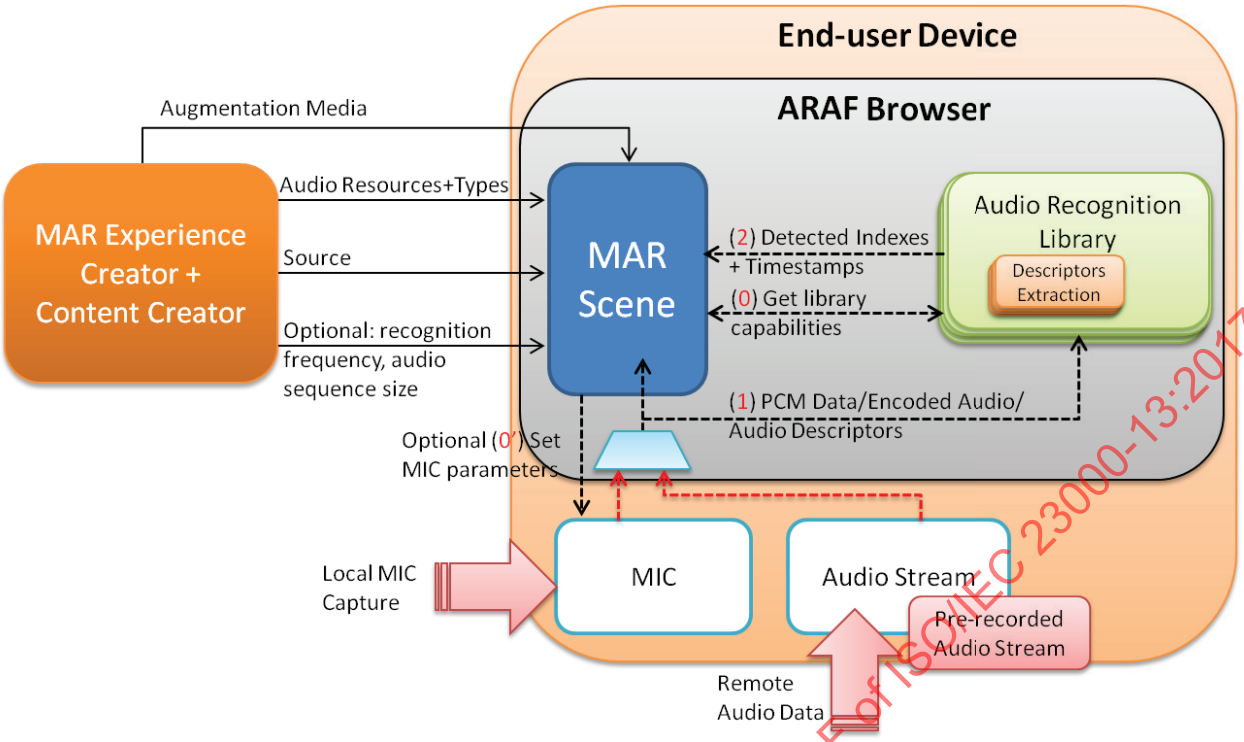


Figure 10 — Local audio recognition

5.1.4.13.3.2 BIFS Textual description

```
EXTERNPROTO LocAud [
    exposedField SFString   source                ""
    exposedField SFBool     enabled               TRUE
    exposedField MFString   targetResources       []
    exposedField MFString   targetResourcesTypes  []
    exposedField SFFloat    recognitionFreq       5000 #milliseconds
    exposedField SFFloat    recognitionSize       5000 # milliseconds
    eventOut     MFInt32    onRecognizedAudio
    eventOut     MFFloat    onTimestamp
    eventOut     SFInt32    onError
]"org:mpeg:locaud"
```

5.1.4.13.3.3 Functionality and semantics

The audio recognition process is performed locally, on the device where the MAR Experience is running. The ARAF browser shall be able to retrieve audio data from the capturing device (microphone) or from a broadcast server which is also specified by the MAREC in the ARAF scene, compress/convert the audio data and/or extract the audio features if necessary and send it to the audio recognition library considering the MAREC preferences. In the case when the audio data comes from external resources, one of the communication protocols presented in Table 23 has to be used for retrieving the audio data.

The ARAF browser is allowed to choose how the audio data is received from the capturing device (if applicable), therefore, this document does not impose any limitation regarding this aspect, as long as the final audio data being sent to the audio recognition library is compliant with this document.

The **source** field is a SFString specifying the URI/URL to a local or remote device that is capable of providing an audio stream. The audio feed can be live feed or prerecorded audio. The ARAF browser receives the audio data coming from the capturing device (e.g. microphone, broadcast server) and sends it to the audio recognition library in one of the accepted formats. The ARAF browser is in charge

converting/encoding the audio data in a format that is compatible with the recognition library. The source can be one of the following:

a) a URI to one of the microphones available on the end user's device. The possible values are specified in Table 28;

b) a URL to an external device (or broadcast audio server) providing audio feed.

The accepted audio formats are specified in Table 27.

The accepted communication protocols are specified in Table 23.

The **enabled** is a SFInt32 value indicating if the audio recognition process is enabled (running). MAREC can control the status of the audio recognition process or he can let the ARAF browser to decide whether the recognition process should be running or not. Table 29 specifies the supported integer values of the *enabled* field.

Table 29 — LocAud: enabled

enabled	Description
–1	ARAF browser decides when the audio recognition process is enabled/disabled. If not supported, the audio recognition process is always disabled unless a value of 0 or 1 is set by the MAREC.
0 (default)	The audio recognition process is disabled.
1	The audio recognition process is enabled.

targetResources is an MFString where the audio target resources to be recognized within the MAR experience are specified. A URI can point to a local or remote audio resource file. The accepted communication protocols for the remote resources are the ones specified in Table 23. Any of the below combinations describes a valid *targetResources* assignment:

— URIs pointing to target audio files;

— URIs pointing to files where target audio descriptors are found. A file contains descriptors of one single target audio;

— URLs pointing to files where multiple target audio descriptors are found. One file contains descriptors of multiple target images;

— any combination of the cases described above can coexist in the same *targetResources* field.

The file formats specified in Table 27 are the ones accepted for any of the options described above.

The **targetResourcesTypes** field is an MFInt32 containing an array which specifies the type of each target resource defined in the *targetResources* field. Each target resource shall have associated a type in order for the ARAF browser to know how to interpret the data. The possible pre-defined keywords of the *targetResourcesTypes* and their meaning are listed in Table 27. If the target resource is a proprietary descriptor file and in addition to the actual audio descriptors data the proprietary recognition library needs some other data (for e.g. an XML file), the related content should be stored in a directory that has the same name as the resource type (for e.g. *target_resource_type_keywod/*) in order for the ARAF browser to know where the required files can be found. The names of the files within the directory have to be the same name as the corresponding descriptor file name.

The **recognitionFreq** field is a SFFloat value specifying the frequency (in milliseconds) of the audio data being processed by the audio recognition library. In other words, the ARAF browser sends audio data to the audio recognition library with a frequency given by the field's value. For example, a value of 5 000 indicates that every 5 s a new audio data set is transmitted by the ARAF browser to the recognition library.

recognitionSize is a SFFloat value specifying the size (in milliseconds) of the audio data being transmitted by the ARAF browser to the audio recognition library. The size of the audio data should be equal or less than the value of *recognitionFreq.* If both the recognition size and the recognition

frequency have the same value K, it means that the whole audio stream is transmitted to the audio recognition library, every K milliseconds. If the recognition size is less that the recognition frequency the ARAF browser transmits audio data of "size" milliseconds every "freq" milliseconds.

onRecognizedAudio is an output event of type MFInt32 specifying the indexes of the target resources that have been recognized. An index is an integer value representing the position of a target resource in the *targetResources* array (0 indexed). The index of the first target resource is 0 and it is incremented by one for each next target resource. If a target resource is a file containing descriptors for several images, each target image descriptor within the descriptor file is assigned the next index as they were separately specified.

onTimestamp is an output event of type MFFloat specifying where the external audio data has been recognized within the target audio resource. It is possible that the recognition library detects the audio but is not able to compute the timestamp. In this case, the corresponding index of the field is set to –1.

onError is an output event of type SFInt32.

Table 30 specifies onError possible values and their meaning.

Table 30 — LocAud: error codes

onError	Description
–1	The audio source URL is invalid or not supported.
–2	At least one target audio resource is invalid or not supported. This error can be triggered in the cases when the ARAF browser is not able to read/access any of the target resources or the processing library does not support the format of at least one target resource.
–3	Unavailable audio recognition library for at least one target resource that has been specified by the MAREC.
–4	Unknown error.

5.1.4.13.3.4 RemAud (remote audio recognition)

Remote audio recognition functionality is necessary in those scenarios where the audio data is totally unknown or the audio sequences to be recognized are part of very big databases. A local audio recognition process is out of the question because the mobile devices nowadays don't have enough storage memory or enough processing power for these specific use-cases.

The MAREC specifies the source of the audio data, the processing server URL and optionally the audio sequence size (in milliseconds) which is transmitted to the audio recognition server and the frequency of the process that is being performed on the server. The ARAF browser interrogates the audio recognition server that is specified by the MAREC and retrieves the supported audio format (see Table 27). As long as the recognition process is active, the ARAF browser sends audio data to the processing server in one of the complaint formats that are accepted by the server. The MAREC preferences (processing frequency and sequence size if specified) should be considered by the ARAF browser.

A general schema of the connection between the MAREC, ARAF browser and MAR Scene and the Processing Server is presented in Figure 11.

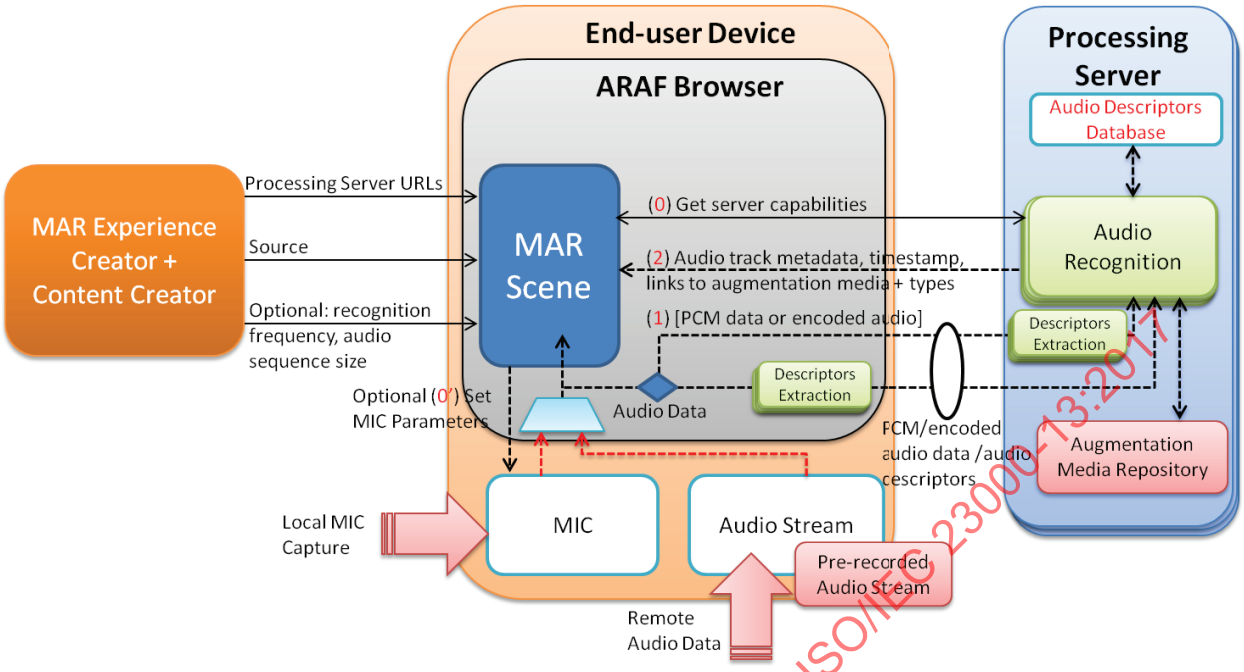


Figure 11 — Remote audio recognition

5.1.4.13.3.5 BIFS Textual description

```
EXTERNPROTO RemAud [
    exposedField SFString   source                  ""
    exposedField MFString   processingServerURL     []
    exposedField SFBool     enabled                 TRUE
    exposedField SFFloat    recognitionFreq         5000 #milliseconds
    exposedField SFFloat    recognitionSize         5000 #milliseconds
    eventOut     MFString   onRecognizedAudio
    eventOut     SFInt32    onTimestamp
    eventOut     MFString   onAugmentationMediaURL
    eventOut     MFInt32    onAugmentationMediaType
    eventOut     SFInt32    onError
]"org:mpeg:remaud"
```

5.1.4.13.3.6 Functionality and semantics

The audio recognition process is performed remotely, on one of the processing servers that are specified by the MAREC. In order for the processing machine to be ARAF compliant, it has to support the communication workflow that is presented in Table 33.

The ARAF browser shall be able to retrieve audio data from the capturing device (microphone) or from a broadcast server which is also specified by the MAREC in the ARAF scene, compress/convert the audio data and/or extract the audio features if necessary and send it to the recognition server considering the MAREC preferences. In the case when the audio data comes from external resources, one of the communication protocols presented in Table 23 has to be used for retrieving the audio data.

The ARAF browser is allowed to choose how the audio data is received from the capturing device (if applicable), therefore, this document does not impose any limitation regarding this aspect, as long as the final audio data being sent to the processing server is compliant and understood by the server.

The **source** field is a SFString specifying the URI/URL to a local or remote device that is capable of providing an audio stream. The audio feed can be live feed or prerecorded audio. The ARAF browser receives the audio data coming from the capturing device (e.g. microphone, broadcast server) and sends

it to the recognition server in the expected format. The expected format is given by the recognition server as detailed in Table 33. The source can be one of the following:

— a URI to one of the microphones available on the end user's device. The possible values are specified in Table 28;

— a URL to an external device (or broadcast audio server) providing audio feed.

The accepted audio formats are specified in Table 27.

The accepted communication protocols are specified in Table 23.

The **processingServerURL** is a MFString used by the MAREC to specify one or multiple web addresses where ARAF compliant processing servers are available. A valid URL is one that points to a processing server that handles at least one audio resource type and is able to understand ARAF browser requests, as defined in Table 33. Because a processing server can handle requests from multiple clients in the same time, a unique key is generated by the server and transmitted to the ARAF browser. The ARAF browser sends the unique generated key in each request to the processing server. This way, the processing server knows the source of the request and therefore it can improve the speed of the audio recognition process.

The **enabled** is a SFInt32 value indicating if the audio recognition process is enabled (running). MAREC can control the status of the audio recognition process or he can let the ARAF browser to decide whether the recognition process should be running or not. Table 31 specifies the supported integer values of the *enabled* field.

Table 31 — RemAud: enabled

enabled	Description
–1	ARAF browser decides when the audio recognition process is enabled/disabled. If not supported, the audio recognition process is always disabled unless a value of 0 or 1 is set by the MAREC.
0 (default)	The audio recognition process is disabled.
1	The audio recognition process is enabled.

The **recognitionFreq** field is a SFFloat value specifying the frequency (in milliseconds) of the audio data being processed by the remote server. In other words, the ARAF browser sends audio data to the audio recognition server with a frequency given by the field's value. For example, a value of 5 000 indicates that every 5 s a new audio data set is transmitted by the ARAF browser to the recognition server.

recognitionSize is a SFFloat value specifying the size (in milliseconds) of the audio data being transmitted by the ARAF browser to the audio recognition server. The size of the audio data should be equal or less than the value of *recognitionFreq*. If both the recognition size and the recognition frequency have the same value K, it means that the whole audio stream is transmitted to the processing server, every K milliseconds. If the recognition size is less that the recognition frequency the ARAF browser transmits audio data of "size" milliseconds every "freq" milliseconds.

onRecognizedAudio is an output event of type MFString that is triggered whenever the processing server recognizes an audio sequence. The processing server may send as an answer an array of strings that is related to the recognized audio sequence such as the metadata of a song. The meaning of the server's answer is not defined by this document; it's up to the MAREC to inform himself how the array string should be interpreted.

The following example shows what a server response might look like:

— the name of the recognized track;

— the author/band;

— the album;

— release date;

— any number of links separated by comas.

EXAMPLE ["Nothing else matters" "Metallica" "Metallica" "NULL" "http://en.wikipedia.org/wiki/Nothing_Else_Matters, http://youtu.be/W00b54UY7uE"].

As the MAREC is the one providing the server URL, it's his job to know what is the meaning of each keyword of the array.

onTimestamp is an output event of type SFInt32 which is set when a recognized audio sequence has been found within the full audio track that is part of. The matching algorithm and how the descriptors of the audio sequence are stored in the database it's not the scope of this document. The server should always provide the metadata of the recognized audio track (*onRecognizedAudio*) whenever the timestamp has been computed. The returned value specifies the milliseconds in the original audio track where the audio chunk (raw PCM data, audio descriptors) has been recognized. The processing server returns the millisecond matching the beginning of the audio sequence that has been sent by the ARAF browser to the audio recognition server.

If the timestamp has been computed, implicitly the audio track has been recognized, therefore, the metadata related to the audio track should be also part of the server's answer.

onAugmentationMediaURL is an output event of type MFString where augmentation media URLs associated to the recognized target resource are referenced. Several URLs can be provided when an audio sequence is recognized. The type of the media found at the specified URL is found in the output event *onAugmentationMediaTypes*; the correlation between the media and its type is given by the array indexes.

onAugmentationMediaTypes is an output event of type MFInt32 where the code of each augmentation media format referenced in the *onAugmentationMediaURL* field is specified. See Table 24 for the description of the supported types and their associated codes.

onError is an output event of type SFInt32.

Table 32 specifies onError possible values and their meaning.

Table 32 — RemAud: error codes

onError	Description
–1	The audio source URL is invalid or not supported.
–2	The processing server URL is not valid or the server is not compliant.
–3	The ARAF browser is not able to provide accepted audio format to the audio recognition server.
–4	Unknown error

5.1.4.13.3.7 ARAF Browser — Processing server communication workflow

In order to be a compliant ARAF audio processing server, the remote machine where the audio recognition is performed has to support the HTTP communication workflow described below.

The accepted communication protocols are specified in Table 23 (see 5.1.4.13.2).

Table 33 — RemAud: communication workflow

ARAF browser request	Request type	Description	Processing server response	Description
pServer? *alive*	GET	Get the unique key, the server parameters and capabilities	unique key 64-bit, one or several codes of the audio formats specified in table Audio Formats.	The unique key shall be used to indentify future requests from client. An array containing the corresponding codes of the supported audio formats, separated by comma
pServer *key&data_format*	POST	Inform the processing server about the chosen audio data format	True/false	The server reponse is True if the data is correctly received and False otherwise
pServer? *audio data*	POST	Send the audio data in the chose format, every X milliseconds (considering the MAREC preferences)	onRecognizedAudio, onTimestamp, onAugmentationMediaURL, onAugmentationMediaType	The server response includes the metadata of the recognized audio sequence, the timestamp where the audio sequence has been detected in the original audio track, one or several URLs to pointing to augmentation media and their types.

Communication workflow:

a) ARAF browser interrogates the Processing Server (GET /alive) in order to detect its status and to receive the server parameters and capabilities. The server returns:

   1) a unique key that shall be transmitted by ARAF browser in future requests;

   2) the list of codes describing the supported audio formats (file types or audio descriptors). See Table 27 for the supported codes and their meaning.

The server response shall be in the following format:

***key**=unique key 64-bit*

*&**audio_code**=[audio format codes];*

Example of a possible server response:

***key**=2e45325f4f&**audio_code**=0,2*

b) Once the key has been received, the ARAF browser knows that the processing server is ready to perform the audio recognition process. The ARAF browser decides on one audio encoding type, based on the available audio data format and on the server's response then it informs the processing server about the chosen format. The audio data of this session is sent only in the specified encoding type. The POST request is in the following format:

***key**=unique_key*

*&**audio_code**=the chosen audio encoding format code*

The processing server returns TRUE if the data is correctly received or FALSE otherwise.

c) The ARAF browser sends audio data to the processing server in the format that the audio processing server has been previously informed. The audio data length (size) is computed based on the recognitionSize value.

***key**=unique_key*

*&**audio_data**=the audio data*

The processing server's response can be:

— **onRecognizedAudio** (proprietary string): an array of strings that is related to the recognized audio sequence such as the metadata of a song;

— **onTimestamp**: the second matching the beginning of the recognized audio sequence that has been sent by the ARAF browser to the audio recognition server (the Browser should “compensate” the delay);

— **onAugmentationMediaURL**: one or several augmentation media URLs associated to the recognized audio sequence;

— **onAugmentationMediaType**: the type of each augmentation media referenced in onAugmentationMediaURL. Therefore, the lengths of the two arrays have to be the same.

An example of how the server's response may look like as presented below:

audio_metadata="proprietary string"

&timestamp=integer_value

&media_url=url1;url2

&media_type=0;2

The case where the processing server returns a proprietary string is not covered by this document. The MAREC should know how the string has to be interpreted, because he is the one providing the server URL. In this case the server response is transmitted to the MAR Scene exactly in the form as it comes.

d) The loop starts over from point 3 whenever the ARAF browser has to send new audio data to the processing server. The frequency of audio data being transmitted to the audio processing server is given by the field *recongitionFreq.*

5.1.4.14 Switch

5.1.4.14.1 XSD description

```
<complexType name="SwitchType">
 <all>
   <element ref="xmta:IS" minOccurs="0"/>
   <element name="choice" form="qualified" minOccurs="0">
      <complexType>
        <group ref="xmta:SF3DNodeType" minOccurs="0" maxOccurs="unbounded"/>
      </complexType>
   </element>
 </all>
 <attribute name="whichChoice" type="xmta:SFInt32" use="optional" default="-1"/>
 <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="Switch" type="xmta:SwitchType"/>
```

5.1.4.14.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.46.

The Switch grouping node traverses zero or one of the nodes specified in the *choice* field.

ISO/IEC 14772-1:1997, 4.6.5 describes details on the types of nodes that are legal values for *choice*.

The *whichChoice* field specifies the index of the child to traverse, with the first child having index 0. If *whichChoice* is less than zero or greater than the number of nodes in the *choice* field, nothing is chosen.

All nodes under a Switch continue to receive and send events regardless of the value of *whichChoice*. For example, if an active TimeSensor is contained within an inactive choice of a Switch, the TimeSensor sends events regardless of the Switch's state.

With the following restriction specified in ISO/IEC 14496-11:2015, 7.2.2.122.2.

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 the sound that corresponds to **Sound** nodes in the **whichChoice'th** subgraph of this node is played. The others are muted.

5.1.4.15 Transform

5.1.4.15.1 XSD description

```
      <complexType name="TransformType">
        <all>
          <element ref="xmta:IS" minOccurs="0"/>
          <element name="children" form="qualified" minOccurs="0">
             <complexType>
               <group ref="xmta:SF3DNodeType" minOccurs="0" maxOccurs="unbounded"/>
             </complexType>
          </element>
       </all>
       <attribute name="center" type="xmta:SFVec3f" use="optional" default="0 0 0"/>
       <attribute name="rotation" type="xmta:SFRotation" use="optional" default="0 0 1
0"/>
       <attribute name="scale" type="xmta:SFVec3f" use="optional" default="1 1 1"/>
       <attribute name="scaleOrientation" type="xmta:SFRotation" use="optional" default="0
0 1 0"/>
       <attribute name="translation" type="xmta:SFVec3f" use="optional" default="0 0 0"/>
       <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="Transform" type="xmta:TransformType"/>
```

5.1.4.15.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.52.

The Transform node is a grouping node that defines a coordinate system for its children that is relative to the coordinate systems of its ancestors. See ISO/IEC 14772-1:1997, 4.4.4 and ISO/IEC 14772-1:1997, 4.4.5 for a description of coordinate systems and transformations.

ISO/IEC 14772-1:1997, 4.6.5 provides a description of the *children*, *addChildren*, and *removeChildren* fields and eventIns.

The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the children of the Transform node. This is a hint that may be used for optimization purposes. The results are undefined if the specified bounding box is smaller than the actual bounding box of the children at any time. A default *bboxSize* value, (−1, −1, −1), implies that the bounding box is not specified and, if needed, shall be calculated by the browser. The bounding box shall be large enough at all times to enclose the union of the group's children's bounding boxes; it shall not include any transformations performed by the group itself (i.e. the bounding box is defined in the local coordinate system of the children). The results are undefined if the specified bounding box is smaller than the true bounding box of the group. A description of the *bboxCenter* and *bboxSize* fields is provided in ISO/IEC 14772-1:1997, 4.6.4.

The *translation*, *rotation*, *scale*, *scaleOrientation* and *center* fields define a geometric 3D transformation consisting of (in order):

— a (possibly) non-uniform scale about an arbitrary point;

— a rotation about an arbitrary point and axis;

— a translation.

The *center* field specifies a translation offset from the origin of the local coordinate system (0,0,0). The *rotation* field specifies a rotation of the coordinate system. The *scale* field specifies a non-uniform scale of the coordinate system. *scale* values shall be greater than zero. The *scaleOrientation* specifies a rotation of the coordinate system before the scale (to specify scales in arbitrary orientations). The *scaleOrientation* applies only to the scale operation. The *translation* field specifies a translation to the coordinate system.

As specified in ISO/IEC 14496-11:2015, 7.2.2.131.2.

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. 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.

5.1.4.16 Transform2D

5.1.4.16.1 XSD description

```
        <complexType name="Transform2DType">
         <all>
           <element ref="xmta:IS" minOccurs="0"/>
           <element name="children" form="qualified" minOccurs="0">
              <complexType>
                 <group ref="xmta:SF2DNodeType" minOccurs="0" maxOccurs="unbounded"/>
              </complexType>
           </element>
         </all>
         <attribute name="center" type="xmta:SFVec2f" use="optional" default="0 0"/>
         <attribute name="rotationAngle" type="xmta:SFFloat" use="optional" default="0"/>
         <attribute name="scale" type="xmta:SFVec2f" use="optional" default="1 1"/>
         <attribute name="scaleOrientation" type="xmta:SFFloat" use="optional"
default="0"/>
         <attribute name="translation" type="xmta:SFVec2f" use="optional" default="0 0"/>
         <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="Transform2D" type="xmta:Transform2DType"/>
```

5.1.4.16.2 Functionality and semantics

As specified in ISO/IEC 14496-11:2015, 7.2.2.132.2.

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 **Transform2D** node applies to the local coordinate system of the **Sound2D** nodes to alter the apparent spatial location and direction. If the children are not spatially presented but have equal numbers of channels, the **Transform2D** node has no effect on the childrens' sounds. After any such transformation, the combination of sounds is performed as described in ISO/IEC 14496-11:2015, 7.2.2.117.2.

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.

5.1.4.17 Viewpoint

5.1.4.17.1 XSD description

```
      <complexType name="ViewpointType">
       <all>
         <element ref="xmta:IS" minOccurs="0"/>
       </all>
       <attribute name="fieldOfView" type="xmta:SFFloat" use="optional"
default="0.785398"/>
       <attribute name="jump" type="xmta:SFBool" use="optional" default="true"/>
       <attribute name="orientation" type="xmta:SFRotation" use="optional" default="0 0 1
0"/>
       <attribute name="position" type="xmta:SFVec3f" use="optional" default="0 0 10"/>
       <attribute name="description" type="xmta:SFString" use="optional"
default="&amp;quot;&amp;quot;"/>
       <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="Viewpoint" type="xmta:ViewpointType"/>
```

5.1.4.17.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.53.

The Viewpoint node defines a specific location in the local coordinate system from which the user may view the scene. Viewpoint nodes are bindable children nodes (see ISO/IEC 14772-1:1997, 4.6.10) and thus there exists a Viewpoint node stack in the browser in which the top-most Viewpoint node on the stack is the currently active Viewpoint node. If a TRUE value is sent to the *set_bind* eventIn of a Viewpoint node, it is moved to the top of the Viewpoint node stack and activated. When a Viewpoint node is at the top of the stack, the user's view is conceptually re-parented as a child of the Viewpoint node. All subsequent changes to the Viewpoint node's coordinate system change the user's view (e.g. changes to any ancestor transformation nodes or to the Viewpoint node's *position* or *orientation* fields). Sending a *set_bind* FALSE event removes the Viewpoint node from the stack and produces *isBound* FALSE and *bindTime* events. If the popped Viewpoint node is at the top of the viewpoint stack, the user's view is re-parented to the next entry in the stack. More details on binding stacks can be found in ISO/IEC 14772-1:1997, 4.6.10. When a Viewpoint node is moved to the top of the stack, the existing top of stack Viewpoint node sends an *isBound* FALSE event and is pushed down the stack.

An author can automatically move the user's view through the world by binding the user to a Viewpoint node and then animating either the Viewpoint node or the transformations above it. Browsers shall allow the user view to be navigated relative to the coordinate system defined by the Viewpoint node (and the transformations above it) even if the Viewpoint node or its ancestors' transformations are being animated.

The *bindTime* eventOut sends the time at which the Viewpoint node is bound or unbound. This can happen:

a) during loading;

b) when a *set_bind* event is sent to the Viewpoint node;

c) when the browser binds to the Viewpoint node through its user interface described below.

The *position* and *orientation* fields of the Viewpoint node specify relative locations in the local coordinate system. *Position* is relative to the coordinate system's origin (0,0,0), while *orientation* specifies a rotation relative to the default orientation. In the default position and orientation, the viewer is on the Z-axis looking down the -Z-axis toward the origin with +X to the right and +Y straight up. Viewpoint nodes are affected by the transformation hierarchy.

Navigation types (see ISO/IEC 14772-1:1997, 6.29) that require a definition of a down vector (e.g. terrain following) shall use the negative Y-axis of the coordinate system of the currently bound Viewpoint node. Likewise, navigation types that require a definition of an up vector shall use the positive Y-axis of the coordinate system of the currently bound Viewpoint node. The *orientation* field of the Viewpoint node does not affect the definition of the down or up vectors. This allows the author to separate the viewing direction from the gravity direction.

The *jump* field specifies whether the user's view "jumps" to the position and orientation of a bound Viewpoint node or remains unchanged. This jump is instantaneous and discontinuous in that no collisions are performed and no ProximitySensor nodes are checked in between the starting and ending jump points. If the user's position before the jump is inside a ProximitySensor the *exitTime* of that sensor shall send the same timestamp as the bind eventIn. Similarly, if the user's position after the jump is inside a ProximitySensor the *enterTime* of that sensor shall send the same timestamp as the bind eventIn. Regardless of the value of *jump* at bind time, the relative viewing transformation between the user's view and the current Viewpoint node shall be stored with the current Viewpoint node for later use when *un-jumping* (i.e. popping the Viewpoint node binding stack from a Viewpoint node with *jump* TRUE*)*. The following summarizes the bind stack rules (see ISO/IEC 14772-1:1997, 4.6.10) with additional rules regarding Viewpoint nodes (displayed in boldface type).

a) During read, the first encountered Viewpoint node is bound by pushing it to the top of the Viewpoint node stack. If a Viewpoint node name is specified in the URL that is being read, this named Viewpoint node is considered to be the first encountered Viewpoint node. Nodes contained within Inline nodes, within the strings passed to the Browser.createVrmlFromString() method, or within files passed to the Browser.createVrmlFromURL() method (see ISO/IEC 14772-1:1997, 4.12.10) are not candidates for the first encountered Viewpoint node. The first node within a prototype instance is a valid candidate for the first encountered Viewpoint node. The first encountered Viewpoint node sends an *isBound* TRUE event.

b) When a *set_bind* TRUE event is received by a Viewpoint node:

   — If it is not on the top of the stack: The relative transformation from the current top of stack Viewpoint node to the user's view is stored with the current top of stack Viewpoint node. The current top of stack node sends an *isBound* FALSE event. The new node is moved to the top of the stack and becomes the currently bound Viewpoint node. The new Viewpoint node (top of stack) sends an *isBound* TRUE event. If *jump* is TRUE for the new Viewpoint node, the user's view is instantaneously "jumped" to match the values in the *position* and *orientation* fields of the new Viewpoint node.

   — If the node is already at the top of the stack, this event has no affect.

c) When a set_bind FALSE event is received by a Viewpoint node in the stack, it is removed from the stack. If it was on the top of the stack,

   — it sends an isBound FALSE event,

   — the next node in the stack becomes the currently bound Viewpoint node (i.e. pop) and issues an isBound TRUE event,

   — if its jump field value is TRUE, the user's view is instantaneously "jumped" to the position and orientation of the next Viewpoint node in the stack with the stored relative transformation of this next Viewpoint node applied.

d) If a set_bind FALSE event is received by a node not in the stack, the event is ignored and isBound events are not sent.

e) When a node replaces another node at the top of the stack, the isBound TRUE and FALSE events from the two nodes are sent simultaneously (i.e. with identical timestamps).

f) If a bound node is deleted, it behaves as if it received a set_bind FALSE event (see c).

The *jump* field may change after a Viewpoint node is bound. The rules described above still apply. If *jump* was TRUE when the Viewpoint node is bound, but changed to FALSE before the *set_bind* FALSE is sent, the Viewpoint node does not *un-jump* during unbind. If *jump* was FALSE when the Viewpoint node is bound, but changed to TRUE before the *set_bind* FALSE is sent, the Viewpoint node does perform the *un-jump* during unbind.

Note that there are two other mechanisms that result in the binding of a new Viewpoint:

— an Anchor node's *url* field specifies a "#ViewpointName";

— a script invokes the loadURL() method and the URL argument specifies a "#ViewpointName".

Both of these mechanisms override the *jump* field value of the specified Viewpoint node (#ViewpointName) and assume that *jump* is TRUE when binding to the new Viewpoint. The behaviour of the viewer transition to the newly bound Viewpoint depends on the currently bound NavigationInfo node's *type* field value (see ISO/IEC 14772-1:1997, 6.29).

The *fieldOfView* field specifies a preferred minimum viewing angle from this viewpoint in radians. A small field of view roughly corresponds to a telephoto lens; a large field of view roughly corresponds to a wide-angle lens. The field of view shall be greater than zero and smaller than π. The value of *fieldOfView* represents the minimum viewing angle in any direction axis perpendicular to the view. For example, a browser with a rectangular viewing projection shall have the following relationship:

$$\frac{\text{display width}}{\text{display height}} = \frac{\tan(FOV_{horizontal}/2)}{\tan(FOV_{vertical}/2)}$$

where the smaller of display width or display height determines which angle equals the *fieldOfView* (the larger angle is computed using the relationship described above). The larger angle shall not exceed π and may force the smaller angle to be less than *fieldOfView* in order to sustain the aspect ratio.

The *description* field specifies a textual description of the Viewpoint node. This may be used by browser-specific user interfaces. If a Viewpoint's *description* field is empty it is recommended that the browser not present this Viewpoint in its browser-specific user interface.

The URL syntax ".../scene.wrl#ViewpointName" specifies the user's initial view when loading "scene.wrl" to be the first Viewpoint node in the VRML file that appears as **DEF ViewpointName Viewpoint {...}**. This overrides the first Viewpoint node in the VRML file as the initial user view, and a *set_bind* TRUE message is sent to the Viewpoint node named "ViewpointName". If the Viewpoint node named "ViewpointName" is not found, the browser shall use the first Viewpoint node in the VRML file (i.e. the normal default behaviour). The URL syntax "#ViewpointName" (i.e. no file name) specifies a viewpoint within the existing VRML file. If this URL is loaded (e.g. Anchor node's *url* field or loadURL() method is invoked by a Script node), the Viewpoint node named "ViewpointName" is bound (a *set_bind* TRUE event is sent to this Viewpoint node).

The results are undefined if a Viewpoint node is bound and is the child of an LOD, Switch, or any node or prototype that disables its children. If a Viewpoint node is bound that results in collision with geometry, the browser shall perform its self-defined navigation adjustments as if the user navigated to this point (see ISO/IEC 14772-1:1997, 6.8).

5.1.4.18 Viewport

5.1.4.18.1 XSD description

```
        <complexType name="ViewportType">
         <all>
           <element ref="xmta:IS" minOccurs="0"/>
         </all>
         <attribute name="position" type="xmta:SFVec2f" use="optional" default="0 0"/>
         <attribute name="size" type="xmta:SFVec2f" use="optional" default="-1 -1"/>
         <attribute name="orientation" type="xmta:SFFloat" use="optional" default="0"/>
         <attribute name="alignment" type="xmta:MFInt32" use="optional" default="0 0"/>
         <attribute name="fit" type="xmta:SFInt32" use="optional" default="0"/>
         <attribute name="description" type="xmta:SFString" use="optional"
default="&amp;quot;&amp;quot;"/>
         <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="Viewport" type="xmta:ViewportType"/>
```

5.1.4.18.2 Functionality and semantics

As specified in ISO/IEC 14496-11:2015, 7.2.2.137.2.

A **Viewport** node can be placed in the **viewport** field of a **Layer2D** or **CompositeTexture2D** node or in the scene tree as a 2D node. It defines a new viewport and implicitly establishes a new local coordinate system. The bounds of the new viewport are defined by the **size** and **position** field. The new local coordinate system's origin is at the center of the parent node in the parent's local coordinate system.

The **orientation** field specifies the rotation which is applied to the viewport in the parent node's local coordinate system with respect to the X-axis. **Viewport** nodes are bindable nodes (see ISO/IEC 14496-11, 7.1.1.2.14) and thus there exists a **Viewport** node stack which follows the same rules than other bindable nodes (e.g. **Background2D**).

The **description** field specifies a textual description of the **Viewport** node. The **alignment** and **fit** fields specify how the viewing area is mapped to the rendering area of the parent node (i.e. **Layer2D**, **CompositeTexture2D**, or the 2D top-node).

If the **fit** field is set to 0, the viewing area is scaled to fit the rendering area without preserving the aspect ratio. If the **fit** field is set to 1, the viewing area is scaled preserving the aspect ratio to fit entirely inside the rendering area. The scaling operation is performed possibly after rotation as specified by the **orientation** field. If the **fit** field is set the 2, the viewing area is scaled preserving the aspect ratio to cover entirely the rendering area. The scaling operation is performed possibly after rotation as specified by the **orientation** field.

The **alignement** field is an MFInt32 field that contains two values. The first value specifies alignment along the X-axis and the second value specifies alignment along the Y-axis. The first value belongs to the following set of SFInt32: –1, 0, 1. The second value belongs to the following set of SFInt32: –1, 0, 1. An empty **alignement** field is equivalent to the default value. When the **fit** field is set to 0, the **alignment** field is ignored.

5.1.4.19 Form

5.1.4.19.1 XSD description

```
        <complexType name="FormType">
         <all>
           <element ref="xmta:IS" minOccurs="0"/>
           <element name="children" form="qualified" minOccurs="0">
             <complexType>
                <group ref="xmta:SF2DNodeType" minOccurs="0" maxOccurs="unbounded"/>
             </complexType>
           </element>
        </all>
        <attribute name="size" type="xmta:SFVec2f" use="optional" default="-1 -1"
```

```
/>
        <attribute name="groups" type="xmta:MFInt32" use="optional"/>
        <attribute name="constraints" type="xmta:MFString" use="optional"/>
        <attribute name="groupsIndex" type="xmta:MFInt32" use="optional"/>
        <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="Form" type="xmta:FormType"/>
```

5.1.4.19.2 Functionality and semantics

As specified in ISO/IEC 14496-11:2015, 7.2.2.62.2.

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. A value of 0 in the groupsIndex field references the form node itself, otherwise a groupsIndex field value is a 1-based index into the group field.

Constraints belong to two categories: alignment and distribution constraints.

Groups referred to in the tables below are groups 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.

In case the form itself is specified in alignment constraint (group index 0), the form rectangle shall be used as the base of the alignment computation and other groups in the constraint list shall be aligned as specified by the constraint.

5.1.5 Dynamic and animated scene

5.1.5.1 General

The following animation related nodes are used in ARAF: OrientationInterpolator, ScalarInterpolator, CoordinateInterpolator, ColorInterpolator, PositionInterpolator, Valuator.

5.1.5.2 OrientationInterpolator

5.1.5.2.1 XSD description

```
<complexType name="OrientationInterpolatorType">
  <all>
    <element ref="xmta:IS" minOccurs="0"/>
  </all>
  <attribute name="key" type="xmta:MFFloat" use="optional"/>
  <attribute name="keyValue" type="xmta:MFRotation" use="optional"/>
  <attributeGroup ref="xmta:DefUseGroup"/>
```

```
    </complexType>
<element name="OrientationInterpolator" type="xmta:OrientationInterpolatorType"/>
```

5.1.5.2.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.32.

The OrientationInterpolator node interpolates among a list of rotation values specified in the *keyValue* field. These rotations are absolute in object space and therefore are not cumulative. The *keyValue* field shall contain exactly as many rotations as there are keyframes in the *key* field.

An orientation represents the final position of an object after a rotation has been applied. An OrientationInterpolator interpolates between two orientations by computing the shortest path on the unit sphere between the two orientations. The interpolation is linear in arc length along this path. The results are undefined if the two orientations are diagonally opposite.

If two consecutive *keyValue* values exist such that the arc length between them is greater than π, the interpolation will take place on the arc complement. For example, the interpolation between the orientations (0, 1, 0, 0) and (0, 1, 0, 5.0) is equivalent to the rotation between the orientations (0, 1, 0, 2π) and (0, 1, 0, 5.0).

5.1.5.3 ScalarInterpolator

5.1.5.3.1 XSD description

```
  <complexType name="ScalarInterpolatorType">
    <all>
      <element ref="xmta:IS" minOccurs="0"/>
    </all>
    <attribute name="key" type="xmta:MFFloat" use="optional"/>
    <attribute name="keyValue" type="xmta:MFFloat" use="optional"/>
    <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="ScalarInterpolator" type="xmta:ScalarInterpolatorType"/>
```

5.1.5.3.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.39.

This node linearly interpolates among a list of SFFloat values. This interpolator is appropriate for any parameter defined using a single floating point value. Examples include width, radius, and intensity fields. The *keyValue* field shall contain exactly as many numbers as there are keyframes in the *key* field.

5.1.5.4 CoordinateInterpolator

5.1.5.4.1 XSD description

```
  <complexType name="CoordinateInterpolatorType">
    <all>
      <element ref="xmta:IS" minOccurs="0"/>
    </all>
    <attribute name="key" type="xmta:MFFloat" use="optional"/>
    <attribute name="keyValue" type="xmta:MFVec3f" use="optional"/>
    <attributeGroup ref="xmta:DefUseGroup"/>
</complexType>
<element name="CoordinateInterpolator" type="xmta:CoordinateInterpolatorType"/>
```

5.1.5.4.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.13.

This node linearly interpolates among a list of MFVec3f values. The number of coordinates in the *keyValue* field shall be an integer multiple of the number of keyframes in the *key* field. That integer multiple defines how many coordinates will be contained in the *value_changed* events.

5.1.5.5 ColorInterpolator

5.1.5.5.1 XSD description

```
        <complexType name="ColorInterpolatorType">
         <all>
           <element ref="xmta:IS" minOccurs="0"/>
         </all>
         <attribute name="key" type="xmta:MFFloat" use="optional"/>
         <attribute name="keyValue" type="xmta:MFColor" use="optional"/>
         <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="ColorInterpolator" type="xmta:ColorInterpolatorType"/>
```

5.1.5.5.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1997, 6.10.

This node interpolates among a list of MFColor key values to produce an SFColor (RGB) *value_changed* event. The number of colours in the *keyValue* field shall be equal to the number of keyframes in the *key* field. The *keyValue* field and *value_changed* events are defined in RGB colour space. A linear interpolation using the value of *set_fraction* as input is performed in HSV space. The results are undefined when interpolating between two consecutive keys with complementary hues.

5.1.5.6 PositionInterpolator

5.1.5.6.1 XSD description

```
      <complexType name="PositionInterpolatorType">
        <all>
          <element ref="xmta:IS" minOccurs="0"/>
        </all>
        <attribute name="key" type="xmta:MFFloat" use="optional"/>
        <attribute name="keyValue" type="xmta:MFVec3f" use="optional"/>
        <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="PositionInterpolator" type="xmta:PositionInterpolatorType"/>
```

5.1.5.6.2 Functionality and semantics

As specified in ISO/IEC 14772-1:1996, 6.37.

The PositionInterpolator node linearly interpolates among a list of 3D vectors. The *keyValue* field shall contain exactly as many values as in the *key* field.

See ISO/IEC 14772-1:1997, 4.6.8, Interpolator nodes, contains a more detailed discussion of interpolators.

5.1.5.7 Valuator

5.1.5.7.1 XSD description

```
     <complexType name="ValuatorType">
      <all>
       <element ref="xmta:IS" minOccurs="0"/>
      </all>
      <attribute name="Factor1" type="xmta:SFFloat" use="optional" default="1"
/>
      <attribute name="Factor2" type="xmta:SFFloat" use="optional" default="1"
/>
      <attribute name="Factor3" type="xmta:SFFloat" use="optional" default="1"
/>
      <attribute name="Factor4" type="xmta:SFFloat" use="optional" default="1"
/>
      <attribute name="Offset1" type="xmta:SFFloat" use="optional" default="0"
/>
```



```
      <attribute name="Offset2" type="xmta:SFFloat" use="optional" default="0"
/>
      <attribute name="Offset3" type="xmta:SFFloat" use="optional" default="0"
/>
      <attribute name="Offset4" type="xmta:SFFloat" use="optional" default="0"
/>
      <attribute name="Sum" type="xmta:SFBool" use="optional" default="false"
/>
      <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="Valuator" type="xmta:ValuatorType"/>
```

5.1.5.7.2 Functionality and semantics

As specified in ISO 14496-11:2015, 7.2.2.135.1.

The **Valuator** node serves as a simple type casting method. It can receive events of multiple types. On reception of such an event, eventOuts of many different types can be generated. Both the eventIn and the eventOut values can be single field (SF) or multiple field (MF) types. In addition, the possible eventIn and eventOut types include both scalar types, like SFBool, and vector types, like SFVec2f.

Each component of the (possibly vector) eventOut value is calculated from the corresponding component of the (possibly vector) eventIn value with the following relationship:

output.i = factor.i * input.i + offset.i

All values specified in the above equation are floating point values.

input.i is the value of the ith component of the eventIn type and output.i is the value of the ith component of one of the eventOut types specified in the node interface. input.i shall be extended by zeros for all components i that do not exist in the input type (e.g. input.z=0.0 in case an SFVec2f is cast to an SFVec3f).

factor.i and offset.i are the exposedField values for the ith component of the vectorial calculation. In the special case of a scalar input type (e.g. SFBool, SFInt32) that is cast to a vectorial output type (e.g. SFVec2f), for all components i of output.i, input.i shall take the value of the scalar input type, after appropriate type conversion.

Depending on the number of dimensions of the data type, there may be one up to four input values. For example, an eventIn of type SFRotation will require four input paths but SFInt32 will only require the first input path. Each input path operates identically.

Each input value is converted to a floating-point value using a simple typecasting rule as illustrated in Table 34. After conversion, the values are multiplied by the corresponding factor.i value and added to the corresponding offset.i value as specified above. Depending on whether the summer is enabled, either the summed value or the individual values are presented at the output. The summer sums all four computed input paths independent of the number of dimensions of the eventIn type.

Table 34 — 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
String	Convert if the content of the string represensts an int, float or a double value. “Boolean” string values “true” and “false” are converted to 1.0 and 0.0 respectively. Any other string is converted to 0.0.

Table 35 — 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
String	Convert to a string representing the float

For conversion of data types to and from strings the values of multiple valued data types, such as SFColor, are separated by spaces.

Depending on the dimension of the eventOut type, the corresponding number of output values are computed and converted to the output types according to Table 34 and as detailed below.

If the eventIn is of an SF type then an eventOut for an MF type shall consist of just one element, i.e. the MF type collapses to a SF type.

If the eventIn is of an MF type then an eventOut for an SF type shall be created by using the first element of the MF input only.

If the eventIn is of an MF type then an eventOut for an MF type shall be created by using each element of the MF input to generate one element of the MF output type, respecting the order of the elements in the eventIn MF type.

If the eventIn is of SFTime type then the conversion to string format shall be in the format "hh:mm:ss" where "hh", "mm", "ss" are respectively hours, minutes and seconds of the input SFTime value.

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**.

SFVec4f fields cannot be routed to Valuator node.

5.1.6 Communication and compression

5.1.6.1 General

The following communication and compression related nodes are used in ARAF: BitWrapper, MediaControl.

5.1.6.2 BitWrapper

5.1.6.2.1 XSD description

```
      <complexType name="BitWrapperType">
        <all>
          <element ref="xmta:IS" minOccurs="0"/>
          <element name="node" form="qualified" minOccurs="0">
            <complexType>
              <group ref="xmta:SFWorldNodeType" minOccurs="0"/>
            </complexType>
          </element>
        </all>
        <attribute name="type" type="xmta:SFInt32" use="optional" default="0"/>
        <attribute name="url" type="xmta:MFUrl" use="optional"/>
        <attribute name="buffer" type="xmta:SFString" use="optional"
default="&amp;quot;&amp;quot;"/>
        <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="BitWrapper" type="xmta:BitWrapperType"/>
```


5.1.6.2.2 Functionality and semantics

As specified in ISO/IEC 14496-11:2015, 7.2.2.23.2.

A node may have a dedicated node compression scheme. This compressed representation may be carried in the BIFS stream or in a separate stream.

The **node** field contains the node that has a compressed representation. The BitWrapper node can be used in lieu and place of the **node** it wraps. The **type** field indicates which node compression scheme shall be used, 0 being the default. It is envisioned that future node compression schemes may be developed for the same node. For this specification, AFX object code table of ISO/IEC 14496-1 defines the default schemes.

The compressed representation is carried either in a separate stream or within the scene stream. The **url** field indicates the stream that contains the compressed representation and the **buffer** field contains the compressed representation when carried within the scene. When the compressed representation is carried in separate streams by using **url** field, node decoders shall be configured.

In the object descriptor stream, a node decoder is indicated in the DecoderConfig descriptor for streamType 0x03, objectTypeIndication 0x05, and code defined in AFX object code table of ISO/IEC 14496-1. The decoder is configured with a AFXConfig descriptor.

Note that **buffer** is an array of 8-bit values. It shall not be interpreted as a UTF-8 string. For in-band scenario, compressed media stream is transmitted within a scene description stream through **buffer** field.

For out-band scenario, compressed media stream is transmitted outside scene description stream through **url** field. It is used when the specific node requires upstream to send a specific information to a server.

5.1.6.3 MediaControl

5.1.6.3.1 XSD description

```
    <complexType name="MediaControlType">
      <all>
        <element ref="xmta:IS" minOccurs="0"/>
      </all>
      <attribute name="url" type="xmta:MFUrl" use="optional"/>
      <attribute name="mediaStartTime" type="xmta:SFTime" use="optional" default="-1"/>
      <attribute name="mediaStopTime" type="xmta:SFTime" use="optional" default="1.797693
1348623157E308"/>
      <attribute name="mediaSpeed" type="xmta:SFFloat" use="optional" default="1"/>
      <attribute name="loop" type="xmta:SFBool" use="optional" default="false"/>
      <attribute name="preRoll" type="xmta:SFBool" use="optional" default="true"/>
      <attribute name="mute" type="xmta:SFBool" use="optional" default="false"/>
      <attribute name="enabled" type="xmta:SFBool" use="optional" default="true"/>
      <attributeGroup ref="xmta:DefUseGroup"/>
    </complexType>
    <element name="MediaControl" type="xmta:MediaControlType"/>
```

5.1.6.3.2 Functionality and semantics

As specified in ISO/IEC 14496-11:2015, 7.2.2.84.2.

The **MediaControl** node controls the play back and, hence, delivery of a media stream referenced by a media node. The **MediaControl** node allows selection of a time interval within one or more stream objects for play back, modification of the playback direction and speed, as well as pre-rolling and muting of the stream.

A media node may be used with or without an associated **MediaControl** node. A media node for which no **MediaControl** node is present shall behave as if a **MediaControl** node for that media stream were present in the scene, with default values set.

The **url** field contains a reference to one or more stream objects ("OD:n#segment" or "OD:n"), called the controlled stream objects, all of which shall belong to the same media stream. This media stream is called the controlled stream. When any media node referring to a media stream in its **url** field is active, the associated media stream is said to be active.

This means that the controlled stream becomes active exactly when some media node pointing to it becomes active. The controlled stream becomes inactive, when all media nodes referring to it become inactive. When a controlled media stream becomes active, the associated controlled stream objects in the **url** field of the MediaControl node shall be played sequentially.

The **mediaStartTime** and **mediaStopTime** fields define the time interval, in media time, of each controlled stream object to be played back. If media time of the media stream is undefined, selection of a time interval of the controlled stream object for play back is not supported. In that case, the **mediaStartTime** and **mediaStopTime** fields shall be ignored. The following values have special meaning for **mediaStartTime** and **mediaStopTime**:

— 0 indicates the beginning of the controlled stream object;

— –1 indicates the media time of the controlled stream object when the associated media node becomes active;

— +I, or any value greater than the duration of the controlled stream object indicates its end.

Semantics of **mediaStartTime** and **mediaStopTime** depend on the delivery scenario. Semantics in case of delivery scenarios that permit seeking: Play back of the controlled stream object shall start at **mediaStartTime** of the first controlled media object when the controlled stream becomes active. When the controlled stream becomes inactive and then active again, then if **mediaStartTime** is –1 the stream starts playing from the point where it was last stopped. Otherwise, the first controlled stream object in the **url** field restarts playing from **mediaStartTime**. If the **loop** field is TRUE, all the controlled stream objects are played in a loop, each in the range **mediaStartTime** to **mediaStopTime** while the controlled stream is active. If **mediaStartTime** is –1, each stream object will start from the beginning.

In all delivery scenarios, play back of the controlled stream object shall occur only in the range defined by **mediaStartTime** and **mediaStopTime**. Outside this range the play back shall be muted. The **loop** field shall be ignored in delivery scenarios that do not permit seeking. The **mediaSpeed** is a requested multiplication factor to the normal speed of each controlled stream object. Negative values for **mediaSpeed** request that the controlled stream object plays backward from **mediaStartTime** to **mediaStopTime**. When this field is zero, the controlled stream shall be paused.

NOTE 1 All streams, independent of speed, are only played in the range defined by **mediaStartTime** and **mediaStopTime**. When **mediaSpeed** < 0, the stream object can only be played if the server reassigns time stamps to be increasing from **mediaStopTime** to **mediaStartTime**.

If **mediaSpeed** > 0 (forward play back) and **mediaStopTime** < **mediaStartTime,** then the controlled stream object will play until the end.

If **mediaSpeed** < 0 (backward play back) and **mediaStopTime** > **mediaStartTime,** then the controlled stream object will play to the beginning.

In these equations, the special value –1 is substituted by the actual value of media time that it represents. There is no requirement that a delivery service supports specific ranges of **mediaSpeed** other than **mediaSpeed** = 1. Media content shall comply with maximum and average bit rates specified for the stream, irrespective of the value of the **mediaSpeed** field.

If the **preRoll** field is set to TRUE the controlled stream should be pre-rolled in order to be ready to start instantly when the controlled stream becomes active. All streams that are associated to the same object time base as the stream that is pre-rolled should also be pre-rolled. If the delivery scenario does not permit seeking, **preRoll** = TRUE means that the controlled stream object should be delivered and

recently received access units should be stored in the decoding buffer in order to enable instantaneous play back when the media node becomes active.

NOTE 2 Play back of stream objects in media nodes that are not controlled by **MediaControl** or where **preRoll** is FALSE may suffer an unspecified startup delay if play back is requested by an unpredictable action (e.g. user interaction, script).

The **isPreRolled** event sends a TRUE value when the controlled stream object has completed pre-rolling. If the **mute** field is set to TRUE, the stream objects in the **url** field are not rendered when they are played. However, their media clock is not stopped. For visual streams, whether natural video or synthetic such as animation streams or **Inline** nodes, **mute** means that the visual texture remains unchanged; for audio streams, the audio is not played.

If the **enabled** field is set to TRUE, the **MediaControl** node controls the stream object it refers to. More than one **MediaControl** node may be used to control a stream object within the same stream. At most one of these **MediaControl** nodes shall be enabled at any time. If one of these **MediaControl** nodes becomes enabled, the **enabled** field of all other **MediaControl** nodes that refer to the same stream shall automatically be set to FALSE.

If the **enabled** field is set to FALSE, the **MediaControl** node shall cease to control the play back and muting of the controlled stream object, however, **preRoll** shall still be evaluated. If the controlled stream object is playing when **enabled** is set to FALSE and no other **MediaControl** node takes control of the stream, the stream object shall continue playing as if it were still controlled by the disabled **MediaControl** node. Only one **MediaControl** node shall refer to any of the set of media streams that are associated to a single object time base.

NOTE 3 **MediaControl** affects the OTB of the controlled stream and therefore affects all the streams that are associated to the same OTB. Therefore, changing play position, speed or direction of one stream will correspondingly affect all the active streams that are associated to the same OTB.

5.1.6.4 Support for Maps

5.1.6.4.1 General

MAPS are supported in ARAF by three PROTOs: Map, MapOverlay, MapMarker and MapPlayer. As for other elements in the scene, the node interface and the functionality and semantics are normative. Annex A presents an informative implementation of these PROTOs.

5.1.6.4.2 Map

5.1.6.4.2.1 XSD description

```
<ProtoDeclare name="Map" locations="org:mpeg:Map">
  <field name="name" type="String" vrml97Hint="exposedField" stringValue=""/>
  <field name="addOverlays" type="Nodes" vrml97Hint="eventIn"/>
  <field name="removeOverlays" type="Nodes" vrml97Hint="eventIn"/>
  <field name="translate" type="Vector2" vrml97Hint="eventIn"/>
  <field name="mapGPSCenter" type="Vector2" vrml97Hint="exposedField" vector2Value="0 0"/>
  <field name="zoomIn" type="Boolean" vrml97Hint="eventIn"/>
  <field name="zoomOut" type="Boolean" vrml97Hint="eventIn"/>
  <field name="overlays" type="Nodes" vrml97Hint="exposedField">
    <nodes></nodes>
  </field>
  <field name="mode" type="Strings" vrml97Hint="exposedField" stringArrayValue="ROADMAP"/>
  <field name="provider" type="Strings" vrml97Hint="exposedField" stringArrayValue="ANY"/>
  <field name="mapSize" type="Vector2" vrml97Hint="exposedField" vector2Value="768 768"/>
  <field name="mapTranslation" type="Vector2" vrml97Hint="exposedField" vector2Value="0
0"/>
  <field name="mapWidth" type="Float" vrml97Hint="exposedField" floatValue="0"/>
  <field name="zoomLevel" type="Integer" vrml97Hint="exposedField" integerValue="0"/>
</ProtoDeclare>
```

5.1.6.4.2.2 BIFS textual description

```
EXTERNPROTO Map [
 exposedField  SFString  name                ""
 exposedField  SFVec2f   mapTranslation      0.0 0.0
 exposedField  SFVec2f   mapGPSCenter        0.0 0.0
 exposedField  MFNode    overlays            []
 exposedField  MFString  mode                ["ROADMAP"]
 exposedField  MFString  provider            ["ANY"]
 exposedField  SFVec2f   mapSize             768 768
 exposedField  SFFloat   mapWidth            0
 exposedField  SFInt32   zoomLevel           0
 eventIn       MFNode    addOverlays
 eventIn       MFNode    removeOverlays
 eventIn       SFVec2f   translate
 eventIn       SFBool    zoomIn
 eventIn       SFBool    zoomOut
]"org:mpeg:Map"
```

5.1.6.4.2.3 Functionality and semantics

The **Map** node provides map display capabilites to a scene. The node detects pointer device dragging and enables the dragging of the map image. The dragging operation changes the mapGPSCenter corresponding to the drag operation and translates all the associated **Map** items along with the image as a single unit.

The **name** field of the map specifies a unique name of the **Map** instance. As multiple **Map** instances can coexist in the same scene, this field allows the identification of a specific MAP node by name.

mapTranslation specifies a (x, y) translation in the local coordinate system of the **Map** image instance.

NOTE mapTranslation does not modify the GPS center of the **Map** (mapGPSCenter) position or any other GPS related value.

addOverlays specifies one or more **MapOverlay** nodes that shall be added to the Map **overlays** field. The MapOverlay instances are inserted after the already existing ones.

removeOverlays specifies one or more **MapOverlay** nodes that shall be removed from the Map **overlays** field. If a **MapOverlay** instance is not found, its removal fails silently. Removing a **MapOverlay** implies the deletion of all the **MapMarkers** already attached to the indicated **MapOverlay** instance.

translate specifies a translation that is to be applied to the **Map** image. The values are represented in the local coordinate system of the Map node. The event also modifies the mapGPSCenter field.

mapGPSCenter specifies the GPS position (latitude, longitude) of the Map center.

zoomIn increases the zoomLevel of the **Map** by one.

zoomOut decreases the zoomLevel of the **Map** by one.

mapSize is a 2D vector which specifies the width and the height of the map image.

zoomLevel represents the resolution of the current view. The minimal value of zoom level is 0, while the maximal value is defined by the map provider depending on its capabilities. Zoom level 0 encompasses the entire earth. Each succeeding zoom level doubles the precision in both horizontal and vertical dimensions.

mapWidth represents the length in meters on the longitude axis of the desired visible map. The client calculates the maximum zoom level that contains the desired map and sets that value in the zoomLevel field. If mapWidth is set to 0, then the zoomLevel field values is used.

provider specifies the desired map provider to be used. The provider field is a multi-value field enabling designers to specify fallback map providers in the case the desired one is not supported by the client. The “ANY” choice allows the client to select its provider.

mode specifies the type of map that is to be displayed. The possible values are: “SATELLITE”, “PLANE”, “ROADMAP” and “TERRAIN”. Satellite mode should display map images that are practically shot from a vertical viewpoint, usually by a satellite. Plane mode should display map images that are taken by an angle close to 45°, usually shoot by an airplane. Map should display images that are vector drawings of streets, buildings and other similar features. Terrain mode should display images that represent physical relief map image, showing terrain and vegetation.

If multiple values are specified in the map field, then the resulting image should be a combination of all desired modes as long as they are supported by the map provider. If a certain combination is not supported, then the map view falls back to the closest supported one.

The map image is made up by a 3 × 3 matrix of tile images. The GPS position and size of each tile should be automatically computed once the mapSize and the mapGPSCenter are set.

5.1.6.4.3 MapOverlay

5.1.6.4.3.1 XSD description

```
<ProtoDeclare name="MapOverlay" locations="org:mpeg:MapOverlay">
    <field name="name" type="String" vrml97Hint="exposedField" stringValue = ""/>
    <field name="visible" type="Boolean" vrml97Hint="exposedField" booleanValue = "TRUE"/>
    <field name="enabled" type="Boolean" vrml97Hint="exposedField" booleanValue = "TRUE"/>
    <field name="clickable" type="Boolean" vrml97Hint="exposedField" booleanValue =
"TRUE"/>
    <field name="children" type="Nodes" vrml97Hint="exposedField">
    <nodes></nodes>
    </field>
     <field name="keywords" type="Strings" vrml97Hint="exposedField stringArrayValue=""/>
    <field name="addOverlayItems" type="Nodes" vrml97Hint="eventIn"/>
    <field name="removeOverlayItems" type="Nodes" vrml97Hint="eventIn"/>
</ProtoDeclare>
```

5.1.6.4.3.2 BIFS rextual description

```
EXTERNPROTO MapOverlay [
    exposedField  SFString   name           ""
    exposedField  SFBool     visible        TRUE
    exposedField  SFBool     enabled        TRUE
    exposedField  SFBool     clickable      TRUE
    exposedField  MFNode     children       []
    exposedField  MFString   keywords       []
    eventIn       MFNode     addOverlayItems
    eventIn       MFNode     removeOverlayItems
]"org:mpeg:MapOverlay"
```

5.1.6.4.3.3 Semantics

A **MapOverlay** instance acts like a container for any number of items of the same type (**MapMarkers**) that should be added to the **Map**. It also provides an easy way of executing a specified action on all the items it contains at a time, as indicated below.

The **name** field specifies a unique name of the **MapOverlay** instance that may be used to identify a specific overlay item for further actions.

The **visible** field is a Boolean value which specifies if the **MapOverlay** instance is visible on the **Map**. This field is used to display/hide all the items (**MapMarker** instances) of the current **MapOverlay** instance at a time.

The **enabled** field is a Boolean value which specifies if all the MapMarkers of the current overlay are enabled or not. An enabled **MapMarker** will output an event each time the **MapPlayer** enters/exits its active region (the area “covered” by it). This value can be set using the **radius** field of the **MapMarker**. Review the semantics of the **MapMarker** PROTO (see 5.1.6.4.4.3 for details).

The **clickable** field specifies if all the **MapMarkers** of a given **MapOverlay** are clickable or not. A clickable **MapMarker** can be tapped by the user. When tapped, “onClick” output event is triggered

by the corresponding **MapMarker** instance. Review the semantics of the **MapMarker** PROTO (see 5.1.6.4.4.3 for details).

children is a list of **MapMarker** instances. The list contains all the items that have been already added to the current **MapOverlay** instance.

The **keywords** field specifies a semantic description of the specific **MapOverlay** node (e.g. "restaurant", "museum", etc.).

addOverlayItems is an input event which can recieve one or multiple **MapMarker** instances to be added to the **MapOverlay children** field. If the children field is not empty the specified **MapMarker** instances are inserted after the ones that already exist.

removeOverlayItems is an input event that removes the specified **MapMarker** instances from its **children** field. If a **MapMarker** instance is not found among the children of the **MapOverlay**, the removal of the unknown **MapMarker** instance fails silently.

Visible, **clickable** and **enabled** fields may give wrong information about the corresponding **MapMarker** fields. If, for example, the visible field of the **MapOverlay** has been used to set all the **MapMarkers** visible but in the meantime one or more **MapMarkers** have been individually set invisible (using their own **visible** field) then the **MapOverlay** visible field (which is still TRUE) will give false information that all the **MapMarkers** are still visible. The same rule applies to clickable and enabled fields. Writing a new value to any of these fields will set the new value to all the **MapMarker** instances no matter their previous value of the specified field. Therefore, the user should be careful when **reading** any of these fields.

addOverlayItems and **removeOverlayItems** are input events of type MFNode but adding or removing one single **MapMarker** instance at a time shall be also valid.

5.1.6.4.4 MapMarker

5.1.6.4.4.1 XSD description

```
<ProtoDeclare name="MapMarker" locations="urn:mpeg:MapMarker">
    <field name="name" type="String" vrml97Hint="exposedField" stringValue = ""/>
    <field name="visible" type="Boolean" vrml97Hint="exposedField" booleanValue = "TRUE"/>
    <field name="enabled" type="Boolean" vrml97Hint="exposedField" booleanValue = "TRUE"/>
    <field name="clickable" type="Boolean" vrml97Hint="exposedField" booleanValue =
"TRUE"/>
    <field name="position" type="Vector2" vrml97Hint="exposedField" vector3Value = "0 0
0"/>
<field name="radius" type="Float" vrml97Hint="exposedField" floatValue="0"/>
    <field name="rotation" type="Rotation" vrml97Hint="exposedField" rotationValue="0 0 1
0"/>
    <field name="markerShape" type="Nodes" vrml97Hint="exposedField"
<nodes></nodes>
<field>
    <field name="keywords" type="Strings" vrml97Hint="exposedField stringArrayValue=""/>
    <field name="doClick" type="Boolean" vrml97Hint="eventIn"/>
    <field name="setPlayerGPS" type="Vector2" vrml97Hint="eventIn"/>
<field name="setMapGPSCenter" type="Vector2" vrml97Hint="eventIn"/>
<field name="setMapZoomLevel" type="Integer" vrml97Hint="eventIn"/>
    <field name="onClick" type="Boolean" vrml97Hint="eventOut"/>
    <field name="onPlayerAround" type="Boolean" vrml97Hint="eventOut"/>
    <field name="onPlayerLeft" type="Boolean" vrml97Hint="eventOut"/>
</ProtoDeclare>
```

5.1.6.4.4.2 BIFS textual description

```
EXTERNPROTO MapMarker [
exposedField  SFString    name          ""
exposedField  SFVec2f     position      0.0 0.0
exposedField  SFFloat     radius        5.0
exposedField  SFRotation  rotation      0 0 1 0
exposedField  SFBool      clickable     TRUE
exposedField  SFBool      visible       TRUE
```

```
exposedField  SFBool      enabled        TRUE
exposedField  MFNode      markerShape    []
exposedField  MFString    keywords       []
eventIn       SFBool      doClick
eventIn       SFVec2f     setPlayerGPS
eventIn       SFVec2f     setMapGPSCenter
eventIn       SFInt32     setMapZoomLevel
eventOut      SFBool      onClick
eventOut      SFBool      onPlayerAround
eventOut      SFBool      onPlayerLeft
]"org:mpeg:MapMarker"
```

5.1.6.4.4.3 Functionality and semantics

The **MapMarker** proto allows creating marker instances that may be used to represent additional information placed on the Map at a specified GPS position. In order for a **MapMarker** to be overlaid on the map a **MapOverlay** instance is needed. The visual representation of a **MapMarker** can be any 2D or 3D object (e.g. an image, a video, a sphere, a complex 3D graphical object, etc.).

name specifies a unique name of the **MapMarker** instance. It helps identifying a specific **MapMarker** instance for further actions.

The **visible** field stores a Boolean value which specifies if the **MapMarker** node is visible on the map or not. A MapMarker is considered to be visible when its corresponding appearance node is displayed over the Map image instance.

The **clickable** field stores a Boolean value which specifies if the **MapMarker** node is clickable. A clickable **MapMarker** instance generates a TRUE Boolean output event, "onClick", when tapped.

enabled specifies if the **MapMarker** is enabled. An enabled **MapMarker** generates two Boolean output events always TRUE (onPlayerAround, onPlayerLeft) each time the player enters/exits the area covered by the **MapMarker**. This area is a circle centered in the **MapMarker** GPS position with a specified radius.

position is a 2D vector that specifies the GPS location of the **MapMarker**. The first value of the vector specifies the latitude and the second one specifies the longitude.

radius is a float value that defines a circle centered in **MapMarker's position** with a radius of X meters, where X is the value given to the radius field. The area of the circle defines the active zone (active region) of the **MapMarker**. A value of 0 specifies that the **MapMarker** does not have an active region.

rotation specifies an arbitrary rotation of the marker. The first three values specify a normalized rotation axis vector about which the rotation takes place whilst the forth value specifies the amount of right-handed rotation about that axis in radians.

The **keywords** field specifies a semantic description of the specific **MapOverlay** node (e.g. "restaurant", "museum", etc.).

markerShape is a list of nodes representing the visual appearance of the current **MapMarker** instance that should be overlaid on the **Map** image if its visible field is TRUE.

doClick input event simulates a click action on the **MapMarker** visual representation.

setPlayerGPS is a 2D vector eventIn representing the current GPS location of the player, the latitude respectively the longitude. The player GPS position should be used to compute the distance between the **MapMarker** and the player. Based on the computed distance **onPlayerAround** and **onPlayerLeft** output events are triggered whenever the distance conditions are fulfilled.

setMapZoomLevel represents the current zoom level of the Map. The zoom level of the Map is needed to compute the (x, y) coordinates of the **MapMarker** in the local coordinate system of the Map instance.

setMapGPSCenter is a 2D vector input event specifying the GPS position of the **Map** center. Beside the zoomLevel (described above), the GPS center of the **Map** is also required in order to compute the location of the MapMarker in the current coordinate system of the **Map** instance.

Each **MapMarker** shall be attached to a **MapOverlay**. There has to be at least one **MapOverlay** attached to the **Map** instance in order to be able to add and eventually display **MapMarkers** on the **Map**. Review **Map** PROTO (see 5.1.6.4.2) and **MapOverlay** PROTO (see 5.1.6.4.2) for details.

5.1.6.4.5 MapPlayer

5.1.6.4.5.1 XSD description

```
<ProtoDeclare name="MapPlayer" locations="org:mpeg:MapPlayer">
    <field name="name" type="String" vrml97Hint="exposedField" stringValue = ""/>
    <field name="visible" type="Boolean" vrml97Hint="exposedField" booleanValue = "TRUE"/>
    <field name="position" type="Vector2" vrml97Hint="exposedField" vector2Value = "0 0"/>
    <field name="playerShape" type="Nodes" vrml97Hint="exposedField">
       <nodes></nodes>
    <field>
    <field name="setMapGPSCenter" type="Vector2" vrml97Hint="eventIn"/>
    <field name="setMapZoomLevel" type="Integer" vrml97Hint="eventIn"/>
</ProtoDeclare>
```

5.1.6.4.5.2 BIFS textual description

```
EXTERNPROTO MapPlayer [
    exposedField  SFString   name             ""
    exposedField  SFVec2f    position         0 0
    exposedField  SFBool     visible          TRUE
    exposedField  MFNode     playerShape      []
    eventIn       SFInt32    setMapZoomLevel
    eventIn       SFVec2f    setMapGPSCenter
]" org:mpeg:MapPlayer"
```

5.1.6.4.5.3 Functionality and semantics

The **MapPlayer** proto allows creating a visual representation of the player on the Map. The player location on the Map is represented by the real GPS position of the device. Each location change (GPS position) of the device should also affect the player location on the **Map**. The visual representation of a **MapPlayer** can be any 2D/3D object (e.g. an image, a video, a sphere, a complex 3D graphical object, etc.).

name specifies a unique name of the **MapPlayer** instance. It helps identifying a specific **MapPlayer** instance for further actions. The name may be useful in a multiplayer application. A standalone application should not have more than one **MapPlayer** instance.

position is a 2D vector that specifies the GPS location of the **MapPlayer**.

The **visible** field stores a Boolean value which specifies if the **MapPlayer** node is visible on the map or not. A **MapPlayer** is considered to be visible when its corresponding appearance node is displayed over the **Map** image instance.

playerShape is a list of nodes representing the visual appearance of the **MapPlayer** instance that should be overlaid on the **Map**. The playerShape should be displayed on the **Map** only when “visible” is TRUE.

setMapZoomLevel represents the current zoom level of the **Map**. The zoom level of the **Map** is needed to compute the (x, y) coordinates of the **MapPlayer** in the local coordinate system of the Map instance.

setMapGPSCenter is a 2D vector input event specifying the GPS position of the **Map** center. Beside the zoomLevel (described above), the GPS center of the **Map** is also required in order to compute the location of the **MapPlayer** in the current coordinate system of the **Map** instance.

The implementation of **MapPlayer** automatically computes the player translation in the local coordinate system of the **Map** instance whenever the GPS location of the device changes. The recommended way of adding a **MapPlayer** instance to a **Map** is using a dedicated **MapOverlay**. The **MapPlayer** is nothing else but a special marker that has a slightly different behavior compared to an ordinary **MapMarker** (see 5.1.6.4.4).

5.1.6.4.6 Map example

```
Map {
        name            "CatMap"
        mapSize         768 768
        mapGPSCenter    48.625252 2.442515
        zoomLevel       19
        overlays        [
           DEF AngryCatOverlay MapOverlay {
             name "AngryCatOverlay"
             children [
             DEF AngryCat1 MARKER {
                 name "AngryCat1"
                 markerShape [USE AC_MARKER]
                 position 48.625240 2.442301
                 radius 5.0
                 clickable FALSE
                 enabled TRUE
                 visible TRUE
             }
             DEF AngryCat2 MARKER {
                 name "AngryCat2"
                 markerShape [USE AC_MARKER]
                 position 48.625006 2.442843
                 radius 5.0
                 clickable FALSE
                 enabled TRUE
                 visible TRUE
             }
             DEF AngryCat3 MARKER {
                 name "AngryCat3"
                 markerShape [USE AC_MARKER]
                 position 48.624974, 2.442407
                 radius 5.0
                 clickable FALSE
                 enabled TRUE
                 visible TRUE
             }
             DEF AngryCat4 MARKER {
                 name "AngryCat4"
                 markerShape [USE AC_MARKER]
                 position 48.624974, 2.443407
                 radius 5.0
                 clickable FALSE
                 enabled TRUE
                 visible TRUE
                 }
               ]
             }
             DEF SleepyCatOverlay MapOverlay {
                 name "SleepyCatOverlay"
                 children [
                   DEF SleepyCat1 MARKER {
                     name "SleepyCat1"
                     markerShape [USE SC_MARKER]
                     position 48.625540 2.442501
                     radius 5.0
                     clickable FALSE
                     enabled TRUE
                     visible TRUE
                   }
                   DEF SleepyCat2 MARKER {
                     name "SleepyCat2"
                     markerShape [USE SC_MARKER]
                     position 48.625306 2.442243
                     radius 5.0
                     clickable FALSE
                     enabled TRUE
                     visible TRUE
                   }
                   DEF SleepyCat3 MARKER {
```

```
                    name "SleepyCat3"
                    markerShape [USE SC_MARKER]
                    position 48.624774, 2.442707
                    radius 5.0
                    clickable FALSE
                    enabled TRUE
                    visible TRUE
                  }
                ]
              }
              DEF CheeseOverlay MapOverlay {
                name "CheeseOverlay"
                children [
                    DEF CheesePiece1 MARKER {
                      name "CheesePiece1"
                      markerShape [USE CHEESE_MARKER]
                      position 48.625352 2.442455
                      radius 5.0
                      clickable FALSE
                      enabled TRUE
                      visible TRUE
                    }
                    DEF CheesePiece2 MARKER {
                      name "CheesePiece2"
                      markerShape [USE CHEESE_MARKER]
                      position 48.625822 2.442365
                      radius 5.0
                      clickable FALSE
                      enabled TRUE
                      visible TRUE
                    }
                    DEF CheesePiece3 MARKER {
                      name "CheesePiece3"
                      markerShape [USE CHEESE_MARKER]
                      position 48.625552 2.442015
                      radius 5.0
                      clickable FALSE
                      enabled TRUE
                      visible TRUE
                    }
                ]
            }
        ]
    }
```

5.1.7 Terminal

5.1.7.1 General

The following terminal related node is used in ARAF: TermCap.

5.1.7.2 TermCap

5.1.7.2.1 XSD description

```
<complexType name="TermCapType">
    <all>
     <element ref="xmta:IS" minOccurs="0"/>
    </all>
    <attribute name="capability" type="xmta:SFInt32" use="optional" default="0"
/>
    <attributeGroup ref="xmta:DefUseGroup"/>
  </complexType>
  <element name="TermCap" type="xmta:TermCapType"/>
```

5.1.7.2.2 Functionality and semantics

As defined in ISO/IEC 14496-11:2015, 7.2.2.125.

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 specified in Table 36.

Table36 — TermCap capabilities

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

6 ARAF for sensors and actuators

6.1 General

The data captured from sensors or used to command actuators in ARAF are based on ISO/IEC 23005-5.

MPEG-V provides an architecture and specifies associated information representations to enable the representation of the context and to ensure interoperability between virtual worlds. Concerning ARAF, MPEG-V specifies the interaction between the virtual world and the real world by implementing support for accessing different input/output devices, e.g. sensors, actuators, vision and rendering, robotics.

ARAF supports two manners to connect the scene to the sensor/actuators. A first manner is by using the InputSensor and OutputActuator nodes. The second manner is based on dedicate nodes in the scene graph that maps directly the sensor/actuator (e.g. the CameraSensor PROTO).

6.1.1 Usage of InputSensor and script nodes

6.1.1.1 Overview

The InputSensor node is used to receive the MPEG-V sensor data in a scene or to transmit data to MPEG-V actuators from the scene. It should be noted that the data is pushed in the scene and it is applied immediately when received. Figure 12 represents the architecture for accessing MPEG-V sensor data in ARAF scenes.

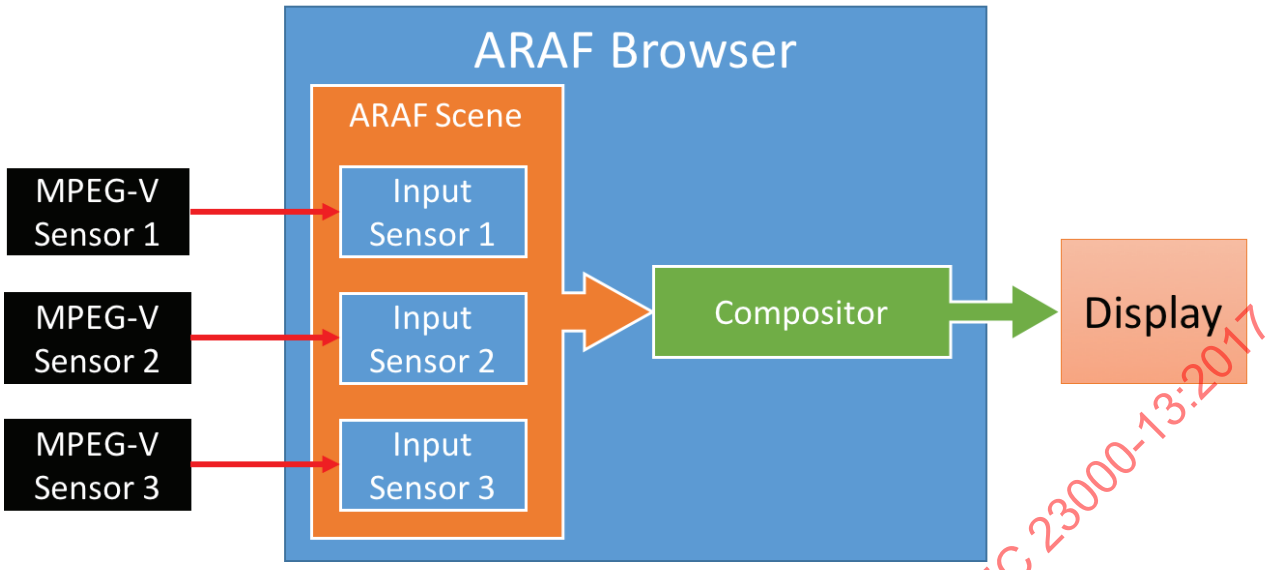


Figure 12 — Diagram of the architecture for accessing MPEG-V sensor data

As specified in ISO/IEC 14496-1, in order to add new devices for the InputSensor node, it is necessary to define:

— the content of the Device Data Frame (DDF) definition: this sets the order and type of the data coming from the device and then mandates the content of the InputSensor buffer;

— **deviceName** string which will designate the new device;

— optional **devSpecInfo** of **UIConfig.**

6.1.1.2 Orientation sensor

The definition of MPEG-V Orientation Sensor DDF is the following:

```
MPEGVOrientationSensorType [
   SFVec3F angles
]
```

The angles are specified as Euler angles as defined in ISO/IEC 23005-5. The deviceName is "MPEG-V:siv: OrientationSensorType". The UIConfig.devSpecInfo contains one 32-bit integer specifying the desired refresh frame-rate for the sensor.

6.1.1.3 Position sensor

The definition of MPEG-V Position Sensor DDF is the following:

```
MPEGVPositionSensorType [
   SFVec3F position
]
```

The *position* is specified in meters. The deviceName is "MPEG-V:siv: PositionSensorType". The UIConfig. devSpecInfo contains one 32-bit integer specifying the desired refresh frame-rate for the sensor.

6.1.1.4 Acceleration sensor

The definition of MPEG-V Acceleration Sensor DDF is the following:

```
MPEGVAccelerationSensorType [
   SFVec3F acceleration
]
```

The deviceName is “MPEG-V:siv: AccelerationSensorType”. The UIConfig.devSpecInfo contains one 32-bit integer specifying the desired refresh frame-rate for the sensor.

6.1.1.5 Angular velocity

The definition of MPEG-V Angular Velocity Sensor DDF is the following:

```
MPEGVAngularVelocitySensorType [
   SFVec3F AngularVelocity
]
```

The deviceName is “MPEG-V:siv: AngularVelocitySensorType”. The UIConfig.devSpecInfo contains one 32-bit integer specifying the desired refresh frame-rate for the sensor.

6.1.1.6 Global position system sensor

The definition of MPEG-V Global Position System Sensor DDF is the following:

```
MPEGVGPSSensorType [
   SFVec2F location
]
```

The deviceName is “MPEG-V:siv:GPSSensorType”. The UIConfig.devSpecInfo contains one 32-bit integer specifying the desired refresh frame-rate for the sensor.

6.1.1.7 Altitude sensor

The definition of MPEG-V Altitude Sensor DDF is the following:

```
MPEGVAltitudeSensorType [
   SFFloat altitude
]
```

The deviceName is “MPEG-V:siv:AltitudeSensorType”. The UIConfig.devSpecInfo contains one 32-bit integer specifying the desired refresh frame-rate for the sensor.

6.1.1.8 Geomagnetic sensor

The definition of MPEG-V Geomagnetic Sensor DDF is the following:

```
MPEGVGeomagneticSensorType [
   SFVec3F geomagnetic
]
```

The deviceName is “MPEG-V:siv: GeomagneticSensorType”. The UIConfig.devSpecInfo contains one 32-bit integer specifying the desired refresh frame-rate for the sensor.

6.1.1.9 Example of integrating sensors in the ARAF scene

In the following example, it is shown how the InputSensor and Script node can be used to access MPEG-V sensors.

```
DEF SCRIPT Script {
   eventIn SFVec3f updateOrientation
   . . . .
   url ["javascript:

      function updateOrientation(rot)
       {
         if ( objrot.children.length == 0 )
           return;
         Azimuth = rot.x;
         Pitch = rot.y;
         Roll = rot.z;
         conv = 3.14/180/2;
         c1 = Math.cos(Azimuth * conv);
         s1 = Math.sin(Azimuth * conv);
         c2 = Math.cos(Pitch * conv);
```

```
                s2 = Math.sin(Pitch * conv);
                c3 = Math.cos(Roll * conv);
                s3 = Math.sin(Roll * conv);

                c1c2 = c1*c2;
                s1s2 = s1*s2;

                w = c1c2*c3 - s1s2*s3;
                x = c1c2*s3 + s1s2*c3;
                y = s1*c2*c3 + c1*s2*s3;
                z = c1*s2*c3 - s1*c2*s3;

                angle = 2 * Math.acos(w);
                norm = x*x + y*y + z*z;

                if ( norm < 0.001 ) {
                   x = 1;
                   y = z = 0;
                }
                else {
                   norm = Math.sqrt(norm);
                   x /= norm;
                   y /= norm;
                   z /= norm;
                }

                objrot.rotation = new SFRotation( x, z, y, angle);
            }
            . . . .
     "]
    }
    DEF ORIENT_SENS InputSensor {
          url [50]
           buffer {
                REPLACE SCRIPT.updateOrientation BY 0 0 0
   }
}
```

where `url [50]` is the object descriptor for the orientation sensor defined as follows:

```
      ObjectDescriptor {
        objectDescriptorID 50
        esDescr [
          ES_Descriptor {
            ES_ID 50
            decConfigDescr DecoderConfigDescriptor {
              streamType 10
              decSpecificInfo UIConfig {
                deviceName "MPEG-V:siv:OrientationSensorType"
             }
            }
        }
    ]
}
```

6.2 Access to local camera sensor

The camera frames are directly accessed by the ARAF player. Figure 13 presents the diagram for accessing the camera video stream.

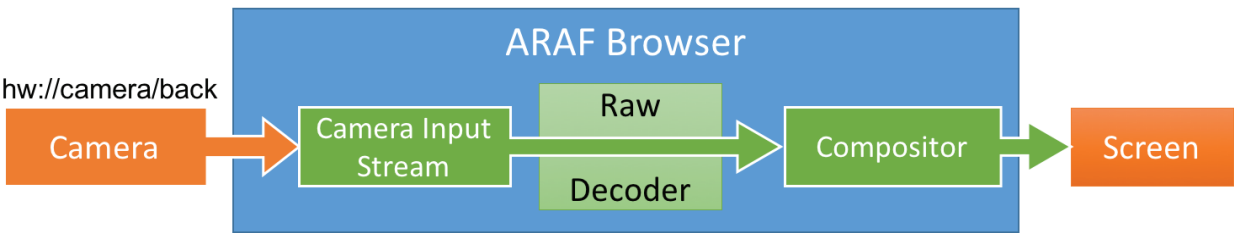


Figure 13 — Diagram of the architecture for accessing the camera frames

The following are defined:

a) An URN for the camera in order to initialize the input stream:

   — for the back camera of the device: *hw://camera/back*;

   — for the front camera of the device: *hw://camera/front*.

b) A type of video stream that doesn't need to be decoded (RAW decoder). As specified in ISO/IEC 14496-1, the following decoder specific info for the RAW decoder is defined:

```
class RAWVideoConfig extends DecoderSpecificInfo : bit(8)
tag=DecSpecificInfoTag {
unsigned int(16)  width;
unsigned int(16)  height;
unsigned int(8)   bit_depth;
unsigned int(32)  stride;
unsigned int(32)  coding4CC;
unsigned int(8)   fps;
unsigned int(1)   use_frame_packing;
unsigned int(7)   frame_packing;
}
```

6.3 Usage of outputactuator and script nodes

6.3.1 General

The OutputActuator proto is used to transmit data to MPEG-V actuators from the scene. It should be noted that the data produced by the scene is applied immediately when received by the actuators. Figure 14 represents the architecture for commanding MPEG-V actuators from the ARAF scenes.

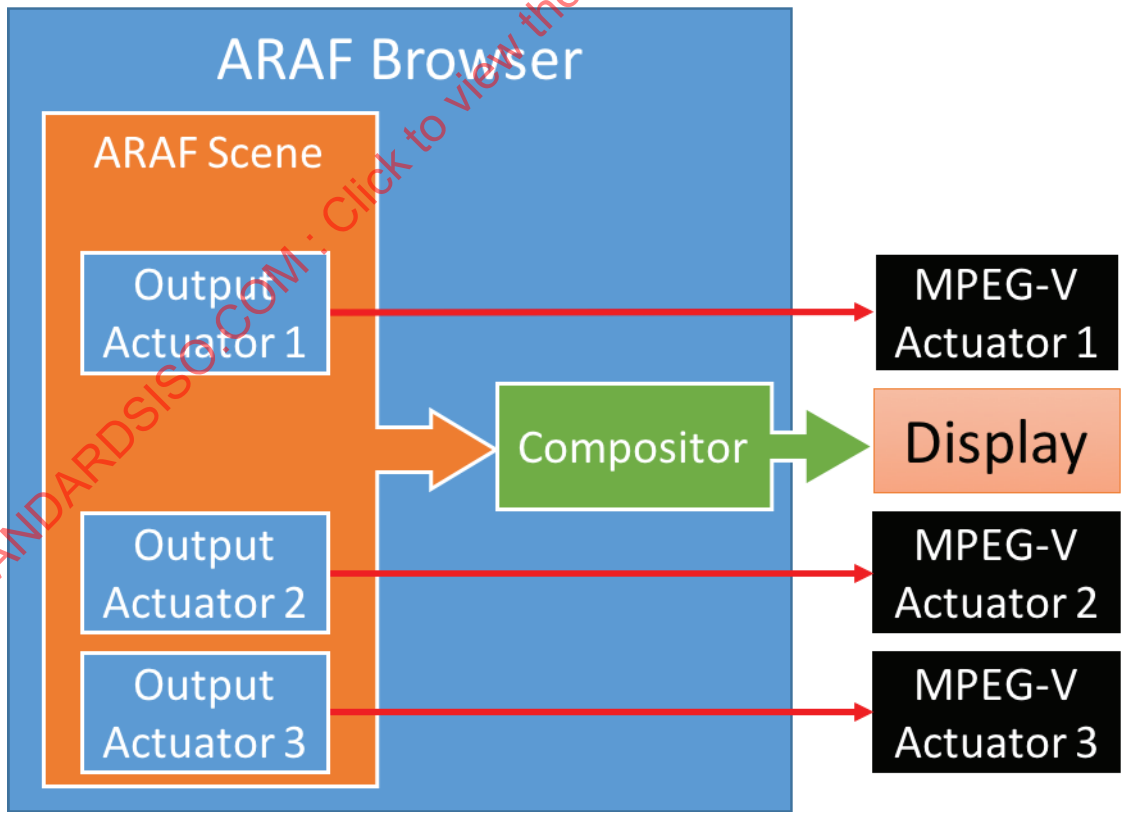


Figure 144 — Diagram of the architecture for commanding MPEG-V actuators