



Technology under Consideration for ISO/IEC 23090-14

WG3 Scene Description BoG

MDS24736_WG03_N01436

Table of Contents

1. Extensions	1
1.1. Support of Spatial Computing in SD*	1
1.1.1. Introduction	1
1.1.2. API to access to an external spatial computing module	1
1.1.3. Semantic Update of the MPEG_anchor extension	4
1.1.4. Processing Model	9
1.1.5. References	9
2. Codec Support	10
2.1. Dynamic mesh support in scene description	10
2.2. Support for multiple atlases for MIV applications (MPEG142)	10
2.2.1. Multiple atlases	10
2.2.2. References	17
2.3. Support for multi-view video and multi-channel audio sources	17
2.3.1. Introduction	17
2.3.2. Potential Solution to Support Multi-view Video	18
2.3.3. References	19
3. Interfaces	20
3.1. Supporting Multiple Viewers in the Media Access Function	20
3.1.1. General	20
3.1.2. Proposed Updates to MAF API	20
3.2. Generic API for Presentation Engine	21
3.2.1. Generic Render Control API	22
3.2.2. Extension for Audio Node Mapping	25
4. MPEG-I Audio in Scene Description	27
4.1. On spatial synchronization between graphs	27
4.1.1. Attempt problem definition for the spatial synchronization	27
4.1.2. Approach proposal for the spatial synchronization	29
4.1.3. Conclusion	31
4.1.4. References	31
4.2. Immersive audio support in Scene Description	31
4.2.1. Introduction	31
4.2.2. Main assumptions	32
4.2.3. Support of immersive audio in Scene Description	33
4.2.4. References	36
5. Interactivity framework	37
5.1. On event-based scene update	37
5.1.1. General	37
5.1.2. A use case for event based updates	38

5.1.3. JSON patch limitations	39
5.1.4. Semantics for event-based update	40
6. Collected problem statements and industry needs	42
6.1. On the support of real environment data	42
6.1.1. General	42
6.1.2. Representation of the real environment	42
6.1.3. Storing a representation of the real environment	43
6.1.4. Examples of framework for real environment handling	44
7. Avatar	47
7.1. Update of the Description of the MPEG reference avatar model Morgan	47
7.1.1. Introduction	47
7.1.2. References	53
Appendix A: Disclaimer	54

Chapter 1. Extensions

1.1. Support of Spatial Computing in SD*

Source [m70188](#)

1.1.1. Introduction

The m70186 contribution [1] proposes a use case related to spatial computing and some requirements related to the support of spatial computing in MPEG-SD.

The m67595 contribution presented some configuration examples of XR spatial computing in recent AR devices (e.g. Apple “*scene reconstruction and understanding*” API and Microsoft *XR_MSFT_scene_understanding* Khronos OpenXR vendor extension). It also proposed a high-level architecture to address the need of time and spatial synchronizations between the Scene Description graph managed by a Presentation Engine and the representation of the user real environment managed by an external Spatial Computing module.

This contribution proposes an API that can be exposed by such a Spatial Computing module to configure and retrieve the representation of the real scene around the user and to integrate a virtual scene into this real scene. The proposed API is inspired by the one specified in the Microsoft OpenXR scene understanding extension ([3]).

This contribution also proposes new semantics for the MPEG_anchor extension of MPEG scene description ([1]), to provide configuration parameters for the spatial computing module.

1.1.2. API to access to an external spatial computing module

The high-level architecture proposed in contribution m67595 is provided in Figure 1.

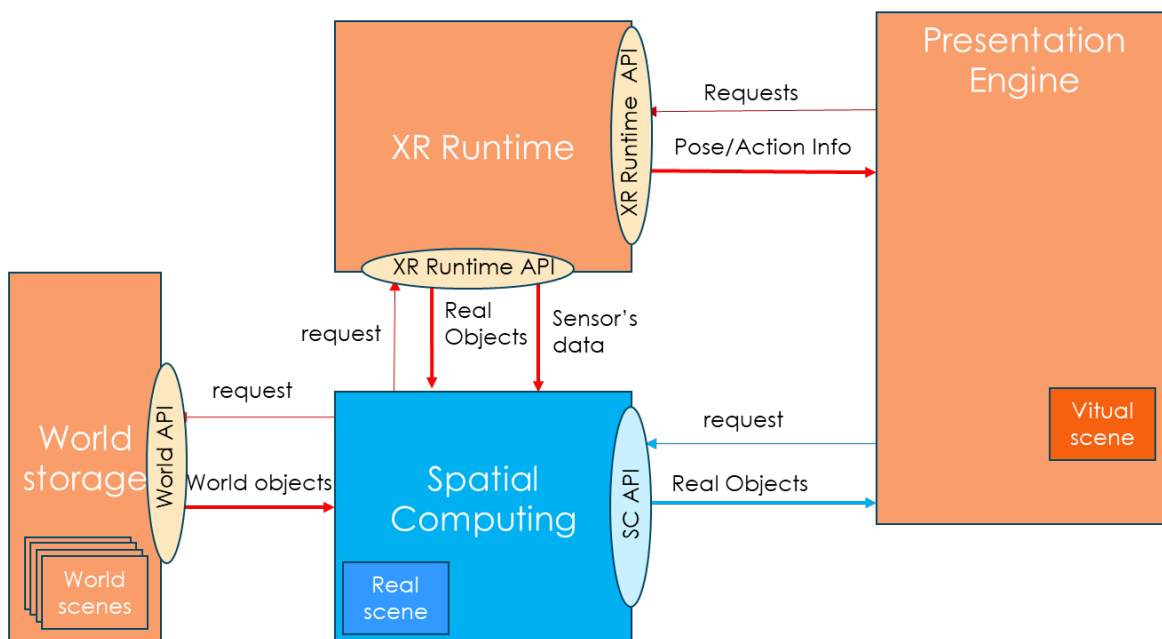


Figure 1:High-level architecture for XR Spatial Computing Support

A Spatial Computing API exposes to the presentation engine a set of functions:

- To initialize and configure the external spatial computing module: creation of a session between the two modules, setting of parameters needed by the external module to handle its scene.
- To start or stop the external module processing.
- To notify the external module that update occurred in the scene managed by the presentation engine.
- To register callbacks for the external module to request status from the PE module or to notify updates occurred in the real scene.

A spatial computing module maintains a representation of the user environment. It is the unique entry point for a presentation engine module to get the knowledge of the real scene around the user and to integrate a virtual scene into this real scene.

The following table describes the functionalities that may be provided by this Spatial Computing API.

Method	Description
init ()	Initialization of the SC module and creation of a session between the PE and the SC module.
configure ()	<p>Configures the way the real scene must be scanned, by specifying a set of parameters. Those parameters should be linked to a reference XR space, e.g. the one related to a trackable or an Anchor.</p> <p>In case of multiple trackables or Anchors defined for the XR experience, multiple set should be specified.</p> <p>For each XR space, the parameters to this method may include:</p> <ul style="list-style-type: none">• An id for the XR space (xrSpaceId)• An occurrence value specifying the rate at which the scan must be updated.• A set of options for the scan processing, such as level of details, mesh types...),• A set of volumes where scanned objects must be provided. Real scan objects that intersect one or more of the bounding volumes should be used, and all other objects ignored. Scan volumes may overlap between 2 XR spaces. In that case, real object contained in the overlap section may be provided twice, or only once for one of the spaces.
start ()	Starts or stops the creation and update of the real scenes. The input parameter should be a set of xrSpaceId to specify the trackables around which the scan of real objects must be performed.
stop ()	

Method	Description
registerCallbacks ()	<p>Provides callback functions to the SC module to allow it to send updates or status of the world scene.</p> <p>A list of callbacks is given as input, with the following parameters for each one:</p> <ul style="list-style-type: none"> • A value giving the goal of the callback, e.g. an enumeration with the following possible values: UPDATES, STATUS..., • a pointer to a callback function <p>The parameters of an UPDATES callback are a list of objects that may include:</p> <ul style="list-style-type: none"> • The operation to perform, e.g. ADD, UPDATE or REMOVE • A timestamp value • For ADD, the reference and the xrSpaceId of the new object as well as a dictionary of attributes and their initialization values. • For UPDATE, the reference and the xrSpaceId of the object as well as a dictionary of attributes and their new values. • For REMOVE, the reference of the object to remove. • The SC module may automatically call this UPDATE callback at the same occurrence of the scan process, or it may call it when significative changes has occurred. <p>The parameters of a STATUS callback are a list of objects that may include:</p> <ul style="list-style-type: none"> • xrSpaceId • an object reference (if absent, all the scene related to xrSpaceId is considered) • the status of the referenced object e.g. ONGOING, COMPLETE, NOT_VALID, PARTIAL... • The SC module may send this STATUS callback multiple times after a scan process has been started or each time this status changes.

Method	Description
getSceneComponents ()	<p>Requests one or more element(s) of the world scene. The following set of optional input parameters give filtering criteria for this request:</p> <ul style="list-style-type: none"> • A time value, to request the world scene at a given historical time or predicted time. This parameter may be used with an XR device able to store past versions of the world scene or able to predict it in the future. A time limit in both directions may be specified, depending on the capacity of the XR UE (e.g. a few second in the past and a few tens of ms in the future). • xrSpaceId to request elements related to a given XR space. • A list of scan volumes to request elements contained in one of the specified volumes. • A list of object semantics to request elements with one of the given semantics (wall”, “floor”, “table”, “chair”, “light”, “sound”, “freespace” ...). • A list of object references to request elements with one of the given ids. <p>}</p> <p>This method should return a status of the world scene (same possible values as in the STATUS callback) and if OK (status = COMPLETE), should return the graph of the world scene or of some word scene elements, each returned element coming with a set of attributes that may include:</p> <ul style="list-style-type: none"> • An object reference • A set of visual, lighting, and physical attributes • A set of semantic attributes

Table 1: Spatial Computing API

1.1.3. Semantic Update of the MPEG_anchor extension

A content creator may want to express in a scene description file how its AR content interacts with the user environment. For that, some new spatial computing parameters are needed, that will give some recommended parameters to configure the spatial computing module.

In MPEG-SD, MPEG_anchor extension specifies an element of the real world where virtual elements should be positioned.

New parameters can be added in *MPEG_anchor* extension to configure the 3D Model Construction. The following tables gives the new semantics and possible values for each configuration parameters.

Table 2 – Definition of the Anchor object (table 27 of [1])

Name	Type	Usage	Default	Description
trackable	integer	M		Index of the trackable in the trackables array that will be used for this anchor.
requiresAnchoring	boolean	M		<p>Indicates if AR anchoring is required for the rendering of the associated nodes.</p> <p>If TRUE, the application shall skip the virtual assets attached to this anchor until the pose of this anchor in the real world is known.</p> <p>if FALSE, the application shall process the virtual assets attached to this anchor</p>
minimumRequiredSpace	vec3	O	(0,0,0)	Space required to anchor the AR asset (x, y, z in meters). This space corresponds to an axis-aligned bounding box, placed at the origin of the trackable space and extending along the positive x+,y+, and z+ axes, expressed in the trackable local space. This value shall be compared to the bounding box of the real-world available space associated with the trackable as estimated by the XR runtime.
aligned	enumeration	O	NOT_USED	<p>the aligned flag may take one of the following values: NOT_USED=0, ALIGNED_NOTSCALED=1, ALIGNED_SCALED=2.</p> <p>If ALIGNED_SCALED is set, the bounding box of the virtual assets attached to that anchor is aligned and scaled to match the bounding box of the real-world available space associated with the trackable as estimated by the XR runtime.</p>

Name	Type	Usage	Default	Description
actions	array(number)	O	N/A	Indices of the actions in the actions array of the interactivity extension to be executed once the pose of this anchor is determined. An example is a setTransform action to place the virtual assets attached to that anchor.
light	integer	O	N/A	Reference to an item in the lights array of the MPEG_lights_texture_based extension.
useReal	Boolean	O	False	<p>If true, the scan of the real word is needed for the integration of the virtual objects:</p> <ul style="list-style-type: none"> • As simplified meshes to be used for physics simulation (collision, occlusion...) • As full mesh/texture to handle relighting. • ...
If (useReal)				
scanOptions	Array	O	N/A	Array of options (enumeration) for the scan computation: possible values are given in Table 3.
scanDetails	Object	O	N/A	Specifies the required level of detail for the mesh (number/type of primitives/m3) and for the texture of the visual mesh. The semantics is presented in Table 4.
scanOccurence	Enum	O	N/A	Specifies when the scan of real objects must be updated (Table 5).
scanVolumes	Array	O	N/A	Array of bounding volumes that determine the spaces where scanned objects must be used. Real scan objects that intersect one or more of the bounding volumes should be used, and all other objects ignored. The semantics for these volumes are presented in Table 6.

Name	Type	Usage	Default	Description
realSemantic	Array	O	N/A	Semantic descriptions of nodes that are needed (“table”, “room”, “chair”, “wall”, “light”, “freespace” ...)

The possible values for a ***scanOptions*** item are given in the following table, with some of them based on the MSFT OpenXR extension [3].

Enumeration value	Description
PLANE	Request plane data for scanned objects
PLANAR_MESH	Request planar meshes for scanned objects
VISUAL_MESH	Request 3D visualization meshes for scanned objects
COLLIDER_MESH	Request 3D collider meshes for scanned objects
FREE_VOLUME	Request to get the available space around a trackable
POINT_CLOUD	Request a points cloud representation
BOUNDING_BOX	Request a simplified collider mesh
TEXTURED_MESH	Request mesh with a texture

Table 3: scanOptions item values

The semantics of a ***scanDetail*** object is given in the following table:

Parameter name	Type	Usage	Description
primitivesNumber	number	O	Gives the quantity of geometric primitives per m3
primitiveType	Enum	O	Gives the types of primitives: see Table 5
TextureOptions	Array	O	<p>Array of options (e.g. enumeration) for the texturing details. Possible values may be:</p> <ul style="list-style-type: none"> • LOW_RES, HIGH_RES for the resolution of the texture or the exact resolution (e.g. 1024x768), • LOW_LIGHT, HIGH_LIGHT for the level of lightning of the texture • RGB, RGBA...for the texture format • ...

Table 4: scanDetail objects semantics

The possible values for a primitive type are given in the following table:

Enumeration value	Description
QUAD	Mesh of 3D Model is represented by an array of Quad

Enumeration value	Description
TRIANGLE	Mesh of 3D Model is represented by an array of TRIANGLE
POINT CLOUD	3D Model is represented by a point cloud
GAUSSIAN_SPLAT	3D Model is represented by an array of gaussian

Table 5: primitiveType values

The possible values for the **scanOccurence** parameter are given in the following table. Several of these possible values may be combined to address different scenarios (e.g. ONCE AND AUTO).

Enumeration value	Description
ONCE	the scan is performed only once, for instance in case of a static real scene
N_FRAME	the scan is performed periodically, every N rendering frames, N depending on the dynamism of the real scene.
AUTO	the scan occurrence is managed by the SU module. The SU module may perform a light pre-analysis to detect significant moves in the real world and start a complete new scan. This pre-analysis may be performed from raw images data, like RGB images from a camera or depth images from a depth sensor.

Table 6:scan occurrence

Here is the semantics for a **scanVolumes** object (example based on the MSFT OpenXR extension [3]). 3D coordinates are expressed in the XR space related to trackable associated to the anchor.

Parameter name	Type	Usage	Description
type	Enum	M	SPHERE, BOX, FRUSTUM
If (type == SPHERE)			
Center	array	M	3D coordinate of the center of the sphere
Radius	number	M	Radius of the sphere in meters.
If (type == BOX)			
pose	matrix	M	4*4 matrix representing the center position and orientation of the box
extents	array	O	Edge-to-edge length of the box along each dimension.
If (type == FRUSTUM)			
pose	matrix	M	4*4 matrix representing the position and orientation of the tip of the frustum
fov	Vec4	M	Angles of the four sides of the frustum

Parameter name	Type	Usage	Description
far	number	M	Positive distance of the far plane or the frustum
near	number	M	Positive distance of the near plane or the frustum

Table 7: scanVolumes object semantics

1.1.4. Processing Model

To be added to the processing model of the anchor:

When processing the MPEG_anchor extension, if *useReal* is set to True, the Presentation Engine shall configure the Spatial Computing module with specifics parameters such as scanOptions, scanDetails, scanOccurence, scanVolumes, scanSemantic

At runtime, the Presentation Engine shall then use the API to request elements of the world scene or update the configuration.

1.1.5. References

[1] Text of ISO/IEC FDIS 23090-14 2nd edition Scene description, April 2024

[2] [OpenXR™](#)

[3] [XR_MSFT_scene_understanding](#) OpenXR extension

Chapter 2. Codec Support

2.1. Dynamic mesh support in scene description

V-DMC is considered for future Amendment

2.2. Support for multiple atlases for MIV applications (MPEG142)

Source: [m62515](#)

2.2.1. Multiple atlases

2.2.1.1. Motivation

A V3C bitstream can be decomposed into one or more atlas sub-bitstreams and their associated video sub-bitstreams. The video sub-bitstreams for each atlas may include video-coded occupancy, geometry, and attribute components. In the V3C parameter set (sub-clause 8.4.4.1 in [3]), `vps_atlas_count_minus1` plus 1 indicates the total number of atlases in the current bitstream. The value of `vps_atlas_count_minus1` is in the range of 0 to 63, inclusive.

With the proposal in Section 2.2.1 to support multiple atlases in the `MPEG_primitive_V3C` extension, MPEG-I SD remains future proof to any future derivation of V3C specification which may depend on multiple atlases along with common atlas data. One derived V3C specification in ISO/IEC 23090-12, specified the use of common atlas data which is common to atlases in the V3C bitstream.

2.2.1.2. Overview

The proposals take the following aspects into consideration:

- Logical grouping of the relevant syntax to describe an atlas in the `MPEG_primitive_V3C` extension.
- Use of `atlasID` property to identify the atlas identifier which is equal to `vps_atlas_id[k]` specified in 8.4.4.1 of ISO/IEC 23090-5[3]. In case there are multiple atlases in the V3C bitstream, `atlasID` provides a unique identifier stored in the bitstream to uniquely identify an atlas in `_MPEG_primitive_v3c` extension and establishes a corresponding relation with atlas definition in the bitstream.

2.2.1.3. Array of atlases

A new property is defined under the `_MPEG_primitive_V3C` extension named `atlases`. The `atlases` property is an array of components corresponding to an atlas. The length of the `atlases` array shall be equal to the number of atlases for a V3C object. The properties for an object in the `atlases` array describe the atlas data component and corresponding video-coded components such as attribute, occupancy, and geometry for a V3C object.

The `atlasID` property is an integer values, where each integer value refers to the `vps_atlas_id`

specified in sub-clause 8.4.4 in [3] for each atlas in the V3C bitstream.

2.2.1.3.1. MPEG_primitive_V3C

glTF extension to specify support for V3C compressed primitives.

Table 1. MPEG_primitive_V3C Properties

	Type	Description	Required
atlases	MPEG_primitive_V3C.atlas [1-*]	An array of atlases	✓ Yes
_MPEG_V3C_CAD	MPEG_primitive_V3C._MPEG_V3C_CAD	This object lists different properties described for the Common Atlas Data in ISO/IEC 23090-5.	No
extensions	object	JSON object with extension-specific objects.	No
extras	any	Application-specific data.	No

Additional properties are allowed.

- **JSON schema:** MPEG_primitive_V3C.schema.json

2.2.1.3.1.1. MPEG_primitive_V3C.atlases

An array of atlases

- **Type:** MPEG_primitive_V3C.atlas [1-*]
- **Required:** ✓ Yes

2.2.1.3.1.2. MPEG_primitive_V3C._MPEG_V3C_CAD

This object lists different properties described for the Common Atlas Data in ISO/IEC 23090-5.

- **Type:** MPEG_primitive_V3C._MPEG_V3C_CAD
- **Required:** No

2.2.1.3.1.3. MPEG_primitive_V3C.extensions

JSON object with extension-specific objects.

- **Type:** object
- **Required:** No
- **Type of each property:** Extension

2.2.1.3.1.4. MPEG_primitive_V3C.extras

Application-specific data.

- **Type:** any
- **Required:** No

2.2.1.3.2. MPEG_primitive_V3C._MPEG_V3C_CAD

defines the common atlas data for a v3c object

Table 2. MPEG_primitive_V3C._MPEG_V3C_CAD Properties

	Type	Description	Required
MIV_view_parameters	integer	indicates the accessor index which is used to refer to the list of MIV view parameters.	✓ Yes
extensions	object	JSON object with extension-specific objects.	No
extras	any	Application-specific data.	No

Additional properties are allowed.

- **JSON schema:** MPEG_primitive_V3C._MPEG_V3C_CAD.schema.json

2.2.1.3.2.1. MPEG_primitive_V3C._MPEG_V3C_CAD.MIV_view_parameters

indicates the accessor index which is used to refer to the list of MIV view parameters.

- **Type:** integer
- **Required:** ✓ Yes
- **Minimum:** >= 1

2.2.1.3.2.2. MPEG_primitive_V3C._MPEG_V3C_CAD.extensions

JSON object with extension-specific objects.

- **Type:** object
- **Required:** No
- **Type of each property:** Extension

2.2.1.3.2.3. MPEG_primitive_V3C._MPEG_V3C_CAD.extras

Application-specific data.

- **Type:** *any*
- **Required:** No

2.2.1.3.3. MPEG_primitive_V3C.atlas

glTF extension to specify support for V3C compressed primitives.

Table 3. *MPEG_primitive_V3C.atlas Properties*

	Type	Description	Required
_MPEG_V3C_CONFIG	<i>integer</i>		✓ Yes
_MPEG_V3C_AD	<i>integer</i>		✓ Yes
_MPEG_V3C_GVD_MAPS	<i>integer [1-*</i>	an array of references to video texture maps.	✓ Yes
_MPEG_V3C_OVD_MAP	<i>integer [0-*</i>	a reference to a video texture that provides the occupancy map	No
_MPEG_V3C_AVD	<i>MPEG_primitive_V3C.attribute [0-*</i>		No
_MPEG_V3C_CAD	<i>object</i>	This object lists different properties described for the Common Atlas Data in ISO/IEC 23090-5.	No
extensions	<i>object</i>	JSON object with extension-specific objects.	No
extras	<i>any</i>	Application-specific data.	No

Additional properties are allowed.

- **JSON schema:** *MPEG_primitive_V3C.atlas.schema.json*

2.2.1.3.3.1. MPEG_primitive_V3C.atlas._MPEG_V3C_CONFIG

- **Type:** *integer*
- **Required:** ✓ Yes
- **Minimum:** *>= 0*

2.2.1.3.3.2. MPEG_primitive_V3C.atlas._MPEG_V3C_AD

a reference to the accessor that points to the atlas data.

- **Type:** *integer*

- **Required:** ✓ Yes
- **Minimum:** ≥ 0

2.2.1.3.3.3. MPEG_primitive_V3C.atlas._MPEG_V3C_GVD_MAPS

an array of references to video textures that provide the geometry maps.

- **Type:** `integer [1-*)`
 - Each element in the array **MUST** be greater than or equal to `0`.
- **Required:** ✓ Yes

2.2.1.3.3.4. MPEG_primitive_V3C.atlas._MPEG_V3C_OVD_MAP

a reference to a video texture that provides the occupancy map

- **Type:** `integer [0-*)`
 - Each element in the array **MUST** be greater than or equal to `0`.
- **Required:** No

2.2.1.3.3.5. MPEG_primitive_V3C.atlas._MPEG_V3C_AVD

An array of references to the video textures that provide the attribute data

- **Type:** `MPEG_primitive_V3C.attribute [0-*)`
- **Required:** No

2.2.1.3.3.6. MPEG_primitive_V3C.atlas._MPEG_V3C_CAD

This object lists different properties described for the Common Atlas Data in ISO/IEC 23090-5.

- **Type:** `object`
- **Required:** No

2.2.1.3.3.7. MPEG_primitive_V3C.atlas.extensions

JSON object with extension-specific objects.

- **Type:** `object`
- **Required:** No
- **Type of each property:** Extension

2.2.1.3.3.8. MPEG_primitive_V3C.atlas.extras

Application-specific data.

- **Type:** `any`
- **Required:** No

2.2.1.3.4. MPEG_primitive_V3C.attribute

defines the attribute of a V3C object.

Table 4. MPEG_primitive_V3C.attribute Properties

	Type	Description	Required
type	integer	provides the type of the attribute.	No
maps	integer [1-*]		✓ Yes
extensions	object	JSON object with extension-specific objects.	No
extras	any	Application-specific data.	No

Additional properties are allowed.

- **JSON schema:** MPEG_primitive_V3C.attribute.schema.json

2.2.1.3.4.1. MPEG_primitive_V3C.attribute.type

provides the type of the attribute.

- **Type:** integer
- **Required:** No
- **Minimum:** ≥ 0
- **Maximum:** ≤ 255

2.2.1.3.4.2. MPEG_primitive_V3C.attribute.maps

provides the references to the corresponding video texture maps.

- **Type:** integer [1-*]
 - Each element in the array **MUST** be greater than or equal to 0.
- **Required:** ✓ Yes

2.2.1.3.4.3. MPEG_primitive_V3C.attribute.extensions

JSON object with extension-specific objects.

- **Type:** object
- **Required:** No
- **Type of each property:** Extension

2.2.1.3.4.4. MPEG_primitive_V3C.attribute.extras

Application-specific data.

- **Type:** any
- **Required:** No

Following is an example illustrating the use of the syntax described in [Section 2.2.1.3.3](#)

```
{
  "meshes": [{
    "name": "v3c_mesh",
    "primitives": [{
      "attributes": {
        "POSITION": 0,
        "COLOR_0": 1
      },
      "mode": 0,
      "extensions": {
        "MPEG_primitive_V3C": {
          "atlases": [{
            "atlasID": 1,
            "_MPEG_V3C_OVD_MAPS": [2],
            "_MPEG_V3C_GVD_MAPS": [3, 4],
            "_MPEG_V3C_AVD": [{
              "type": 0,
              "maps": [5, 6]
            },
            {
              "type": 4,
              "maps": [7, 8]
            }
          ],
            "_MPEG_V3C_CONFIG": 9,
            "_MPEG_V3C_AD": {
              "buffer_format": "baseline",
              "accessor": 10
            }
          },
          "_MPEG_V3C_CAD": {
            "MIV_view_parameters": 114
          }
        }
      }
    }
  ]
}
```

2.2.2. References

[1] m61138, "Support for multiple atlases for MIV application", MPEG 140, Mainz Meeting, October 2022.

[2] WG7N00553, "Technologies under Consideration on Scene description", MPEG 141, Online, January 2023.

[3] ISO/IEC 23090-5:2021 Information technology — Coded representation of immersive media — Part 5: Visual volumetric video-based coding (V3C) and video-based point cloud compression (V-PCC), Online, <https://www.iso.org/standard/73025.html>

2.3. Support for multi-view video and multi-channel audio sources

Source: [m71320](#)

2.3.1. Introduction

The MPEG-I Scene Description extensions to glTF 2.0, specifically `MPEG_texture_video` and `MPEG_audio_spatial`, enhance the integration of dynamic media elements within 3D scenes, facilitating more immersive and interactive experiences.

The `MPEG_texture_video` extension enables the incorporation of video textures on 3D models. By linking a glTF texture object to external media and its respective track, this extension allows for the dynamic updating of textures in real-time, such as applying a live video feed to a surface within the 3D environment. This is achieved through a reference to a timed accessor, which provides access to the decoded video frames for seamless integration.

The `MPEG_audio_spatial` extension introduces support for spatial audio within glTF scenes. It allows for the definition of audio sources (`source`), reverb effects (`reverb`), and listeners (`listener`) within the scene graph. Audio sources can be of type 'Object' for mono audio or 'HOA' for Higher-Order Ambisonics, enabling 3D positional audio rendering. Reverb effects can be applied to audio sources to simulate environmental acoustics, and listeners, typically attached to camera nodes, represent the audio output in the scene, ensuring that audio perception aligns with the viewer's position and orientation.

Recently, Spatial video, a 3D format that adds depth and dimension to traditional 2D videos, has gained popularity among users due to the immersive experience that it provides and the ease of capturing on Apple devices. Unfortunately, rendering of spatial video textures that are integrated in 3D scenes is not supported yet.

Similarly, multi-channel audio formats have become integral to modern entertainment, delivering immersive sound experiences across various platforms. These formats are prevalent in home theater systems, streaming services, and gaming consoles, providing listeners with rich, surround sound that enhances the realism of audio content. Unfortunately, the `MPEG_audio_spatial` extension currently lacks support for multi-channel audio sources.

In this contribution, we propose to start working on the necessary extensions to the

MPEG_texture_video and MPEG_audio_spatial to add support for multi-view and multi-channel sources in Scene Description.

2.3.2. Potential Solution to Support Multi-view Video

In order to support multi-view video sources in 3D scenes, it is important to provide both all view textures as well as the necessary metadata to enable the Presentation Engine to properly render the video texture.

In the simplest scenario, a stereo video is used as a texture, where the intrinsic and extrinsic parameters of the scene camera match those of the stereo camera used to capture the stereo video. In such a scenario, the left video texture is shown to the left eye and the right video texture is shown to the right eye. No further processing would be needed.

However, for scenarios where there are multiple views or where the stereo rendering cameras do not match the capture cameras, re-projection of the views would be required.

To enable these different scenarios, we propose the following signaling as part of the **MPEG_texture_video** glTF extensions.

Name	Type	Default	Usage	Description
extras				Extensions go into extras element of MPEG_texture_video extension.
group_id	number	N/A	M	A group identifier that indicates that multiple video textures are associated together.
group_type	Enum	N/A	M	Indicates the type of the group, currently “stereo” and “multi-view” types are defined.
camera	Object	N/A	O	Intrinsic and extrinsic camera parameters object, that is assumed to be the target for rendering the content
mask	Enum	right	O	Mask that indicates to which eye this video texture is visible

For cases where all views are multiplexed into a single track, the track array in the **alternatives** element of the **MPEG_media** may reference a particular layer by using the following syntax:

reference = track_id “:” layer_id

The receiver starts by grouping all textures that belong to the same group. It is the case that only the main view’s texture is referenced in a material element. The metadata is then parsed to understand the nature of the relationship between the different textures. If camera parameters are present, the Presentation Engine has a choice of either fixing the rendering camera to match these camera parameters, effectively limiting the experience to a 3DoF experience, or configure proper processing to adjust the video texture depending on the current viewer’s pose by performing re-projection.

The relationship to MIV and to the 3GPP extension for signaling split rendering output still needs to be studied.

Support for Multi-channel Audio Sources

In order to support multi-channel audio sources, the Presentation Engine needs to recreate the speaker setup of the multi-channel source virtually around the audio source node. Each speaker in the speaker layout provides a transform matrix that places that speaker in the correct position with respect to the audio source node.

Two new audio source types are defined “stereo” and “multi-channel”. When the type is set to “stereo” or “multi-channel”, an array of transformations is also provided as `layoutTransforms`, where each element matches the speaker of a specific channel. When rendering such an audio source, a set of sub audio sources is created with the correct placement around that audio source using the `layoutTransforms`. This is effectively creating several audio sources per multi-channel audio source.

The layout may be signaled using a standardized speaker layout as defined in ISO/IEC 23091-8 [2] clause 8.2 on the output channel positions. In this case, a single global transform is sufficient and is applied to all loudspeakers to generate the virtual audio sources. Hence, for standardized speaker layouts, the information needed will be the channel mapping (as described in table 7 of [2]) and a global transform matrix.

2.3.3. References

[1] ISO/IEC 23090-14 2nd Edition, MPEG-I Scene Description

[2] ISO/IEC 23091-8, Information technology — MPEG systems technologies — Part 8: Coding-independent code points

==

Chapter 3. Interfaces

3.1. Supporting Multiple Viewers in the Media Access Function

Source: [m58510](#)

3.1.1. General

In the Presentation Engine of the MPEG-I Scene Description architecture, the viewer's view of the scene is determined by the camera used for rendering the scene from the viewer's viewpoint. In many use cases, the Presentation Engine runs on the end user's device and therefore there is only one viewer for the scene and one camera object is used at any given point in time for composition and rendering. Using the camera information provided by the Presentation Engine, the MAF can identify which objects in the scene are within the viewing frustum of the camera at a given time instance.

However, in some scenarios multiple cameras are used for rendering the scene from a number of viewpoints corresponding to different viewers of the same scene (e.g., in multi-viewer applications such as online conferencing applications with multiple users). In such scenarios, information about the cameras used to generate each viewer's view of the scene, including both intrinsic and extrinsic camera parameters, are required by the MAF to identify and request the appropriate media or media parts for each viewer.

Since a media pipeline is tightly coupled with the type of the media, it may not be desirable to have multiple media pipelines for the same content for different viewers. Rather, the MAF should allow a single media pipeline for a media content to be used for composition and rendering for different viewers.

3.1.2. Proposed Updates to MAF API

To support media fetching for multi-viewer applications, where each viewer may have their own extrinsic and intrinsic camera parameters, relevant methods in the MAF API and their definition should be updated as follows (updates are in **bold**).

3.1.2.1. Methods

Table 5. n/a

Methods	State after success	Description
startFetching()	ACTIVE	<p>Once initialized and in READY state, the Presentation Engine may request the media pipeline to start fetching the requested data.</p> <p>The initialization may be performed using view information for one or more viewers.</p>
updateView()	ACTIVE	<p>Update the current view information. This function is called by the Presentation Engine to update the current view information, if the pose or object position have changed significantly enough to impact media access. It is not expected that every pose change will result in a call to this function.</p> <p>A call to this function shall include the view information for only those views whose parameters have significantly changed.</p>

3.1.2.2. IDL for media pipeline

```

interface Pipeline {
    readonly attribute Buffer          buffers[];
    readonly attribute PipelineState  state;
    attribute          EventHandler  onstatechange;
    void    initialize.  (MediaInfo mediaInfo, BufferInfo bufferInfo[]);
    void    startFetching (TimeInfo timeInfo, ViewInfo viewInfo[]);
    void    updateView.  (ViewInfo viewInfo[]);
    void    stopFetching. ();
    void    destroy.     ();
};

```

3.2. Generic API for Presentation Engine


Source: [m66705](#)

3.2.1. Generic Render Control API

The Generic Render Control API is an abstract API that is offered by external renderers to enable applications, such as Presentation Engines, to control the rendering process by aligning and synchronizing their rendering state to that of the Presentation Engine. This API is used by the Presentation Engine to configure and update the status of the external renderer.

The following table describes the functionality provided by the Render Lock-in API:

Method	Description
init()	<p>Initializes the external renderer by providing the related media source information and their corresponding buffers. It also establishes a session between the Presentation Engine and the external renderer.</p> <p>The inputs to this method call should be:</p> <ul style="list-style-type: none">• A media source object that contains a handler to the buffer(s), where the source media will be made available by the MAF. A description of the media source and the contents of each buffer shall also be provided.

Method	Description
configure()	<p data-bbox="427 165 1458 286">Configures the external renderer to establish an initial alignment and synchronization between the Presentation Engine and the external renderer.</p> <p data-bbox="427 324 1023 358">The parameters to this method may include:</p> <ul data-bbox="451 398 1458 1361" style="list-style-type: none"> <li data-bbox="451 398 1458 562">• A mapping between the initial timestamp of the common Presentation Engine timeline and that of the media associated with the external renderer. It also provides information about the clock rate of the Presentation Engine. <li data-bbox="451 584 1458 913">• A list of mapped nodes in the source media rendered by the external renderer. This list shall at least contain one object with a mapping to the main camera of the main scene description. For audio renderers, this may be the audio listener. The information is provided by the the MPEG_node_mapping extension in the scene description document. It should also provide the initial position and transformation of the mapped nodes after applying the transformations associated with these node mappings. . <li data-bbox="451 936 1458 1301">• A description of the scene bounding box using the glTF 2.0 spatial coordinate system. The external renderer uses this information to establish a spatial alignment between the scene coordinate system and the coordinate system that is used by the source media. The external renderer may align the bounding box of the scene to that of its media stream, which establishes the transformation that needs to be applied to all spatial coordinates exchanged over the API, in order to determine the corresponding coordinates in the coordinate system of the media stream. <li data-bbox="451 1323 1430 1361">• A list of tracked AR anchors that may be used by the external renderer. <p data-bbox="427 1400 1458 1520">The external renderer may then subscribe for updates to specific aligned nodes or it may specifically ask for current state for these nodes, using the referenceId.</p> <div data-bbox="477 1655 542 1720">  </div> <p data-bbox="620 1570 1426 1812">all exchanges over this API are based on the scene (glTF2.0) coordinate system. It is the responsibility of the external renderer to convert into their own coordinate system. The Presentation Engine does not consider any other coordinate systems other than the one established by the scene description.</p>

Method	Description
start() pause() resume() stop()	Allows the Presentation Engine to control the playback of selected media sources associated with the external renderer for interactivity purposes.
update()	<p>Used by the Presentation Engine to update node positions and orientations for which there is a mapping with the external renderer. Updates may result from received scene updates, user interactions, animations, physics simulations, or any other events.</p> <p>The parameters passed to this method are an array of objects consisting of:</p> <ul style="list-style-type: none"> • The <code>referenceId</code> of the node to which this update applies • The transform matrix that sets the current pose of the tracked object after applying the transform operation as described by the corresponding <code>MPEG_node_mapping</code>. Any further adjustments need to be applied by the external renderer to align with its internal coordinate system.
updateGraph()	<p>The Presentation Engine uses the <code>updateGraph</code> function to add, update, or remove a set of nodes to the internal representation of the scene that is maintained by the external renderer. Only nodes that have a mapping with the external renderer can be passed through this method.</p> <p>The parameters to this method are an array of objects that include:</p> <ul style="list-style-type: none"> • The graph operation: ADD, REMOVE, UPDATE • For ADD: the <code>referenceId</code> and the initialization information for the associated media data to the object that is to be added. • For REMOVE: the <code>referenceId</code> of the object to be removed. • For UPDATE: the <code>referenceId</code> of the object to be updated, as well as a dictionary of attributes and their update values.
registerCallback()	<p>The Presentation Engine may provide a callback function to the external renderer to allow it to query the status of certain parameters at any time. This may for example include asking for the current user pose.</p> <p>The Presentation Engine shall register a callback function whenever possible.</p>

The following is a description for the API in IDL (ISO/IEC 19516):

```
interface GenericRenderControl {
    void init();
    void configure();
}
```

```

void start();
void pause();
void resume();
void stop();
update();
void updateGraph();
void registerCallback();
};

```

3.2.2. Extension for Audio Node Mapping

3.2.2.1. General

The MPEG node mapping extension, identified by `MPEG_node_mapping`, establishes a mapping between the node in the scene description document and an external entity. An example is the mapping between a node that contains a car and an external audio node in an MPEG-I Audio bitstream, with a simplified geometry of that car and the attached audio sources. The following figure depicts that example:

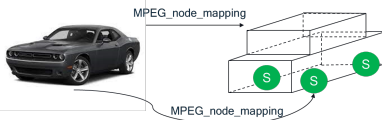


Figure 1. A black and white image of a camera Description automatically generated

When present, the `MPEG_node_mapping` extension shall be included in a node object.

3.2.2.2. Semantics

The definition of all objects with the `MPEG_node_mapping` extension is provided in the following table:

Name	Type	Default	Usage	Description
mappings	array(object)		M	An array of mappings associated with the containing node.
Role	string	“urn:mpeg:sd:role:default”	O	An identifier of the role associated with this mapping. The role may for instance be “urn:mpeg:sd:role:audio-renderer” to indicate that the component is an audio renderer.
source	number	N/A	M	The index in the <code>MPEG_media</code> that provides the media resource that contains the mapped element.
referenceId	number	N/A	M	An identifier of the element in the referenced resource.

Name	Type	Default	Usage	Description
transform	array(number)	Identity	O	A 4x4 matrix that supplies the transform used to align the referenced element to the current node.
supportsInteractivity	boolean	false	O	Indicates if interactivity actions applied to the node should be exposed if an API is made available to the Presentation Engine by the renderer of the resource.

3.2.2.3. Processing Model

When processing the MPEG_node_mapping extension, the Presentation Engine shall identify nodes in the scene description that have a node mapping. The Presentation Engine shall determine if the component identified by the indicated role supports the Rendering Alignment API as defined in contribution m65395. If it does, the Presentation Engine shall pass the mapping information to the identified component.

The Presentation Engine shall then use the API to align the rendering with the component as configured over the API.

Chapter 4. MPEG-I Audio in Scene Description

4.1. On spatial synchronization between graphs

Source: [m67011](#)

4.1.1. Attempt problem definition for the spatial synchronization

4.1.1.1. Virtual Reality (VR) use case

The VR use case corresponds to an animated virtual car. Each wheel can be animated individually. Spatial sounds are generated by the motor of the car, and by the contact of each wheel on the road.

[Figure 2](#) provides the SD and the immersive audio graph representations of the virtual car.

It can be noticed that these two graphs have not the same topology and not the same global XR Space (i.e., the global frame of reference in which 3D coordinates are expressed).

The following node mappings have been created:

- Between the root nodes of the car to ensure a consistent car animation
- Between each node related to a wheel to ensure a consistent wheel animation

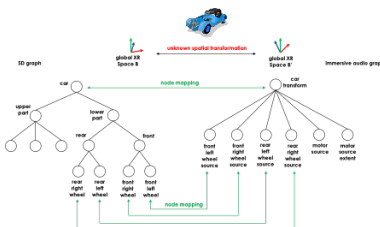


Figure 2. SD and immersive audio graph representations of a virtual car

Note 1: The node mapping needs to be investigated, when an extent is added to an audio source, to ensure the spatial synchronization of both the audio source and its extent. For example, the two following approaches may be envisaged if an extent is added to a wheel of the car:

- To allow nested spatial transformation nodes in the immersive audio graph [Figure 3](#)
 - The audio source and its extent would then be the children of a mapped spatial transformation node
- Or to allow the extent to be a child of the mapped audio source [Figure 4](#)

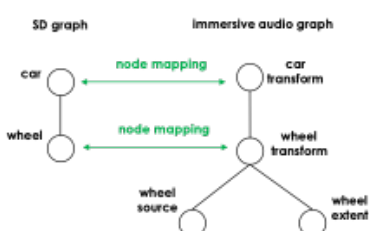


Figure 3. Possible approaches to ensure a spatial synchronization for both an audio source and its extent

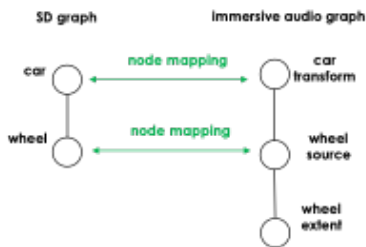


Figure 4. Possible approaches to ensure a spatial synchronization for both an audio source and its extent

The following issues need to be addressed to ensure a spatial synchronization between the two graphs:

- the knowledge of the transformation matrix between the global XR Space B and B',
- the identification of which initial parameters to be provided to the immersive audio renderer through the render control API at the configuration step,
- the identification of which parameters to be provided to the immersive audio renderer through the render control API to maintain the spatial synchronization during the VR experience.

4.1.1.2. Augmented Reality (AR) use case

In this use case, the virtual car of section 2 is inserted to the user's real environment using AR anchoring.

MPEG-I Scene Description has defined a dedicated MPEG_anchor glTF extension to support AR anchoring of virtual assets represented by a node graph.

The MPEG_anchor extension defines the Trackable and Anchor objects as follows (Figure 5):

Trackable: a real-world object that can be tracked by the XR runtime. Each trackable provides a local reference space, also known as a trackable space, in which an anchor can be expressed.

Anchor: a virtual element for which its position, orientation, scale and other properties are expressed in the trackable space defined by the trackable. A virtual asset's position, orientation, scale and other properties are expressed in relation to an anchor.

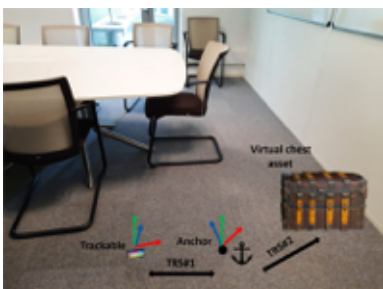


Figure 5. Trackable and Anchor for AR

In this AR use case, both the SD and the immersive audio graph may define a Trackable to insert the virtual car into the user's real environment.

Note 2: The immersive audio group uses a single Anchor object for the AR anchoring of the scene. This Anchor object corresponds to a Trackable object of an MPEG Scene Description. In other words, the transformation matrix between the Trackable and the Anchor objects (TRS#1 in Figure 5

) is always the Identity matrix in the immersive audio graph.

Figure 6 illustrates the AR anchoring of the SD and immersive audio graphs representing the virtual car using a 2D marker by assuming that a common shared Trackable is defined in both the SD and immersive audio graphs.

Note 3: The root nodes of the car for the two graphs need to have identical initial transformation matrices to ensure a consistent spatial positioning with respect to the Trackable.

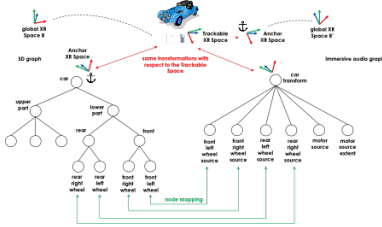


Figure 6. SD and immersive audio graph representations of a virtual car with AR anchoring using a 2D marker

The pose of the Trackable is retrieved from the XR Runtime API of the device (e.g. Khronos OpenXR).

The XR Runtime needs to be configured through the XR Runtime API at the beginning of the AR session to be able to detect and track the Trackable at runtime.

It is assumed that the Presentation Engine related to the SD graph configures the XR Runtime. An approach would be that the poses of the Trackables are provided to the immersive audio renderer by the Presentation Engine through the render control API to ensure the spatial consistency between the two graphs.

4.1.2. Approach proposal for the spatial synchronization

This section proposes an approach to address the following issues for ensuring a spatial synchronization between the SD and the immersive audio graphs:

- the knowledge of the transformation matrix between the global XR Space B and B',
- the identification of which initial parameters to be provided to the immersive audio renderer through the render control API at the configuration step,
- the identification of which parameters to be provided to the immersive audio renderer through the render control API to maintain the spatial synchronization during the VR experience.

For the AR case, it is assumed that the Presentation Engine related to the SD graph configures the XR Runtime. Then, the poses of the Trackables are provided to the immersive audio renderer by the Presentation Engine through the render control API.

4.1.2.1. Determination the transformation matrix between the global XR Space of each graph

This spatial transformation corresponds to the matrix $P_{B'}^B$ which transforms the input 3D coordinates expressed in the global XR Space B of the SD graph to 3D coordinates expressed in the global Space B' of the immersive audio graph (1):

$$(x', y', z')_{B'} = P_{B'}^B (x, y, z)_B \quad (1)$$

The proposed approach uses the node mappings between the two graphs to obtain a common XR Space from which the calculation of this matrix $P_{B'}^B$ can be done.

Figure 7 illustrates this matrix calculation process with:

- The node *ref* of the SD graph used as the node mapping of reference, defining a local XR Space B_{ref} and a mapping transform matrix $P_{B_{ref}}^{B_{ref}'}$ (i.e., the transform parameter of the node mapping glTF extension of [1])
- The node *ref'* of the immersive audio graph referenced by the *referenceId* parameter of the node mapping glTF extension of [1]

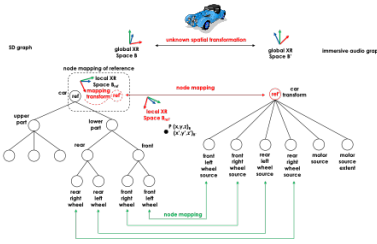


Figure 7. Transformation matrix determination using a node mapping

For any point P:

$$(x, y, z)_B = P_B^{B_{ref}'} (x_{ref}', y_{ref}', z_{ref}')_{B_{ref}'} = P_B^{B_{ref}'} P_{B_{ref}'}^{B_{ref}'} (x_{ref}', y_{ref}', z_{ref}')_{B_{ref}'} \quad (2)$$

$$(x', y', z')_{B'} = P_{B'}^{B_{ref}'} (x_{ref}', y_{ref}', z_{ref}')_{B_{ref}'} \quad (3)$$

Then, with (2) and (3):

$$(x', y', z')_{B'} = P_{B'}^{B_{ref}'} [P_{B_{ref}'}^{B_{ref}'}]^{-1} [P_B^{B_{ref}'}]^{-1} (x, y, z)_B \quad (4)$$

With (1) and (4):

$$P_{B'}^B = P_{B'}^{B_{ref}'} [P_{B_{ref}'}^{B_{ref}'}]^{-1} [P_B^{B_{ref}'}]^{-1} \quad (5)$$

For a sake of clarity, the matrix product $[P_{B_{ref}'}^{B_{ref}'}]^{-1} [P_B^{B_{ref}'}]^{-1}$ may be called alignment matrix P_{align}

$$P_{align} = [P_{B_{ref}'}^{B_{ref}'}]^{-1} [P_B^{B_{ref}'}]^{-1} \quad (6)$$

And finally, with (5) and (6):

$$P_{B'}^B = P_{B'}^{B_{ref}'} P_{align} \quad (7)$$

In formula (7), it has to be noted that:

- The Presentation Engine does not know the matrix $P_{B'}^{B_{ref}'}$
- The immersive audio renderer does not know the alignment matrix P_{align} and which node *ref'* of the immersive audio graph has been used for the calculation of the transformation matrix $P_{B'}^B$

4.1.2.2. Parameters to be provided to the immersive audio renderer during the configuration step

The following parameters need to be provided to the immersive audio renderer during the configuration step:

- The alignment matrix P_{align} ,
- The unique identifier (i.e., the referenceId of the node mapping glTF extension of [1]) of the node of the immersive audio graph used for the calculation of the transformation matrix $P_{B'}^B$

By receiving the alignment matrix P_{align} and the referenceId of the node *ref*, the immersive audio renderer can calculate and store the transformation matrix $P_{B'}^B$ using the formula (7).

Then, when receiving the initial poses of the mapped nodes and the Trackables expressed in the global XR Space B of the SD graph, the immersive audio renderer can convert these poses to the global XR Space B' of the immersive audio graph by using the formula (1).

4.1.2.3. Parameters to be provided to the immersive audio renderer to maintain the spatial synchronization

The spatial synchronization between the two graphs is maintained by providing the current poses of the mapped nodes and the Trackables expressed in the global XR Space B of the SD graph. Then, the immersive audio renderer can convert these poses to the global XR Space B' of the immersive audio graph by using the formula (1).

4.1.3. Conclusion

We propose to discuss on the content of the sections 2 and 3 with the immersive audio experts. If the proposed approach is agreeable, we propose to add the content of sections 3 to the TuC for further investigations.

4.1.4. References

[1] MPEG-I WG3 m66705, generic API for Presentation Engine, January 2024

4.2. Immersive audio support in Scene Description

Source: [m70205](#)

4.2.1. Introduction

A generic Render Control API and a related MPEG_node_mapping glTF extension at node level is defined in the section 3.2 of the Technology under Consideration document [1] from the contribution m66705 [2].

Additional contributions on the spatial synchronization between graphs [3] and on the support of AR anchors [4] have been provided during past MPEG meetings.

Based on the assumptions provided in section 2, this contribution proposes an update of the generic

Render Control API and related MPEG_node_mapping glTF extension at node level for the amd1 document of the second edition (section 3). The modifications with respect to the TuC version are highlighted in yellow.

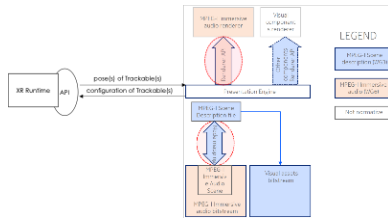
4.2.2. Main assumptions

Based on the architecture of Figure 1 for the processing of the MPEG-I immersive audio and Scene Description data, the main assumptions are the following:

- The *transform* parameter of the MPEG_node_mapping glTF extension corresponds to the 4x4 TRS matrix which transforms the 3D coordinates of the node having this glTF extension expressed in the glTF2.0 scene coordinate system to the 3D coordinates of the node of the external renderer graph referenced by the *referenceId* identifier expressed in the external renderer scene coordinate system
 - In that case, there is no need of further data (e.g., bounding boxes as defined in the TuC version) to ensure a spatial alignment between the graphs as the 4x4 TRS matrix of the *transform* parameter encompasses all the required spatial transformations.
- There is no need of exposing dedicated data on the AR anchors/Trackables (e.g., a list of tracked AR anchors as defined in the TuC version) to the external renderer
 - The information on AR anchors/Trackables are duplicated in the immersive audio bitstream and in the Scene Description file.
 - This information is used by the Presentation Engine to configure the XR Runtime to detect and track the AR Anchors/Trackables
 - If a node having the MPEG_node_mapping glTF extension is a child of an AR Anchor/Trackable, this node and the node of the external renderer graph referenced by the *referenceId* identifier are positioned in a common AR Anchor/Trackable XR Space based on the spatial transformations from their respective node graph hierarchy
 - In that case, the *transform* parameter of the MPEG_node_mapping glTF extension corresponds to the 4x4 TRS matrix which transforms the 3D coordinates of the node having this glTF extension expressed in the common AR Anchor/Trackable coordinate system to the 3D coordinates of the node of the external renderer graph referenced by the *referenceId* identifier expressed in the common AR Anchor/Trackable coordinate system
- For a sake of processing efficiency, additional parameters may be added to the MPEG_node_mapping glTF extension for defining the synchronization occurrence for that node. The objects of the scene may require different synchronization requirements. For example,
 - A wall having acoustic properties remains static and does not require further spatial synchronization at runtime
 - A draggable object (e.g. a door) may only require a spatial synchronization based on an event, i.e., a trigger activation such as collision, user input
 - An animated object (e.g., a car) may require periodic, rendering frame-based spatial synchronization
 - The control (e.g., play, pause, stop) of an audio source (e.g., an alarm) may depend on a logical combination of events such as a logical AND combination between visibility and

proximity triggers

Baseline architecture for the processing of the MPEG-I immersive audio and Scene Description data



4.2.3. Support of immersive audio in Scene Description

4.2.3.1. Generic Render Control API

The Generic Render Control API is an abstract API that is offered by external renderers to enable applications, such as Presentation Engines, to control the rendering process by aligning and synchronizing their rendering state to that of the Presentation Engine. This API is used by the Presentation Engine to configure and update the status of the external renderer.

The following table describes the functionality provided by the Render Lock-in API:

Method	Description
init()	<p>Initializes the external renderer by providing the related media source information and their corresponding buffers. It also establishes a session between the Presentation Engine and the external renderer.</p> <p>The inputs to this method call should be:</p> <ul style="list-style-type: none"> A media source object that contains a handler to the buffer(s), where the source media will be made available by the MAF. A description of the media source and the contents of each buffer shall also be provided.

Method	Description
configure()	<p>Configures the external renderer to establish an initial alignment and synchronization between the Presentation Engine and the external renderer.</p> <p>The parameters to this method may include:</p> <ul style="list-style-type: none"> • A mapping between the initial timestamp of the common Presentation Engine timeline and that of the media associated with the external renderer. It also provides information about the clock rate of the Presentation Engine. • A list of mapped nodes in the source media rendered by the external renderer. This list shall at least contain one object with a mapping to the main camera of the main scene description. For audio renderers, this may be the audio listener. The information is provided by the [line-through]#the MPEG_node_mapping extension in the scene description document. It should also provide the initial [.mark]#pose [line-through]#position and transformation of the mapped nodes after applying the [.mark][line-through]transformations associated with these# 4x4 matrix of the <i>transform</i> parameter provided in the node mappings. • [line-through]#A description of the scene bounding box using the glTF 2.0 spatial coordinate system. The external renderer uses this information to establish a spatial alignment between the scene coordinate system and the coordinate system that is used by the source media. The external renderer may align the bounding box of the scene to that of its media stream, which establishes the transformation that needs to be applied to all spatial coordinates exchanged over the API, in order to determine the corresponding coordinates in the coordinate system of the media stream.# • [line-through]#A list of tracked AR anchors that may be used by the external renderer.# <p>The external renderer may then subscribe for updates to specific aligned nodes or it may specifically ask for current state for these nodes, using the <i>referenceId</i>.</p> <p>[line-through]#NOTE: all exchanges over this API are based on the scene (glTF2.0) coordinate system. It is the responsibility of the external renderer to convert into their own coordinate system. The Presentation Engine does not consider any other coordinate systems other than the one established by the scene description.#</p>

Method	Description
start() pause() resume() stop()	Allows the Presentation Engine to control the playback of selected media sources associated with the external renderer for interactivity purposes.
update()	<p>Used by the Presentation Engine to update node positions and orientations for which there is a mapping with the external renderer. Updates may result from received scene updates, user interactions, animations, physics simulations, or any other events. The Presentation Engine uses this update() method when one or more mapped nodes need to be spatially synchronized, depending on their <i>synchronizationOccurrence</i> and <i>synchronizationOccurrenceCombination</i> parameters provided in their MPEG_node_mapping extension.</p> <p>The parameters passed to this method are an array of objects consisting of:</p> <ul style="list-style-type: none"> • The referenceId of the node to which this update applies • [line-through]#The transform matrix that sets the current pose of the tracked object after applying the transform operation as described by the corresponding MPEG_node_mapping. Any further adjustments need to be applied by the external renderer to align with its internal coordinate system.# • The current pose of the mapped node after applying the 4x4 matrix of the <i>transform</i> parameter provided in the corresponding MPEG_node_mapping
updateGraph()	<p>The Presentation Engine uses the updateGraph function to add, update, or remove a set of nodes to the internal representation of the scene that is maintained by the external renderer. Only nodes that have a mapping with the external renderer can be passed through this method.</p> <p>The parameters to this method are an array of objects that include:</p> <ul style="list-style-type: none"> • The graph operation: ADD, REMOVE, UPDATE • For ADD: the referenceId and the initialization information for the associated media data to the object that is to be added. • For REMOVE: the referenceId of the object to be removed. • For UPDATE: the referenceId of the object to be updated, as well as a dictionary of attributes and their update values.

Method	Description
registerCallback()	<p>The Presentation Engine may provide a callback function to the external renderer to allow it to query the status of certain parameters at any time. This may for example include asking for the current user pose.</p> <p>The Presentation Engine shall register a callback function whenever possible.</p>

The following is a description for the API in IDL (ISO/IEC 19516):

```

interface GenericRenderControl \{
  void init();
  void configure();
void start();
void pause();
void resume();
void stop();
update();
void updateGraph();
  void registerCallback();
};

```

4.2.4. References

- [1] Technology under Consideration for ISO/IEC 23090-14, MDS24160_WG03_N01314
- [2] Generic API for Presentation Engine, Qualcomm, m66705, Online meeting of January 2024
- [3] On spatial synchronization between graphs, InterDigital, m67011, Rennes meeting of April 2024
- [4] AR Anchor and LSDF generation in MPEG-I immersive audio, Nokia, m68566, Sapporo meeting of July 2024

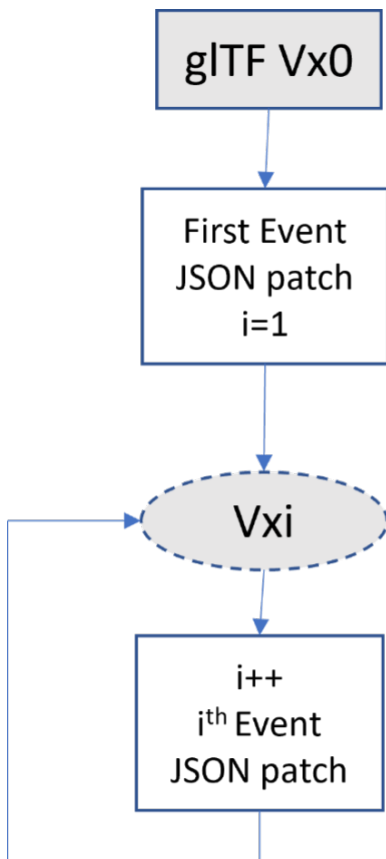


Figure 10. Event-based update diagram

5.1.2. A use case for event based updates

This update diagram is illustrated in the IDCC demo, presented during the last MPEG meeting in Mainz:

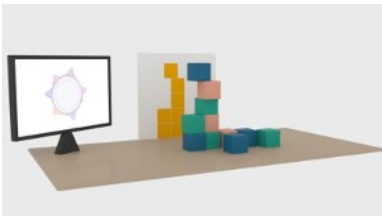


Figure 11. n/a



Figure 12. n/a

The demo presents a game application. An initial scene is first displayed, containing a plane surface, a TV screen displaying a video content and a vertical surface displaying a pattern. The user can add a new cube in the scene by touching the screen, in order to build a cubes stack that matches the displayed pattern. Each time a match occurs, a new scene is loaded with a new pattern and a new video. The game may be multiplayer with the same scene shared between all the connected clients. The scene is synchronized each time an update is performed in one client. A

game server handles the scene synchronization each time an update is performed by a client.

The creation of the cube and the loading of a new scene is currently implemented using proprietary solution, but it could be possible to build a mechanism in line with the MPEG-SD dynamic scene framework.

Two kinds of updates are triggered during the game:

1. During a game phase, each time the user touches the screen to create a cube in front of the pattern, a same scene update/patch is applied. The difference is the position of the user's finger that gives the position where the cube is created and from which it falls. Using the current scene update mechanism, with JSON patch, the creation of a new cube would be performed with 2 patch operations:
 - An “add” operation, that adds a new node in the glTF node array, for instance with a path equal to “/nodes/-“, i.e. a new node created at the end of the array. A new node created in the middle of the nodes array (i.e., with a path equal to “/nodes/2”) would leave the scene in an erroneous status and would need extra patch operations to fix it. We would face other issues if the new “cube” nodes must be created as children of another “cubesStack” node: We would not know in advance the index of the new node since it depends on the number of updates that have already been triggered.
 - A “place” operation that does not exist in the JSON patch specifications. We could use a “replace” operation to set the “translation” or/and “rotation” elements of the new node but:
 - Same as above, we do not know in advance the index of the new node!
 - The value to be applied must be retrieved from user's finger position on the screen! And there is no way to pass this value as an input to the “replace” operation.
2. When the cubes stack matches the pattern, a new scene is loaded with a new pattern:
 - It could be a JSON patch, removing the cube nodes and replacing the pattern with a new one. As above, we do not know the indexes of all the cube nodes and these indexes are needed to remove the nodes. If the nodes have been created as children of a unique parent node, we could just empty the children array of this node. The cube nodes description would remain in the description file.
 - It could be a complete update and a new glTF file is used.

5.1.3. JSON patch limitations

A JSON patch is not a “glTF patch” and does not consider all the characteristics of the JSON tree in a glTF scene description file and particularly the interdependence between elements of different branches of the glTF tree (a node referencing a mesh that references a material, or a node referencing one or more child nodes). It is fine if you know in advance the scene description you want to update and the resulting scene description: The JSON patch can be generated by comparing the 2 JSON description files.

For repetitive event-based updates as described in [Section 5.1.2](#), we don't know the resulting scene and care should be taken when writing the JSON patch. Furthermore, the application, that applies the patch, may need to perform extra operations to complete the update:

- check the consistency of the resulting glTF scene,
- get the index of an array item created with the “-“ JSON patch alias,
- perform extra glTF modifications not handled by JSON patches (set newly created nodes as child of another node, set JSON element to a value only determined at run-time...).

5.1.4. Semantics for event-based update

A new semantic is needed to describe event-based scene update: A semantic that would address the use case (related to pre-defined timed scene updates) as well as the new one introduced in [Section 5.1.2](#).

An approach would be to keep using the JSON patch mechanism, which is already used for the pre-defined timed scene updates. As explained above, the definition of extra parameters would then be required.

Furthermore, the description of the event and its relationship with the scene update could be described with the interactivity framework specified in [ISO/IEC JTC 1/SC 29/WG 3 N0725]. It defines a set of action types that can be executed following a trigger activation. As a reminder, the table above gives the action types that are already specified:

Table 6. Type of action

Action type	Description
“ACTION_ACTIVATE”	Set activation status of a node
“ACTION_TRANSFORM”	Set transform to a node
“ACTION_BLOCK”	Block the transform of a node
“ACTION_ANIMATION”	Select and control an animation
“ACTION_MEDIA”	Select and control a media
“ACTION_MANIPULATE”	Select a manipulate action
“ACTION_SET_MATERIAL”	Set new material to nodes
“ACTION_SET_HAPTIC”	Get haptic feedbacks on a set of nodes

An event-based scene update may be described in a glTF scene description file, using the interactivity extensions specified in [ISO/IEC JTC 1/SC 29/WG 3 N0725]: A trigger element may describe the event (for instance, a “TRIGGER_USER_INPUT” trigger, as defined in [ISO/IEC JTC 1/SC 29/WG 3 N0725]), and an action element (of a new type, to be defined) may describe the update information (a patch to be applied (an array of JSON patch operations) and other parameters used by the application to complete this update). Here is a list of such parameters that may be defined:

- Parameters to place one or more nodes in a position not known in advance. For instance, it may include a position information and a list of nodes. The position parameter may be related to a user input, or a user pose and may use the [OpenXR interaction profile path semantic](#). Each node to position may be identified by one of the patch operations that created or modified it.
- Parameters identifying one or more nodes to be used as parent of one or more newly created nodes. For instance, a list of parent nodes and a list of child nodes. Same as above, each child

node may be identified by one of the patch operations that created or modified it.

- Any other parameters that may be needed for other use cases: flag to share or not a local update with other connected users sharing the same scene, strategy in case the patch fails or gives an inconsistent glTF tree (rollback, fix...), ...

Chapter 6. Collected problem statements and industry needs

6.1. On the support of real environment data

Source: [m61811](#)

6.1.1. General

In Augmented Reality (AR) experiences, virtual content is seamlessly inserted into the user's real environment using optical or video-see-through devices. The knowledge of the user's real environment is then required for:

- * The positioning of the virtual objects based on AR anchors
- * Consistent handling of collisions between virtual and real objects
- * Consistent rendering of virtual and real objects including occlusion and lighting/shadowing aspects

This contribution provides an overview of how real environment data are handled (captured, computed, stored and loaded) in some AR frameworks and proposes to investigate the support of real environment data in MPEG-I Scene Description for transmission purposes.

6.1.2. Representation of the real environment

As shown in [Figure 13](#), the real environment data are computed from embedded-sensor raw data. An AR device may have several embedded sensors to scan the user environment, such as color camera(s) and Light Detection and Ranging (LiDAR). The generated raw data are typically point clouds, depth maps, pictures. An Inertial Measurement Unit (IMU) is also required to estimate the current pose of the AR device when acquiring these data. Based on these sensor raw data, a representation of the real environment is computed and the resulting real environment data may have various formats:

- A single mesh, optionally textured, issued from a spatial mapping computation
- A semantic representation, optionally associated with a mesh segmentation, issued from a scene understanding computation
- A real light mapping

Depending on the AR experiences, the most appropriate representation of the real environment is computed:

- A single mesh representation may be sufficient for coherent collision handling and lighting
- A semantic representation (e.g. “desk”, “laptop”, “screen”, “floor”, “ceiling”, “wall”) may be required for the definition of advanced anchoring and/or interaction
- A mesh segmentation is required for individual real object handling, such as object removal in a diminished reality application

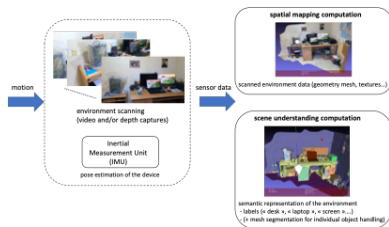


Figure 13. Computation of real environment data

The computation of the real environment data may either be done locally in the AR device or remotely in a Spatial Computing Server. In the case of remote computation, the transmission of such kind of data is in line with the Spatial Computing Server (SCS) requirements for eXtended reality (XR) of the MPEG-I Phase 2 requirement document especially the requirement #134:

“The SCS shall provide XR Spatial Description in a standard representation format (e.g. scene description) upon request of XR devices (UEs) on different platforms (desktop and mobile).”

6.1.3. Storing a representation of the real environment

The process of scanning the real environment and generating the corresponding representation may be done prior to runtime. This approach is often related to quasi-static environment and has the following main advantages:

- Availability of the real environment data at the beginning of the AR session
- Resource optimization of the AR devices resulting to power savings as no or limited scans are required at runtime
- Support of low-end AR devices having no efficient sensors
- Consistency of the representation of a shared real environment between several heterogenous AR devices
- Ability to build a scalable library of real environments (rooms, buildings, cities...)

Note: Having an initial scan may also be relevant for time-evolving real environments. Updating some parts of the initial scan could be less time-consuming than performing a complete scan.

Generating real environment data before runtime requires efficient storage. Storing real environment data in the Cloud has been investigated by ETSI Augmented Reality Framework (ARF). As shown in Figure 14, a World Knowledge server is located in the Cloud and stores the real environment data to be used by

- a Vision Engine for AR anchoring positioning/localization aspects
- a 3D Rendering Engine for consistent collision handling and rendering between virtual and real objects

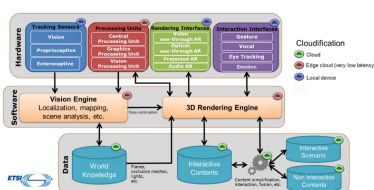


Figure 14. Global overview of the architecture of an AR system (from ETSI ARF)

Note: there is a need for a format to transmit real environment data between the World Knowledge storage server and the 3D Rendering Engine in complement to the transmission of virtual contents, which is already the scope of MPEG-I SD.

6.1.4. Examples of framework for real environment handling

Several frameworks are available to scan, compute, store and load real environment data for AR experiences. An overview of the following frameworks is provided in this section:

- Microsoft's Mixed Reality framework
- Apple's ARKit framework
- Meta/Oculus framework

6.1.4.1. Microsoft's Mixed Reality framework

The Microsoft Mixed Reality framework has been developed for the HoloLens 2 device. It is composed of

- a spatial computing module, generating a mesh representation of the real environment as shown in [Figure 15](#)
- a scene understanding module from Mixed Reality Toolkit (MRTK) version 2.7 based on OpenXR, detecting and labeling planar surfaces for the placement of virtual content as shown in [Figure 16](#)

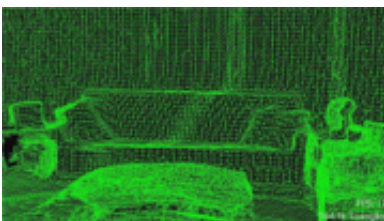


Figure 15. Mesh representation of the real environment after a spatial mapping computation



Figure 16. Semantic representation of the real environment after a scene understanding computation

A complete Microsoft's Scene Understanding SDK for Unity is available. An example of a C# code to scan, load and store real environment data based on the Scene Observer object is shown below

```
if (!SceneObserver.IsSupported())
{
    // Handle the error
}

// This call should grant the access we need.
await SceneObserver.RequestAccessAsync();

// Create Query settings for the scene update
SceneQuerySettings querySettings;
```

```

querySettings.EnableSceneObjectQuads = true;
// Requests that the scene updates quads.
querySettings.EnableSceneObjectMeshes = true;
// Requests that the scene updates watertight mesh data.
querySettings.EnableOnlyObservedSceneObjects = false;
// Do not explicitly turn off quad inference.
querySettings.EnableWorldMesh = true;
// Requests a static version of the spatial mapping mesh.
querySettings.RequestedMeshLevelOfDetail = SceneMeshLevelOfDetail.Fine; // Requests
the finest LOD of the static spatial mapping mesh

// Initialize a new Scene
Scene myScene = SceneObserver.ComputeAsync(querySettings, 10.0f).GetAwaiter()
.GetResult();

// Create Query settings for the scene update
SceneQuerySettings querySettings;

// Compute a scene but serialized as a byte array
SceneBuffer newSceneBuffer = SceneObserver.ComputeSerializedAsync(querySettings, 10
.0f).GetAwaiter().GetResult();

// If we want to use it immediately we can de-serialize the scene ourselves
byte[] newSceneData = new byte[newSceneBuffer.Size];
newSceneBuffer.GetData(newSceneData);
Scene mySceneDeSerialized = Scene.Deserialize(newSceneData);

// Save newSceneData for later

```

6.1.4.2. Apple's ARKit framework

On a fourth-generation iPad Pro running iPad OS 13.4 or later, Apple's ARKit uses the LiDAR Scanner to create a mesh representation of the user real environment. Then this mesh is further segmented and multiple anchors, called ARMeshAnchor, are assigned to the resulting set of segmented meshes. As shown in [Figure 17](#), a semantic labeling is performed for the real objects that ARKit can identify such as ceiling, door, floor, seat, table, wall and window labels.

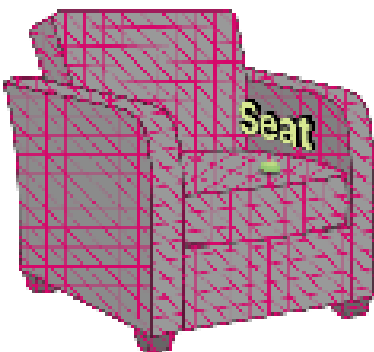


Figure 17. Semantic labeling of Apple's ARKit

These real environment data attached to the ARMeshAnchors can be saved and loaded by

serializing/deserializing an ARWorldMap as shown in [Figure 18](#).

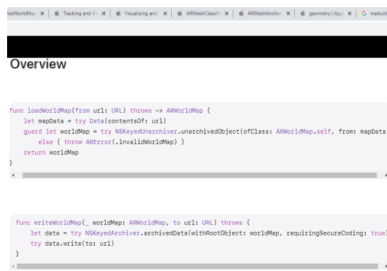


Figure 18. Saving and loading an Apple's ARKit ARWorldMap

6.1.4.3. Meta/Oculus framework

The Meta/Oculus framework has been developed for Meta Quest 2 and Meta Quest Pro devices. The scene understanding computation provides a scene model, which is a representation of the user real environment. The scene model contains Scene Anchors, with each anchor being attached to geometric components and semantic labels. The floor, ceiling, wall_face, desk, couch, door_frame and window_frame labels are currently supported as shown in [Figure 19](#).

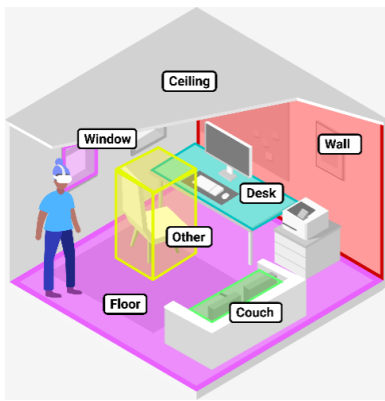


Figure 19. Semantic labeling of the Meta/Oculus Scene Understanding

The scene understanding computation is based on the Khronos OpenXR standard and relies on the Meta OpenXR XR_FB_scene extension. By using Unity as Presentation Engine, an OVRSceneManager allows access to the scene model. An OVRSceneAnchor component corresponds to a scene anchor. The semantic classification of a scene anchor is managed by the OVRSemanticClassification.

A Scene Model is generated by the Scene Capture system flow that lets users walk around and capture their scene. Users have complete control over the manual capture experience and decide what they want to share about their environment.

As shown below, the OVRSceneManager provides functions

- to launch a scene capture to generate a Scene Model
- to load an existing Scene Model

```
OVRSceneManager.RequestSceneCapture()  
OVRSceneManager.LoadSceneModel()
```

Chapter 7. Avatar

7.1. Update of the Description of the MPEG reference avatar model Morgan

Source: [m69577](#)

7.1.1. Introduction

The MPEG-IoMT group presented a revision of data format and APIs for body keypoints extractor [1]. In parallel, the MPEG-I-SD group presented a reference avatar, *Morgan* [2]. The goal of this contribution is to propose an extension of the previous version of the description of *Morgan*, which does not change the previous one. This is compatible with IoMT developments.

This contribution includes:

- semantics of body mesh parts (Section 1) of *Morgan*
- semantics of skeleton joints (Section 2) of *Morgan*
- mapping between IoMT keypoints and *Morgan* (Section 3)

7.1.1.1. Semantics of Body Mesh Parts

This section provides a description of Morgan body parts. These body parts, as presented in Annex H [2], are mesh sub-parts including vertices and faces.

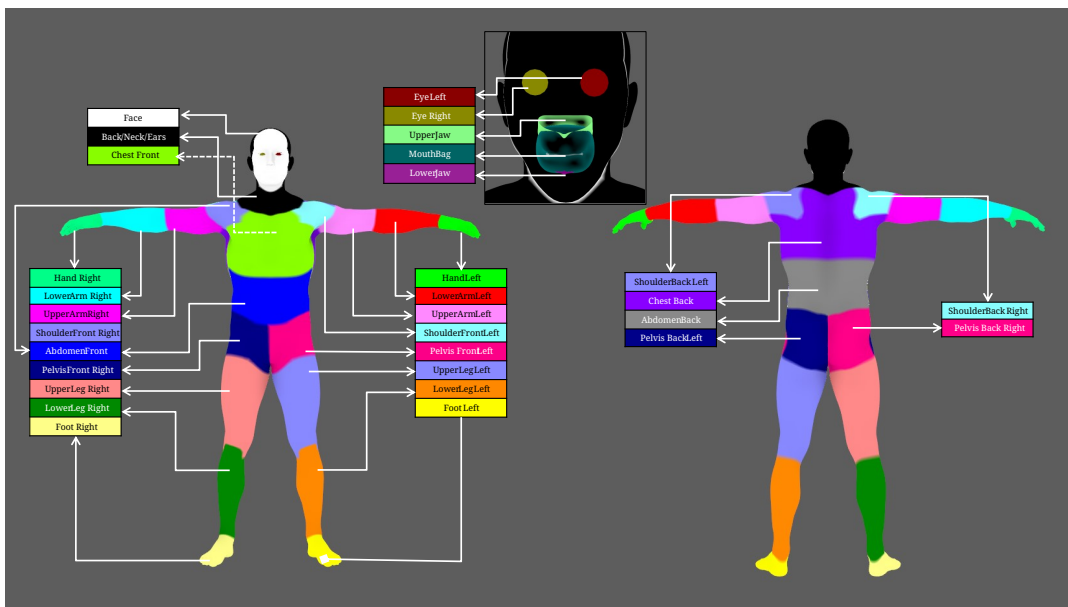


Figure 1: body semantic areas

Name	Definition
Head	It describes points of skeleton and articulation of head part.
Full	It describes mesh vertices/faces of the whole body.

<i>Name</i>	<i>Definition</i>
UpperBody	It describes mesh vertices/faces of the upper body part.
LowerBody	It describes mesh vertices/faces of the lower body part.
Head	It describes mesh vertices/faces of the head.
Face	It describes mesh vertices/faces of the face.
Back/Neck/Ears	It describes mesh vertices/faces of the back of the head, the neck and the ears.
MouthBag	It describes mesh vertices/faces of the mouth bag.
LowerJaw	It describes mesh vertices/faces of the lower part of the jaw.
UpperJaw	It describes mesh vertices/faces of the upper part of the jaw.
EyeLeft	It describes mesh vertices/faces of the left eye
EyeRight	It describes mesh vertices/faces of the right eye
Thorax	It describes mesh vertices/faces of the thorax
Chest	It describes mesh vertices/faces of the chest
UpperBack	It describes mesh vertices/faces of the upper back
Shoulders	It describes mesh vertices/faces of the shouldrs
ShoulderFront	It describes mesh vertices/faces of the front part of the shoulders
ShoulderFrontLeft	It describes mesh vertices/faces of the left front shoulder
ShoulderFrontRight	It describes mesh vertices/faces of the right front shoulder
ShoulderBack	It describes mesh vertices/faces of the back part of the shoulders
ShoulderBackLeft	It describes mesh vertices/faces of the left part of the shoulder
ShoulderBackRight	It describes mesh vertices/faces of the right part of the shoulder
ArmLeft	It describes mesh vertices/faces of the left arm
UpperArmLeft	It describes mesh vertices/faces of the upper part of the left arm
LowerArmLeft	It describes mesh vertices/faces of the lower part of the left arm
HandLeft	It describes mesh vertices/faces of the left hand
UpperArmRight	It describes mesh vertices/faces of the upper part of the right arm
LowerArmRight	It describes mesh vertices/faces of the lower part of the right arm.
HandRight	It describes mesh vertices/faces of the right hand.
Abdomen	It describes mesh vertices/faces of the abdomen.
AbdomenFront	It describes mesh vertices/faces of the front part of the abdomen.
LowerBack	It describes mesh vertices/faces of the lower part of the back.
LowerBody	It describes mesh vertices/faces of the lower body part.
Pelvis	It describes mesh vertices/faces of the pelvis.
PelvisFront	It describes mesh vertices/faces of the front part of the pelvis.

<i>Name</i>	<i>Definition</i>
PelvisFrontLeft	It describes mesh vertices/faces of the left front part of the pelvis.
PelvisFrontRight	It describes mesh vertices/faces of the right front part of the pelvis.
PelvisBack	It describes mesh vertices/faces of the back of the pelvis.
PelvisBackLeft	It describes mesh vertices/faces of the left back part of the pelvis.
PelvisBackRight	It describes mesh vertices/faces of the right back part of the pelvis.
LegLeft	It describes mesh vertices/faces of the left leg.
UpperLegLeft	It describes mesh vertices/faces of the upper part of the left leg.
LowerLegLeft	It describes mesh vertices/faces of the lower part of the left leg.
FootLeft	It describes mesh vertices/faces of the left foot.
LegRight	It describes mesh vertices/faces of the right leg.
UpperLegRight	It describes mesh vertices/faces of the upper part of the right leg.
LowerLegRight	It describes mesh vertices/faces of the lower part of the right leg.
FootRight	It describes mesh vertices/faces of the right foot.

• + *

7.1.1.2. Semantics of Skeleton Joints

This section provides a description of Morgan's skeleton. This skeleton, as presented in Annex H [2], is a hierarchy of joints.

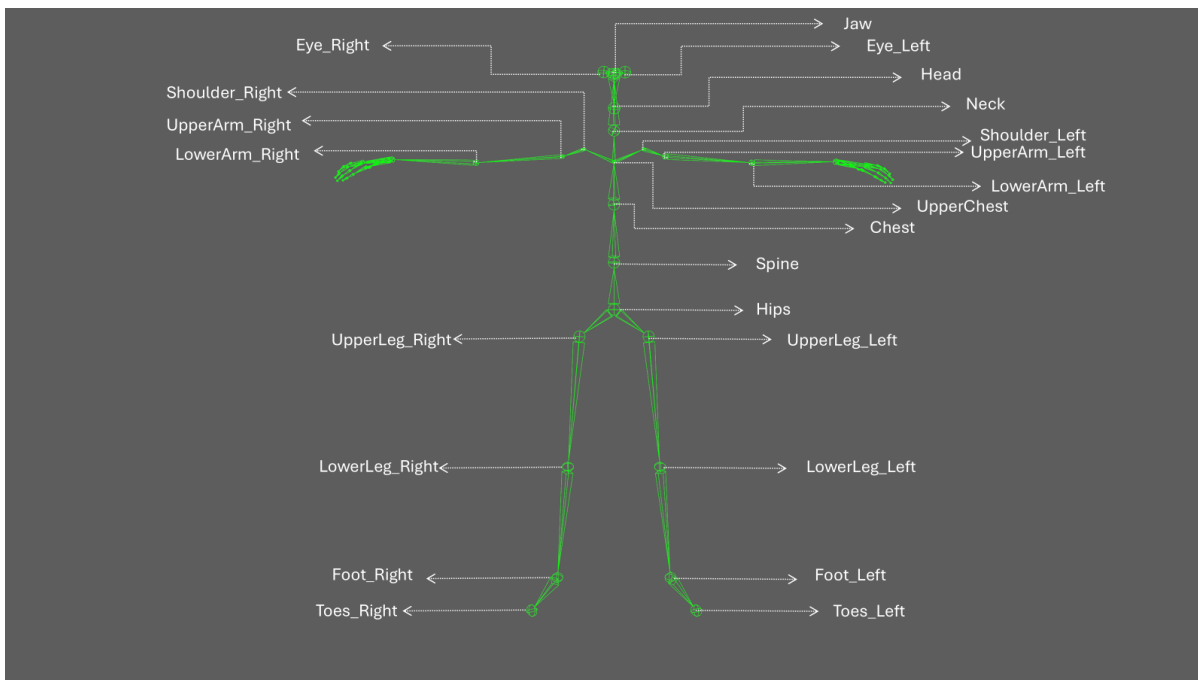


Figure 2: Body skeleton joints hierarchy on Morgan

Hands

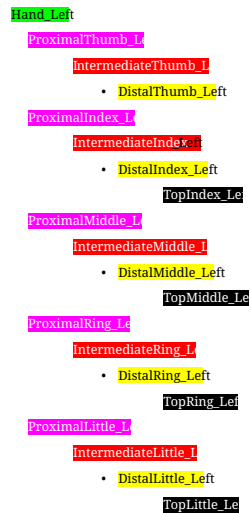
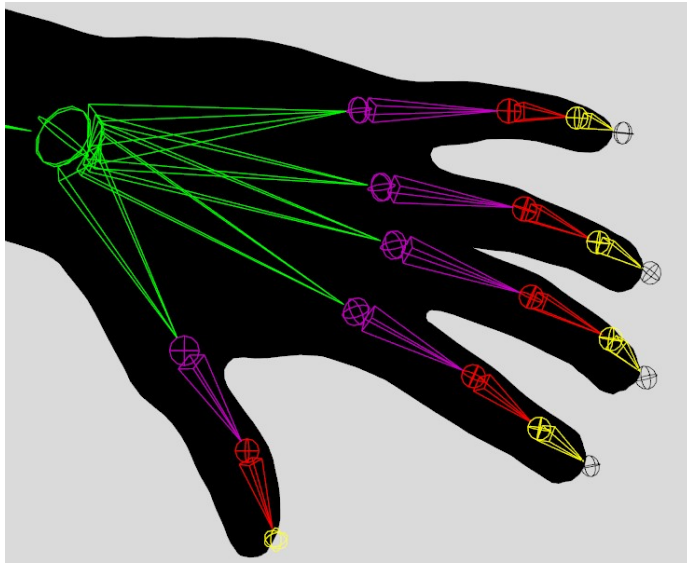


Figure 3: Hand skeleton joints hierarchy on Morgan

==

Name	Definition
Hips	It describes the skeleton joint of the hips
Spine	It describes the skeleton joint of the spine
Chest	It describes the skeleton joint of the chest
UpperChest	It describes the skeleton joint of the upper chest
Shoulder_Left	It describes the skeleton joint of the left shoulder
UpperArm_Left	It describes the skeleton joint of the upper part the left arm
LowerArm_Left	It describes the skeleton joint of the lower part of the left arm
Hand_Left	It describes the skeleton joint of the left hand
ProximalThumb_Left	It describes the skeleton joint of the proximal part of the left thumb
IntermediateThumb_Left	It describes the skeleton joint of the intermediate part of the left thumb
DistalThumb_Left	It describes the skeleton joint of the distal part of the left thumb
ProximalIndex_Left	It describes the skeleton joint of the proximal part of the left index
IntermediateIndex_Left	It describes the skeleton joint of the intermediate part of the left index
DistalIndex_Left	It describes the skeleton joint of the distal part of the left index finger
TopIndex_Left	It describes the skeleton joint of the top part of the left index finger
ProximalMiddle_Left	It describes the skeleton joint of the proximal part of the left middle finger

<i>Name</i>	<i>Definition</i>
IntermediateMiddle_Left	It describes the skeleton joint of the intermediate part of the left middle finger
DistalMiddle_Left	It describes the skeleton joint of the distal part of the left middle finger
TopMiddle_Left	It describes the skeleton joint of the top part of the left middle finger
ProximalRing_Left	It describes the skeleton joint of the proximal part of the left ring finger
IntermediateRing_Left	It describes the skeleton joint of the intermediate part of the left ring finger
DistalRing_Left	It describes the skeleton joint of the distal part of the left ring finger
TopRing_Left	It describes the skeleton joint of the top part of the left ring finger
ProximalLittle_Left	It describes the skeleton joint of the proximal part of the left little finger
IntermediateLittle_Left	It describes the skeleton joint of the intermediate part of the left little finger
DistalLittle_Left	It describes the skeleton joint of the distal part of the left little finger
TopLittle_Left	It describes the skeleton joint of the top part of the left little finger
Shoulder_Right	It describes the skeleton joint of the right shoulder
UpperArm_Right	It describes the skeleton joint of the upper part of the right arm
LowerArm_Right	It describes the skeleton joint of the lower part of the right arm
Hand_Right	It describes the skeleton joint of the right hand
ProximalThumb_Right	It describes the skeleton joint of the proximal part of the right thumb
IntermediateThumb_Right	It describes the skeleton joint of the intermediate part of the right thumb
DistalThumb_Right	It describes the skeleton joint of the distal part of the right thumb
ProximalIndex_Right	It describes the skeleton joint of the proximal part of the right index
IntermediateIndex_Right	It describes the skeleton joint of the intermediate part of the right index
DistalIndex_Right	It describes the skeleton joint of the distal part of the right index finger
TopIndex_Right	It describes the skeleton joint of the top part of the right index finger
ProximalMiddle_Right	It describes the skeleton joint of the proximal part of the right middle finger
IntermediateMiddle_Right	It describes the skeleton joint of the intermediate part of the right middle finger

<i>Name</i>	<i>Definition</i>
DistalMiddle_Right	It describes the skeleton joint of the distal part of the right middle finger
TopMiddle_Right	It describes the skeleton joint of the top part of the right middle finger
ProximalRing_Right	It describes the skeleton joint of the proximal part of the right ring finger
IntermediateRing_Right	It describes the skeleton joint of the intermediate part of the right ring finger
DistalRing_Right	It describes the skeleton joint of the distal part of the right ring finger
TopRing_Right	It describes the skeleton joint of the top part of the right ring finger
ProximalLittle_Right	It describes the skeleton joint of the proximal part of the right little finger
IntermediateLittle_Right	It describes the skeleton joint of the intermediate part of the right little finger
DistalLittle_Right	It describes the skeleton joint of the distal part of the right little finger
TopLittle_Right	It describes the skeleton joint of the top part of the right little finger
Neck	It describes the skeleton joint of the neck
Head	It describes the skeleton joint of the head
Eye_Left	It describes the skeleton joint of the left eye
Eye_Right	It describes the skeleton joint of the right eye
Jaw	It describes the skeleton joint of the jaw
UpperLeg_Left	It describes the skeleton joint of the upper part of the left leg
LowerLeg_Left	It describes the skeleton joint of the lower part of the left leg
Foot_Left	It describes the skeleton joint of the left foot
Toes_Left	It describes the skeleton joint of the left toes
UpperLeg_Right	It describes the skeleton joint of the upper part of the right leg
LowerLeg_Right	It describes the skeleton joint of the lower part of the right leg
Foot_Right	It describes the skeleton joint of the right foot
Toes_Right	It describes the skeleton joint of the right toes

7.1.1.3. Mapping with IoMT

The following table provides a mapping between IoMT skeleton joints [1] and Morgan's skeleton joints. The left column indicates the IoMT skeleton joint and, on the right column, the corresponding Morgan's skeleton joint.

IoMT skeleton joint	Morgan skeleton joint
Pelvis	Hips
RightPelvis	UpperLeg_Right
LeftPelvis	UpperLeg_Left
RightKnee	LowerLeg_Right
LeftKnee	LowerLeg_Left
RightAnkle	Foot_Right
LeftAnkle	Foot_Left
LumbarVertebra1	Spine
ThoracicVertebra8	Chest
ThoracicVertebra1	Upper_Chest
CervicalVertebra5	Neck
CervicalVertebra1	Head
RightClavicle	Shoulder_Right
RightShoulder	UpperArm_Right
RightElbow	LowerArm_Right
RightWrist	Hand_Right
LeftClavicle	Shoulder_Left
LeftShoulder	UpperArm_Left
LeftElbow	LowerArm_Left
LeftWrist	Hand_Left

7.1.2. References

[1] m67324 - Revision of data format and APIs for IoMT body keypoint extractor. Rennes, France - April 2023.

[2] N01025 - Revised text of ISO/IEC 23090-14 DAM 2: Support for Haptics, Augmented Reality, Avatars, Interactivity, MPEG-I Audio, and Lighting

Appendix A: Disclaimer



The formatting of the document is based on the Khronos glTF specification formatting under CC-BY 4.0.



The extensions information are automatically generated using [wetzel](#) tool under Apache License 2.0.