



Technology under Consideration for ISO/IEC 23090-14

WG3 Scene Description BoG

MDS25295_WG03_N01548

Table of Contents

1. Extensions	1
2. Codec Support	2
2.1. Dynamic mesh support in scene description	2
2.2. Support for multiple atlases for MIV applications (MPEG142)	2
2.2.1. Multiple atlases	2
2.2.2. References	9
2.3. Support for multi-view video and multi-channel audio sources	9
2.3.1. Introduction	9
2.3.2. Potential Solution to Support Multi-view Video	10
2.3.3. References	11
2.4. On the support of immersive codec application in MPEG-SD	11
2.4.1. General	11
2.4.2. Background	11
2.4.3. Motivation	12
2.4.4. Proposal	13
3. Interfaces	17
3.1. Supporting Multiple Viewers in the Media Access Function	17
3.1.1. General	17
3.1.2. Proposed Updates to MAF API	17
4. MPEG-I Audio in Scene Description	19
4.1. On spatial synchronization between graphs	19
4.1.1. Attempt problem definition for the spatial synchronization	19
4.1.2. Approach proposal for the spatial synchronization	21
4.1.3. Conclusion	23
4.1.4. References	23
5. Interactivity framework	24
5.1. On event-based scene update	24
5.1.1. General	24
5.1.2. A use case for event based updates	25
5.1.3. JSON patch limitations	26
5.1.4. Semantics for event-based update	27
5.2. Mapping interactivity to Khronos extension	28
5.2.1. Introduction	28
5.2.2. Overview of both Interactivity Frameworks	28
6. Collected problem statements and industry needs	41
6.1. On the support of real environment data	41
6.1.1. General	41
6.1.2. Representation of the real environment	41

6.1.3. Storing a representation of the real environment	42
6.1.4. Examples of framework for real environment handling	43
Appendix A: JSON Schema for extensions	46
A.1. JSON Schema for MPEG_primitive_V3C	46
A.2. JSON Schema for MPEG_primitive_V3C._MPEG_V3C_CAD.....	47
A.3. JSON Schema for MPEG_primitive_V3C.atlas	48
A.4. JSON Schema for MPEG_primitive_V3C.attribute	50
Appendix B: Disclaimer	51

Chapter 1. Extensions

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

==

2.4. On the support of immersive codec application in MPEG-SD

Source: [m73419](#)

2.4.1. General

The proposal focuses on the support of immersive codec applications in MPEG-I Scene Description, ISO/IEC 23090-14.

An immersive codec application refers to an application with video content designed to engage the viewer in a more interactive or enveloping experience, often by simulating a sense of presence within the scene.

2.4.2. Background

The MPEG-I Scene Description group has adopted glTF as a scene graph format for ISO/IEC 23090-

HEVC, etc. Such immersive video formats describe different components to express artistic intent which enable rich experiences for the user. The different components of immersive video formats can be declared in a glTF2.0 document with semantic meaning and associate texture information for the final rendering. In addition to the texture information, supplementary information such as camera calibration, stereo information, projection type and more can be described to accurately perform the final rendering of the immersive media.

To maintain the flexibility of texture combinations and ensure consistent visual fidelity across immersive experiences, a material extension could be enriched to support immersive representations.

2.4.4. Proposal

To integrate the support of immersive codec application in MPEG Scene Description, the proposal is to add an extension to the “material” in glTF 2.0, named “MPEG_Material”, as shown in [Figure 2](#) .

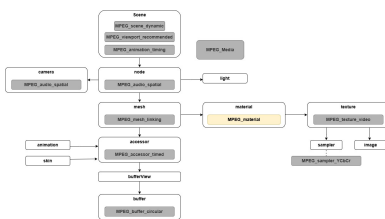


Figure 2. MPEG_material extension definition

Each mesh primitive has a reference to a material that defines how the surface of a 3D object looks when rendered. The MPEG_material extension is applied to the “material” property and signals one immersive codec application among a set of supported immersive applications. The “mesh” defines the geometry of the object, whether it is a 2D plane, a sphere or a 3D mesh object, while the MPEG_material extension aggregates all relevant information to a specific immersive media content within the same container, facilitating the scene reconstruction process. The definition of the MPEG_material extension is shown in Table 1.

Table 1 - Definition of MPEG_material extension.

Name	Type	Default	Usage	Description
codecApplication	string	N/A	M	Provides an information on the immersive codec application. Possible values include: "multiview", "stereo", "3D", and "surround".
layers	array[Layer]	N/A	M	Provides an array of layer objects for “multiview”, “stereo”, “3D”, and “surround” immersive codec applications

Name	Type	Default	Usage	Description
Legend: For attributes: M=mandatory, O=optional, OD=optional with default value, CM=conditionally mandatory.				

In Table 1, a “layer” is an object which combines a texture with its associated rendering properties. Below are some examples of how layers can be used in the context of an immersive codec application:

- In the case of “multiview” video, each view is represented by a separate layer, identified by a unique nal_layer_id within the NAL units; typically, a base layer provides a primary view, while non-base (enhancement) layers deliver additional views.
- In the case of “3D” video, layers are used to separate views and depth information, where a base layer provides the primary view and non-base (enhancement) layers add stereoscopic or multiview data.
- In “surround” video, layers are used to separate the full-scene base view and optional stereo components.

The definition of Layer is described in Table 2.

Table 2 - Definition of Layers in the MPEG_material extension.

Name	Type	Default	Usage	Description
primaryReference	boolean	N/A	M	Indicates whether this layer component is a primary component among other layers. For example, it can be used to refer to base (True) layer or non-base (False) layer. In case of stereo, it refers to the hero (True) or non-hero (False) layer.
texture	integer	N/A	M	Provides index of a texture.
cameraNode	integer	N/A	M	Provides index to a node containing a camera property used for reprojection. The camera property describes the internal characteristics of the capture camera (intrinsics). A camera node is a floating node with transform information (extrinsics) of the capture camera. It shall be only used for reprojection in the context of immersive codec applications.
eyeChannel	integer	N/A	O	Indicates if the information refers to left (0) or right (1) eye used in the case of stereoscopic application.
projectionType	integer	N/A	O	Indicates the type of projection used for deprojection in the case of surround video. It can be equirectangular(0), cubemap(1), pyramidal(2), parametric(3).

Name	Type	Default	Usage	Description
Legend: For attributes: M=mandato ry, O=optional, OD=optiona l with default value, CM=conditi onally mandatory.				

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. ();
};

```

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 3](#) 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

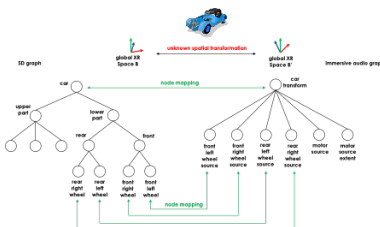


Figure 3. SD and immersive audio graph representations of a virtual car

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 4](#)
 - 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 5](#)

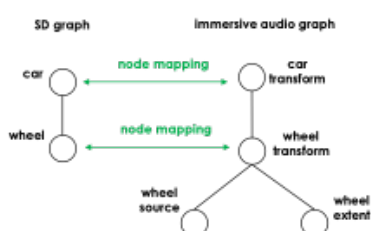


Figure 4. Possible approaches to ensure a spatial synchronization for both an audio source and its extent

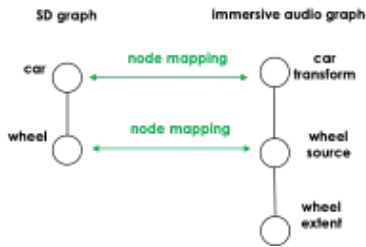


Figure 5. Possible approaches to ensure a spatial synchronization for both an audio source and its extent

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 6):

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.

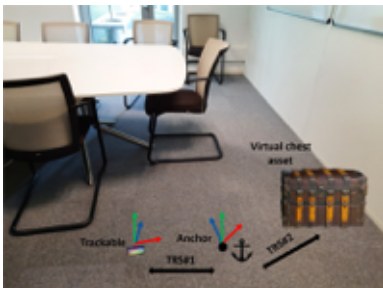


Figure 6. Trackable and Anchor for AR

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 6

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

Figure 7 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.

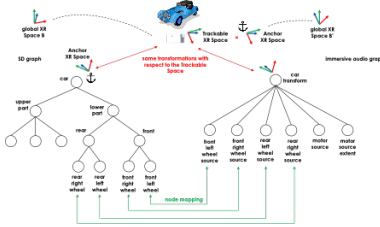


Figure 7. SD and immersive audio graph representations of a virtual car with AR anchoring using a 2D marker

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 8 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]

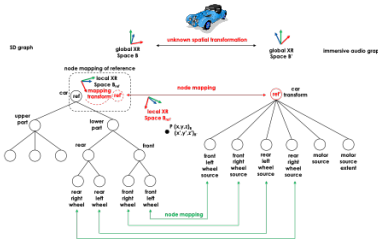


Figure 8. Transformation matrix determination using a node mapping

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

Chapter 5. Interactivity framework

5.1. On event-based scene update

Source: [m61812](#)

5.1.1. General

In the 23090-14 DIS document, a scene update mechanism is proposed, with predefined timed updates: A special track in a media content (for instance an ISOBMFF file), provides timed samples that contain patch (i.e., [JSON patch](#)) to be apply to the original scene description file.

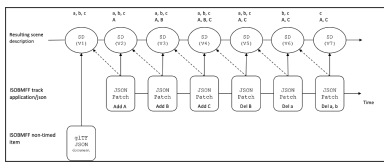


Figure 9. n/a

This mechanism handles pre-defined scene evolution but does not allow describing event-based update, following for instance a user action or any event that may occurred amongst the scene objects at any time. In the MPEG-I Scene Description output document on scene update [ISO/IEC JTC 1/SC 29/WG 3 N0315], a potential solution is presented for event-based scene updates : while a predefined timed scene update is in progress, an event may occur that updates the scene description. Several scenarios are then proposed: apply a patch and switch to a new timed samples track or apply a patch and skip one or more versions in the same track.

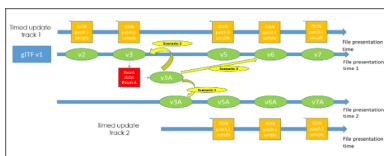


Figure 10. n/a

This mechanism is still strongly related to pre-defined scene evolutions and does not specify how the event that triggers the update is described in the scene description document.

Furthermore, it does not handle the case where the same event that creates a new node may be fired multiple times, like illustrated in the following diagram: A glTF scene contains a description of an event-based update mechanism with the same patch applied each time an event is fired. Some elements of the glTF scene are modified (adding, changing or removing nodes, meshes parameters) but not the event-based update description.

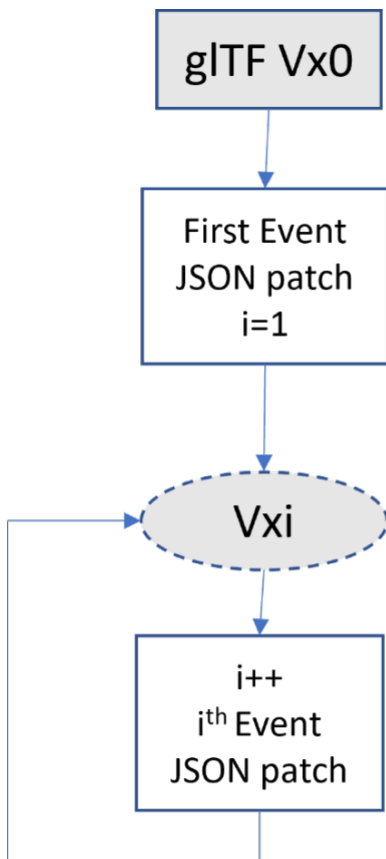


Figure 11. Event-based update diagram

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:

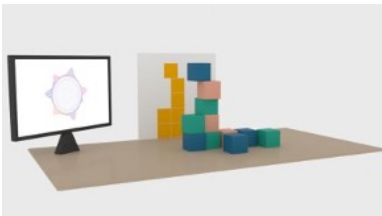


Figure 12. n/a



Figure 13. 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...), ...

5.2. Mapping interactivity to Khronos extension

Source: [m72365](#)

5.2.1. Introduction

In this contribution, we present a detailed mapping between the MPEG_interactivity extension (defined in ISO/IEC 23090-14) and the KHR_interactivity extension (the Khronos Group’s glTF 2.0 interactivity extension). We begin with an overview of each extension, including their structure and purpose. Next, we define a one-to-one mapping of core interactivity constructs – such as triggers, actions, and variables in MPEG_interactivity – to their equivalents in KHR_interactivity (event nodes, flow/logic nodes, variables, operations, etc.). We then discuss the processing rules for each system, explaining how events are triggered, conditions evaluated, and actions executed in MPEG’s model versus Khronos’s behavior graph model. Finally, we illustrate the mapping with a fully detailed example scenario: a user interacts with a “remote control” object in a 3D scene to turn on a TV and start video playback. The scenario is described narratively and shown in both MPEG_interactivity JSON syntax and KHR_interactivity JSON (behavior graph) syntax, accompanied by diagrams of the event flow. Throughout the report, we use clear section headings and concise explanations for readability.

5.2.2. Overview of both Interactivity Frameworks

5.2.2.1. 2.1 Overview of MPEG_interactivity

The MPEG_interactivity extension is part of the MPEG-I Scene Description standard and is designed to integrate interactive behavior into 3D scenes (built on glTF 2.0). Interactivity in MPEG-I is defined at two levels: a scene-level extension (MPEG_scene_interactivity) and a node-level extension (MPEG_node_interactivity). The scene-level extension contains the global definitions of triggers, actions, and behaviors that apply to the scene. The node-level extension can complement these by providing additional data or specialized parameters for specific nodes (for example, physics or collision properties for that node). In essence, MPEG_interactivity allows authors to define interactive behaviors that link user or environment events to changes in the scene, all within the MPEG scene description.

Triggers and Actions: In MPEG’s model, a behavior is the fundamental unit of interactivity, defined as a pairing of one or more triggers with one or more actions. Triggers represent events or conditions that can occur – for example, a collision between objects, an object coming within a certain distance of another (proximity), a user input event, or an object’s visibility state. These triggers can originate from user interactions (e.g. controller input or gestures), temporal/spatial conditions, or system events. Actions represent the responses that occur when triggers are activated – for example, enabling or disabling an object, transforming or moving an object, playing an animation or media, changing a material, applying haptic feedback, etc. MPEG_interactivity

defines a fixed set of trigger types and action types. Below is a summary of the main types:

- Trigger types:
 - *TRIGGER_COLLISION*: Fires when a collision is detected between specified nodes.
 - *TRIGGER_PROXIMITY*: Fires when one node comes within a defined distance of another.
 - *TRIGGER_USER_INPUT*: Fires on a user input or gesture (e.g. a button press or hand motion).
 - *TRIGGER_VISIBILITY*: Fires when a node becomes visible or hidden (e.g. entering/exiting the camera frustum).
- Action types:
 - *ACTION_ACTIVATE*: Enable or disable a node's active state in the scene.
 - *ACTION_TRANSFORM*: Apply a specified transformation to target node(s).
 - *ACTION_BLOCK*: Lock or freeze a node's transform.
 - *ACTION_ANIMATION*: Control an animation, e.g. play, pause, resume, stop a glTF animation by index.
 - *ACTION_MEDIA*: Control a media asset, e.g. video or audio playback of media defined in an MPEG_media extension).
 - *ACTION_MANIPULATE*: Initiate a user manipulation of a node, e.g. grabbing an object with a controller.
 - *ACTION_SET_MATERIAL*: Swap the material of target node(s), e.g. to change an object's appearance.
 - *ACTION_HAPTIC*: Trigger haptic feedback via haptic devices, with parameters for the type of feedback.
 - *ACTION_SET_AVATAR*: Invoke an avatar-specific action.

A behavior in MPEG_interactivity ties together one or more triggers with one or more actions, along with some logic controls. The behavior definition can specify a logical combination of triggers (using AND, OR, NOT operators) that must be satisfied to activate the behavior. For example, a behavior might be set to trigger only when Trigger A AND Trigger B are true, or Trigger C OR Trigger D is true, etc. This logical expression is given as a string (e.g. "#1 & ~#2 | (#3 & #4)" where numbers refer to trigger indices). The behavior also defines a triggersActivationControl, which indicates when the trigger condition is considered activated. This allows behaviors to fire one-time or repeatedly, and to detect both the "enter" and "exit" of a condition (e.g. when a user enters a zone vs leaves it). Additionally, a behavior can specify whether multiple actions execute sequentially or in parallel, an optional interruptAction (an action to execute if an ongoing behavior is interrupted or removed), and a priority value to arbitrate if multiple behaviors conflict.

In summary, MPEG's interactivity model is a rule-based system, where behaviors are essentially rules of the form "if these trigger conditions are satisfied, then perform these actions."

5.2.2.2. Overview of Khronos Interactivity

The Khronos KHR_interactivity extension for glTF 2.0 introduces a behavior graph system to incorporate interactivity into glTF assets. Instead of pre-defined trigger-action lists,

KHR_interactivity uses a node graph (behavior graph) model similar to Unreal Engine’s Blueprints or Unity’s visual scripting. The extension allows content creators to define logic within the glTF file itself, so that the asset can respond to events in a consistent way across different viewers. The primary motivation is to make interactive 3D assets portable and self-contained. The focus is on safety and sandboxing; by using a limited set of graph nodes and not arbitrary scripting, the behaviors remain predictable and secure across platforms.

A KHR_interactivity behavior graph is essentially a directed acyclic graph (DAG) of nodes connected by links (edges) that pass execution flow or data. Each node performs a simple function (such as detecting an event, evaluating a condition, or performing an action), and nodes are connected such that an event triggers a flow through the graph, causing actions to happen. Nodes in the graph fall into one of several categories:

- **Event Nodes:** these are entry points that listen for certain events and start the execution flow when the event occurs. Examples of events include *lifecycle events* (like “On Start” when the scene/asset loads, or “On Tick” each frame) and *custom or user-defined events*. Custom events are essentially named events that can be triggered by external application.
- **Action/Operation Nodes:** Nodes that perform an action or change in the scene. These correspond to things like starting an animation, changing a material or variant, transforming a node, playing a sound or video, etc. They typically consume input values and produce an effect. Often, these nodes will interface with other glTF extensions or properties (e.g., a node to play an animation will reference a glTF animation, etc.).
- **Logic/Flow Control Nodes:** Nodes that direct the flow of execution or make decisions. Examples include *Branch* (an if-else node that routes the flow based on a boolean condition), loops, and sequence or delay nodes to chain or defer actions. These nodes don’t directly affect the scene; they control which actions happen and when.
- **Variable/State Nodes:** Nodes that store or manipulate state. The extension introduces the concept of variables that can hold values during the execution of the graph. Variable nodes might include setting a variable, reading a variable, or modifying it (e.g., incrementing a counter). These are useful for keeping track of state across events. There are also query nodes which can retrieve information from the glTF scene or runtime.
- **Math/Conversion Nodes:** Basic arithmetic or logic comparison nodes (e.g., add, subtract, boolean AND/OR, compare values) and type converters. These help build conditions and compute values to use in decisions or actions.

All nodes have defined input sockets and output sockets. There are typically two kinds of connections: flow connections that pass along execution order and data connections that pass values. The use of flow sockets means the graph’s execution is explicit: an event node emits a flow, which travels through connected nodes in sequence. This is how behaviors are executed. In other words, when an event occurs, it triggers the next node via a flow link, and so on, forming a chain of execution.

KHR_interactivity by itself defines the graph framework (nodes, events, variables), but it works in concert with other glTF mechanisms to actually affect the 3D scene. For example, to change the state of an object, an operation node might use the KHR_animation_pointer extension to target a specific property of a glTF node (like its translation, or a custom “visibility” flag). Likewise, an action node that plays a video might rely on a video texture extension or media extension to handle

the media resource. This modular approach means KHR_interactivity can be extended by introducing new node types via additional extensions. The initial set of node types covers common needs (UI events, animation control, variant switching, simple logic, etc.), and more specialized interactions (like physics or complex UI) may involve additional glTF extensions.

5.2.2.3. Mapping MPEG_node/scene_interactivity to KHR_interactivity

Despite differences in approach (rule-based vs node-graph), MPEG_interactivity and KHR_interactivity address similar needs and concepts. Below, we map the key constructs one-to-one between MPEG's design and Khronos's design:

1. Triggers (MPEG) \square Event Nodes (KHR): A trigger in MPEG corresponds to an event that can start a behavior. In KHR_interactivity this is represented by an Event node in the behavior graph. For example:
 - *MPEG TRIGGER_USER_INPUT* maps to a Custom Event node or a specific input event node in KHR. Khronos does not hard-code user input events in the initial spec; instead, one would use a *Custom Event* node that is fired by the viewer when the user performs that input. In practice, this means an MPEG user input trigger like "left controller trigger pulled" would translate to a custom event named "left_trigger_pull" in the glTF behavior graph, which the viewer knows to send on that input.
 - *MPEG TRIGGER_COLLISION* has no direct built-in equivalent in the base KHR_interactivity since physics/collision aren't covered in the initial draft. The analogous concept would be an event triggered by a physics engine. This could be achieved with an extension: for instance, a KHR_physics extension might generate a custom event when a collision occurs.
 - *MPEG TRIGGER_PROXIMITY* similarly maps to a custom event from an engine when an object enters/exits a zone. In essence, a proximity trigger can be implemented in KHR_interactivity using the building blocks of event + logic nodes, since there isn't a single "OnProximity" node by default.
 - *MPEG TRIGGER_VISIBILITY* would map to an event or condition related to the camera view. Khronos could handle this by using an On Tick event and a query node to check if the object is in the camera frustum, combined with a branch node. Alternatively, a custom event could be fired by the viewer when an object enters/exits view. This again shows that MPEG provides a specific trigger type, whereas KHR_interactivity might rely on general mechanisms or future extensions to achieve the same.
 - *Lifecycle triggers*: Although not explicitly named "triggers" in MPEG_interactivity, one can consider the scene start as an implicit trigger. In KHR_interactivity, there is a built-in OnStart event node, which fires when the asset is loaded, and an OnTick which have no direct MPEG analog, since MPEG's triggers are more content-centric.
2. Actions (MPEG) \square Action/Operation Nodes (KHR): An MPEG action corresponds to a node in the behavior graph that performs the equivalent operation. Many MPEG action types have clear counterparts or ways to achieve them in KHR_interactivity:
 - *ACTION_ACTIVATE*: This maps to toggling a node's active state or visibility. glTF core doesn't have an "enabled" flag, but a parallel extension (KHR_node_visibility) is in development to allow hiding/showing nodes. In a KHR_interactivity graph, one could use an operation node that sets a node's visibility to true or false via the KHR_node_visibility property.

- *ACTION_TRANSFORM*: This corresponds to directly manipulating a node's translation/rotation/scale. In KHR_interactivity, this would likely be done via an Animation or Pointer node, e.g. using KHR_animation_pointer to target a node's transform and setting it. The graph might use a node that takes a matrix or vector value and applies it to the target node.
- *ACTION_BLOCK*: In MPEG, this prevents a node from moving. There isn't a direct single node in KHR_interactivity for this, but it could be interpreted as setting certain physics or interaction constraints. If we consider a physics extension, an equivalent would be to change the body to static or disable user interaction on that node.
- *ACTION_ANIMATION*: This maps well to KHR_interactivity's abilities. A likely implementation is an Animation Control node that can play/pause/stop a glTF animation.
- *ACTION_MEDIA*: Khronos is working on incorporating media (video & audio) into glTF. Using an extension like MPEG_texture_video and MPEG_media for video textures, the behavior graph would control media similarly to animations. A KHR action node for media might take a media/texture ID and a play/pause command.
- *ACTION_MANIPULATE*: This is quite interactive and likely relies on continuous input. In KHR_interactivity, continuous interactions, like dragging an object with the mouse or a controller, might be handled outside the behavior graph logic or by a combination of event + continuous update.
- *ACTION_SET_MATERIAL*: glTF has a dedicated system for material variants (KHR_materials_variants). In a KHR_interactivity graph, changing a material can be done by setting the active variant of an object. For instance, there could be an Action node that sets a variant index on an object.
- *ACTION_HAPTIC*: Currently, glTF has no haptic feedback features in standard extensions. This is a specialized case where MPEG defines parameters for haptic devices. KHR_interactivity doesn't cover this yet; a future extension or external API would be needed. One could envision a custom event node that communicates with a haptic device when triggered. Thus, an MPEG haptic action would correspond to triggering some external haptic system in the Presentation Engine.
- *ACTION_SET_AVATAR*: Similarly, glTF doesn't have built-in avatar systems. MPEG's avatar actions, like toggling an avatar's microphone or performing an animation on an avatar, would require an external avatar system..

In general, MPEG's action types map onto either specific KHR_interactivity nodes or combinations of nodes. Khronos's design tends to break down effects into simpler pieces, whereas MPEG sometimes has a single action that encapsulates a multi-step process, like manipulate or haptic, which involve continuous feedback or external devices. The simpler actions, like activate, transform, play animation, set material, have clear counterparts in the graph model.

- Behavior (MPEG) □ Behavior Graph / Event Flow (KHR): An MPEG behavior object as a whole corresponds to an event flow in the KHR_interactivity graph. In other words, a single MPEG behavior can be thought of as a little program "if [trigger conditions] then [do actions]." In a node graph, this would be represented by wiring the event nodes for those triggers through logic nodes into the sequence of action nodes. For a simple behavior with one trigger and one action, the mapping is straightforward: one Event node connected to one Action node. For a

more complex behavior, i.e. multiple triggers and multiple actions with logic, the contents of the MPEG behavior need to be expanded into multiple graph nodes, e.g., several Event nodes feeding into a Logic/AND node to replicate a combined condition, then that logic node feeding into multiple Action nodes. The `triggersCombinationControl` string in MPEG is effectively replaced by a network of boolean logic nodes in `KHR_interactivity`, AND, OR, NOT nodes linking the outputs of event nodes, which then feed into a branch. Likewise, the `triggersActivationControl` modes, such as `FIRST_ENTER`, `EACH_ENTER`, `ON`, etc., do not exist explicitly in KHR's design. Instead, achieving the equivalent behavior relies on how the graph is constructed:

- “`FIRST_ENTER`” could be mimicked by using a variable as a flag to remember it fired, or by using an Event node that only triggers once.
- “`EACH_ENTER`” is effectively how event nodes naturally behave if you use instantaneous events. A combination of conditions would need edge detection logic, which could be done with variables.
- “`ON`” could be achieved by using a continuous event like `OnTick` and inside it, use an IF (Branch) to continuously do something while a condition holds..
- “`FIRST_EXIT/EACH_EXIT`” similarly would require tracking state and using logic in the graph, like using a combination of `OnTick` + a stored boolean to detect the transition.
- **Variables and State:** `MPEG_interactivity` doesn't define general-purpose variables for behaviors; it relies on triggers and some internal state like whether an action is ongoing. Conditions are directly encoded as triggers or combinations thereof, rather than allowing arbitrary state checks. In contrast, `KHR_interactivity` provides variables as a first-class concept. This means some logic that would require a custom trigger in MPEG can be done by checking a variable in KHR.
- **Processing Model Differences:** In MPEG, behaviors are evaluated by the engine each frame: all triggers are polled/evaluated, and when conditions match, the associated actions are launched. It's a data-driven approach where the scene description lists behaviors and the engine continuously checks them. In `KHR_interactivity`, the behavior graph sits idle until an event node is invoked; then it propagates execution along the linked nodes. There isn't a concept of continuously checking a condition unless you explicitly set that up with an `OnTick` event. So effectively, MPEG's triggers that are continuously monitored map to either event nodes that are inherently continuous (`OnTick`) or to having multiple event nodes fire as needed. This means some things that are automatic in MPEG (like collision detection triggering an action) will, in a glTF context, depend on the viewer providing those events or the graph explicitly querying.

In summary, `MPEG_interactivity` and `KHR_interactivity` cover the same functionality but through different paradigms. To map MPEG to Khronos: triggers correspond to events, possibly requiring additional logic nodes in Khronos's interactivity extension. Variables and custom logic in Khronos can be used to achieve complex conditions of MPEG's interactivity.

5.2.2.4. Processing and Execution Rules

MPEG defines a clear processing model for how interactive behaviors are handled at runtime. When the scene is loaded, the Presentation Engine (the runtime) will parse the glTF and set up all the behaviors described in the `MPEG_scene_interactivity` extension. Each behavior knows its triggers and actions (and any node-level parameters from `MPEG_node_interactivity`). At runtime

(typically each frame or each time the scene updates), the engine iterates through all defined behaviors and evaluates them as follows:

1. It checks the status of each trigger in that behavior – meaning it evaluates whether each trigger condition is currently active or has occurred (e.g., is collision X happening? is the button pressed? is object Y visible?). There is an underlying procedure or algorithm for each trigger type to determine its boolean state (the spec even provides a flowchart for trigger activation in the standard).
2. It then evaluates the logical combination of those triggers as specified by the behavior (applying the AND/OR/NOT from the triggersCombinationControl). This results in an overall true/false evaluation for the condition of the behavior in the current frame.
3. The engine then checks this result against the triggersActivationControl setting for the behavior. For example, if triggersActivationControl is “EACH_ENTER,” the behavior should fire *at the moment* the condition goes from false to true. So the engine might compare the current condition state with the previous frame’s state to decide if this is a newly true event. If it’s “ON,” it would consider the behavior active continuously while the condition is true (possibly triggering actions continuously or at least keeping them active). The specifics are defined in the standard’s Table 12 (as listed in the extension): FIRST_ENTER triggers once on true, EACH_ENTER triggers every time it becomes true, etc. If the condition meets the criteria (e.g., it just became true for a FIRST_ENTER, or it is true for ON), then
4. the engine launches the actions associated with the behavior. Launching actions means it executes each action in either sequential or parallel order as specified. Sequential actions might be executed over multiple frames if they have delays, whereas parallel actions are initiated together (for instance, starting an animation and a sound at the same time). Some actions (like play animation or play media) are instantaneous triggers that then proceed over time. The MPEG model accounts for actions that continue over time (e.g., an animation playing) by considering a behavior “ongoing” until those complete. If a scene update removes an ongoing behavior, the engine will execute the defined interruptAction (if any) to gracefully stop the behavior. Also, MPEG specifies that if multiple behaviors try to affect the same node simultaneously, the one with higher priority wins and the other is suppressed. This ensures determinism when two rules conflict (for example, one behavior says “move object up” and another says “move object down” at the same time – the one with higher priority will take effect). The entire cycle of checking triggers and updating actions repeats each frame or whenever the scene state changes. In effect, MPEG_interactivity acts as a continuous rule evaluator: at any moment, it will respond to the current state of inputs by initiating the appropriate outputs.

The KHR_interactivity behavior graph has a different execution model more akin to an event-driven system. Rather than continuously scanning conditions, it waits for events and propagates changes through the graph. Here’s how it works: When an interactive glTF asset is loaded, the viewer will initialize the behavior graph. This likely involves creating runtime representations of all the nodes (events, variables, etc.) and possibly initializing default variable values. Some event nodes may fire immediately upon load – notably, the OnStart (or equivalent) event node will trigger as soon as the graph is ready, which can start certain behaviors (e.g., an introductory animation) without any user input. During runtime, events occur either because of user interaction or as part of the system (for example, each frame an OnTick event may fire, or a custom event is emitted when an external condition is met). When an Event node fires, it emits a flow signal that travels

along its outgoing connections in the graph. The nodes connected to that event's output will then execute in order. Execution in the graph is generally instantaneous in the sense that in a single frame tick, an event can trigger a whole chain of nodes to run sequentially. Each node, upon execution, may do something and then pass along flow to the next. For example, if an Event node is connected to an Action node: as soon as the event happens, the action node executes. If that action node has a flow output to another node (like another action), it will then trigger that next node, and so on – all within the same event invocation. There isn't an external loop checking all nodes; instead, nodes call one another via these links. This is a reactive execution: something happens (event), then reactions propagate.

During this propagation, flow control nodes can decide to branch or loop. A Branch (if/else) node will receive the flow, check a condition (e.g., compare a variable's value), and then send the flow down one of its two outputs (true path or false path). This is how conditions are evaluated in KHR_interactivity – the condition check is just another node in the chain. For looping, the extension avoids infinite loops by design (no direct cycle in the graph), but a node might have multiple flow outputs (like a loop node could output back to an earlier part but that's likely forbidden to keep acyclic). Instead, repeating behavior is usually achieved by using OnTick events or by an action re-triggering an event. For instance, a repeating animation loop might be handled by having an animation node that upon finishing emits a custom event that loops back to start it again (that “loop” was described in the sofa example using a custom event to repeat). So loops are achieved by scheduling events rather than actual graph cycles.

5.2.2.5. Example Scenario: Remote Control Triggers TV On (Video Playback)

To illustrate the mapping, consider a scenario in an interactive 3D scene: A user reaches out and presses a virtual remote control object, which causes a 3D television in the scene to turn on and start playing a video. We will describe this scenario and then show how it can be implemented using both MPEG_interactivity and KHR_interactivity, including example JSON syntax and a diagram of the event flow for each.

The scene consists of a television set and a remote control as separate objects. Initially, the TV is “off”, perhaps its screen is dark and no video is playing. The remote control is an object the user can interact with. When the user performs the appropriate input, e.g. clicking on the remote control object, that interaction is detected by the system. The interactivity logic then triggers two changes: (1) the TV's power state turns on, i.e. change its material to a lit screen), and (2) a video begins playing on the TV's screen.

This scenario involves one primary event (user presses remote) and two actions (turn on TV, play video). There could also be a condition, e.g. only do it if the TV was off, but for simplicity we'll assume the TV is off and the remote always turns it on.

Below we show how the MPEG_interactivity JSON might look like, and then the equivalent KHR_interactivity behavior graph JSON.

MPEG_interactivity Implementation

In MPEG_interactivity, we define a trigger for the remote-control input, actions for turning on the TV and playing the video, and a behavior that links them. We also assume a media is defined in the MPEG_media extension.

The JSON snippet might look like this:

```
"extensionsUsed": [
  "MPEG_scene_interactivity",
  "MPEG_media"
],
"extensions": {
  "MPEG_media": {
    "media": [
      {
        "uri": "tv_video.mp4",
        "mimeType": "video/mp4"
      }
    ]
  }
},
"scene": 0,
"scenes": [
  {
    "extensions": {
      "MPEG_scene_interactivity": {
        "triggers": [
          {
            "type": "TRIGGER_USER_INPUT",
            "userInputDescription": "/user/hand/right/input/select/click"
          }
        ]
      }
    }
  }
]
```

```

    ],
    "actions": [
        {
            "type": "ACTION_ACTIVATE",
            "activationStatus": "ENABLED",
            "nodes": [ 1 ]    // assume node 1 = TV
        },
        {
            "type": "ACTION_MEDIA",
            "media": 0,        // play media index 0 (tv_video.mp4)
            "mediaControl": "MEDIA_PLAY"
        }
    ],
    "behaviors": [
        {
            "triggers": [ 0 ],
            "actions": [ 0, 1 ],
            "triggersCombinationControl": "",
            "triggersActivationControl": "TRIGGER_ACTIVATE_EACH_ENTER",
            "actionsControl": "SEQUENTIAL"
        }
    ]
}

},
"nodes": [ ...
    { "name": "RemoteControl", /* remote node index 0 */ },

```



```

    { "name": "TV", /* TV node index 1, initially off/inactive */ }
  ... ]
}
]

```

In this example, we use a `TRIGGER_USER_INPUT` for the remote control. The `userInputDescription` is given as an OpenXR path `"/user/hand/right/input/select/click"`, which represents a generic “select” action (like pulling a trigger or clicking) with the right hand. This implies that when the user performs the select action (while pointing at or near the remote, presumably), the trigger fires. We list two actions: an `ACTION_ACTIVATE` targeting the TV’s node (index 1) with `activationStatus: ENABLED` to turn it on, and an `ACTION_MEDIA` referencing media 0 (which in `MPEG_media.media[0]` is `tv_video.mp4`) with control `MEDIA_PLAY` to start playback of the video.

KHR_interactivity Implementation

Now, we implement the same scenario with `KHR_interactivity`. We need to create a behavior graph that listens for the remote press event and triggers the TV on + video play actions. In the glTF, this would be done inside the `KHR_interactivity` extension object. We will use a Custom Event node for the remote press, and two Operation nodes for the actions. We’ll also assume we have an extension for visibility or activation, and an extension to play the video. We assume the following:

- The Presentation Engine will send a custom event named `"remote_pressed"` when the user clicks the remote object.
- The TV’s visibility is controlled by a boolean property visible on the TV node.
- The video playback can be started by setting a parameter on a video texture.

Given those assumptions, the behavior graph JSON could look like:

```

"extensionsUsed": [
  "KHR_interactivity",
  "KHR_node_visibility",
  "EXT_texture_video",

"extensions": {

"KHR_interactivity": {

  "nodes": [

    {

      "id": 0,

      "type": "Event",

```

```

    "eventType": "custom",

    "name": "remote_pressed",

    "outputs": {

        "flow": [ 1 ]

    }

},

{

    "id": 1,

    "type": "Operation",

    "operation": "setVisibility",

    "target": \{ "node": 1, "property": "visible" \},

    "value": true,

    "outputs": \{

        "flow": [ 2 ]

    }

},

{

    "id": 2,

    "type": "Operation",

    "operation": "playMedia",

    "target": { "node": 1, "media": 0 },

    "outputs": {}

}

]

}

```

```
}
```

This example highlights how the same interactive outcome is achieved through each extension. In MPEG_interactivity, we relied on the predefined trigger and action types and simply listed them in a behavior object. In KHR_interactivity, we manually built the logic using behavior graph nodes. Both realizations require integration with a media extension for the video and (in Khronos's case) a visibility/activation mechanism.

[1] WG03 N01221, 23090-14 2nd Edition, MPEG Scene Description

==

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 14](#), 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

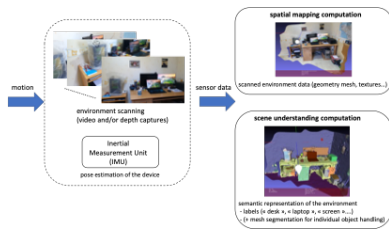


Figure 14. Computation of real environment data

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 15, 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

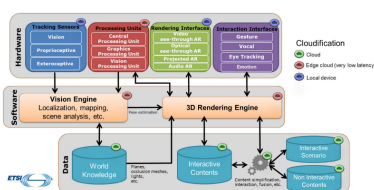


Figure 15. Global overview of the architecture of an AR system (from ETSI ARF)

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 16](#)
- 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 17](#)

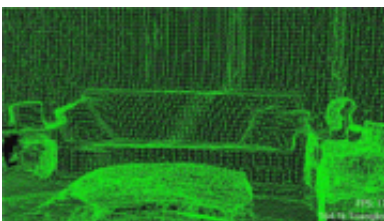


Figure 16. Mesh representation of the real environment after a spatial mapping computation



Figure 17. 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 18](#), a semantic labeling is performed for the real objects that ARKit can identify such as ceiling, door, floor, seat, table, wall and window labels.

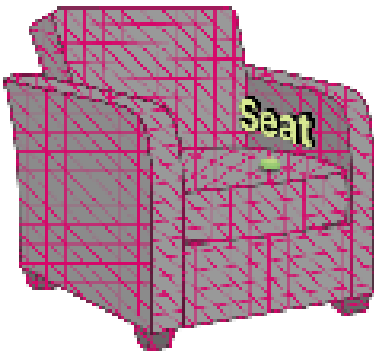


Figure 18. Semantic labeling of Apple's ARKit

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

serializing/deserializing an ARWorldMap as shown in [Figure 19](#).

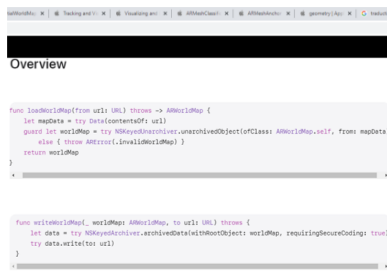


Figure 19. Saving and loading an Apple's ARKit ARWorldMap

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 20](#).

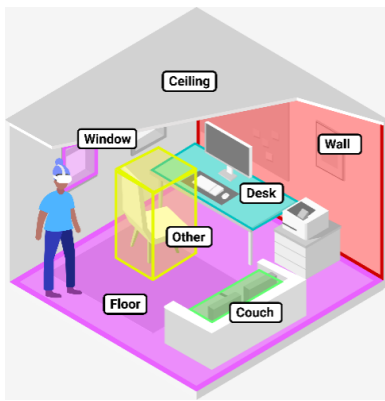


Figure 20. Semantic labeling of the Meta/Oculus Scene Understanding

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()
```


Appendix A: JSON Schema for extensions

A.1. JSON Schema for MPEG_primitive_V3C

```
{
  "$schema" : "http://json-schema.org/draft-07/schema",
  "title" : "MPEG_primitive_V3C",
  "type" : "object",
  "description": "glTF extension to specify support for V3C compressed primitives.",
  "allOf": [ { "$ref": "glTFProperty.schema.json" } ],
  "properties" : {
    "atlases": {
      "type": "array",
      "description": "An array of atlases",
      "gltf_detailedDescription": "An array of atlases",
      "items": {
        "$ref": "MPEG_primitive_V3C.atlas.schema.json"
      },
      "minItems" : 1
    },
    "_MPEG_V3C_CAD": {
      "type" : "object",
      "allOf": [{ "$ref": "MPEG_primitive_V3C._MPEG_V3C_CAD.schema.json"}],
      "description": "This object lists different properties described for the  
Common Atlas Data in ISO/IEC 23090-5." ,
      "gltf_detailedDescription" : "This object lists different properties  
described for the Common Atlas Data in ISO/IEC 23090-5.",
      "minItems":0
    },
    "extensions": {},
    "extras": {}
  },
  "required": ["atlases"]
}
```

A.2. JSON Schema for MPEG_primitive_V3C._MPEG_V3C_CAD

```
{
  "$schema" : "http://json-schema.org/draft-07/schema",
  "title" : "MPEG_primitive_V3C._MPEG_V3C_CAD",
  "type" : "object",
  "description": "defines the common atlas data for a v3c object",
  "allOf": [ { "$ref": "glTFProperty.schema.json"} ],
  "properties" : {
    "MIV_view_parameters": {
      "type": "integer",
      "description": "indicates the accessor index which is used to refer to the  
list of MIV view parameters.",
      "gltf_detailedDescription": "indicates the accessor index which is used to  
refer to the list of MIV view parameters.",
      "minimum": 1
    }
  },
  "required": [
    "MIV_view_parameters"
  ]
}
```

A.3. JSON Schema for MPEG_primitive_V3C.atlas

```
{
  "$schema" : "http://json-schema.org/draft-07/schema",
  "title" : "MPEG_primitive_V3C.atlas",
  "type" : "object",
  "description": "glTF extension to specify support for V3C compressed primitives.",
  "allOf": [ { "$ref": "glTFProperty.schema.json" } ],
  "properties" : {
    "_MPEG_V3C_CONFIG": {
      "allOf": [ { "$ref": "glTFid.schema.json" } ],
      "description": "",
      "gltf_detailedDescription": ""
    },
    "_MPEG_V3C_AD": {
      "allOf": [ { "$ref": "glTFid.schema.json" } ],
      "description": "",
      "gltf_detailedDescription": "a reference to the accessor that points to
the atlas data."
    },
    "_MPEG_V3C_GVD_MAPS": {
      "type": "array",
      "description": "an array of references to video texture maps.",
      "gltf_detailedDescription": "an array of references to video textures that
provide the geometry maps.",
      "items": {
        "allOf": [ { "$ref": "glTFid.schema.json" } ]
      },
      "minItems": 1
    },
    "_MPEG_V3C_OVD_MAP": {
      "type": "array",
      "description": "a reference to a video texture that provides the occupancy
map",
      "gltf_detailedDescription": "a reference to a video texture that provides
the occupancy map",
      "items": {
        "allOf": [ { "$ref": "glTFid.schema.json" } ]
      },
      "minItems": 0
    },
    "_MPEG_V3C_AVD": {
      "type": "array",
      "description": "",
      "gltf_detailedDescription": "An array of references to the video textures
that provide the attribute data",
      "items": {
        "$ref": "MPEG_primitive_V3C.attribute.schema.json"
      },
      "minItems": 0
    }
  }
}
```

```

    },
    "_MPEG_V3C_CAD": {
      "type" : "object",
      "description": "This object lists different properties described for the  
Common Atlas Data in ISO/IEC 23090-5." ,
      "gltf_detailedDescription" : "This object lists different properties  
described for the Common Atlas Data in ISO/IEC 23090-5.",
      "minItems":0
    },
    "extensions": {},
    "extras": {}
  },
  "required": [ "_MPEG_V3C_CONFIG", "_MPEG_V3C_AD", "_MPEG_V3C_GVD_MAPS" ]
}

```

A.4. JSON Schema for MPEG_primitive_V3C.attribute

```
{
  "$schema" : "http://json-schema.org/draft-07/schema",
  "title" : "MPEG_primitive_V3C.attribute",
  "type" : "object",
  "description": "defines the attribute of a V3C object.",
  "allOf": [ { "$ref": "glTFProperty.schema.json" } ],
  "properties" : {
    "type": {
      "type": "integer",
      "description": "provides the type of the attribute.",
      "gltf_detailedDescription": "provides the type of the attribute.",
      "minimum": 0,
      "maximum": 255
    },
    "maps": {
      "type": "array",
      "description": "",
      "gltf_detailedDescription": "provides the references to the corresponding
video texture maps.",
      "items": {
        "allOf": [ { "$ref": "glTFid.schema.json" } ]
      },
      "minItems": 1
    }
  },
  "required": ["maps"]
}
```

Appendix B: 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.