ISO/IEC JTC 1/SC 29/WG 3 N1438



**ISO/IEC JTC 1/SC 29/WG 3**

**MPEG Systems   
Convenorship: KATS (Korea, Republic of)**

**Document type:** Output Document

**Title: Procedures and Test Formats for Avatar Representation Format**

**Status:** Approved

**Date of document:** 2025-01-23

**Source:** ISO/IEC JTC 1/SC 29/WG 3

**Expected action:** ACT

**Action due date:** 2025-01-24

**No. of pages:** 14 (with cover page)

**Email of Convenor:** young.L@samsung.com

**Committee URL:** <https://isotc.iso.org/livelink/livelink/open/jtc1sc29wg3>

**INTERNATIONAL ORGANISATION FOR STANDARDISATION**

**ORGANISATION INTERNATIONALE DE NORMALISATION**

**ISO/IEC JTC 1/SC 29/WG 3**

**CODING OF MOVING PICTURES AND AUDIO**

**ISO/IEC JTC 1/SC 29/WG 3 N01438**

**Geneva, CH – January 2025**

|  |  |
| --- | --- |
| **Source** | **WG03 (MPEG Systems)** |
| **Title** | **Procedures and Test Formats for Avatar Representation Formats** |
| **Editor** | **João Regateiro (Interdigital)** |
| **MPEG number** | **24738** |

Contents

[1 Scope 4](#_Toc181914369)

[2 Time Plans and Projects 4](#_Toc181914370)

[3 Requirements, Scenarios and Test Assets 4](#_Toc181914371)

[3.1 Requirements 4](#_Toc181914372)

[3.1.1 Overview 4](#_Toc181914373)

[3.1.2 Phase 1 Requirements 5](#_Toc181914374)

[3.1.3 Phase 1 Requirements Priorities 6](#_Toc181914375)

[3.2 Test Scenarios 7](#_Toc181914376)

[3.2.1 Template for Test Scenario 7](#_Toc181914377)

[3.2.2 Call for Test Data 7](#_Toc181914378)

[3.2.3 Available Test Assets 8](#_Toc181914379)

[3.2.4 Test Scenario – Telepresence and Immersive Experiences 8](#_Toc181914380)

[3.3 Validation Procedure 9](#_Toc181914381)

[4 Candidates for Solutions 9](#_Toc181914382)

[4.1 Solution 1 for Reference Software 10](#_Toc181914383)

[4.2 Solution 2 for Reference Software 10](#_Toc181914384)

[4.3 Solution 2 for Data Model Representation 11](#_Toc181914385)

[4.4 Solution 2 for Animation Formats 12](#_Toc181914386)

[5 Market-relevant Representation Formats 12](#_Toc181914387)

[5.1 Metahuman DNA Format 12](#_Toc181914388)

[5.2 VRM Representation Format 13](#_Toc181914389)

[5.3 ARKit Face and Body Tracking 14](#_Toc181914390)

[5.4 Google’s MediaPipe 14](#_Toc181914391)

[6 Initial Considerations on Solutions 15](#_Toc181914392)

[6.1 Overview 15](#_Toc181914393)

[6.2 Representation Format 15](#_Toc181914394)

[6.2.1 Avatar JSON Interchange Format 15](#_Toc181914395)

[6.2.2 Landmarks Format 16](#_Toc181914396)

[7 Gitlab Management 16](#_Toc181914397)

[8 Coordinators for Efforts until MPEG#148 16](#_Toc181914398)

# Scope

This document provides information and processes in order to support the development of a MPEG avatar representation format as part of MPEG-I.

This part covers the following aspects:

* An interchange representation format.
* A CG-based avatar format and associated container.
* A representation of animation stream format.
* Signaling information for interoperability.
* Descriptive metadata.

The work is executed in the *AHG for MPEG-I Scene Description and Avatar Representation Formats*.

# Time Plans and Projects

The timeline for the development of the MPEG avatar representation format is divided into phases. This document only expresses phase 1 timeline and requirements (Figure 1). The remaining requirements and their priorities are differed to later phases.

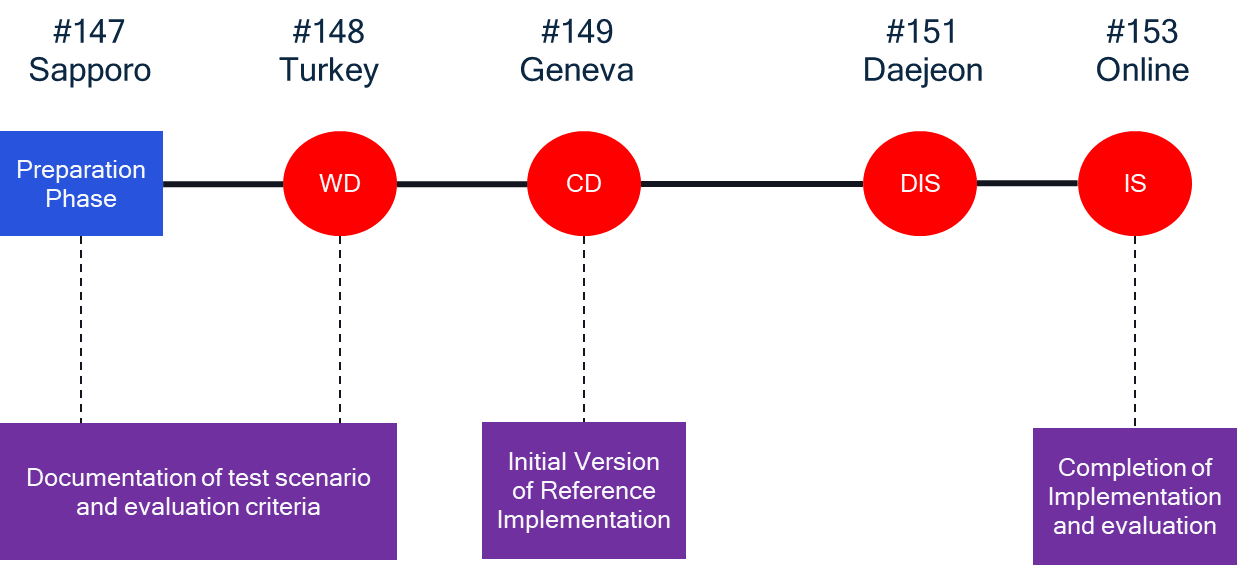


Figure 1 - Phase 1 timeline for MPEG avatar representation format.

# Requirements, Scenarios and Test Assets

# Requirements

# Overview

The work of the MPEG avatar representation format is based on the requirements defined in N00359.

The requirements defined in N00359 are currently organized into five different categories including: Avatar Representation, Coding format, Transport/Synchronisation/Carriage, Storage and Privacy and Security requirements.

The requirements related to the “Avatar Representation” are summarized as follows:

* Interoperability: allow converting to and from other avatar representation.
* 3D Scenes: to allow integration into scene description solution.
* Attributes: requirements applicable to the features supported by the avatar, such as, CGI features as textures, geometries, skeletons, accessories, semantic definitions.

The requirements related to the “Coding Format” are summarized as follows:

* General: requirements applicable to the whole format and its use
* Geometry: requirements applicable to the geometry of the base avatar
* Interaction: requirements that relate to the interaction between Avatars and between Avatars and other objects in the scene
* Animation and control: requirements on the facial and body animation

Transport/Synchronisation/Carriage: requirements related to the transport of the base avatar model and the associated animation streams

* Storage: requirements that pertain to the storage of the base avatar model
* Security and privacy: requirements on the protection of the avatar in different scenarios

A more detailed description of each of the above requirements can be found in N00359. The requirements have been pruned to allow the possibility of multiple working phases and at this stage the document defines the phase 1 requirements and defer the remaining requirements for future phases.

# Phase 1 Requirements

The Phase 1 requirements are selected from document N00359 based on the timeline in Figure 1, as follows:

* Avatar Representation:
* Interoperability: allow converting to and from other avatar representations. (requirement 194, 197, 198 and 201)
* 3D Scenes: allow integration into scene description solution. (requirement 200)
* Attributes: requirements applicable to the features supported by the avatar, such as CGI features as textures, geometries, skeletons, accessories, semantic definitions. (requirement 196.a and 199)
* Coding Format:
* General: requirements applicable to the whole format and its use
  + requirements 205.
* Geometry: requirements applicable to the geometry of the base avatar
  + Requirement 208, 210.
* Interaction: requirements that relate to the interaction between Avatars and between Avatars and other objects in the scene
  + Requirements 211 and 218.
* Animation and control: requirements on the facial and body animation
  + Requirements 219, 220, 221 and 223.
* Transport: requirements related to the transport of the base avatar model and the associated animation streams
  + Requirements 225, 227, 229, 230 and 234.
* Storage: requirements that pertain to the storage of the base avatar model
  + Requirements 235, 236, 237, 238, 239, 240 and 24.
* Security and privacy: requirements on the protection of the avatar in different scenarios
  + Requirements 234.

# Phase 1 Requirements Priorities

This section identifies the priority of the requirements by means of labelling the requirements with the following:

* Low – should be considered once the higher-level requirements are fulfilled or partially acceptable.
* Medium – should be considered once higher-level requirements partially acceptable.
* High – should be the first to be considered for the technical solution.

The following table identifies the requirements with the metrics above:

|  |  |  |  |
| --- | --- | --- | --- |
| **Category** | **Req #** | **Description** | **Priority** |
| **Avatar Representation** | **194** | A suitable exchange format for conversion between avatar representation formats in market | High |
|  | **196a** | Representation format includes description of body, skeletal, etc. | High |
|  | **198** | Representation format does not prevent DRM protection | Medium |
|  | **199** | Mesh-based format for representation and animation | High |
|  | **200** | Integration into scene description | Medium |
|  | **201** | Signal coding format | High |
| **General** | **205** | Semantic and signal representation   * Semantic - high level features * Signal - low level features | High |
| **Geometry** | **208** | Multiple levels of detail | High |
|  | **210** | Reference known 3D models | High |
| **Interaction** | **211** | Avatar-avatar, user-avatar, avatar-scene interactions | Low |
|  | **218** | Integrates with existing Interaction frameworks | Low |
| **Animation and control** | **219** | Facial and body animation | High |
|  | **220** | Storage and replay of animation streams | Low |
|  | **221** | Efficient animation through:   * External sensors * Pre-defined actions | * High * Low -> Medium |
|  | **223** | Animation through   * Pre-defined actions * External sensors | * Low -> Medium * High |
| **Transport** | **225** | Delay sensitive animation streams | High |
|  | **227** | Partial transport of base avatar | High |
|  | **229** | Uniquely identifiable avatar representation and animation streams | High |
|  | **230** | MPEG format as fallback | High |
|  | **234** | Reuse of animation streams for different formats (potentially with loss) | High |
| **Storage** | **235** | Base avatar stored in single file | High |
| **236** | Partial access | High |
| **237** | Independent component compression | High |
| **238** | Independent component encryption | High |
| **239** | Different levels of detail | High |
| **240** | Entry point to describe components and relationships | High |
| **241** | Identify format for interoperability | High |
| **Privacy/Security** | **243** | Protection of authenticity and association with user | Medium |

# Test Scenarios

# Template for Test Scenario

The following table should be used to propose test scenarios for the MPEG avatar representation format:

|  |  |
| --- | --- |
| **Item** | **Description** |
| **Title** | <give it a catchy title, e.g., as those listed in clause 2> |
| **Description** | * What is the basic use case? * How does it relate to MPEG-I Requirements [1] and Avatar Use Cases [2]? |
| **Required test assets** | * Avatar asset, scene description, real-time assets for media (2D/3D) * Anything else * References to test assets |
| **Current Support** | * How can the test scenario be supported today. |
| **Potential Gaps** | * What are gaps and potential improvements this scenario? |
| **Criteria** | * What are relevant criteria for the user experience/QoE? * What are relevant criteria for passing the test scenario? |

# Call for Test Data

Among others, we solicit the following material to be used as content for the creation and validation of the MPEG avatar representation format:

* 3D avatar assets that serve as content for evaluation and testing of the MPEG avatar representation format.
* 3D scene that can support avatar content

# Available Test Assets

The following table lists the available assets and provides a brief description:

|  |  |
| --- | --- |
| **Asset** | **Description** |
| MPEG\_Morgan.zip | A glTF asset that represents an avatar model, including a face model, a semantically define body parts mode, and a full body without semantical partitioning. |
| Lionel\_base\_avatar.zip | An asset that represents the base avatar model using the reference avatar model (“MPEG\_Morgan.zip”). |
| Unity.scene.zip | Unity scene description asset. |
| Unreal.scene.zip | Unreal scene description asset |
| Animation.zip | Pre-recorded animation parameters of blendshape weights to be applied to the base avatar (“Lionel\_base\_avatar.zip”). |

Scenes are preferred to be in glTF 2.0.

# Test Scenario – Telepresence and Immersive Experiences

A scene is used to describe the environment, avatars can be employed to represent people (users) interacting in virtual 3D environments.

A user can use traditional video feeds through webcam technology to drive the animation of an avatar by capturing facial expressions of the user and transfer them to animate the avatar.

In Figure 2 each captured frame of the user's head (Figure 2 (a)) is analyzed using a model that predicts blendshape (or morph target) weights (Figure 2 (b)). This model can directly predict the weights or can detect landmarks, which are then used to infer blendshape weights. The weights are finally sent to the 3D virtual world environment and used to animate the user avatar (Figure 2 (c)).

|  |  |
| --- | --- |
| **Item** | **Description** |
| **Title** | Telepresence and Immersive Experiences |
| **Description** | * Represent users in 3D scenes with avatar representations. * This test should cover the following requirement from MPEG-I Requirements:   + 196.a, 200, 205, 210, 211, 219, 221, 223, 225, 226, 230, 235, 236, 239 and 241. |
| **Required test assets** | * Unreal.scene.zip, Unity.scene.zip, Animation.zip and Lionel\_base\_avatar.zip assets |
| **Current Support** | * Unity scene integrating the SD avatar representation. * Animation based on glTF 2.0 morph targets. * Dedicated format for animation streaming. * An interactivity framework that supports webcam user input. |
| **Potential Gaps** | * Format for streaming of features to provide real-time facial expression. (Requirements in 225, 226 and 230) |
| **Criteria** | * What are relevant criteria for the user experience/QoE?   + Synchronization between user input and avatar should be accurate and realistic. * What are relevant criteria for passing the test scenario?   + User can drive an avatar through facial expression and others can visualize his avatar mimicking is facial deformations. |

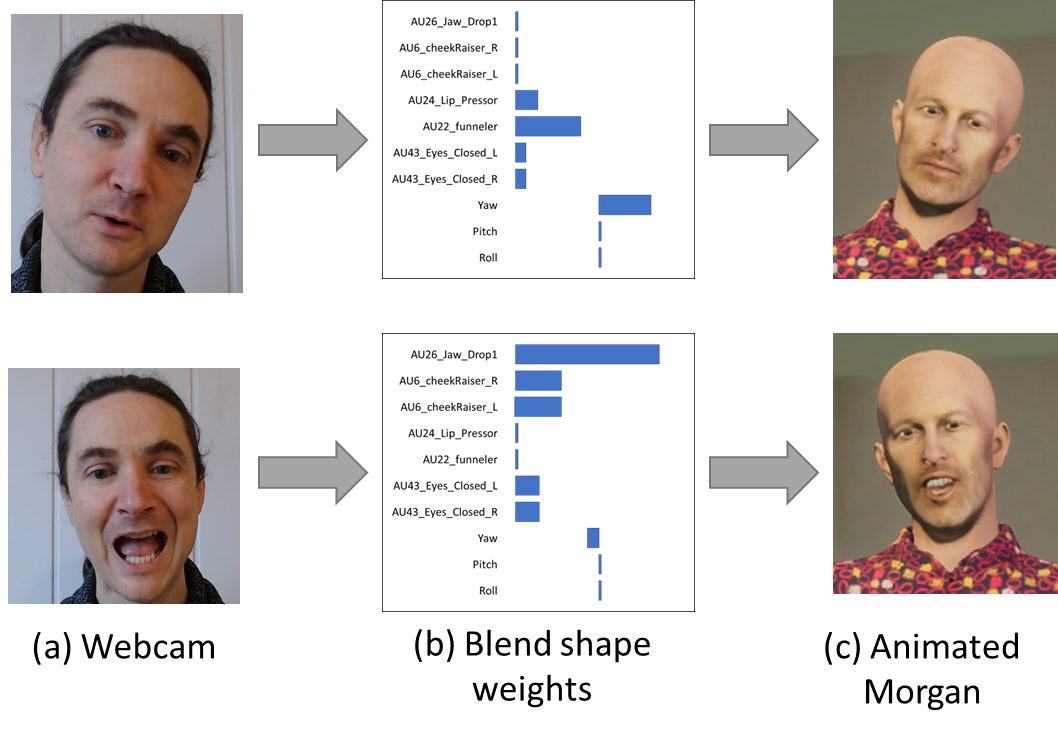


Figure 2 - Interactive application for immersive realities.

# Validation Procedure

A test scenario is composed of the following pieces:

* A description of the test scenario.
* A description of the test criteria: test criteria determine if a test scenario has been successfully addressed by the avatar format solution or not.
* A collection of assets that will be used for the test scenario.
* Cross check integration with MPEG-I SD technology.

# Candidates for Solutions

**[Note: the work in progress for the different solution and draft of the specification can be found in the document:** [**779**](https://git.mpeg.expert/MPEG/Systems/SceneDescription/MPEG-Contributions/-/issues/779)**]**

# Solution 1 for Reference Software

Avatar animation is typically performed as part of rendering a 3D scene, where the user(s) is/are represented by their avatar(s). The assumption is that the animation process will generate a dynamic mesh that is then rendered as a node of the scene graph. Optimizations may combine animation and rendering to directly produce image renderings of the user’s avatar based on the current camera pose.

The straightforward approach to the reference implementation would be to develop a dedicated pipeline for the animation and reconstruction of the avatar. This approach is depicted in the following diagram:

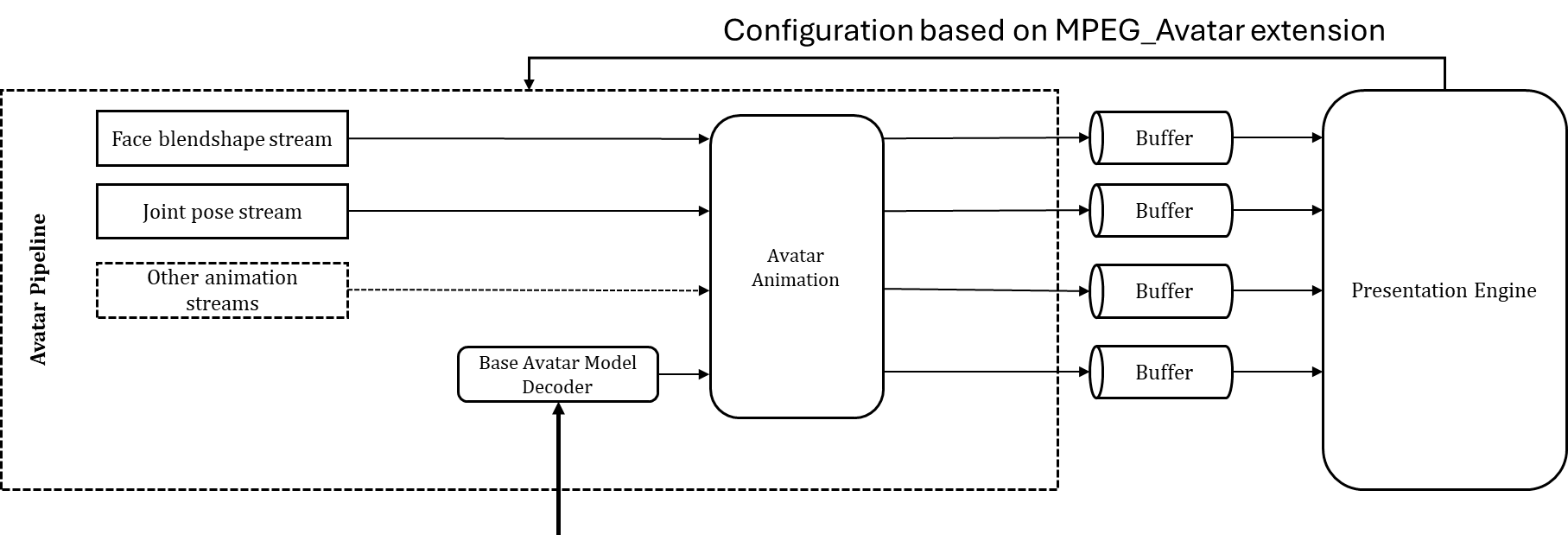


Figure 3 - Reference software candidate for MPEG avatar representation format.

The MARF proposal as described in this document has been partially implemented and made available as a new branch (**qcom-marf**) under the mpegsd reference software. The implementation is fully compatible with the OpenXR tracking framework. Scripts are included to capture through OpenXR (e.g. using a Quest Pro HMD) the facial and body expressions and use them for the animation.

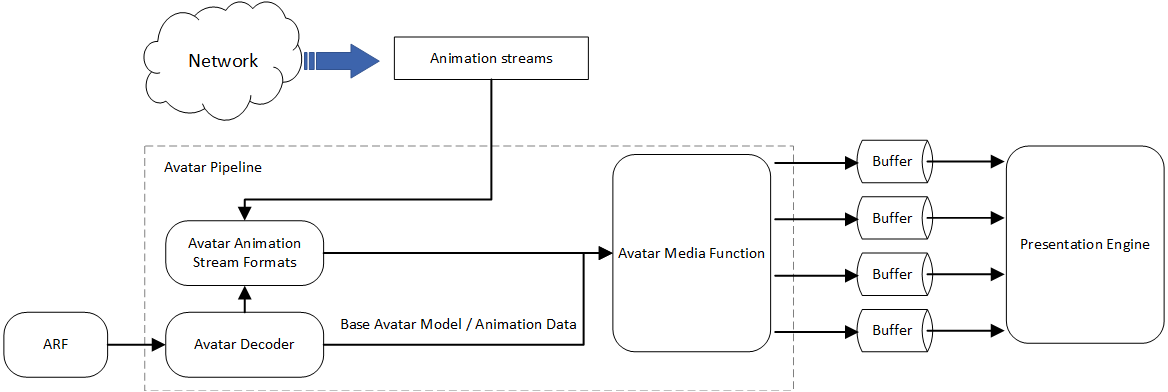
The reconstruction is performed in an Avatar pipeline that receives the animation streams via a WebSockets connection  and accesses the MARF container locally and then reconstructs and animates the base avatar model and passes it through buffers as a dynamic mesh to the presentation engine for rendering.

This is work in progress and it will be maintained in accordance with the agreements made on MARF.

# Solution 2 for Reference Software

The AJIF is a dedicated avatar format that is disentangled from any scene or intricate file formats e.g., the contents and information in the file only refers to the description of a digital human (avatar).

Therefore, a new ecosystem and avatar related functionality need to be developed. The following figure presents a candidate architecture for the avatar pipeline.



The avatar pipeline is expected to receive “.ajif” files containing the avatar representation.

The pipeline is initiated by the “Avatar Decoder” which parses and decodes the AJIF file. Once the data is decoded/parsed is divided into three main processes.

* Avatar Animation Stream Formats – Contains the expected format of animation data. This animation is expected to be timed data that is received from external sources e.g., face controllers’ stream, joint transforms stream and/or other animation streams. The data received from external sources should agree with the format extracted from the avatar representation.
* Base Avatar Model – Contains the assets and metadata relative to the “static” representation of the avatar e.g., mesh, topology, textures, metadata and others.
* Avatar Animation Data – Contains pre-defined/pre-recorded animations of the base avatar model, this can be in the form of skeletal animations, morph targets/blendshapes and others.

Lastly, the “Avatar Media Function” has the objective to encapsulate or process all the information provided by the previous functions and prepare the necessary structure to deliver the avatar data to the application (“Presentation Engine”).

This is work in progress and it will be maintained and converged to a single solution in accordance with the agreements made between the solutions.

# Solution 2 for Data Model Representation

This section introduces the general concept of the data model for the avatar representation format, as illustrated in the figure below.

**A black background with a blue line

Description automatically generated**

ARF Document

metadata

components

animations

preamble

other

joint\_animation\_sample

skeletons

skins

structure

blendshapes

This is work in progress and it will be maintained and converged to a single solution in accordance with the agreements made between the solutions.

# Solution 2 for Animation Formats

This is work in progress and it will be maintained and converged to a single solution in accordance with the agreements made between the solutions.

# Solution 2 for Reference Software Description

This section describes the branch of the reference software, where it is provided an initial implementation of the Avatar Representation format. The code is available under the mpeg-arf branch of the pympegsd reference software.

# Pyrenderer Python 3.11 and PyOpenGL Updates

The Pyrender library has undergone significant updates to ensure full compatibility with Python 3.11 while maintaining backward compatibility with previous Python versions. The build system has been modernized through the adoption of the pyproject.toml configuration format, replacing the traditional setup.py approach.

To install Pyrender, you will need to use the fork that includes these updates as follows:

pip install git+https://github.com/i.bouazizi/pyrender.git

The project’s dependency management has been restructured to ensure compatibility across Python versions. The core dependencies are specified in the pyproject.toml file:

dependencies = [  
 "freetype-py",  
 "imageio",  
 "networkx",  
 "numpy",  
 "Pillow",  
 "pyglet>=1.4.10",  
 "PyOpenGL~=3.1.7",  
 "scipy",  
 "trimesh",  
]

The integration between Pyrender and PyOpenGL has been enhanced to provide better stability and performance across different platforms. The PyOpenGL dependency has been specifically fixed to version 3.1.7 to ensure consistent behavior and compatibility with modern OpenGL features. This version requirement is enforced through the dependency specification in pyproject.toml.

The graphics pipeline for PBR rendering has also been updated to improve compatibility with modern OpenGL features. The shader code now supports newer GLSL versions and provides better error handling.

# Avatar Functionality in Scene Description

# ARF (Avatar Representation Format) Implementation

The avatar functionality is built based on the WD of the MPEG Avatar Representation Format (ARF) specification, which provides a standardized container format for avatar assets and animations. The ARF implementation follows the MPEG ARF specification and integrates with the OpenXR facial animation tracking system. Here’s a detailed look at the ARF structure and implementation:

from pympegsd.arf import ARFContainer, ARFLoader  
from pympegsd.scene import Scene, Node  
import json  
  
class AvatarSystem:  
 def \_\_init\_\_(self):  
 self.arf\_loader = ARFLoader()  
   
 def load\_arf\_avatar(self, arf\_path: str) -> Scene:  
 # Load and parse ARF container  
 arf\_container = self.arf\_loader.load(arf\_path)  
   
 # Validate ARF signature and version  
 if not arf\_container.validate\_signature("MPEG-ARF-2025"):  
 raise ValueError("Invalid ARF signature")  
   
 # Create scene from ARF data  
 scene = Scene()  
   
 # Process ARF components  
 self.\_process\_blendshapes(arf\_container, scene)  
 self.\_process\_nodes(arf\_container, scene)  
   
 return scene  
   
 def \_process\_blendshapes(self, arf: ARFContainer, scene: Scene):  
 # Load blendshape definitions  
 for blendshape\_set in arf.components.blendshapes:  
 # Load basis mesh  
 basis\_mesh = self.\_load\_gltf(  
 arf.get\_data\_uri(blendshape\_set.basisMesh)  
 )  
   
 # Load and register blendshapes  
 for shape\_id in blendshape\_set.shapes:  
 shape\_data = arf.get\_data(shape\_id)  
 self.\_register\_blendshape(  
 scene,  
 basis\_mesh,  
 shape\_data.name,  
 shape\_data.uri  
 )

The ARF container structure is defined in the arf.json configuration file, which specifies the avatar’s components, assets, and animation capabilities:

{  
 "preamble": {  
 "signature": "MPEG-ARF-2025",  
 "version": "1.0.0",  
 "supportedAnimation": {  
 "faceAnimation": ["urn:khronos:openxr:facial-animation:fb-tracking2"]  
 }  
 },  
 "metadata": {  
 "name": "Alexis",  
 "id": "alexis-avatar"  
 },  
 "structure": {  
 "assets": [  
 {  
 "name": "head",  
 "lods": [  
 {  
 "name": "medium\_quality",  
 "blendshapes": [1]  
 }  
 ]  
 }  
 ]  
 }  
}

# MPEG\_node\_avatar Extension Integration

The MPEG\_node\_avatar GLTF extension is implemented to provide advanced avatar animation capabilities. This extension enables the integration of facial expressions, body animations, and real-time tracking data. This implementation provides a pipeline for avatar animation, starting by loading ARF containers to real-time tracking updates through the MPEG\_node\_avatar extension and the usage of WebSockets for carriage of animation samples. The integration between ARF and MPEG\_node\_avatar needs to be updated to follow the developments in the MPEG ARF specification.

# Tracking and Capture Implementation

# Face Tracking

The face tracking implementation leverages the Meta Quest 3’s facial tracking system through the OpenXR API. The system captures and processes facial expressions in real-time, providing detailed blendshape coefficients that can be used to drive avatar animations. The implementation is built around the XrTracking class, which manages the OpenXR face tracking lifecycle and data processing.

Here’s a complete example of initializing and using the face tracking system:

import xr  
import numpy as np  
import pickle  
from ctypes import cast, byref  
  
class FaceTrackingSystem:  
 def \_\_init\_\_(self):  
 # Initialize OpenXR instance with face tracking extension  
 self.instance = xr.Instance(  
 enabled\_extensions=[  
 xr.KHR\_OPENGL\_ENABLE\_EXTENSION\_NAME,  
 xr.FB\_FACE\_TRACKING2\_EXTENSION\_NAME  
 ]  
 )  
   
 # Initialize face tracking functions  
 self.face\_tracker = xr.FaceTracker2FB()  
 self.init\_face\_tracking\_functions()  
   
 def init\_face\_tracking\_functions(self):  
 # Get function pointers for face tracking operations  
 self.create\_tracker = cast(  
 xr.get\_instance\_proc\_addr(  
 self.instance,  
 "xrCreateFaceTracker2FB"  
 ),  
 xr.PFN\_xrCreateFaceTracker2FB  
 )  
   
 self.get\_expressions = cast(  
 xr.get\_instance\_proc\_addr(  
 self.instance,  
 "xrGetFaceExpressionWeights2FB"  
 ),  
 xr.PFN\_xrGetFaceExpressionWeights2FB  
 )  
   
 def start\_tracking(self, session):  
 # Configure and create the face tracker  
 create\_info = xr.FaceTrackerCreateInfo2FB(  
 face\_expression\_set=xr.FaceExpressionSet2FB(),  
 requested\_data\_source\_count=1,  
 requested\_data\_sources=[xr.FaceTrackingDataSource2FB.VISUAL]  
 )  
 self.create\_tracker(session, byref(create\_info), self.face\_tracker)  
   
 def get\_expression\_weights(self, frame\_state):  
 # Prepare buffers for expression data  
 weights = [0.0] \* xr.FaceExpression2FB.COUNT  
 confidences = [0.0] \* xr.FaceConfidence2FB.COUNT  
   
 # Create expression weights structure  
 expression\_weights = xr.FaceExpressionWeights2FB(  
 weights=weights,  
 confidences=confidences,  
 weight\_count=xr.FaceExpression2FB.COUNT,  
 confidence\_count=xr.FaceConfidence2FB.COUNT  
 )  
   
 # Get current expression data  
 expression\_info = xr.FaceExpressionInfo2FB(  
 time=frame\_state.predicted\_display\_time  
 )  
 self.get\_expressions(  
 self.face\_tracker,  
 byref(expression\_info),  
 byref(expression\_weights)  
 )  
   
 return expression\_weights  
  
 def save\_expressions(self, weights, filename='expressions.pkl'):  
 with open(filename, 'wb') as f:  
 pickle.dump(weights, f)

The face tracking data is processed at approximately 25 frames per second and includes 70 different blendshape coefficients that correspond to various facial expressions. The system also provides confidence metrics for each expression, allowing applications to filter out unreliable data.

# Body Tracking

The body tracking implementation provides comprehensive full-body pose estimation through the OpenXR body tracking extension. The system tracks 17 key joints in the upper body, providing both position and orientation data in real-time. Here’s a detailed implementation example:

import xr  
import numpy as np  
import json  
from ctypes import cast, byref  
from typing import List, Dict  
  
class BodyTrackingSystem:  
 def \_\_init\_\_(self):  
 self.body\_tracker = xr.BodyTrackerFB()  
 self.joint\_locations: List[xr.BodyJointLocationFB] = [  
 xr.BodyJointLocationFB()  
 ] \* xr.BodyJointFB.COUNT  
 self.skeleton\_joints: List[xr.BodySkeletonJointFB] = [  
 xr.BodySkeletonJointFB()  
 ] \* xr.BodyJointFB.COUNT  
   
 # Initialize tracking space  
 self.tracking\_space = self.create\_tracking\_space()  
   
 def create\_tracking\_space(self):  
 # Create a reference space for body tracking  
 space\_info = xr.ReferenceSpaceCreateInfo(  
 reference\_space\_type=xr.ReferenceSpaceType.STAGE,  
 pose\_in\_reference\_space=xr.Posef(  
 orientation=xr.Quaternionf(0, 0, 0, 1),  
 position=xr.Vector3f(0, 0, 0)  
 )  
 )  
 return xr.create\_reference\_space(space\_info)  
   
 def update\_tracking(self, frame\_state):  
 # Prepare joint location structure  
 locations = xr.BodyJointLocationsFB(  
 joint\_count=xr.BodyJointFB.COUNT,  
 joint\_locations=self.joint\_locations  
 )  
   
 # Get current joint locations  
 locate\_info = xr.BodyJointsLocateInfoFB(  
 base\_space=self.tracking\_space,  
 time=frame\_state.predicted\_display\_time  
 )  
   
 self.locate\_joints(locate\_info, locations)  
 return self.process\_joint\_data(locations)  
   
 def process\_joint\_data(self, locations) -> Dict:  
 joint\_data = {}  
 for i in range(xr.BodyJointFB.COUNT):  
 joint = locations.joint\_locations[i]  
 joint\_data[f"joint\_{i}"] = {  
 "position": [  
 joint.pose.position.x,  
 joint.pose.position.y,  
 joint.pose.position.z  
 ],  
 "orientation": [  
 joint.pose.orientation.x,  
 joint.pose.orientation.y,  
 joint.pose.orientation.z,  
 joint.pose.orientation.w  
 ],  
 "confidence": joint.confidence  
 }  
 return joint\_data  
   
 def save\_tracking\_data(self, data, filename='body\_tracking.json'):  
 with open(filename, 'w') as f:  
 json.dump(data, f, indent=2)

The body tracking system maintains a hierarchical skeleton structure where each joint’s position and orientation are defined relative to its parent joint. This allows for accurate representation of body movements and poses.

# Hand Tracking

The hand tracking implementation provides detailed finger-level tracking and gesture recognition capabilities. The system tracks individual finger joints and can recognize various hand poses and gestures. Here’s an implementation example:

import xr  
from ctypes import cast, byref  
from typing import Dict, List  
  
class HandTrackingSystem:  
 def \_\_init\_\_(self):  
 self.hand\_trackers = {  
 'left': xr.HandTrackerEXT(),  
 'right': xr.HandTrackerEXT()  
 }  
 self.joint\_locations = {  
 'left': [xr.HandJointLocationEXT() \* xr.HandJointEXT.COUNT],  
 'right': [xr.HandJointLocationEXT() \* xr.HandJointEXT.COUNT]  
 }  
   
 self.init\_hand\_tracking()  
   
 def init\_hand\_tracking(self):  
 # Initialize hand tracking for both hands  
 for hand in ['left', 'right']:  
 create\_info = xr.HandTrackerCreateInfoEXT(  
 hand=xr.HandEXT.LEFT if hand == 'left' else xr.HandEXT.RIGHT,  
 hand\_joint\_set=xr.HandJointSetEXT.DEFAULT  
 )  
 xr.create\_hand\_tracker(create\_info, self.hand\_trackers[hand])  
   
 def update\_hand\_tracking(self, frame\_state) -> Dict[str, List]:  
 hand\_data = {}  
 for hand in ['left', 'right']:  
 # Get joint locations  
 locations = xr.HandJointLocationsEXT(  
 joint\_count=xr.HandJointEXT.COUNT,  
 joint\_locations=self.joint\_locations[hand]  
 )  
   
 # Locate joints  
 locate\_info = xr.HandJointLocateInfoEXT(  
 base\_space=self.reference\_space,  
 time=frame\_state.predicted\_display\_time  
 )  
   
 self.hand\_trackers[hand].locate\_joints(locate\_info, locations)  
 hand\_data[hand] = self.process\_hand\_joints(locations)  
   
 return hand\_data  
   
 def process\_hand\_joints(self, locations) -> List[Dict]:  
 joints = []  
 for i in range(xr.HandJointEXT.COUNT):  
 joint = locations.joint\_locations[i]  
 joints.append({  
 'position': [  
 joint.pose.position.x,  
 joint.pose.position.y,  
 joint.pose.position.z  
 ],  
 'orientation': [  
 joint.pose.orientation.x,  
 joint.pose.orientation.y,  
 joint.pose.orientation.z,  
 joint.pose.orientation.w  
 ],  
 'radius': joint.radius,  
 'confidence': joint.confidence  
 })  
 return joints

# Eye Tracking

The eye tracking system provides tracking of eye movements, gaze direction, and blink detection. The implementation uses the OpenXR eye tracking extension to capture detailed eye movement data. Here’s a comprehensive example:

import xr  
from ctypes import cast, byref  
import numpy as np  
from typing import Dict, Tuple  
  
class EyeTrackingSystem:  
 def \_\_init\_\_(self):  
 self.eye\_tracker = xr.EyeTrackerFB()  
 self.init\_eye\_tracking()  
   
 def init\_eye\_tracking(self):  
 # Initialize eye tracking functions  
 self.create\_tracker = cast(  
 xr.get\_instance\_proc\_addr(  
 self.instance,  
 "xrCreateEyeTrackerFB"  
 ),  
 xr.PFN\_xrCreateEyeTrackerFB  
 )  
   
 # Create eye tracker  
 create\_info = xr.EyeTrackerCreateInfoFB()  
 self.create\_tracker(self.session, byref(create\_info), self.eye\_tracker)  
   
 def get\_eye\_data(self, frame\_state) -> Dict:  
 # Prepare eye data structure  
 eye\_data = xr.EyeGazesFB()  
   
 # Get current eye tracking data  
 gaze\_info = xr.EyeGazesInfoFB(  
 base\_space=self.reference\_space,  
 time=frame\_state.predicted\_display\_time  
 )  
   
 self.eye\_tracker.get\_gazes(gaze\_info, eye\_data)  
   
 return {  
 'left\_eye': self.process\_eye\_data(eye\_data.left),  
 'right\_eye': self.process\_eye\_data(eye\_data.right),  
 'combined\_gaze': self.calculate\_combined\_gaze(eye\_data)  
 }  
   
 def process\_eye\_data(self, eye\_data) -> Dict:  
 return {  
 'gaze\_direction': [  
 eye\_data.gaze.direction.x,  
 eye\_data.gaze.direction.y,  
 eye\_data.gaze.direction.z  
 ],  
 'pupil\_diameter': eye\_data.pupil\_diameter,  
 'confidence': eye\_data.confidence  
 }  
   
 def calculate\_combined\_gaze(self, eye\_data) -> Dict:  
 # Calculate combined gaze direction from both eyes  
 left\_dir = np.array([  
 eye\_data.left.gaze.direction.x,  
 eye\_data.left.gaze.direction.y,  
 eye\_data.left.gaze.direction.z  
 ])  
 right\_dir = np.array([  
 eye\_data.right.gaze.direction.x,  
 eye\_data.right.gaze.direction.y,  
 eye\_data.right.gaze.direction.z  
 ])  
   
 # Weight by confidence  
 combined\_dir = (  
 left\_dir \* eye\_data.left.confidence +  
 right\_dir \* eye\_data.right.confidence  
 ) / (eye\_data.left.confidence + eye\_data.right.confidence)  
   
 # Normalize  
 combined\_dir = combined\_dir / np.linalg.norm(combined\_dir)  
   
 return {  
 'direction': combined\_dir.tolist(),  
 'confidence': (  
 eye\_data.left.confidence +  
 eye\_data.right.confidence  
 ) / 2.0  
 }

The implementation leverages Meta Quest 3’s advanced tracking capabilities.

Each tracking system (face, body, hands, and eyes) operates independently but can be easily integrated into a unified tracking solution. The system provides high-frequency updates (typically 25-90 Hz depending on the tracking type) and includes confidence metrics for all tracked features, allowing applications to filter and smooth the data as needed.

The tracking data can be used directly to drive avatar animations or can be recorded for later playback and analysis. The scripts support both real-time streaming and batch processing of tracking data, making it suitable for the development of the ARF standard as well as for live avatar demos.

# Market-relevant Representation Formats

# Human Digital Representations Formats

# Metahuman DNA Format

The DNA format is a file format that is used to store all the details of the shape and rig of the Metahuman Avatar head. DNA format currently does not support the storage of the complete Avatar.

DNA file format comes in both a JSON representation and a binary representation and there is a tool to convert between both.

The structure of the DNA file can be analyzed by inspecting some of the sample DNA files. The following is the rough hierarchy of the file:

* Signature: an identifying signature of the file
* Version: contains version information
* Sections: lists the included sections in the file
* Descriptor: provides some metadata about the file, such as gender and age. It also indicates the number of levels of detail stored in the file
* Definition: includes the following information:
  + mappings between assets (joints, blendshapes, animated maps, meshes) and the levels of detail
  + control information describing the supported controls.
  + joint and mesh names that describe the head and facial joints that can be controlled.
  + joint mappings and hierarchy as well as their transforms to the neutral T-pose
* Behavior: provides definitions of the Avatar behavior, including:
  + Controls: that assign transform paths for the control elements upon an animation
  + Joints: provides transforms for the corresponding joint groups, identified by their joint indices
  + Blendshape channels: provides transforms for blendshapes during a specific behavior
  + Animated maps: provides transforms for the references maps during a specific behavior
* Geometry: contains all mesh descriptions for the Avatar, where each mesh may have:
  + Positions: provides the coordinates of all vertices
  + Texture coordinates: provides the UV coordinates for the textures
  + Normals: provides the vertex normal
  + Faces: includes the face definitions for the mesh
  + Skin weights: provides the weight associated with each specific joint, which are used to control the skinning
  + Blendshape Targets: links the mesh to the corresponding blendshapes
* End of file: a terminating element to indicate the end of the file

# VRM Representation Format

The VRM specification [1] is a platform-independent file format designed for use with 3D characters and avatars in the modern VR landscape. It is an open-source file format specifically designed for humanoid character avatars. The VRM specification is based on glTF 2.0, a cross-platform format that has been gaining a lot of traction in recent years.

The VRM specification provides a unified way to store and exchange avatar data, including the 3D model, textures, skeletal rigging, facial expressions, and even additional metadata information, such as licensing and copyright. By integrating into glTF 2.0, VRM aims to make 3D avatars more usable on all platforms, such as distribution and games.

The VRM specification is an extension of the glTF 2.0 format, designed specifically for humanoid avatars. The following list describes the VRM avatar representation and components:

* VRM Humanoid Structure: The VRM specification defines a humanoid bone structure as a glTF extension. This structure allows for the standardization of bone names and locations, making it easier for applications to control avatars in a uniform manner.
* VRM Emotion: To support anime-like styles, expressions are defined as changes of blend shapes (morph) and materials with respect to vowels and preset expressions (angry, fun, etc.). These preset patterns can be extended.
* VRM SpringBone: This is a simple secondary animation system intended to be used for non-realistic hairs and clothes. It prioritizes simplicity to ensure portability rather than being feature-rich.
* VRM MToon: This is an anime look material definition provided as part of the default implementation. It has also been ported to WebGL for deployments in Web environments.
* VRM License: The EULA/License of distributing avatars can be defined and included. This is important to protect user’s appearance in the virtual world.

The VRM specification defines a “model space” for each VRM model, that observes relative transforms from the root of the glTF scene. This model space is distinct from the world space defined in the application that uses the VRM model.

VRM Animation is a format for describing animations of humanoid models defined in VRM. The same VRM animation file can be used for any VRM Avatar file. The format is described in glTF, and is a cross-platform format. A standard implementation for importing and exporting VRM animations in Unity is provided through UniVRM.

The following is an example VRM file, which was simply renamed to the binary glTF 2.0 file extension “glb”. It can then be rendered easily but without the VRM features.

# ARKit Face and Body Tracking

# ARKit Face and Body Tracking

ARKit is Apple's augmented reality (AR) development platform for iOS devices, offering robust tools for AR experiences. Two key features of ARKit are Face Tracking and Body Tracking.

ARKit Face Tracking uses the TrueDepth camera system available on devices like the iPhone and later models. It provides detailed tracking of facial expressions and movements, making it ideal for applications like facial recognition, Animoji, and AR filters.

ARKit face tracking has the following features:

* Tracks over 50 facial muscles in real time
* Provides blend shape coefficients for facial expressions
* Detects eye gaze and face topology
* Works with both front and back cameras

ARKit Body Tracking provides real-time tracking of the human body using the rear camera. It can recognize and track up to two bodies at a time, identifying 19 body joints. This feature is useful for applications such as fitness tracking, interactive gaming, and more.

ARKit body tracking has the following features:

* Tracks 19 body joints
* Recognizes multiple bodies simultaneously
* Provides real-time skeletal tracking
* Integrates seamlessly with ARKit's AR capabilities

# Google’s MediaPipe

MediaPipe is an open-source framework from Google designed for building multimodal applied machine learning pipelines. It is highly optimized for performance and supports various platforms like Android, iOS, web, and desktop. Two of its popular packages are Pose and Face Tracking.

MediaPipe Pose is a machine learning solution that tracks the human body by detecting 33 pose landmarks on the whole body (upper and lower). It uses a holistic approach, combining body, hands, and face landmarks to achieve accurate pose estimation.

MediaPipe pose has the following features:

* Real-time pose estimation
* Tracks 33 pose landmarks
* Works on various platforms

MediaPipe Face Tracking is a solution that provides real-time face detection and tracking. It can detect 468 face landmarks, which makes it useful for various applications like face mesh, augmented reality, and more.

MediaPipe face tracking has the following features:

* Real-time face detection and tracking
* Detects 468 face landmarks
* Suitable for AR, face filters, and more

# Initial Considerations on Solutions

# Overview

This section provides the principal agreements for the MPEG avatar representation format:

Principle Agreements

* start with an abstract data model that describes the relations
* Agree to use JSON for data type representation and expressing data model (hierarchies, dependencies)
* We expect to have a JSON document that will include the base avatar representation
* Further mappings are for future study

# Representation Format

# Avatar JSON Interchange Format

This section provides a high-level description of what the JSON avatar format should contain as information related to an avatar and serve as a baseline for the continuation of the work development.

The representation can contain the following data:

* High level avatar information:
  + Name
  + Identifier
  + Age
  + Gender
* Representation data:
  + reference template
  + mesh: one or more geometries (LoDs).
  + body mappings: a list of string with associated geometries to identify the body parts
  + skeletons: one or more skeleton skins
  + controllers: one or more animation controllers (inc. facial blendshapes, joints, morphtarget…)
* Additional information:
  + Model mappings: one or more mapping function (reference template to B)
  + Animations: a set of predefined animations
  + Processing functions
  + Avatar gaze
  + Parametric textures
  + Hair
  + Accessories

# Landmarks Format

Facial landmarks are facial key points on a human face that have specific semantic positions, marking locations such as the corners of the eyes, nose, lips, and other features. As specified in MPEG-I Amd2/Annex H, the number of facial key points is usually 68 (see Figure 4). Additionally, note that the Facial Landmarks section of 3GPP SA4 points out that besides the set of 68 facial landmarks provided in MORGAN may also be used for different levels of implementation complexities.



Figure 4 - MPEG Morgan's landmarks.

The document m68848 provides an initial format to address the landmarks representation, although its definition is focused on the implementation in MPEG-I Scene Description and needs further revision. The landmarks are currently under investigation in the EE3: Other animation sample formats for Animation streams.

# Exploratory Experiments Study

The work on the avatar representation format as resulted in five exploratory experiments (EE), described as follows:

* **EE-1: Compression**: compression for animation streams, and provide evaluations , formats, algorithms and metrics.
* **EE-2: GeoRep** : study on geometric representations to investigate the need for additional avatar data components represented in the format.
* **EE-3: Additional Animation**: study on animation samples (controllers, landmarks etc.) additional to blendshape and joint animation.
* **EE-4: Content Discovery and Partial Access**: study aims to evaluate solutions to address the issue of content discovery and partial access.
* **EE-5: Animation Controllers**: study on animation controllers to allow the combination of both blendshape and joint animation.

# Gitlab Management

Two new projects were created under the SD git.

* ARF reference software was created as a branch of MPEG-SD reference software [[mpeg-arf](https://git.mpeg.expert/MPEG/Systems/SceneDescription/software/reference/-/tree/mpeg-arf?ref_type=heads)]
* ARF evaluation platform [[git](https://git.mpeg.expert/MPEG/Systems/SceneDescription/software/avatar-evaluation)]

# Coordinators for Efforts until MPEG#149

* Editors:
  + João Regateiro ([joao.regateiro@interdigital.com](mailto:joao.regateiro@interdigital.com) )
  + Imed Bouazizi ([bouazizi@qti.qualcomm.com](mailto:bouazizi@qti.qualcomm.com))
  + Anthony Trioux ([anthony\_trioux@xidian.edu.cn](mailto:anthony_trioux@xidian.edu.cn) )