**ISO 23090-29:2024(E)**

ISO/IEC JTC 1/SC 29/WG 7

Date: 2024-06-03

**Information technology — Coded representation of immersive media — Part 29: Video-based dynamic mesh coding (V-DMC)**

CD stage

**Warning for WDs and CDs**

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

*A model manuscript of a draft International Standard (known as “The Rice Model”) is available at* [*https://www.iso.org/iso/model\_document-rice\_model.pdf*](https://www.iso.org/iso/model_document-rice_model.pdf)

© ISO/IEC 2024

All rights reserved. Unless otherwise specified, or required in the context of its implementation, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO’s member body in the country of the requester.

ISO copyright office

CP 401 • Ch. de Blandonnet 8

CH-1214 Vernier, Geneva

Phone: +41 22 749 01 11

Email: copyright@iso.org

Website: www.iso.org

Published in Switzerland

Contents

*This template allows you to work with default MS Word functions and styles. You can use these if you want to maintain the Table of Contents automatically and apply auto-numbering.*

*To update the Table of Contents please select it and press "F9".*

[Foreword viii](#_Toc168308368)

[Introduction ix](#_Toc168308369)

[1 Scope 1](#_Toc168308370)

[2 Normative references 1](#_Toc168308371)

[3 Terms and definitions 1](#_Toc168308372)

[4 Abbreviated terms 3](#_Toc168308373)

[5 Conventions 3](#_Toc168308374)

[5.1 General 3](#_Toc168308375)

[5.2 Arithmetic operators 3](#_Toc168308376)

[5.3 Logical operators 3](#_Toc168308377)

[5.4 Relational operators 3](#_Toc168308378)

[5.5 Bit-wise operators 3](#_Toc168308379)

[5.6 Assignment operators 3](#_Toc168308380)

[5.7 Other operators 3](#_Toc168308381)

[5.8 Mathematical functions 3](#_Toc168308382)

[5.9 Order of operation precedence 6](#_Toc168308383)

[5.10 Variables, syntax elements, and tables 6](#_Toc168308384)

[6 Overall V-DMC characteristics, decoding operations, and post-decoding processes 6](#_Toc168308385)

[6.1 V-DMC characteristics 6](#_Toc168308386)

[6.2 V-DMC bitstream characteristics, decoding operations, and post-decoding processes 7](#_Toc168308387)

[7 Bitstream format, partitioning, and scanning processes 8](#_Toc168308388)

[7.1 General 8](#_Toc168308389)

[7.2 V3C bitstream formats 8](#_Toc168308390)

[7.3 NAL bitstream formats 8](#_Toc168308391)

[7.4 Partitioning of atlas frames into tiles 8](#_Toc168308392)

[7.5 Tile partition scanning processes 8](#_Toc168308393)

[7.6 Extension of partitioning process for Attribute tile components 8](#_Toc168308394)

[7.7 Attribute tile partition scanning processes 9](#_Toc168308395)

[8 Syntax and semantics 10](#_Toc168308396)

[8.1 Method of specifying syntax in tabular form 10](#_Toc168308397)

[8.2 Specification of syntax functions and descriptors 10](#_Toc168308398)

[8.3 Syntax in tabular form 10](#_Toc168308399)

[8.3.1 General 10](#_Toc168308400)

[8.3.2 V3C unit syntax 10](#_Toc168308401)

[8.3.3 Byte alignment syntax 10](#_Toc168308402)

[8.3.4 V3C parameter set syntax 10](#_Toc168308403)

[8.3.5 NAL unit syntax 12](#_Toc168308404)

[8.3.6 Raw byte sequence payloads, trailing bits, and byte alignment syntax 12](#_Toc168308405)

[8.3.7 Atlas tile data unit syntax 19](#_Toc168308406)

[8.3.8 Supplemental enhancement information message syntax 19](#_Toc168308407)

[8.3.9 VDMC atlas tile data unit syntax 19](#_Toc168308408)

[8.4 Semantics 24](#_Toc168308409)

[8.4.1 General semantics 24](#_Toc168308410)

[8.4.2 V3C unit semantics 24](#_Toc168308411)

[8.4.3 Byte alignment semantics 24](#_Toc168308412)

[8.4.4 V3C parameter set semantics 24](#_Toc168308413)

[8.4.5 NAL unit semantics 26](#_Toc168308414)

[8.4.6 Raw byte sequence payloads, trailing bits, and byte alignment semantics 26](#_Toc168308415)

[8.4.7 Atlas tile data unit semantics 36](#_Toc168308416)

[8.4.8 Supplemental enhancement information message semantics 36](#_Toc168308417)

[8.4.9 VDMC atlas tile data unit semantics 36](#_Toc168308418)

[9 Decoding process 45](#_Toc168308419)

[9.1 General decoding process 45](#_Toc168308420)

[9.2 Atlas data decoding process 45](#_Toc168308421)

[9.2.1 General atlas decoding process 45](#_Toc168308422)

[9.2.2 Decoding process for a coded atlas frame 45](#_Toc168308423)

[9.2.3 Atlas NAL unit decoding process 46](#_Toc168308424)

[9.2.4 Atlas tile header decoding process 46](#_Toc168308425)

[9.2.5 Decoding process for patch data units 46](#_Toc168308426)

[9.2.6 Decoding process of the block to patch map 47](#_Toc168308427)

[9.2.7 Conversion of tile level patch information to atlas level patch information 47](#_Toc168308428)

[9.2.8 Decoding process for meshpatch data units 47](#_Toc168308429)

[9.2.9 Conversion of tile level meshpatch information to atlas level meshpatch information 60](#_Toc168308430)

[9.3 Occupancy video decoding process 63](#_Toc168308431)

[9.4 Geometry video decoding process 63](#_Toc168308432)

[9.5 Attribute video decoding process 63](#_Toc168308433)

[9.6 Packed video decoding process 63](#_Toc168308434)

[9.7 Common atlas decoding process 63](#_Toc168308435)

[9.8 Sub-bitstream extraction process 63](#_Toc168308436)

[9.9 Basemesh decoding process 63](#_Toc168308437)

[10 Pre-reconstruction process 65](#_Toc168308438)

[11 Reconstruction process 65](#_Toc168308439)

[11.1 General 65](#_Toc168308440)

[11.2 Submesh Reconstruction Process 69](#_Toc168308441)

[11.2.1 General 69](#_Toc168308442)

[11.2.2 Texture coordinate derivation 70](#_Toc168308443)

[11.2.3 Subdivision process 79](#_Toc168308444)

[11.2.4 Inverse image packing of transform coefficients 92](#_Toc168308445)

[11.2.5 Inverse quantization 93](#_Toc168308446)

[11.2.6 Inverse transform 95](#_Toc168308447)

[11.2.7 Normal, tangent, and bitangent vector generation 96](#_Toc168308448)

[11.2.8 Vertex coordinates reconstruction 99](#_Toc168308449)

[11.2.9 Texture coordinate adaptation process 100](#_Toc168308450)

[11.3 Append submesh to a mesh process 100](#_Toc168308451)

[12 Post-reconstruction process 101](#_Toc168308452)

[12.1 General 101](#_Toc168308453)

[12.2 Zippering process 102](#_Toc168308454)

[12.2.1 Zippering method by distance 103](#_Toc168308455)

[12.2.2 Zippering method by index matching 103](#_Toc168308456)

[12.2.3 Identification of boundary vertices 104](#_Toc168308457)

[12.2.4 Identification of matched boundary vertices by distance 105](#_Toc168308458)

[12.2.5 Identification of matched boundary vertices by explicit signaling 107](#_Toc168308459)

[12.2.6 Border vertex fusion 108](#_Toc168308460)

[12.2.7 Handling of unmatched LoD vertices 110](#_Toc168308461)

[13 Adaptation process 118](#_Toc168308462)

[14 Parsing process 118](#_Toc168308463)

[Annex A (normative) Profile, tier, and levels 119](#_Toc168308464)

[A.1 Overview of profiles, tiers, and levels 119](#_Toc168308465)

[A.2 Profile, tier and level structure 119](#_Toc168308466)

[A.3 CodecGroup profile components 119](#_Toc168308467)

[A.4 Toolset profile components 119](#_Toc168308468)

[A.5 Reconstruction profile components 119](#_Toc168308469)

[A.6 Tiers and Levels 119](#_Toc168308470)

[A.7 Decoder instantiations 119](#_Toc168308471)

[Annex B (informative) Post-decoding conversion to nominal video formats 120](#_Toc168308472)

[B.1 General 120](#_Toc168308473)

[B.2 Nominal format conversion 122](#_Toc168308474)

[B.2.8 Arithmetic coded displacement nominal format conversion 126](#_Toc168308475)

[B.3 Conversion operations 127](#_Toc168308476)

[B.4 Unpacking process of a decoded packed video 127](#_Toc168308477)

[Annex C (informative) V3C sample stream format 128](#_Toc168308478)

[Annex D (normative) NAL sample stream format 129](#_Toc168308479)

[Annex E (normative) Atlas hypothetical reference decoder 130](#_Toc168308480)

[Annex F (normative) Supplemental enhancement information 131](#_Toc168308481)

[F.1 General 131](#_Toc168308482)

[F.2 SEI payload syntax 131](#_Toc168308483)

[F.2.1 V-DMC registered SEI message syntax 131](#_Toc168308484)

[F.2.2 V-DMC registered SEI message syntax 131](#_Toc168308485)

[F.2.3 Zippering SEI payload syntax 132](#_Toc168308486)

[F.2.4 Submesh SOI relationship indication SEI payload syntax 133](#_Toc168308487)

[F.2.5 Submesh distortion indication SEI payload syntax 133](#_Toc168308488)

[F.3 SEI payload semantics 134](#_Toc168308489)

[F.3.1 V-DMC registered SEI message semantics 134](#_Toc168308490)

[F.3.2 V-DMC registered SEI payload semantics 134](#_Toc168308491)

[F.3.3 Zippering SEI payload semantics 134](#_Toc168308492)

[F.3.4 Submesh SOI relationship indication SEI message semantics 137](#_Toc168308493)

[F.3.5 Submesh distortion indication SEI message semantics 137](#_Toc168308494)

[Annex G (informative) Volumetric usability information 139](#_Toc168308495)

[G.1 General 139](#_Toc168308496)

[G.2 VUI syntax 139](#_Toc168308497)

[G.2.1 VUI parameters syntax 139](#_Toc168308498)

[G.2.2 HRD parameters syntax 139](#_Toc168308499)

[G.2.3 Sub-layer HRD parameters syntax 139](#_Toc168308500)

[G.2.4 Maximum coded video resolution syntax 139](#_Toc168308501)

[G.2.5 Coordinate system parameters syntax 139](#_Toc168308502)

[G.2.6 V-DMC VUI extension 139](#_Toc168308503)

[G.3 VUI semantics 139](#_Toc168308504)

[G.3.1 VUI parameters semantics 139](#_Toc168308505)

[G.3.2 HRD parameters semantics 139](#_Toc168308506)

[G.3.3 HRD sub-layer parameters semantics 140](#_Toc168308507)

[G.3.4 Maximum coded video resolution semantics 140](#_Toc168308508)

[G.3.5 Coordinate system parameters semantics 140](#_Toc168308509)

[G.3.6 V-DMC VUI extension semantics 140](#_Toc168308510)

[Annex H (normative) Basemesh sub-bistream 141](#_Toc168308511)

[H.1 Scope 141](#_Toc168308512)

[H.2 Normative references 141](#_Toc168308513)

[H.3 Terms and definitions 141](#_Toc168308514)

[H.4 Abbreviated terms 141](#_Toc168308515)

[H.5 Conventions 141](#_Toc168308516)

[H.6 Overall Basemesh Codec characteristics, decoding operations, and post-decoding processes 142](#_Toc168308517)

[H.6.1 Basemesh bitstream characteristics, decoding operations, and post-decoding processes 142](#_Toc168308518)

[H.7 Bitstream format 142](#_Toc168308519)

[H.7.1 NAL bitstream formats 142](#_Toc168308520)

[H.8 Syntax and semantics 142](#_Toc168308521)

[H.8.1 Method of specifying syntax in tabular form 142](#_Toc168308522)

[H.8.2 Specification of syntax functions and descriptors 142](#_Toc168308523)

[H.8.3 Syntax in tabular form 142](#_Toc168308524)

[H.8.4 Semantics 148](#_Toc168308525)

[H.8.5 Raw byte sequence payloads, trailing bits, and byte alignment semantics 156](#_Toc168308526)

[H.9 Decoding process 167](#_Toc168308527)

[H.9.1 General decoding process 167](#_Toc168308528)

[H.9.2 Decoding process for a coded mesh frame 168](#_Toc168308529)

[H.9.3 Mesh NAL unit decoding process 170](#_Toc168308530)

[H.9.4 Submesh header decoding process 170](#_Toc168308531)

[H.10 Parsing process 184](#_Toc168308532)

[H.11 Profiles and levels 185](#_Toc168308533)

[H.12 Basemesh hypothetical reference decoder 185](#_Toc168308534)

[H.13 Supplemental enhancement information 185](#_Toc168308535)

[Annex I (Normative) MPEG EdgeBreaker Static Mesh Coding 186](#_Toc168308536)

[I.1 Scope 186](#_Toc168308537)

[I.2 Normative references 186](#_Toc168308538)

[I.3 Terms and definitions 186](#_Toc168308539)

[I.4 Abbreviated terms 189](#_Toc168308540)

[I.5 Conventions 189](#_Toc168308541)

[I.6 Overall MEB characteristics, decoding operations, and post-decoding processes 190](#_Toc168308542)

[I.6.1 MEB characteristics 190](#_Toc168308543)

[I.6.2 MEB bitstream characteristics, decoding operations, and post decoding processes 190](#_Toc168308544)

[I.7 Bitstream format 190](#_Toc168308545)

[I.8 Syntax and semantics 190](#_Toc168308546)

[I.8.1 Method of specifying syntax in tabular form 190](#_Toc168308547)

[I.8.2 Specification of syntax functions and descriptors 190](#_Toc168308548)

[I.8.3 Syntax in tabular form 192](#_Toc168308549)

[I.8.4 Semantics 200](#_Toc168308550)

[I.9 Decoding Process 209](#_Toc168308551)

[I.9.1 General 209](#_Toc168308552)

[I.9.2 Global initialization 211](#_Toc168308553)

[I.9.3 Connectivity pre pass 216](#_Toc168308554)

[I.9.4 Connectivity decoding 217](#_Toc168308555)

[I.9.5 Auxiliary connectivity decoding 220](#_Toc168308556)

[I.9.6 Decoding primary per vertex attributes 222](#_Toc168308557)

[I.9.7 Decoding auxiliary per vertex attributes 226](#_Toc168308558)

[I.9.8 Decoding auxiliary per face attributes 249](#_Toc168308559)

[I.9.9 Per fan deduplications 252](#_Toc168308560)

[I.9.10 Post reindexing per vertex attributes 253](#_Toc168308561)

[I.9.11 Conversion from corner table to indexed face set 254](#_Toc168308562)

[I.9.12 Duplication of per vertex position 257](#_Toc168308563)

[I.9.13 Adjustment of the decoded mesh 258](#_Toc168308564)

[I.9.14 Vertex degree traversal 260](#_Toc168308565)

[I.9.15 Corner Table 267](#_Toc168308566)

[I.10 Parsing Process 268](#_Toc168308567)

[I.10.1 General 268](#_Toc168308568)

[I.10.2 Parsing process for 0-th order Exp-Golomb codes 268](#_Toc168308569)

[I.10.3 CABAC parsing process 268](#_Toc168308570)

[Annex J (Normative) Arithmetic coded displacement sub-bitstream 271](#_Toc168308571)

[J.1 Scope 271](#_Toc168308572)

[J.2 Normative references 271](#_Toc168308573)

[J.3 Terms and definitions 271](#_Toc168308574)

[J.4 Abbreviated terms 271](#_Toc168308575)

[J.5 Conventions 271](#_Toc168308576)

[J.6 Bitstream format 271](#_Toc168308577)

[J.6.1 NAL bitstream formats 271](#_Toc168308578)

[J.7 Syntax and semantics 272](#_Toc168308579)

[J.7.1 Syntax in tabular form 272](#_Toc168308580)

[J.7.2 Semantics 278](#_Toc168308581)

[J.8 Decoding process 290](#_Toc168308582)

[J.8.1 General Decoding Process 290](#_Toc168308583)

[J.8.2 Decoding process for a coded displacement frame 290](#_Toc168308584)

[J.8.3 Displacement NAL unit decoding process 291](#_Toc168308585)

[J.8.4 Displacement header decoding process 291](#_Toc168308586)

[J.8.5 Decoding process for displacement data units 295](#_Toc168308587)

[J.9 Parsing process 298](#_Toc168308588)

[J.10 Profiles and levels 298](#_Toc168308589)

[J.11 Displacement hypothetical reference decoder 298](#_Toc168308590)

[J.12 Supplemental enhancement information 298](#_Toc168308591)

[Annex K (Normative) Inverse Binarization Process 299](#_Toc168308592)

[K.1 General 299](#_Toc168308593)

[K.2 Parsing processes 300](#_Toc168308594)

[K.2.1 Parsing unsigned fixed-length codes (FL) 300](#_Toc168308595)

[K.2.2 Parsing signed fixed-length codes (FL+S) 300](#_Toc168308596)

[K.2.3 Parsing k-th order exp-Golomb codes (EGk) 301](#_Toc168308597)

[K.2.4 Parsing concatenated truncated unary and k-th order exp-Golomb codes (TU+EGk) 301](#_Toc168308598)

[K.2.5 Parsing signed concatenated truncated unary and k-th order exp-Golomb codes (TU+EGk+S) 302](#_Toc168308599)

[K.2.6 Parsing truncated unary codes (TU) 302](#_Toc168308600)

[K.2.7 Parsing Mesh CLERS symbols 302](#_Toc168308601)

[K.3 CABAC parsing process 303](#_Toc168308602)

[K.3.1 Initialization 303](#_Toc168308603)

[K.3.2 Reading a single arithmetic-coded bin 303](#_Toc168308604)

[K.3.3 Contextual probability models 303](#_Toc168308605)

[K.4 Arithmetic decoding engine 307](#_Toc168308606)

[K.4.1 General 307](#_Toc168308607)

[K.4.2 State variables 307](#_Toc168308608)

[K.4.3 Initial state 308](#_Toc168308609)

[K.4.4 Arithmetic decoding 308](#_Toc168308610)

[References 310](#_Toc168308611)

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

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

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

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

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

This document was prepared by Technical Committee *[or Project Committee]* ISO/TC *[or ISO/PC]* ###, *[name of committee]*, Subcommittee SC ##, *[name of subcommittee]*.

This second/third/… edition cancels and replaces the first/second/… edition (ISO #####:####), which has been technically revised.

The main changes are as follows:

— xxx xxxxxxx xxx xxxx

A list of all parts in the ISO ##### series can be found on the ISO website.

Any feedback or questions on this document should be directed to the user’s national standards body. A complete listing of these bodies can be found at [www.iso.org/members.html](https://www.iso.org/members.html).

Introduction

The advances in 3D capture, modelling, and rendering have promoted the ubiquitous presence of 3D content across several platforms and devices. When uncompressed, 3D content can be costly in terms of storage and transmission. A visual volumetric video-based coding (V3C) standard has been created to address this emerging 3D content, currently addressing volumetric data types such as point clouds (ISO/IEC 23090-5) and immersive video content (ISO/IEC 23090-12).

3D meshes, along with point clouds and immersive video, are also widely used to represent such immersive content. A mesh is composed of several polygons that describe the boundary surface of a volumetric object. Each polygon is defined by its vertices in 3D space and the information on how the vertices are connected, referred to as connectivity information. Optionally, vertex attributes, such as colours, normal, etc., could be associated with the mesh vertices. Attributes could also be associated with the surface of the mesh by exploiting mapping information that describes a parameterization of the mesh onto 2D regions of the plane. Such mapping is usually described by a set of parametric coordinates, referred to as UV coordinates or texture coordinates. 2D attribute maps are used to store high resolution attribute information such as texture, normal, material ID etc. Such information could be used for various purposes such as texture mapping and shading.

To achieve more realism, 3D meshes are becoming ever more sophisticated, and a significant amount of data is linked to the creation and consumption of those meshes. Furthermore, dynamic mesh sequences may require large amounts of data, since it may consist of a significant quantity of information changing in time. Therefore, this document was developed to support compression of 3D mesh data utilizing the ISO V3C standard (ISO/IEC 23090-5).

The International Organization for Standardization (ISO) draws attention to the fact that it is claimed that compliance with this document may involve the use of a patent.

ISO takes no position concerning the evidence, validity and scope of this patent right.

The holder of this patent right has assured ISO that he/she is willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with ISO. Information may be obtained from the patent database available at [www.iso.org/patents](http://www.iso.org/patents).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those in the patent database. ISO shall not be held responsible for identifying any or all such patent rights.

Information technology — Coded representation of immersive media — Part 29: Video-based dynamic mesh coding (V-DMC)

# Scope

This document specifies syntax, semantics, and decoding for video based dynamic mesh coding (V-DMC) methods. Furthermore, this document specifies processes that may be needed for reconstruction of visual volumetric media and may also include additional processes such as post decoding, pre-reconstruction, post reconstruction, and adaptation.

# Normative references

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

ISO /IEC 23090-5(2E), *Information technology — Coded Representation of Immersive Media — Part 5: Visual Volumetric Video-based Coding (V3C) and Video-based Point Cloud Compression (V-PCC)*

# Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 23090-5(2E):2023 and the following apply.

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

* ISO Online browsing platform: available at <https://www.iso.org/obp>
* IEC Electropedia: available at <https://www.electropedia.org/>

3.1

atlas

collection of 2D bounding boxes and their associated information placed onto a rectangular frame and corresponding to a volume in 3D space on which volumetric data is rendered and a list of metadata corresponding to a part of a surface of a mesh in 3D space

3.2

atlas frame

2D rectangular array of atlas samples onto which *patches* (3.91) are projected and additional information related to the *patches* (3.91), corresponding to a *volumetric frame* (3.142) and a list of *meshpatches* (3.9) and additional information related to the *meshpatches* (3.9), corresponding to a *volumetric frame* (3.142)

3.3

atlas sample

position on the rectangular frame onto which *patches* (3.91) that are associated with an *atlas* (3.3) are projected or element of a list of *meshpatches* (3.9) that are associated with an *atlas* (3.3)

3.4

attribute frame

2D rectangular array created through the aggregation of *patches* (3.91) and *meshpatches* (3.9)containing values of a specific *attribute* (3.14)

3.5

attribute map

image for mapping an attribute into a surface of 3D shape

3.6

basemesh frame

array of structures created through the aggregation of *submeshes* (3.15) associated with each *meshpatch* (3.9)

3.7

geometry frame

2D array created through the aggregation of the *geometry* (3.64) information associated with each *patch* (3.91) and *meshpatch* (3.9)

3.8

displacement

set of 3D vectors that are added to the vertices of the subdivided mesh to approximate closely the input mesh surface

3.9

meshpatch

element of an *atlas* (3.3) associated with basemesh information

3.10

meshpatch data

data in an *atlas* (3.3) associated with a *meshpatch* (3.8) that enables the conversion of basemesh into the reconstructed mesh

3.11

motion decoder

process used for inter basemesh decoding

3.12

motion sub-bitstream

sequence of bits extracted from V3C bitstream that contains specified syntax that represents motion of a basemesh

3.13

ortho atlas

orthographic projection and packing of the faces into a texture atlas

3.14

subdivision

process of dividing the mesh faces into a number of sub-faces

3.15

submesh

independently decodable region of a *basemesh* (3.22 in ISO/IEC 23090-5)

3.16

subpatch

a set of parameters in the *meshpatch* (3.9) to construct a homography transform used to map the vertex coordinates of a mesh surface associated with a given faceId to a set of UV texture coordinates and their connectivity indices.

3.17

texture mapping

process of transferring attributes from a 2D attribute (texture) map into surface of a 3D mesh

3.18

texture tile

independently decodable rectangular region of an attribute map frame associated with a texture map

# Abbreviated terms

The following abbreviations apply in addition to the abbreviations in ISO/IEC 23090-5(2E):2023 clause 4.

|  |  |
| --- | --- |
| BMD | Basemesh Data |
| CPM | Contextual Probability Model |
| LOD | Level Of Detail |
| V-DMC | Video-based Dynamic Mesh Coding |

# Conventions

## General

The specifications in ISO/IEC 23090-5(2E):2023 clause 5.1 apply.

## Arithmetic operators

The specifications in ISO/IEC 23090-5(2E):2023 clause 5.2 apply.

## Logical operators

The specifications in ISO/IEC 23090-5(2E):2023 clause 5.3 apply.

## Relational operators

The specifications in ISO/IEC 23090-5(2E):2023 clause 5.4 apply.

## Bit-wise operators

The specifications in ISO/IEC 23090-5(2E):2023 clause 5.5 apply.

## Assignment operators

The specifications in ISO/IEC 23090-5(2E):2023 clause 5.6 apply.

## Other operators

The specifications in ISO/IEC 23090-5(2E):2023 clause 5.7 applies with the following addition.

findIndexInArray( v, A, R) is specified as follows:

* If value or array v is found in the array A and the index is less than or equal to R, the return value is the index in the array A.
* Otherwise, the return value is -1.

## Mathematical functions

The specifications in ISO/IEC 23090-5(2E):2023 clause 5.8 applies with the following addition.

Cross( x, y ) cross product function, operating on two vectors x and y

FindNeighbour( size, array, w ) {  
 for( n = 0; n < size; n++ ) {  
 if( array[ n ] == w ) {  
 return 1  
 }  
 }  
 return 0  
}

FindIndexInArray( value, array1D, sizeofArray ) {  
 for( n = 0; n < sizeofArray ; n++ ) {  
 if( array1D[ n ] == value ) {  
 return n  
 }  
 }  
 return -1  
}

ComputeNeighbours( faceCount, faces, neighbours, neighbourCounts) {  
 for( t = 0; t < faceCount; t++ ) {  
 for( i = 0; i < 3; i++ ) {  
 a = faces[ t ][ i ]  
 b = faces[ t ][ ( i + 1 ) % 3 ]  
 if( FindNeighbour( neighbourCounts[ a ], neighbours[ a ], b ) == 0 ) {  
 neighbours[ a ][ neighbourCounts[ a ]++ ] = b  
 }  
 if( FindNeighbour( neighbourCounts[ b ], neighbours[ b ], a ) == 0 ) {  
 neighbours[ b ][ neighbourCounts[ b ]++ ] = a  
 }  
 }  
 }  
}

ComputeTriangleNeighbours( faceCount, faces, neighbours, neighbourCounts) {  
 for( t = 0; t < faceCount; t++ ) {  
 for( i = 0; i < 3; i++ ) {  
 a = faces[ t ][ i ]  
 b = faces[ t ][ ( i + 1 ) % 3 ]  
 if( FindNeighbour( neighbourCounts[ a ], neighbours[ a ], t ) == 0 ) {  
 neighbours[ a ][ neighbourCounts[ a ]++ ] = t  
 }  
 if( FindNeighbour( neighbourCounts[ b ], neighbours[ b ], t ) == 0 ) {  
 neighbours[ b ][ neighbourCounts[ b ]++ ] = t  
 }  
 }  
 }  
}

edgeTriNeighbours, edgeTriNeighbourCount =   
 ComputeTriangleNeighboursOfEdge( vertex0, vertex1, triNeighbours, triNeighbourCounts){  
 for( tadj0 = 0; tadj0 < triNeighbourCounts[ vertex0 ]; tadj0++ ) {  
 for( tadj1 = 0; tadj1 < triNeighbourCounts[ vertex1 ]; tadj1++ ) {  
 if( triNeighbours[ vertex0 ][ tadj0 ] == triNeighbours[ vertex1 ][ tadj1 ] )˜ {  
 edgeTriNeighbours[ edgeTriNeighbourCount++ ] =  
 triNeighbours[vertex0][tadj0]  
 }  
 }  
 }  
}

findFaceThirdVertex(a,b,face){  
 v = -1  
 for( id = 0; id < 3; id++ ){  
 if( face[ id ] != a && face[ id ] != b ){  
 v  face[ id ]  
 }  
 }  
 return v  
}

Dot( vec0, vec1 ) {  
 out = 0  
 for( d = 0; d < 3; d++ ) {  
 out = out + vec0[d] \* vec1[d]  
 }  
 return out  
}

squareNorm( vec ) {  
 out = 0  
 for( d = 0; d < 3; d++ ) {  
 out = out + vec[d] \* vec[d]   
 }  
 return out  
}

Norm( vec ) {  
 return sqrt( squareNorm( vec ) )  
}

normalize( vec ) {  
 vec = vec / Norm( vec )   
}

isNaN( x ) {  
 return x != x  
}

isInf( x ) {  
 return ( x == 0.5 \* x ) || ( x == -0.5 \* x )   
}

ExtracOddBits( x ) {  
 y = x & 0×55555555  
 y = ( y | ( y >> 1 ) ) & 0×33333333  
 y = ( y | ( y >> 2 ) ) & 0×0F0F0F0F  
 y = ( y | ( y >> 4 ) ) & 0×00FF00FF  
 y = ( y | ( y >> 8 ) ) & 0×0000FFFF  
 return y  
}

( x, y ) = computeMorton2D( i ) {  
 x = extracOddBits( i >> 1 )  
 y = extracOddBits( i )  
}

pow( a, b ) {  
 return ab  
}

## Order of operation precedence

The specifications in ISO/IEC 23090-5(2E):2023 clause 5.9 apply.

## Variables, syntax elements, and tables

The specifications in ISO/IEC 23090-5(2E):2023 clause 5.10 apply with the following addition.

A struct is a composite data type (a set) that may consist of multiple variables, arrays, or even other struct data types. Variables, arrays, and struct data types of a struct S are referred to as members of struct S. A dot operator is used as a postfix with A to denote an existing member. More specifically, a member variable v of a struct S is denoted as S.v. A member array vArr of a struct S is denoted as S.vArr. A member struct vStruct of a struct S is denoted as S.vStruct.

# Overall V-DMC characteristics, decoding operations, and post-decoding processes

## V-DMC characteristics

This document enables the encoding and decoding processes of media representing dynamic mesh by using V3C technology.

This is achieved through first a conversion of input dynamic mesh representation into number of V3C components: a basemesh , a set of displacements, 2D representation of the attributes, and an atlas.

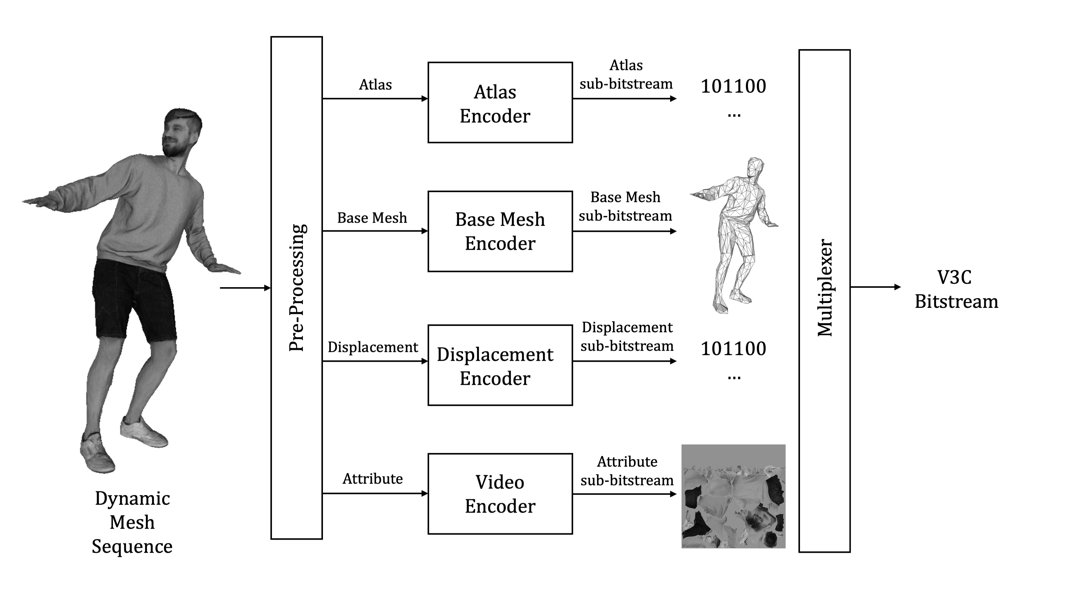
The basemesh component is a simplified low-resolution approximation of the original mesh. The basemesh component can be encoded using any mesh codec. This document defines a static mesh codec in Annex I that can use generic mechanism of the basemesh codec defined in Annex H. Methods of encoding basemesh other than the methods defined in Annex H and Annex I are outside the scope of this document.

The displacements component provides displacement vectors, that can be encoded as V3C geometry video component using any video codec, indicated by the profile or using an SEI message. Alternatively, the profile may indicate that the displacement component is encoded using arithmetic coding defined in Annex J.

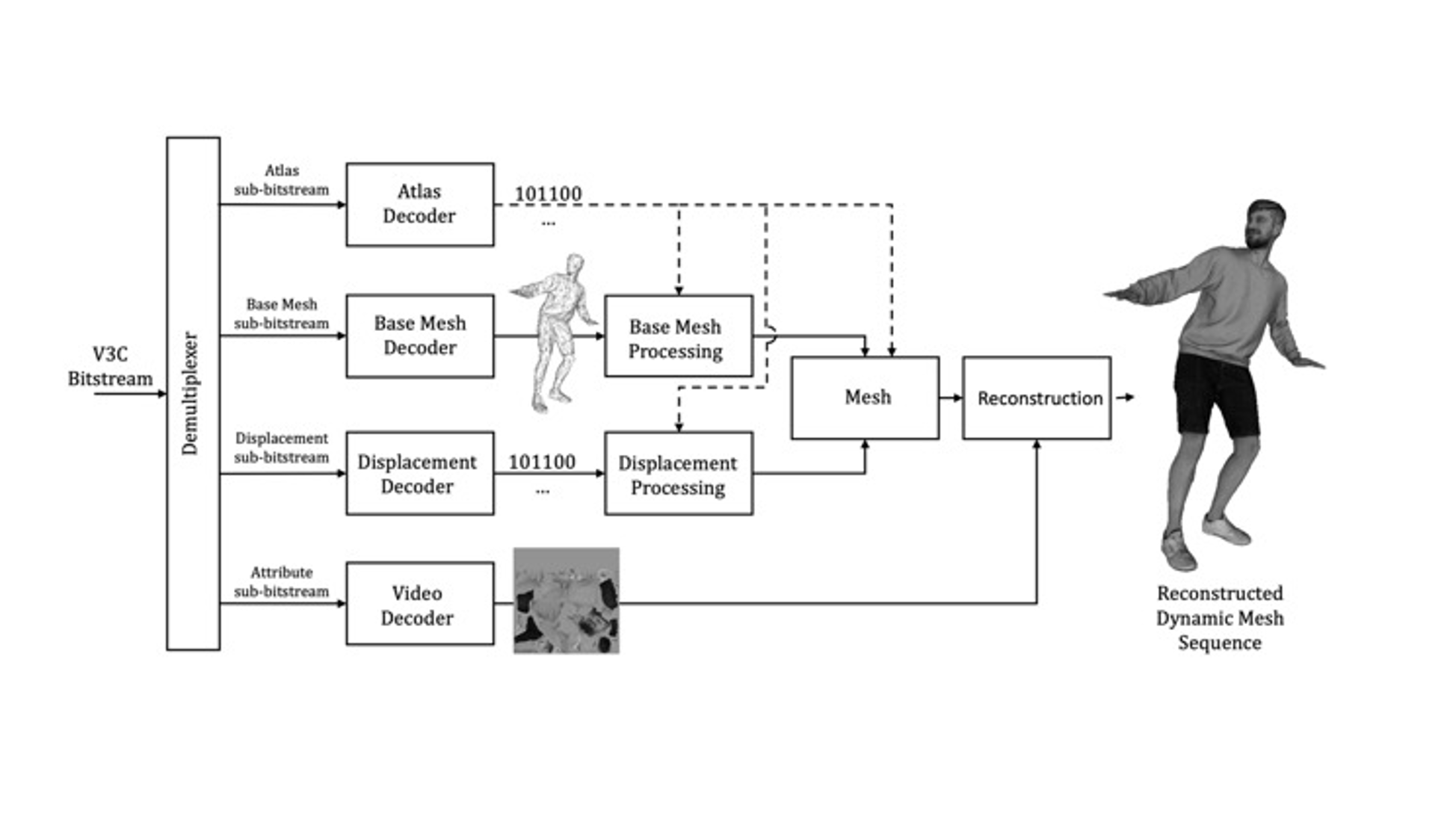
The attribute components can provide additional properties, e.g. texture or material information, and can be encoded by any video codec.

NOTE – Any existing video coding specification such as ISO/IEC 14496-10 or ISO/IEC 23008-2 or any future defined video coding specification can be used.

Finally, the atlas component provides information to a V3C decoding and/or rendering system on how to perform inverse reconstruction. For example, how to perform the subdivision of basemesh and how to apply the displacement vectors to the subdivided mesh vertices and how to apply attributes to the reconstructed mesh. An example of encoder and decoder flow is presented on Figure 1.



(a)



(b)

Figure 1 Volumetric media conversion at (a) encoder and reconstruction at (b) decoder side. The 3D media is converted to a series of sub-bitstreams: basemesh, displacement, and attributes. Additional atlas information is also included in the bitstream to enable inverse reconstruction.

## V-DMC bitstream characteristics, decoding operations, and post-decoding processes

This subclause provides high-level description of the characteristics and the operations needed for the decoding of V3C bitstreams and optional post-decoding processes and reconstruction related processes needed by applications, which may include nominal format conversion, pre-reconstruction, reconstruction, post-reconstruction, and adaptation.

As mentioned in subclause 6.1, a V3C bitstream containing V3C components, such as atlas, basemesh, displacement, and attributes components. Furthermore, to provide additional functionalities and similar flexibility as available in many video specifications, the atlas component may be divided into tiles and is encapsulated into NAL units. Similarly, the basemesh component may be divided into submeshes and is encapsulated into NAL units. Clause 7 provides further details on the encapsulation of V3C component bitstreams into V3C units and NAL units.

The syntax elements and their semantics are specified in Clause 8. Particular focus is placed on the extension to V-DMC and its characteristics, and any constraints that may apply on such syntax.

Clause 9 invokes the decoding process of a V3C bitstream or a collection of V3C sub-bitstreams, with outputs composed of decoded atlas information, decoded basemesh information, decoded displacement information, if available, a set of decoded video sub-bitstreams, corresponding to attribute and displacement, if available, and information from the VPS if available.

The decoded video frames may require the application of additional transformations, as described in Annex B, before any reconstruction operations. For example, the different components may need to be time-aligned and converted to a nominal video format. The outputs of Annex B are the following videos in the nominal format: GeoFramesNF, and, if present, AttrFramesNF.

The decoded basemesh frames may require the application of additional transformations, as described in Annex B, before any reconstruction operations. For example, the different components may need to be time-aligned and converted to a nominal basemesh format. The outputs of Annex B is the basemeshes in the nominal format: BasemeshFramesNF.

With the V3C components in the nominal format, the mesh content is obtained according to the following steps: pre-reconstruction, reconstruction, post-reconstruction, and adaptation.

In the reconstruction stage, described in Clause 11, the basemesh components in the nominal format, BasemeshFramesNF, the video components in the nominal format, GeoFramesNF, and, if present, AttrFramesNF, along with the decoded atlas data, are processed to reconstruct the mesh content and associated information as outputs.

The reconstructed mesh content can be further processed by applying informative post-reconstruction methods, as described in Clause 12.

# Bitstream format, partitioning, and scanning processes

## General

The specifications in ISO/IEC 23090-5(2E):2023 subclause 7.1 apply.

## V3C bitstream formats

The specifications in ISO/IEC 23090-5(2E):2023 subclause 7.2 apply.

## NAL bitstream formats

The specifications in ISO/IEC 23090-5(2E):2023 subclause 7.3 apply.

## Partitioning of atlas frames into tiles

The specifications in ISO/IEC 23090-5(2E):2023 subclause 7.4 apply.

## Tile partition scanning processes

The specifications in ISO/IEC 23090-5(2E):2023 subclause 7.5 apply.

## Extension of partitioning process for Attribute tile components

In the case the atlas frame size for an attribute component is different from the atlas frame size of geometry component, the partitioning of the atlas frame can be extended as follows:

* The attribute frame is first partitioned into NumPartitionColumnsAtt[ attrIdx ] \* NumPartitionRowsAtt[ attrIdx ] number of attribute tile partitions, where NumPartitionColumnsAtt[ attrIdx ] and NumPartitionRowsAtt[ attrIdx ] are derived in subclause 7.7. The width and height of each attribute tile partition, respectively, is indicated in the atlas frame attribute tile information syntax (8.3.6.2.4).
* One or more attribute tile partitions are then combined into attribute tiles by indicating the locations of the attribute tile partitions that correspond to the top-left and bottom-right corners of the attribute tile. All attribute tile partitions within these two attribute tile partitions collectively form an attribute tile, which is essentially a rectangular region of the attribute video frame.

## Attribute tile partition scanning processes

The list PartitionWidthAtt[ attrIdx ][ i ], for i ranging from 0 to NumPartitionColumnsAtt[ attrIdx ] − 1, inclusive, specifying the width of the i-th attribute tile partition column in units of 64 samples, is derived, and the value of NumPartitionColumnsAtt[ attrIdx ] for an attribute with index attrIdx, is inferred, as follows:

if( afati\_uniform\_partition\_spacing\_flag[ attrIdx ] ) {  
 widthPartition =  
 ( afati\_partition\_cols\_width\_minus1[ attrIdx ] + 1 ) \* 64  
 NumPartitionColumnsAtt[ attrIdx ] =  
  asve\_attribute\_frame\_width[ attrIdx ] / widthPartition  
 PartitionPosXAtt[ attrIdx ][ 0 ] = 0  
 PartitionWidthAtt[ attrIdx ][ 0 ] = widthPartition  
 for( i = 1; i < NumPartitionColumnsAtt[ attrIdx ] – 1; i++ ) {  
 PartitionPosXAtt[ attrIdx ][ i ] = PartitionPosXAtt[ attrIdx ][ i – 1 ] +  
 PartitionWidthAtt[ attrIdx ][ i – 1 ]  
 PartitionWidthAtt[ attrIdx ][ i ] = widthPartition  
 }  
 } else {  
 NumPartitionColumnsAtt[ attrIdx ] =  
 afati\_num\_partition\_columns\_minus1[ attrIdx ] + 1  
 PartitionPosXAtt[ attrIdx ][ 0 ] = 0  
 partitionWidthAtt[ attrIdx ][ 0 ] =  
 ( afati\_partition\_column\_width\_minus1[ attrIdx ][ 0 ] + 1 ) \* 64  
 for( i = 1; i < NumPartitionColumnsAtt[ attrIdx ] – 1; i++ ) {  
 PartitionPosXAtt[ attrIdx ][ i ] = PartitionPosXAtt[ attrIdx ][ i – 1 ] +   
 PartitionWidthAtt[ attrIdx ][ i – 1 ]  
 PartitionWidthAtt[ attrIdx ][ i ] =  
 ( afati\_partition\_column\_width\_minus1[ attrIdx ][ i ] + 1 ) \* 64  
 }  
 }  
 if( NumPartitionColumnsAtt[ attrIdx ] > 1 ) {  
 lastIndex = NumPartitionColumnsAtt[ attrIdx ] – 1  
 PartitionPosXAtt[ attrIdx ][ lastIndex ] =   
 PartitionPosXAtt[ attrIdx ][ lastIndex – 1 ] +  
 PartitionWidthAtt[ attrIdx ][ lastIndex – 1 ]  
 PartitionAttributeWidth[ attrIdx ][ lastIndex ] =  
 asve\_attribute\_frame\_width[ attrIdx ] –  
 PartitionPosXAtt[ attrIdx ][ lastIndex ]  
 }

The list PartitionHeightAtt[ attrIdx ][ j ] for j ranging from 0 to NumPartitionRowsAtt[ attrIdx ] - 1, inclusive, specifying the height of the j-th attribute tile partition row in units of 64 samples, is derived, and the value of NumPartitionRowsAtt[ attrIdx ] is inferred, as follows:

if( afati\_uniform\_partition\_spacing\_flag[ attrIdx ] ) {  
 heightPartition =  
 (afati\_partition\_rows\_height\_minus1[ attrIdx ] + 1) \* 64  
 NumPartitionRowsAtt[ attrIdx ] =  
 asve\_attribute\_frame\_height[ attrIdx ] / heightPartition  
 PartitionPosYAtt[ attrIdx ][ 0 ] = 0  
 PartitionHeightAtt[ attrIdx ][ 0 ] = heightPartition  
 for( j = 1; j < NumPartitionRowsAtt[ attrIdx ] – 1; j++ ) {  
 PartitionPosYAtt[ attrIdx ][ j ] = PartitionPosYAtt[ attrIdx ][ j – 1 ] +  
 PartitionHeightAtt[ attrIdx ][ j – 1 ]  
 PartitionHeightAtt[ attrIdx ][ j ] = heightPartition  
 }  
 } else {  
 NumPartitionRowsAtt[ attrIdx ] = afati\_num\_partition\_rows\_minus1[ attrIdx ] + 1  
 PartitionPosYAtt[ attrIdx ][ 0 ] = 0  
 PartitionHeightAtt[ attrIdx ][ 0 ] =  
 ( afati\_partition\_row\_height\_minus1[ attrIdx ][ 0 ] + 1 ) \* 64  
 for( j = 1; j < NumPartitionRowsAtt[ attrIdx ] – 1; j++ ) {  
 PartitionPosYAtt[ attrIdx ][ j ] = PartitionPosYAtt[ attrIdx ][ j – 1 ] +  
 PartitionHeightAtt[ attrIdx ][ j – 1 ]  
 PartitionHeightAtt[ attrIdx ][ j ] =  
 ( afati\_partition\_row\_height\_minus1[ attrIdx ][ j ] + 1 ) \* 64  
 }  
 }  
 if( NumPartitionRowsAtt[ attrIdx ] > 1 ) {  
 lastindex = NumPartitionRowsAtt[ attrIdx ] – 1  
 PartitionPosYAtt[ attrIdx ][ lastIndex ] =  
 PartitionPosYAtt[ attrIdx ][ lastIndex – 1 ] +  
 PartitionHeightAtt[ attrIdx ][ lastIndex – 1 ]  
 PartitionHeightAtt[ attrIdx ][ lastIndex ] =   
 asve\_attribute\_frame\_height[ attrIdx ] –  
 PartitionPosYAtt[ attrIdx ][ NumPartitionRowsAtt[ attrIdx ] – 1 ]  
 }

It is a requirement of bitstream conformance that the values of PartitionWidthAtt[ attrIdx ][ i ] for i ranging from 0 to NumPartitionColumnsAtt[ attrIdx ] – 1, inclusive, and PartitionHeightAtt[ attrIdx ][ j ] for j ranging from 0 to NumPartitionRowsAtt[ attrIdx ] – 1, inclusive, are greater than or equal to 64 and are multiples of PatchPackingBlockSize.

# Syntax and semantics

## Method of specifying syntax in tabular form

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.1 apply.

## Specification of syntax functions and descriptors

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.2 apply.

## Syntax in tabular form

### General

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.1 apply.

### V3C unit syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.2 apply.

### Byte alignment syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.3 apply.

### V3C parameter set syntax

#### General V3C parameter set syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.4.1 apply.

#### Profile, tier, and level syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.4.2 apply.

#### Occupancy information syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.4.3 apply.

#### Geometry information syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.4.4 apply.

#### Attribute information syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.4.5 apply.

#### Profile toolset constraints information syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.4.6 apply.

#### Packing information syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.4.7 apply.

#### VPS extension syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.4.8 apply.

#### Packed video extension syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.4.9 apply.

#### Length alignment syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.4.10 apply.

#### V3C parameter set V-DMC extension syntax

|  |  |
| --- | --- |
| vps\_vdmc\_extension ( ) { | **Descriptor** |
| for( k=0; k<vps\_atlas\_count\_minus1+1; k++ ){ |  |
| j = vps\_atlas\_id[ k ] |  |
| **vps\_ext\_bmesh\_data\_substream\_codec\_id**[ j ] | u(8) |
| **vps\_ext\_bmesh\_geometry\_3d\_bit\_depth\_minus1**[ j ] | u(5) |
| **vps\_ext\_bmesh\_geometry\_3d\_msb\_align\_flag**[ j ] | u(1) |
| **vps\_ext\_bmesh\_data\_attribute\_count**[ j ] | u(8) |
| for( i = 0; i < vps\_ext\_bmesh\_data\_attribute\_count[ j ]; i++ ) { |  |
| **vps\_ext\_bmesh\_attribute\_index**[ j ][ i ] | u(7) |
| **vps\_ext\_bmesh\_attribute\_bit\_depth\_minus1**[ j ][ i ] | u(5) |
| **vps\_ext\_bmesh\_attribute\_msb\_align\_flag**[ j ][ i ] | u(1) |
| **vps\_ext\_bmesh\_attribute\_type**[ j ][ i ] | u(4) |
| } |  |
| for( i = 0; i < ai\_attribute\_count[ j ]; i++ ) { |  |
| **vps\_ext\_attribute\_frame\_width**[ j ][ i ] | ue(v) |
| **vps\_ext\_attribute\_frame\_height**[ j ][ i ] | ue(v) |
| } |  |
| } |  |
| } |  |

### NAL unit syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.5 apply.

### Raw byte sequence payloads, trailing bits, and byte alignment syntax

#### Atlas sequence parameter set RBSP syntax

##### General atlas sequence parameter set RBSP syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.6.1.1 apply.

##### Point local reconstruction information syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.6.1.2 apply.

##### Atlas sequence parameter set V-DMC extension syntax

|  |  |
| --- | --- |
| asps\_vdmc\_extension( ) { | **Descriptor** |
| **asve\_subdivision\_iteration\_count** | u(3) |
| AspsSubdivisionCount = asve\_subdivision\_iteration\_count |  |
| if( AspsSubdivisionCount > 0 ) { |  |
| **asve\_lod\_adaptive\_subdivision\_flag** | u(1) |
| } |  |
| for( i=0; i < AspsSubdivisionCount ; i++) { |  |
| if( asve\_lod\_adaptive\_subdivision\_flag == 1 || i == 0 ) { |  |
| **asve\_subdivision\_method**[ i ] | u(3) |
| } else { |  |
| asve\_subdivision\_method[ i ] = asve\_subdivision\_method[ 0 ] |  |
| } |  |
| AspsSubdivisionMethod[ i ] = asve\_subdivision\_method[ i ] |  |
| } |  |
| **asve\_1d\_displacement\_flag** | u(1) |
| **asve\_displacement\_reference\_qp** | u(7) |
| **asve\_quantization\_parameters\_present\_flag** | u(1) |
| if( asve\_quantization\_parameters\_present\_flag ) |  |
| vdmc\_quantization\_parameters( 0, AspsSubdivisionCount ) |  |
| **asve\_transform\_method** | u(3) |
| if(asve\_transform\_method == LINEAR\_LIFTING) { |  |
| **asve\_lifting\_offset\_present\_flag** | u(1) |
| vdmc\_lifting\_transform\_parameters( 0, AspsSubdivisionCount ) |  |
| } |  |
| **asve\_num\_attribute\_video** | u(7) |
| for(i=0; i< asve\_num\_attribute\_video; i++){ |  |
| **asve\_attribute\_type\_id**[ i ] | u(4) |
| **asve\_attribute\_frame\_width**[ i ] | ue(v) |
| **asve\_attribute\_frame\_height**[ i ] | ue(v) |
| **asve\_****attribute\_subtexture\_enabled\_flag**[ i ] | u(1) |
| } |  |
| **asve\_displacement\_id\_present\_flag** | u(1) |
| **asve\_packing\_method** | u(1) |
| **asve\_projection\_texcoord\_enable\_flag** | u(1) |
| if( asve\_projection\_texcoord\_enable\_flag ){ |  |
| **asve\_projection\_texcoord\_mapping\_attribute\_index\_present\_flag** | u(1) |
| if( asve\_projection\_texcoord\_mapping\_attribute\_index\_present\_flag ) { |  |
| **asve\_projection\_texcoord\_mapping\_attribute\_index** | u(7) |
| } |  |
| **asve\_projection\_texcoord\_output\_attribute\_index** | u(7) |
| **asve\_projection\_texcoord\_output\_bit\_depth\_minus1** | u(5) |
| **asve\_projection\_texcoord\_scale\_factor** | fl(64) |
| } |  |
| **asve\_vdmc\_vui\_parameters\_present\_flag** | u(1) |
| if( asve\_vdmc\_vui\_parameters\_present\_flag ) |  |
| vdmc\_vui\_parameters() |  |
| } |  |

##### Quantization parameters syntax

|  |  |
| --- | --- |
| vdmc\_quantization\_parameters( qpIndex, subdivisionCount ){ | **Descriptor** |
| **vqp\_lod\_quantization\_flag**[ qpIndex ] | u(1) |
| **vqp\_bitdepth\_offset**[ qpIndex ] | se(v) |
| if( vqp\_lod\_quantization\_flag[ qpIndex ] == 0 ) { |  |
| for( k = 0; k < DisplacementDim; k++) { |  |
| **vqp\_quantization\_parameters**[ qpIndex ][ k ] | u(7) |
| for( i=0 ; i < subdivisionCount + 1; i++ ) |  |
| QuantizationParameter[ qpIndex ][ i ][ k ] =  vqp\_quantization\_parameters[ qpIndex ][ k ] |  |
| **vqp\_log2\_lod\_inverse\_scale**[ qpIndex ][ k ] | u(2) |
| } |  |
| } else { |  |
| for( i=0 ; i < subdivisionCount + 1; i++ ) { |  |
| for( k = 0; k < DisplacementDim; k++ ) { |  |
| **vqp\_lod\_delta\_quantization\_parameter\_value**[ qpIndex ][ i ][ k ] | ue(v) |
| if( vqp\_lod\_delta\_quantization\_parameter\_value[ qpIndex ][ i ][ k ] ) |  |
| **vqp\_lod\_delta\_quantization\_parameter\_sign**[ qpIndex ][ i ][ k ] | u(1) |
| if( qpIndex = 0 )  QuantizationParameter[ qpIndex ][ i ][ k ] =  asve\_displacement\_reference\_qp + ( 1 – 2 \*  vqp\_lod\_delta\_quantization\_parameter\_sign[ qpIndex ][ I ][ K ] ) \*   vqp\_lod\_delta\_quantization\_parameter\_value[ qpIndex ][ I ][ K ]  else  QuantizationParameter[ qpIndex ][ i ][ k ] =  QuantizationParameter[ qpIndex – 1 ][ i ][ k ] + ( 1 – 2 \*  vqp\_lod\_delta\_quantization\_parameter\_sign[ qpIndex ][ I ][ K ] ) \*   vqp\_lod\_delta\_quantization\_parameter\_value[ qpIndex ][ I ][ K ] |  |
| } |  |
| } |  |
| } |  |
| **vqp\_direct\_quantization\_enabled\_flag**[ qpIndex ] | u(1) |
| **vqp\_inverse\_quantization\_offset\_enable\_flag**[ qpIndex ] | u(1) |
| if( vqp\_inverse\_quantization\_offset\_enable\_flag[ qpIndex ] ) { |  |
| for( i = 0; i < subdivisionCount + 1; i++ ) { |  |
| for( j = 0; j < DisplacementDim; j++ ) { |  |
| for( k = 0; k < 3; k++ ) { |  |
| vqp\_inverse\_quantization\_offset\_sign\_delta[ qpIndex ] [ i ][ j ][ k ] | se(v) |
| vqp\_inverse\_quantization\_offset\_value\_log2\_prec1\_delta[ qpIndex ] [ i ][ j ][ k ] | se(v) |
| vqp\_inverse\_quantization\_offset\_value\_log2\_prec2\_delta[ qpIndex ] [ i ][ j ][ k ] | se(v) |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |

##### Lifting transform parameters syntax

|  |  |
| --- | --- |
| vdmc\_lifting\_transform\_parameters( ltpIndex, subdivisionCount ){ | **Descriptor** |
| if( asve\_lifting\_offset\_present\_flag && ltpindex ==2 ) { |  |
| **vltp\_lifting\_main\_param\_flag**[ ltpIndex ] | u(1) |
| for( i = 0; i < subdivisionCount + 1; i++ ) { |  |
| **vltp\_lifting\_offset\_values\_num**[ ltpIndex ][ i ] | se(v) |
| **vltp\_lifting\_offset\_values\_deno\_minus1**[ltpIndex ][ i ] | ue(v) |
| } |  |
| } |  |
| if( vltp\_lifting\_main\_param\_flag[ ltpIndex ] ) { |  |
| **vltp\_skip\_update\_flag**[ ltpIndex ] | u(1) |
| **vltp\_lod\_lifting\_parameter\_flag**[ ltpIndex ] | u(1) |
| for( i=0 ; i < subdivisionCount + 1; i++ ) { |  |
| if( vltp\_skip\_update\_flag[ ltpIndex ] ) |  |
| UpdateWeight[ ltpIndex ][ i ] = 0 |  |
| else { |  |
| **vltp\_adaptive\_update\_weight\_flag**[ ltpIndex ][ i ] | u(1) |
| **vltp\_valence\_update\_flag**[ ltpIndex ][ i ] | u(1) |
| if( vltp\_lod\_lifting\_parameter\_flag[ ltpIndex ] == 1 || i == 0) { |  |
| if( vltp\_adaptive\_update\_weight\_flag[ ltpIndex ][ i ] ) { |  |
| **vltp\_lifting\_update\_weight\_numerator**[ ltpIndex ][ i ] | ue(v) |
| **vltp\_lifting\_update\_weight\_denominator\_minus1**[ ltpIndex ][ i ] | ue(v) |
| UpdateWeight[ ltpIndex ][ i ] =  ( vltp\_lifting\_update\_weight\_numerator[ ltpIndex ][ i ] ) ÷   ( vltp\_lifting\_update\_weight\_denominator\_minus1[ ltpIndex ][ i ] + 1 ) |  |
| if ( vltp\_valence\_update\_flag[ ltpIndex ][ i ] ) { |  |
| **vltp\_valence\_update\_weight**[ ltpIndex ][ i ] | ue(v) |
| UpdateWeight[ ltpIndex ][ i ] \*= 1 +   (vltp\_valence\_update\_weight[ ltpIndex ][ i ] \* 0.1 ) |  |
| } |  |
| } else { |  |
| if ( vltp\_valence\_update\_flag[ ltpIndex ][ i ] ) { |  |
| UpdateWeight[ ltpIndex ][ i ] = 1 +   (vltp\_valence\_update\_weight[ ltpIndex ][ i ] \* 0.1 ) |  |
| } |  |
| **vltp\_log2\_lifting\_update\_weight**[ ltpIndex ][ i ] | ue(v) |
| UpdateWeight[ ltpIndex ][ i ] =  1 ÷ ( 1 << vltp\_log2\_lifting\_update\_weight[ ltpIndex ][ i ] ) |  |
| } |  |
| } else { |  |
| UpdateWeight[ ltpIndex ][ i ] = UpdateWeight[ ltpIndex ][ 0 ] |  |
| } |  |
| } |  |
| } |  |
| **vltp\_log2\_lifting\_prediction\_weight**[ ltpIndex ] | ue(v) |
| PredictionWeight[ ltpIndex ] = 1 ÷ ( 1 << vltp\_log2\_lifting\_prediction\_weight[ ltpIndex ] ) |  |
| } |  |
| } |  |

#### Atlas frame parameter set RBSP syntax

##### General atlas frame parameter set RBSP syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.6.2.1

##### Atlas frame tile information syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.6.2.2

##### Atlas frame parameter set V-DMC extension syntax

|  |  |
| --- | --- |
| afps\_vdmc\_extension( ) { | **Descriptor** |
| **afve\_overriden\_flag** | u(1) |
| if( afve\_overriden\_flag ) { |  |
| **afve\_subdivision\_enable\_flag** | u(1) |
| if( asve\_quantization\_parameters\_present\_flag ) { |  |
| **afve\_quantization\_parameters\_enable\_flag** | u(1) |
| } |  |
| **afve\_transform\_method\_enable\_flag** | u(1) |
| **afve\_transform\_parameters\_enable\_flag** | u(1) |
| } |  |
| if( afve\_subdivision\_enable\_flag ) { |  |
| **afve\_subdivision\_iteration\_count** | u(3) |
| AfpsSubdivisionCount = afve\_subdivision\_iteration\_count |  |
| if( AfpsSubdivisionCount > 0) { |  |
| **afve\_lod\_adaptive\_subdivision\_flag** | u(1) |
| } |  |
| for( i=0; i < AfpsSubdivisionCount ; i++ ) { |  |
| if( afve\_lod\_adaptive\_subdivision\_flag == 1 || i == 0 ) { |  |
| **afve\_subdivision\_method**[ i ] | u(3) |
| } else { |  |
| afve\_subdivision\_method[ i ] = afve\_subdivision\_method[ 0 ] |  |
| } |  |
| AfpsSubdivisionMethod[ i ] = afve\_subdivision\_method[ i ] |  |
| } |  |
| } else { |  |
| AfpsSubdivisonCount = AspsSubdivisionCount |  |
| for( i = 0; i < AfpsSubdivisonCount; i++ ){ |  |
| AfpsSubdivisionMethod[ i ] = AspsSubdivisionMethod[ i ] |  |
| } |  |
| } |  |
| if( afve\_quantization\_parameters\_enable\_flag ) |  |
| vdmc\_quantization\_parameters( 1, AfpsSubdivisonCount ) |  |
| if( afve\_transform\_method\_enable\_flag ) |  |
| **afve\_transform\_method** | u(3) |
| if( afve\_transform\_method == LINEAR\_LIFTING && afve\_transform\_parameters\_enable\_flag) |  |
| vdmc\_lifting\_transform\_parameters( 1, AfpsSubdivisonCount ) |  |
| for( attrIdx=0; attrIdx< asve\_num\_attribute\_video; attrIdx++ ) |  |
| atlas\_frame\_attribute\_tile\_information( attrIdx ) |  |
| atlas\_frame\_mesh\_information( ) |  |
| if( asve\_projection\_texcoord\_enable\_flag ) { |  |
| for( i = 0;i < NumSubMeshes; i++ ) { |  |
| **afve\_projection\_texcoord\_present\_flag**[ i ] | u(1) |
| if( afve\_projection\_texcoord\_present\_flag[ i ]) { |  |
| **afve\_projection\_texcoord\_width\_normalization**[ i ] | ue(v) |
| **afve\_projection\_texcoord\_height\_normalization**[ i ] | ue(v) |
| **afve\_projection\_texcoord\_gutter**[ i ] | ue(v) |
| } |  |
| } |  |
| } |  |
| } |  |

##### Atlas frame attribute tile information syntax

[Ed.Note (LK,DB) The discussion on atlas frame attribute tile information and it is mapping to ath\_id is ongoing. Throughout the document it is assumed that number of tiles and their IDs are the same in AFATI and in AFTI. This assumption is used in decoding process 9.2.8.2]

|  |  |
| --- | --- |
| atlas\_frame\_attribute\_tile\_information( attrIdx ) { | **Descriptor** |
| **afati\_single\_tile\_in\_atlas\_frame\_flag**[ attrIdx ] | u(1) |
| if( !afati\_single\_tile\_in\_atlas\_frame\_flag[ attrIdx ] ) { |  |
| **afati\_uniform\_partition\_spacing\_flag**[ attrIdx ] | u(1) |
| if( afati\_uniform\_partition\_spacing\_flag[ attrIdx ] ) { |  |
| **afati\_partition\_cols\_width\_minus1**[ attrIdx ] | ue(v) |
| **afati\_partition\_rows\_height\_minus1**[ attrIdx ] | ue(v) |
| } else { |  |
| **afati\_num\_partition\_columns\_minus1**[ attrIdx ] | ue(v) |
| **afati\_num\_partition\_rows\_minus1**[ attrIdx ] | ue(v) |
| for( i = 0; i < afati\_num\_partition\_columns\_minus1[ attrIdx ]; i++ ) |  |
| **afati\_partition\_column\_width\_minus1**[ attrIdx ][ i ] | ue(v) |
| for( i = 0; i < afati\_num\_partition\_rows\_minus1[ attrIdx ]; i++ ) |  |
| **afati\_partition\_row\_height\_minus1**[ attrIdx ][ i ] | ue(v) |
| } |  |
| **afati\_single\_partition\_per\_tile\_flag**[ attrIdx ] | u(1) |
| if( !afati\_single\_partition\_per\_tile\_flag[ attrIdx ] ) { |  |
| **afati\_num\_tiles\_in\_atlas\_frame\_minus1**[ attrIdx ] | ue(v) |
| for( i = 0; i < afati\_num\_tiles\_in\_atlas\_frame\_minus1[ attrIdx ] + 1; i++ ) { |  |
| **afati\_top\_left\_partition\_idx**[ attrIdx ][ i ] | u(v) |
| **afati\_bottom\_right\_partition\_column\_offset**[ attrIdx ][ i ] | ue(v) |
| **afati\_bottom\_right\_partition\_row\_offset**[ attrIdx ][ i ] | ue(v) |
| } |  |
| } |  |
| else |  |
| afati\_num\_tiles\_in\_atlas\_frame\_minus1[ attrIdx ]=  NumAttributePartitionsInAtlasFrame[ attrIdx ] – 1 |  |
| } |  |
| else |  |
| afati\_num\_tiles\_in\_atlas\_frame\_minus1[ attrIdx ] = 0 |  |
| **afati\_signalled\_tile\_id\_flag**[ attrIdx ] | u(1) |
| if( afati\_signalled\_tile\_id\_flag[ attrIdx ] ) { |  |
| **afati\_signalled\_tile\_id\_length\_minus1**[ attrIdx ] | ue(v) |
| for( i = 0; i < afati\_num\_tiles\_in\_atlas\_frame\_minus1[ attrIdx ] + 1; i++ ) { |  |
| **afati\_tile\_id**[ attrIdx ][ i ] | u(v) |
| TileIDToIndexAtt[ attrIdx ][ afati\_tile\_id[ a ][ i ] ] = i |  |
| TileIndexToIDAtt[ attrIdx ][ i ] = afati\_tile\_id[ a ][ i ] |  |
| } |  |
| } else { |  |
| for( i = 0; i < afati\_num\_tiles\_in\_atlas\_frame\_minus1[ attrIdx ] + 1; i++ ) { |  |
| afati\_tile\_id[ attrIdx ][ i ] = i |  |
| TileIDToIndexAtt[ attrIdx ][ i ] = i |  |
| TileIndexToIDAtt[ attrIdx ][ i ] = i |  |
| } |  |
| } |  |
| } |  |

##### Atlas frame mesh information syntax

|  |  |
| --- | --- |
| atlas\_frame\_mesh\_information( ) { | **Descriptor** |
| **afmi\_use\_single\_mesh\_flag** | u(1) |
| if( !afmi\_use\_single\_mesh\_flag ) { |  |
| **afmi\_num\_submeshes\_minus2** | u(8) |
| NumSubMeshes = afmi\_num\_submeshes\_minus2 +2 |  |
| } |  |
| else |  |
| NumSubMeshes = 1 |  |
| **afmi\_signalled\_submesh\_id\_flag** | u(1) |
| if( afve\_signalled\_submesh\_id\_flag ) { |  |
| **afmi\_signalled\_submesh\_id\_length\_minus1** | ue(v) |
| for( i = 0; i < NumSubMeshes; i++ ) |  |
| **afmi\_submesh\_id**[ i ] | u(v) |
| SubmeshIDToIndex[ afmi\_submesh\_id[ i ] ] = i |  |
| SubmeshIndexToID[ i ] = afmi\_submesh\_id[ i ] |  |
| } |  |
| } else { |  |
| for( i = 0; i < NumSubMeshes; i++ ) { |  |
| afmi\_submesh\_id[ i ] = i |  |
| SubmeshIDToIndex[ i ] = i |  |
| SubmeshIndexToID[ i ] = i |  |
| } |  |
| } |  |
| } |  |

#### Atlas adaptation parameter set RBSP syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.6.3 apply.

#### Supplemental enhancement information RBSP syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.6.4 apply.

#### Access unit delimiter RBSP syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.6.5 apply.

#### End of sequence RBSP syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.6.6 apply.

#### End of bitstream RBSP syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.6.7 apply.

#### Filler data RBSP syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.6.8 apply.

#### Atlas tile layer RBSP syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.6.9 apply.

#### RBSP trailing bit syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.6.10 apply.

#### Atlas tile header syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.6.11 apply.

#### Reference list structure syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.6.10 apply.

#### Common atlas sequence parameter set RBSP syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.6.13 apply.

#### Common atlas frame RBSP syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.6.14 apply.

### Atlas tile data unit syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.7 apply.

### Supplemental enhancement information message syntax

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.3.8 apply.

### VDMC atlas tile data unit syntax

#### General VDMC atlas tile data unit syntax

|  |  |
| --- | --- |
| vmdc\_atlas\_tile\_data\_unit( tileID ) { | **Descriptor** |
| if( ath\_type == SKIP\_TILE ) { |  |
| for( p = 0; p < RefAtduTotalNumMeshpatches[ tileID ]; p++ ) |  |
| skip\_meshpatch\_data\_unit( ) |  |
| } else { |  |
| p = 0 |  |
| do { |  |
| **atdu\_meshpatch\_mode**[ tileID ][ p ] | ue(v) |
| isEnd = ( ath\_type == P\_TILE && atdu\_meshpatch\_mode[ tileID ][ p ] == P\_END) ||  ( ath\_type == I\_TILE && atdu\_meshpatch\_mode[ tileID ][ p ] == I\_END ) |  |
| if( !isEnd ) { |  |
| meshpatch\_information\_data( tileID , p , atdu\_meshpatch\_mode[ tileID ][ p ] ) |  |
| p++ |  |
| } |  |
| } while( !isEnd ) |  |
| } |  |
| AtduTotalNumMeshpatches[ tileID ] = p |  |
| } |  |

#### Meshpatch information data syntax

|  |  |
| --- | --- |
| meshpatch\_information\_data( tileID, patchIdx, meshpatchMode ) { | **Descriptor** |
| if( ath\_type == P\_TILE ) { |  |
| if( meshpatchMode == P\_SKIP ) |  |
| skip\_meshpatch\_data\_unit( ) |  |
| else if( meshpatchMode == P\_MERGE ) |  |
| merge\_meshpatch\_data\_unit( tileID, patchIdx ) |  |
| else if( meshpatchMode == P\_INTRA ) |  |
| meshpatch\_data\_unit( tileID, patchIdx ) |  |
| else if( meshpatchMode == P\_INTER ) |  |
| inter\_meshpatch\_data\_unit( tileID, patchIdx ) |  |
| } |  |
| else if( ath\_type == I\_TILE ) { |  |
| if( meshpatchMode == I\_INTRA ) |  |
| meshpatch\_data\_unit( tileID, patchIdx ) |  |
| } |  |
| } |  |

#### Meshpatch data unit syntax

|  |  |
| --- | --- |
| meshpatch\_data\_unit( tileID, patchIdx ) { | **Descriptor** |
| **mdu\_submesh\_id**[ tileID ][ patchIdx ] | ue(v) |
| if( asve\_displacement\_id\_present\_flag ) { |  |
| **mdu\_displ\_id**[ tileID ][ patchIdx ] | ue(v) |
| } else { |  |
| **mdu\_face\_count\_minus1**[ tileID ][ patchIdx ] | ue(v) |
| **mdu\_2d\_pos\_x**[ tileID ][ patchIdx ] | ue(v) |
| **mdu\_2d\_pos\_y**[ tileID ][ patchIdx ] | ue(v) |
| **mdu\_2d\_size\_x\_minus1**[ tileID ][ patchIdx ] | ue(v) |
| **mdu\_2d\_size\_y\_minus1**[ tileID ][ patchIdx ] | ue(v) |
| } |  |
| **mdu\_parameters\_override\_flag**[ tileID ][ patchIdx ] | u(1) |
| if( mdu\_parameters\_override\_flag[ tileID ][ patchIdx ] ){ |  |
| **mdu\_subdivision\_override\_flag**[ tileID ][ patchIdx ] | u(1) |
| if( asve\_quantization\_parameters\_present\_flag ) { |  |
| **mdu\_quantization\_override\_flag**[ tileID ][ patchIdx ] | u(1) |
| } |  |
| **mdu\_transform\_method\_override\_flag**[ tileID ][ patchIdx ] | u(1) |
| **mdu\_transform\_parameters\_override\_flag**[ tileID ][ patchIdx ] | u(1) |
| } |  |
| if( mdu\_subdivision\_override\_flag[ tileID ][ patchIdx ] ){ |  |
| **mdu\_subdivision\_iteration\_count** | u(3) |
| PatchSubdivisionCount[ tileID ][ patchIdx ] =  mdu\_subdivision\_iteration\_count |  |
| if( PatchSubdivisionCount[ tileID ][ patchIdx ] > 0) { |  |
| **mdu\_lod\_adaptive\_subdivision\_flag**[ tileID ][ patchIdx ] | u(1) |
| } |  |
| for( i = 0; i < PatchSubdivisionCount[ tileID ][ patchIdx ]; i++) { |  |
| if( mdu\_lod\_adaptive\_subdivision\_flag == 1 || i == 0 ) { |  |
| **mdu\_subdivision\_method**[ tileID ][ patchIdx ] [ i ] | u(3) |
| } else { |  |
| mdu\_subdivision\_method[ tileID ][ patchIdx ][ i ] =  mdu\_subdivision\_method[ tileID ][ patchIdx ][ 0 ] |  |
| } |  |
| PatchSubdivisionMethod[ tileID ][ patchIdx ][ i ] =  mdu\_subdivision\_method[ i ] |  |
| } |  |
| } else { |  |
| PatchSubdivisionCount[ tileID ][ patchIdx ] = AfpsSubdivisonCount |  |
| for( i = 0; i < PatchSubdivisionCount[ tileID ][ patchIdx ]; i++ ){ |  |
| PatchSubdivisionMethod[ tileID ][ patchIdx ][ i ] = AfpsSubdivisionMethod[ i ] |  |
| } |  |
| } |  |
| if(mdu\_quantization\_override\_flag[ tileID ][ patchIdx ]) |  |
| vdmc\_quantization\_parameters(2, PatchSubdivisionCount[ tileID ][ patchIdx ] ) |  |
| **mdu\_displacement\_coordinate\_system**[ tileID ][ patchIdx ] | u(1) |
| if(mdu\_transform\_method\_override\_flag[ tileID ][ patchIdx ]) |  |
| **mdu\_transform\_method**[ tileID ][ patchIdx ] | u(3) |
| if(mdu\_transform\_method[ tileID ][ patchIdx ]== LINEAR\_LIFTING &&  mdu\_transform\_parameters\_override\_flag[ tileID ][ patchIdx ]) { |  |
| vdmc\_lifting\_transform\_parameters(2, PatchSubdivisionCount[ tileID ][ patchIdx ] ) |  |
| } |  |
| for( i = 0; i <= PatchSubdivisionCount[ tileID ][ patchIdx ]; i++){ |  |
| **mdu\_block\_countblock\_count\_minus1**[ tileID ][ patchIdx ][ i ]; | u(v) |
| **mdu\_last\_pos\_in\_blockin\_block**[ tileID ][ patchIdx ][ i ] | u(v) |
| } |  |
| for( i=0; i< asve\_num\_attribute\_video; i++ ){ |  |
| if( asve\_attribute\_subtexture\_enabled\_flag[ i ] ){ |  |
| **mdu\_attributes\_2d\_pos\_x**[ tileID ][ patchIdx ][ i ] | ue(v) |
| **mdu\_attributes\_2d\_pos\_y**[ tileID ][ patchIdx ][ i ] | ue(v) |
| **mdu\_attributes\_2d\_size\_x\_minus1**[ tileID ][ patchIdx ][ i ] | ue(v) |
| **mdu\_attributes\_2d\_size\_y\_minus1**[ tileID ][ patchIdx ][ i ] | ue(v) |
| } |  |
| } |  |
| smIdx = SubmeshIDToIndex[ mdu\_submesh\_id[ tileID ][ patchIdx ] ] |  |
| if( afve\_projection\_texcoord\_present\_flag[ smIdx ] ) |  |
| texture\_projection\_information( tileID, patchIdx ) |  |
| } |  |

#### Skip meshpatch data unit syntax

|  |  |
| --- | --- |
| skip\_meshpatch\_data\_unit( ) { | **Descriptor** |
| } |  |

#### Merge meshpatch data unit syntax

|  |  |
| --- | --- |
| merge\_meshpatch\_data\_unit( tileID, patchIdx ) { | **Descriptor** |
| if( NumRefIdxActive ) |  |
| **mmdu\_ref\_index**[ tileID ][ patchIdx ] | ue(v) |
| **mmdu\_patch\_index**[ tileID ][ patchIdx ] | se(v) |
| if( asve\_lifting\_offset\_present\_flag ){ |  |
| vdmc\_lifting\_transform\_parameters( 2, PatchSubdivisionCount[ tileID ][ patchIdx ] ) |  |
| } |  |
| if( asve\_projection\_texcoord\_enable\_flag ){ |  |
| **mmdu\_texture\_projection\_present\_flag**[ tileID ][ patchIdx ] | u(1) |
| if( mmdu\_texture\_projection\_present\_flag[ tileID ][ patchIdx ] ) |  |
| texture\_projection\_merge\_information( tileID, patchIdx ) |  |
| } |  |
| } |  |
| } |  |

#### Inter meshpatch data unit syntax

|  |  |
| --- | --- |
| inter\_meshpatch\_data\_unit( tileID, patchIdx ) { | **Descriptor** |
| if( NumRefIdxActive ) |  |
| **imdu\_ref\_index**[ tileID ][ patchIdx ] | ue(v) |
| **imdu\_patch\_index**[ tileID ][ patchIdx ] | se(v) |
| if ( !asve\_displacement\_id\_present\_flag ) { |  |
| **imdu\_delta\_face\_count\_minus1**[ tileID ][ patchIdx ] | se(v) |
| **imdu\_2d\_delta\_pos\_x**[ tileID ][ patchIdx ] | se(v) |
| **imdu\_2d\_delta\_pos\_y**[ tileID ][ patchIdx ] | se(v) |
| **imdu\_2d\_delta\_size\_x**[ tileID ][ patchIdx ] | se(v) |
| **imdu\_2d\_delta\_size\_y**[ tileID ][ patchIdx ] | se(v) |
| } |  |
| **imdu\_subdivision\_override\_flag**[ tileID ][ patchIdx ] | u(1) |
| if( imdu\_subdivision\_override\_flag[ tileID ][ patchIdx ] ){ |  |
| **imdu\_subdivision\_iteration\_count**[ tileID ][ patchIdx ] | U(3) |
| InterPatchSubdivisionCount[ tileID ][ patchIdx ] =  imdu\_subdivision\_iteration\_count[ tileID ][ patchIdx ] |  |
| }else |  |
| InterPatchSubdivisionCount[ tileID ][ patchIdx ] = AfpsSubdivisonCount |  |
| for( i = 0; i <= InterPatchSubdivisionCount[ tileID ][ patchIdx ]; i++){ |  |
| **imdu\_delta\_block\_count**[ tileID ][ patchIdx ][ i ]; | se(v) |
| **imdu\_delta\_last\_pos\_in\_blockin\_block**[ tileID ][ patchIdx ][ i ] | se(v) |
| } |  |
| for(i=0; i< asve\_num\_attribute\_video; i++ ) { |  |
| if( asve\_attribute\_subtexture\_enabled\_flag[ i ] ) { |  |
| **imdu\_attributes\_2d\_delta\_pos\_x**[ tileID ][ patchIdx ][ i ] | se(v) |
| **imdu\_attributes\_2d\_delta\_pos\_y**[ tileID ][ patchIdx ][ i ] | se(v) |
| **imdu\_attributes\_2d\_delta\_size\_x**[ tileID ][ patchIdx ][ i ] | se(v) |
| **imdu\_attributes\_2d\_delta\_size\_y**[ tileID ][ patchIdx ][ i ] | se(v) |
| } |  |
| } |  |
| if( asve\_lifting\_offset\_present\_flag ){ |  |
| vdmc\_lifting\_transform\_parameters( 2, PatchSubdivisionCount[ tileID ][ patchIdx ] ) |  |
| } |  |
| if( asve\_projection\_texcoord\_enable\_flag ) |  |
| **imdu\_texture\_projection\_present\_flag**[ tileID ][ patchIdx ] | u(1) |
| if( imdu\_texture\_projection\_present\_flag[ tileID ][ patchIdx ] ) |  |
| texture\_projection\_inter\_information( tileID, patchIdx ) |  |
| } |  |

#### Texture projection information syntax

|  |  |
| --- | --- |
| texture\_projection\_information( tileID, patchIdx ) { | **Descriptor** |
| **tpi\_face\_id\_present\_flag**[ tileID ][ patchIdx ] | u(1) |
| fid\_flag = tpi\_face\_id\_present\_flag[ tileID ][ patchIdx ] |  |
| **tpi\_frame\_scale**[ tileID ][ patchIdx ] | fl(64) |
| **tpi\_subpatch\_count\_minus1**[ tileID ][ patchIdx ] | ue(v) |
| numSubpatches = tpi\_subpatch\_count\_minus1[ tileID ][ patchIdx ] + 1 |  |
| for( subIdx = 0; subIdx < numSubpatches; subIdx++) |  |
| subpatch\_information( tileID, patchIdx , subIdx , fid\_flag ) |  |
| } |  |

#### Sub-patch information syntax

|  |  |
| --- | --- |
| subpatch\_information( tileID, patchIdx , subIdx , fid\_flag) { | **Descriptor** |
| if( !fid\_flag ) |  |
| **si\_face\_id**[ tileID ][ patchIdx ][ subIdx ] | ue(v) |
| **si\_projection\_id**[ tileID ][ patchIdx ][ subIdx ] | u(v) |
| **si\_orientation\_id**[ tileID ][ patchIdx ][ subIdx ] | ue(v) |
| **si\_2d\_pos\_x**[ tileID ][ patchIdx ][ subIdx ] | ue(v) |
| **si\_2d\_pos\_y**[ tileID ][ patchIdx ][ subIdx ] | ue(v) |
| **si\_2d\_size\_x\_minus1\_diff**[ tileID ][ patchIdx ][ subIdx ] | se(v) |
| **si\_2d\_size\_y\_minus1\_diff**[ tileID ][ patchIdx ][ subIdx ] | ve(v) |
| **si\_scale\_present\_flag**[ tileID ][ patchIdx ][ subdx ] | u(1) |
| if( si\_scale\_present\_flag[ tileID ][ patchIdx ][ subIdx ] ) |  |
| **si\_scale\_power\_factor**[ tileID ][ patchIdx ][ subIdx ] | ue(v) |
| } |  |

#### Texture projection inter information syntax

|  |  |
| --- | --- |
| texture\_projection\_inter\_information( tileID, patchIdx ) { | **Descriptor** |
| **tpii\_face\_id\_present\_flag**[ tileID ][ patchIdx ] | u(1) |
| fid\_flag = tpii\_face\_id\_present\_flag[ tileID ][ patchIdx ] |  |
| **tpii\_frame\_scale**[ tileID ][ patchIdx ] | fl(64) |
| **tpii\_subpatch\_count\_minus1**[ tileID ][ patchIdx ] | ue(v) |
| **tpii\_subpatch\_inter\_enable\_flag**[ tileID ][ patchIdx ] | u(1) |
| numSubpatches = tpii\_subpatch\_count\_minus1[ tileID ][ patchIdx ] + 1 |  |
| for( subIdx = 0; subIdx < numSubpatches; subIdx++) |  |
| if( tpii\_subpatch\_inter\_enable\_flag[ tileID ][ patchIdx ] ) |  |
| **tpii\_subpatch\_inter\_present\_flag**[ tileID ][ patchIdx ][ subIdx ] | u(1) |
| if( tpii\_subpatch\_inter\_present\_flag[ tileID ][ patchIdx ][ subIdx ] ) |  |
| subpatch\_inter\_information( tileID, patchIdx , subIdx ) |  |
| else |  |
| subpatch\_information( tileID, patchIdx , subIdx , fid\_flag ) |  |
| } |  |

#### Sub-patch inter information syntax

|  |  |
| --- | --- |
| subpatch\_inter\_information( tileID, patchIdx , subIdx ) { | **Descriptor** |
| **sii\_subpatch\_index\_diff**[ tileID ][ patchIdx ][ subIdx ] | se(v) |
| **sii\_2d\_pos\_x\_delta**[ tileID ][ patchIdx ][ subIdx ] | se(v) |
| **sii\_2d\_pos\_y\_delta**[ tileID ][ patchIdx ][ subIdx ] | se(v) |
| **sii\_2d\_size\_x\_delta**[ tileID ][ patchIdx ][ subIdx ] | se(v) |
| **sii\_2d\_size\_y\_delta**[ tileID ][ patchIdx ][ subIdx ] | se(v) |
| } |  |

#### Texture projection merge information syntax

|  |  |
| --- | --- |
| texture\_projection\_merge\_information( tileID, patchIdx ) { | **Descriptor** |
| **tpmi\_subpatch\_merge\_present\_flag**[ tileID ][ patchIdx ] | u(1) |
| if( tpmi\_submesh\_merge\_present\_flag[ tileID ][ patchIdx ] ) { |  |
| **tpmi\_update\_subpatch\_count\_minus1**[ tileID ][ patchIdx ] | ue(v) |
| for( subIdx = 0;   subIdx < tpmi\_update\_subpatch\_count\_minus1[ tileID ][ patchIdx ];  subIdx++) |  |
| subpatch\_merge\_information( tileID, patchIdx, subIdx ) |  |
| } |  |
| } |  |
| } |  |

#### Sub-patch merge information syntax

|  |  |
| --- | --- |
| subpatch\_merge\_information( tileID, patchIdx , subIdx ) { | **Descriptor** |
| **smi\_subpatch\_index**[ tileID ][ patchIdx ][ subIdx ] | se(v) |
| **smi\_2d\_pos\_x\_delta**[ tileID ][ patchIdx ][ subIdx ] | se(v) |
| **smi\_2d\_pos\_y\_delta**[ tileID ][ patchIdx ][ subIdx ] | se(v) |
| **smi\_2d\_size\_x\_delta**[ tileID ][ patchIdx ][ subIdx ] | se(v) |
| **smi\_2d\_size\_y\_delta**[ tileID ][ patchIdx ][ subIdx ] | se(v) |
| } |  |

## Semantics

### General semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4 apply.

### V3C unit semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.2 apply.

### Byte alignment semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.3 apply.

### V3C parameter set semantics

#### General V3C parameter set semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.4.1 apply.

#### Profile, tier, and level semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.4.2 apply.

#### Occupancy information semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.4.3 apply.

#### Geometry information semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.4.4 apply.

#### Attribute information semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.4.5 apply.

#### Profile toolset constraints information semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.4.6 apply.

#### Packing information semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.4.7 apply.

#### VPS extension semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.4.8 apply.

#### Packed video extension semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.4.9 apply.

#### Length alignment semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.4.10 apply.

#### V3C parameter set V-DMC extension semantics

**vps\_ext\_bmesh\_data\_substream\_codec\_id**[ j ] indicates the identifier of the codec used to compress the basemesh data for the atlas with atlas ID j. This codec may be identified through the profiles, a component codec mapping SEI message, or through means outside this document.

**vps\_ext\_bmesh\_geometry\_3d\_bit\_depth\_minus1**[ j ] indicates the bit depth of vertex coordinates in the basemesh for the atlas with atlas ID j. vps\_ext\_bmesh\_geometry\_3d\_bit\_depth\_minus1[ j ]shall be in the range of 0 to 31, inclusive.

**vps\_ext\_bmesh\_geometry\_3d\_msb\_align\_flag**[ j ] indicates how the decoded vertex coordinate samples in the basemesh for the atlas with atlas ID j are converted to vertex coordinates samples at the nominal bit depth, as specified in Annex B.

**vps\_ext\_bmesh\_data\_attribute\_count**[ j ] indicates the number of attributes in the basemesh for the atlas with atlas ID j. vps\_ext\_bmesh\_data\_attribute\_count[ j ] shall be in the range of 0 to 255, inclusive. The value indicated by vps\_ext\_bmesh\_data\_attribute\_count shall be less than or equal to the number of attributes in the basemesh sub-bitstream.

**vps\_ext\_bmesh\_attribute\_index**[ j ][ i ] indicates the index of the basemesh attribute carried in the basemesh bitstream for i-th attribute for atlas with atlasID j.

**vps\_ext\_bmesh\_attribute\_bit\_depth\_minus1**[ j ][ i ] indicates the bit depth of the i-th attribute in the basemesh for the atlas with atlas ID j. vps\_ext\_bmesh\_attribute\_bit\_depth\_minus1 [ j ]shall be in the range of 0 to 31, inclusive.

**vps\_ext\_bmesh\_attribute\_msb\_align\_flag**[ j ][ i ] indicates how the decoded attribute samples in the basemesh for the atlas with atlas ID j and an attribute with index i are converted to attribute samples at the nominal attribute bit depth, as specified in Annex B.

**vps\_ext\_bmesh\_attribute\_type**[ j ][ i ] indicates the attribute type of the i-th attribute signalled through the basemesh for the atlas with atlas ID j. Table 1 describes the list of supported attributes and their relationship with vps\_ext\_bmesh\_attribute\_type [ j ][ i ]

Table 1 – Basemesh attribute types

|  |  |  |
| --- | --- | --- |
| **vps\_ext\_bmesh\_attribute\_type [ j ][ i ]** | **Identifier** | **Attribute type** |
| 0 | ATTR\_TEXTURE | Texture |
| 1 | ATTR\_MATERIAL\_ID | Material ID |
| 2 | ATTR\_TRANSPARENCY | Transparency |
| 3 | ATTR\_REFLECTANCE | Reflectance |
| 4 | ATTR\_NORMAL | Normals |
| 5 | ATTR\_TEXCOORD | Texture coordinate |
| 6 | ATTR\_FACEGROUP\_ID | Face Group ID |
| 7..14 | ATTR\_RESERVED | Reserved |
| 15 | ATTR\_UNSPECIFIED | Unspecified |

**vps\_ext\_attribute\_frame\_width**[ j ][ i ] indicates the width of the Attribute Video Data unit with index i in terms of integer luma samples for the atlas with atlas ID j.

**vps\_ext\_attribute\_frame\_height**[ j ][ i ] indicates the height of the Attribute Video Data unit with index i in terms of integer luma samples for the atlas with atlas ID j.

### NAL unit semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.5 apply.

### Raw byte sequence payloads, trailing bits, and byte alignment semantics

#### Atlas sequence parameter set RBSP semantics

##### General atlas sequence parameter set RBSP semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.6.1.1 apply.

##### Point local reconstruction information semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.6.1.2 apply.

##### Atlas sequence parameter set V-DMC extension semantics

**asve\_subdivision\_iteration\_count** indicates the number of iterations used for the subdivision.

**asve\_lod\_adaptive\_subdivision\_flag** equal to 1 indicates that subdivision method is signaled for each subdivision iteration. asve\_lod\_adaptive\_subdivision\_flag equal to 0 indicates that the same subdivision method is applied for each subdivision iterations.

**asve\_subdivision\_method**[ i ] indicates the identifier of the methodto subdivide the meshes associated with the current atlas sequence parameter set at the subdivision with subdivision index equal to i. Table 2 describes the list of supported subdivision methods and their relationship with asve\_subdivision\_method.

Table 2 – Subdivision methods list

|  |  |
| --- | --- |
| asve\_subdivision\_method  afve\_subdivision\_method  mdu\_subdivision\_method | Name of subdivision method |
| 0 | NONE |
| 1 | MIDPOINT |
| 2 | LOOP |
| 3 | LS3 |
| 4..7 | RESERVED |

**asve\_1d\_displacement\_flag** equal to 1 specifies that only the first component of the displacement is present in the compressed geometry video. The remaining two components are inferred to be 0. asve\_1D\_displacement\_flag equal to 0 specifies that all 3 components of the displacement are present in the compressed geometry video. The following variable is defined:

DisplacementDim = asve\_1d\_displacement\_flag ? 1 : 3

**asve\_displacement\_reference\_qp** specifies the initial value of QuantizationParameter for current frame. When not present asve\_displacement\_reference\_qp is set to be equal to 49.

**asve\_quantization\_parameters\_present\_flag** equal to 1 specifies that the vdmc\_quantization\_parameters( ) syntax structure is present. asve\_quantization\_parameters\_present\_flag equal to 0 specifies that the vdmc\_quantization\_parameters ( ) syntax structure is not present.

**asve\_transform\_method** indicates the identifier of the transform applied to the displacement. Table 3 describes the list of supported transforms and their relationship with asve\_transform\_method.

Table 3 – Transform methods list

|  |  |
| --- | --- |
| asve\_transform\_method  afve\_transform\_method  mdu\_transform\_method | Name of transform method |
| 0 | NONE |
| 1 | LINEAR\_LIFTING |
| 2..7 | RESERVED |

**asve\_lifting\_offset\_present\_flag** equal to 1 indicates that the lifting offset parameters may be present in vdmc\_lifting\_transform\_parameters syntax structure. asve\_lifting\_offset\_present\_flag equal to 0 that the lifting offset parameters are not present in vdmc\_lifting\_transform\_parameters syntax structure.

**asve\_num\_attribute\_video** indicates the number of the attributes signalled through the video sub-bitstreams. It is a requirement of V3C bitstream conformance that the value of asve\_num\_attribute\_video shall be equal to the value of ai\_attribute\_count[ j ], where j is the ID of the current atlas.

**asve\_attribute\_type\_id**[ i ] indicates the attribute type of the Attribute Video Data unit with index i. ISO/IEC 23090-5(2E):2023 Table 4 describes the list of supported attributes. It is a requirement of V3C bitstream conformance that the value of asve\_attribute\_type\_id[ i ] shall be equal to the value of ai\_attribute\_type\_id[ j ][ i ], where j is the ID of the current atlas.

**asve\_attribute\_frame\_width**[ i ]indicates the atlas frame width of the Attribute Video Data unit with index i in terms of integer luma samples for the atlas with atlas ID j. It is a requirement of V3C bitstream conformance that the value of asve\_attribute\_frame\_width[ i ] shall be equal to the value of vps\_ext\_attribute\_frame\_width[ j ][ i ], where j is the ID of the current atlas.

**asve\_attribute\_frame\_height**[ i ] indicates the atlas frame height of the Attribute Video Data unit with index i in terms of integer luma samples for the atlas with atlas ID j. It is a requirement of V3C bitstream conformance that the value of asve\_attribute\_frame\_height[ i ] shall be equal to the value of vps\_ext\_attribute\_frame\_height[ j ][ i ], where j is the ID of the current atlas.

**asve\_attribute\_subtexture\_enabled\_flag**[ i ] equal to 0 specifies that the information of the area in the image corresponding to a submesh is not present for the attribute signalled in the Attribute Video Data unit with index i in meshpatch data unit. asve\_attribute\_subtexture\_enabled\_flag[ i ] equal to 1 specifies the information of the area in the image corresponding to a submesh is present for the attribute signalled in the Attribute Video Data unit with index i in meshpatch data unit.

**asve\_displacement\_id\_present\_flag** equal to 1 displacement data patch is identified by ID. asve\_displacement\_id\_present\_flag equal to 0 displacement data is identified by its position.**asve\_packing\_method** equal to 0 specifies that the displacement component samples are packed in ascending order, asve\_packing\_method equal to 1 specifies that the displacement component samples are packed in descending order.

**asve\_projection\_texcoord\_enable\_flag** equal to 0 specifies that the texture coordinates may be transmitted in the basemesh , asve\_projection\_texcoord\_enable\_flag equal to 1 specifies that the texture coordinates will be derived using projection parameters from the meshpatch data unit.

**asve\_projection\_texcoord\_mapping\_present\_flag** equals to 1 indicates the asve\_projection\_texcoord\_mapping\_attribute\_index is present. If asve\_projection\_texcoord\_mapping\_present\_flag equals to 0 indicates the asve\_projection\_texcoord\_mapping\_attribute\_index is not present, and the face ID is derived implicitly using connected components.

**asve\_projection\_texcoord\_mapping\_attribute\_index** indicates the index of the basemesh attribute, which will be used to map a set of faces to a sub-patch.

**asve\_projection\_texcoord\_output\_attribute\_index** indicates the index of the texture coordinate attribute carried in the reconstructed basemesh. The value indicated by asve\_projection\_texcoord\_output\_attribute\_index shall be less than or equal to the vps\_ext\_bmesh\_data\_attribute\_count. When not present asve\_projection\_texcoord\_output\_attribute\_index is set to be equal to 0.

**asve\_projection\_texcoord\_output\_bit\_depth\_minus1** indicates the bit depth of the texture coordinate generated by orthoAtlas. asve\_projection\_texcoord\_output\_bit\_depth\_minus1shall be in the range of 0 to 31, inclusive.

**asve\_projection\_texcoord\_scale\_factor** indicates the value of the scaling factor variable TexcoordProjectionScaleFactor that is used for texture coordinate derivation from geometry projection.

**asve\_vdmc\_vui\_parameters\_present\_flag** equal to 1 specifies that the vdmc\_vui\_parameters( ) syntax structure, as specified in Annex G, is present. asve\_vdmc\_vui\_parameters\_present\_flag equal to 0 specifies that the vdmc\_vui\_parameters( ) syntax structure, as specified in Annex G, is not present.

##### Quantization parameters semantics

**vqp\_lod\_quantization\_flag**[ qpIndex ] equal to 1 indicates that the quantization parameter will be sent per level-of-detail using delta coding. vqp\_lod\_quantization\_flag[ qpIndex ] equal to 0 indicates that the quantization parameter will be the same for all level-of-details. qpIndex is the index of the quantization parameter set.

**vqp\_bitdepth\_offset**[ qpIndex ] indicates the bit depth offset value applied to the quantization process of the displacements. qpIndex is the index of the quantization parameter set.

**vqp\_quantization\_parameters**[ qpIndex ][ k ] indicates the quantization parameter to be used for the inverse quantization of the kth-component of the displacements. The value of vqp\_quantization\_parameters[ qpIndex ][ k ] shall be in the range of 0 to 100, inclusive. qpIndex is the index of the quantization parameter set.

**vqp\_log2\_lod\_inverse\_scale**[ qpIndex ][ k ] indicates the scaling factor applied to the quantization process of the kth-component of the displacements for each level of detail. qpIndex is the index of the quantization parameter set.

**vqp\_lod\_delta\_quantization\_parameter\_value**[ qpIndex ][ i ][ k ] specifies the absolute difference of quantization parameter value between the value asve\_displacement\_reference\_qp and the quantization parameter for the ith-layer and kth-component. When not present, the value of vqp\_lod\_delta\_quantization\_parameter\_value[ qpIndex ][ i ][ k ] is inferred as 0. The value of QuantizationParameter of each LoD layer shall be in the range of 0 to 100. qpIndex is the index of the quantization parameter set.

**vqp\_lod\_delta\_quantization\_parameter\_sign**[ qpIndex ][ i ][ k ] specifies the sign of difference of quantization parameter value between the value asve\_displacement\_reference\_qp and the quantization parameter for the ith-layer and kth-component. vqp\_lod\_delta\_quantization\_parameter\_sign[ qpIndex ][ i ][ k ] equal to 0 indicate the difference is positive. vqp\_lod\_delta\_quantization\_parameter\_sign[ qpIndex ][ i ][ k ] equal to 1 indicate the difference is negative. When not present, the value of vqp\_lod\_delta\_quantization\_parameter\_sign[ qpIndex ][ i ][ k ] is inferred as 0. qpIndex is the index of the quantization parameter set.

**vqp\_direct\_quantization\_enabled\_flag**[ qpIndex ] equal to 1 indicates that the inverse scale factor is derived from the signaled displacement quantization parameter directly and computed as follows:

InverseScale[ qpIndex ][ i ][ k ] = 1 ÷ QuantizationParameter[ qpIndex ][ i ][ k ]

vqp\_direct\_quantization\_enabled\_flag[ qpIndex ] equal to 0 indicates that the inverse scale factor shall be computed as follows:

bitDepthPosition = asps\_geometry\_3d\_bit\_depth\_minus1 + 1  
 InverseScale[ qpIndex ][ i ][ k ] = pow(0.5, 16 + vqp\_bitdepth\_offset[ qpIndex ] –   
 bitDepthPosition + (4 – QuantizationParameter[ qpIndex ][ i ][ k ]) ÷ 6)

qpIndex is the index of the quantization parameter set.

**vqp\_inverse\_quantization\_offset\_enable\_flag**[ qpIndex ] equal to 0 specifies that the inverse quantization offset may not be applied for compensating the inverse quantized wavelet-transformed coefficients of the displacement. vqp\_inverse\_quantization\_offset\_enable\_flag[ qpIndex ] equal to 1 specifies that the inverse quantization offset is applied for compensating the inverse quantized wavelet-transformed coefficients of the displacement.

**vqp\_inverse\_quantization\_offset\_sign\_delta**[ qpIndex ][ i ][ j ][ k ] indicates the difference of signs of the inverse quantization offset value which is used for compensating the inverse quantized wavelet-transformed displacement coefficients, located in zone k (dead-zone: 0, non-dead-zone-positive: 1, non-dead-zone-negative: 2), with displacement dimension j, between LoD i and LoD i-1 when i is non-zero value. When i is zero, it indicates the absolute value for the sign.

**vqp\_inverse\_quantization\_offset\_value\_log2\_prec1\_delta**[ qpIndex ][ i ][ j ][ k ] indicates difference of the values of the inverse quantization offset value for a first precision level located in zone k (dead-zone: 0, non-dead-zone-positive: 1, non-dead-zone-negative: 2), with displacement dimension j, between LoD i and LoD i-1 when i is non-zero value. When i is zero, it indicates the absolute value for the offset value for a first precision level associated with LoD 0.

**vqp\_inverse\_quantization\_offset\_value\_log2\_prec2\_delta**[ qpIndex ][ i ][ j ][ k ] indicates the value of the inverse quantization offset value for a second precision level located in zone k (dead-zone: 0, non-dead-zone-positive: 1, non-dead-zone-negative: 2), with displacement dimension j, between LoD i and LoD i-1 when i is non-zero value. When i is zero, it indicates the absolute value for the offset value for a second precision level associated with LoD 0.

##### Lifting transform parameters semantics

**vltp\_lifting\_main\_param\_flag**[ ltpIndex ]equal to 1 indicates that the main lifting transform parameters are signaled. vltp\_lifting\_main\_param\_flag[ ltpIndex ] equal to 0 indicates that main lifting parameters are not signaled. When not present, vltp\_lifting\_main\_param\_flag[ ltpIndex ] is inferred to be equal to 1. ltpIndex is the index of the lifting transform parameter set.

**vltp\_lifting\_offset\_values\_num**[ ltpIndex ][ i ] indicates the numerator of the lifting offset used to address the bias in the lifting transform of the ith level of detail. ltpIndex is the index of the lifting transform parameter set.

**vltp\_lifting\_offset\_values\_deno\_minus1**[ ltpIndex ][ i ] plus 1 indicates the denominator of the lifting offset used to address the bias in the lifting transform of the ith level of detail. ltpIndex is the index of the lifting transform parameter set.

if( ltpIndex == 2 ){  
 VltpLiftingOffset[ i ] = vltp\_lifting\_offset\_values\_num[ ltpIndex ][ i ] ÷  
 (vltp\_lifting\_offset\_values\_deno\_minus1[ ltpIndex ][ i ] + 1)  
 } else {  
 VltpLiftingOffset[ i ] = 0  
 }

**vltp\_skip\_update\_flag**[ ltpIndex ] equal to 1 indicates the update step of the lifting transform applied to the displacement is skipped. vltp\_skip\_update\_flag[ ltpIndex ] equal to 0 indicates the update step of the lifting transform applied to the displacement is not skipped. ltpIndex is the index of the lifting transform parameter set.

**vltp\_lod\_lifting\_parameter\_flag**[ ltpIndex ] equal to 1 indicates the lifting transform parameters are signalled at LoD level. vltp\_lod\_lifting\_parameter\_flag[ ltpIndex ] equal to 0 indicates the lifting transform parameters applies across LoDs. ltpIndex is the index to lifting transform parameter set.

**vltp\_adaptive\_update\_weight\_flag**[ ltpIndex ][ i ] equal to 1 indicates the update weight is represented as the ratio of numerator and denominator values. vltp\_adaptive\_update\_weight\_flag[ i ] equal to 0 indicates the update weight at ith level of detail is signalled as single value. ltpIndex is the index of the lifting transform parameter set.

**vltp\_valence\_update\_weight\_flag**[ ltpIndex ][ i ] equal to 1 indicates the valence adaptive lifting update weight is performed. vltp\_lifting\_valence\_update\_weight\_flag[ ltpIndex ][ I ] equal to 0 specifies the valence adaptive lifting update weight is not performed. ltpIndex is the index of the lifting transform parameter set.

**vltp\_lifting\_update\_weight\_numerator**[ ltpIndex ][ i ] indicates the numerator of the weight coefficients used for the update filter of the lifting transform of the ith level of detail. ltpIndex is the index of the lifting transform parameter set.

**vltp\_lifting\_update\_weight\_denominator\_minus1**[ ltpIndex ][ i ] plus 1 indicates the denominator of the weight coefficients used for the update filter of the wavelet transform of the ith level of details. ltpIndex is the index of the lifting transform parameter set.

**vltp\_valence\_update\_weight**[ ltpIndex ][ i ] indicates the weighting coefficients used for the valence adaptive lifting update of the wavelet transform of the ith level of detail. ltpIndex is the index of the lifting transform parameter set.

**vltp\_log2\_lifting\_update\_weight**[ ltpIndex ][ i ] indicates the weighting coefficients used for the update filter of the wavelet transform of the ith level of detail. ltpIndex is the index of the lifting transform parameter set.

**vltp\_log2\_lifting\_prediction\_weight**[ ltpIndex ][ i ] the weighting coefficients used for the prediction filter of the wavelet transform of the ith level of detail. ltpIndex is the index of the lifting transform parameter set.

#### Atlas frame parameter set RBSP semantics

##### General atlas frame parameter set RBSP semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.6.2.1 apply.

##### Atlas frame tile information semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.6.2.2 apply.

##### Atlas frame parameter set V-DMC extension semantics

**afve\_overriden\_flag** equal to 1 indicates the parameters afve\_subdivision\_enable\_flag, afve\_quantization\_parameters\_enable\_flag, afve\_transform\_method\_enable\_flag, and afve\_transform\_parameters\_enable\_flag are present in the atlas frame parameter set extension.

**afve\_subdivision\_enable\_flag** equal to 1 indicates afve\_subdivision\_method and afve\_subdivision\_iteration\_count\_minus1 are present in the atlas frame parameter set extension. When afve\_subdivision\_enable\_flag is not present, its value is inferred to be equal to 0.

**afve\_quantization\_parameters\_enable\_flag** equal to 1 indicates vdmc\_quantization\_parameters(qpIndex, subdivisionCount ) syntax structure is present in the atlas frame parameter set extension. afve\_quantization\_parameters\_enable\_flag equal to 0 indicates vdmc\_quantization\_parameters(qpIndex, subdivisionCount ) syntax structure is not present in the atlas frame parameter set extension. When afve\_quantization\_parameters\_enable\_flag is not present, its value is inferred to be equal to 0.

**afve\_transform\_method\_enable\_flag** equal to 1 indicates afve\_transform\_method is present in the atlas frame parameter set extension. When afve\_transform\_method\_enable\_flag is not present, its value is inferred to be equal to 0.

**afve\_transform\_parameters\_enable\_flag** equal to 1 indicates vdmc\_lifting\_transform\_parameters(lptIndex, subdivisionCount ) syntax structure is present in the atlas frame parameter set extension. When afve\_transform\_parameters\_enable\_flag is not present, its value is inferred to be equal to 0.

**afve\_subdivision\_iteration\_count** indicates the number of iterations used for the subdivision. When afve\_subdivision\_iteration\_count is not present, its value is inferred to be equal to asve\_subdivision\_iteration\_count.

**afve\_lod\_adaptive\_subdivision\_flag** equal to 1 indicates that subdivision method is signaled for each subdivision iteration. afve\_lod\_adaptive\_subdivision\_flag equal to 0 indicates that the same subdivision method is applied for each subdivision iterations.

**afve\_subdivision\_method**[ i ]indicates the identifier of the method to subdivide the meshes associated with the current atlas frame parameter set at the subdivision with subdivision index equal to i. When afve\_subdivision\_method[ i ] is not present, afve\_subdivision\_method[ i ] is equal to asve\_subdivision\_method[ i ]. Table 2 describes the list of supported subdivision methods and their relationship with afve\_subdivision\_method.

**afve\_transform\_method** indicates the identifier of the transform applied to the displacement. When afve\_transform\_method is not present, afve\_transform\_method is equal to asve\_transform\_method. Table 3 describes the list of supported transforms and their relationship with afve\_transform\_method.

**afve\_projection\_texcoord\_present\_flag**[ i ] equal to 1 specifies that the texture coordinates for the submesh with index i will be derived using geometry projection. If afve\_projection\_texcoord\_present\_flag[ i ] is equal to 0, then the texture coordinates are present in the submesh with index i and will not be derived by geometry projection.

TexcoordProjectionFlag[ i ] = afve\_projection\_texcoord\_present\_flag[ i ]

When afve\_projection\_texcoord\_present\_flag[ i ] is not present, then is inferred to be equal to 0

**afve\_projection\_texcoord\_width\_normalization**[ i ] indicates the value used for texture coordinate derivation from geometry projection.

TexcoordProjectionWidthNormalization[ i ] =  
 afve\_projection\_texcoord\_width\_normalization[ i ]

**afve\_projection\_texcoord\_height\_normalization**[ i ] indicates the value used for texture coordinate derivation from geometry projection.

TexcoordProjectionHeightNormalization[ i ] =  
 afve\_projection\_texcoord\_height\_normalization[ i ]

**afve\_projection\_texcoord\_gutter**[ i ] indicates the value used for texture coordinate derivation from geometry projection.

TexcoordProjectionGutter[ i ] = afve\_projection\_texcoord\_gutter[ i ]

##### Atlas frame attribute tile information semantics

**afati\_single\_tile\_in\_atlas\_frame\_flag**[ attrIdx ]equal to 1 specifies that there is only one tile for the attribute signalled in the Attribute Video Data unit with index attrIdx. afati\_single\_tile\_in\_atlas\_frame\_flag[ attrIdx ] equal to 0 specifies that there may be more than one tile in for the attribute signalled in the Attribute Video Data unit with index attrIdx.

**afati\_uniform\_partition\_spacing\_flag**[ attrIdx ] equal to 1 specifies that the tile partitioning of an atlas for the attribute signalled in the Attribute Video Data unit with index attrIdx uses a method that distributes column and row partition boundaries uniformly across the attribute atlas frame. The information corresponding to these boundaries is signalled using the syntax elements afati\_partition\_cols\_width\_minus1[ attrIdx ] and afati\_partition\_rows\_height\_minus1[ attrIdx ], respectively. afati\_uniform\_partition\_spacing\_flag[ attrIdx ] equal to 0 specifies that the tile partitioning of an atlas for the attribute signalled in the Attribute Video Data unit with index attrIdx uses a method that may result in column and row partition boundaries that may or may not be distributed uniformly across the atlas frame. In this case, these boundaries are signalled using the syntax elements afati\_num\_partition\_columns\_minus1[ attrIdx ] and afati\_num\_partition\_rows\_minus1[ attrIdx] and a list of syntax element pairs afati\_partition\_column\_width\_minus1[ attrIdx ][ i ] and afati\_partition\_row\_height\_minus1[ attrIdx ][ i ]. When not present, the value of afati\_ti\_uniform\_partition\_spacing\_flag[ attrIdx ] is inferred to be equal to 1.

**afati\_partition\_cols\_width\_minus1**[ attrIdx ] plus 1 specifies the width of the attribute tile partition columns in the Attribute Video Data unit with index attrIdx, excluding the right-most attribute tile partition column of the attribute atlas frame in units of 64 samples when afati\_uniform\_partition\_spacing\_flag[ attrIdx ] is equal to 1. The value of afati\_partition\_cols\_width\_minus1[ attrIdx ] shall be in the range of 0 to asve\_attribute\_frame\_width[ attrIdx ] / 64 – 1, inclusive. When not present, the value of afati\_partition\_cols\_width\_minus1[ attrIdx ] is inferred to be equal to asve\_attribute\_frame\_width[ attrIdx ] / 64 – 1.

**afati\_partition\_rows\_height\_minus1**[ attrIdx ] plus 1 specifies the height of the attribute tile partition rows in the Attribute Video Data unit with index attrIdx, excluding the bottom attribute tile partition row of the attribute atlas frame in units of 64 samples when afati\_uniform\_partition\_spacing\_flag[ attrIdx ] is equal to 1. The value of afati\_partition\_rows\_height\_minus1[ attrIdx ] shall be in the range of 0 to asve\_attribute\_frame\_height[ attrIdx ] / 64 – 1, inclusive. When not present, the value of afati\_partition\_rows\_height\_minus1[ attrIdx ] is inferred to be equal to asve\_attribute\_frame\_height[ attrIdx ] / 64 – 1.

**afati\_num\_partition\_columns\_minus1**[ attrIdx ] plus 1 specifies the number of attribute tile partition columns in the Attribute Video Data with index attrIdx, used to partition the attribute atlas frame when afati\_uniform\_partition\_spacing\_flag[ attrIdx ] is equal to 0. The value of afati\_num\_partition\_columns\_minus1[ attrIdx ] shall be in the range of 0 to asve\_attribute\_frame\_width[ attrIdx ] / 64 – 1, inclusive. If afati\_single\_tile\_in\_atlas\_frame\_flag[ attrIdx ] is equal to 1, the value of afati\_num\_partition\_columns\_minus1[ attrIdx ] is inferred to be equal to 0. Otherwise, when afati\_uniform\_partition\_spacing\_flag[ attrIdx ] is equal to 1, the value of afati\_num\_partition\_columns\_minus1[ attrIdx ] is inferred as specified in subclause 7.7.

**afati\_num\_partition\_rows\_minus1**[ attrIdx ] plus 1 specifies the number of attribute tile partition rows in the Attribute Video Data with index attrIdx, used to partition the attribute atlas frame when afati\_uniform\_partition\_spacing\_flag[ attrIdx ] is equal to 0. The value of afati\_num\_partition\_rows\_minus1[ attrIdx ] shall be in the range of 0 to asve\_attribute\_frame\_height[ attrIdx ] / 64 – 1, inclusive. If afati\_single\_tile\_in\_atlas\_frame\_flag[ attrIdx ] is equal to 1, the value of afati\_num\_partition\_rows\_minus1[ attrIdx ] is inferred to be equal to 0. Otherwise, when afati\_uniform\_partition\_spacing\_flag[ attrIdx ] is equal to 1, the value of afati\_num\_partition\_rows\_minus1[ attrIdx ] is inferred as specified in subclause 7.7.

The variable NumPartitionsInAtlasFrameAtt[ attrIdx ] is set equal to NumPartitionColumnsAtt[ attrIdx ] \* NumPartitionRowsAtt[ attrIdx ], where NumPartitionColumnsAtt[ attrIdx ] and NumPartitionRowsAtt[ attrIdx ] are derived in subclause 7.6.

When afati\_single\_tile\_in\_atlas\_frame\_flag[ attrIdx ] is equal to 0, NumPartitionsInAtlasFrameAtt[ attrIdx ] shall be greater than 1.

**afati\_partition\_column\_width\_minus1**[ attrIdx ][ i ] plus 1 specifies the width of the i-th attribute tile partition column in the Attribute Video Data with index attrIdx in units of 64 samples.

**afati\_partition\_row\_height\_minus1**[ attrIdx ][ i ] plus 1 specifies the height of the i-th attribute tile partition row in the Attribute Video Data with index attrIdx in units of 64 samples.

**afati\_single\_partition\_per\_tile\_flag**[ attrIdx ] equal to 1 specifies that each attribute tile for the attribute signalled in the Attribute Video Data unit with index attrIdx includes one tile partition. afati\_single\_partition\_per\_tile\_flag[ attrIdx ] equal to 0 specifies that an attribute tile in the Attribute Video Data unit with index attrIdx may include more than one attribute tile partition. When not present, the value of afati\_single\_partition\_per\_tile\_flag[ attrIdx ] is inferred to be equal to 1.

**afati\_num\_tiles\_in\_atlas\_frame\_minus1**[ attrIdx ] plus 1 specifies the number of attribute tiles in each attribute atlas frame for the attribute signalled in the Attribute Video Data unit with index attrIdx. The value of afati\_num\_tiles\_in\_atlas\_frame\_minus1[ attrIdx ] shall be in the range of 0 to NumPartitionsInAtlasFrameAtt[ attrIdx ] – 1, inclusive. When not present and afati\_single\_partition\_per\_tile\_flag[ attrIdx ] is equal to 1, the value of afati\_num\_tiles\_in\_atlas\_frame\_minus1[ attrIdx ] is inferred to be equal to NumPartitionsInAtlasFrameAtt[ attrIdx ] – 1.

**afati\_top\_left\_partition\_idx**[ attrIdx ][ i ] specifies the partition index of the attribute tile partition located at the top-left corner of the i-th tile in the Attribute Video Data with index attrIdx. The value of afati\_top\_left\_partition\_idx[ attrIdx ][ i ] shall be in the range of 0 to NumPartitionsInAtlasFrameAtt[ attrIdx ] – 1, inclusive. When not present, the value of afati\_top\_left\_partition\_idx[ attrIdx ][ i ] is inferred to be equal to i. The length of the afati\_top\_left\_partition\_idx[ attrIdx ][ i ]syntax element is Ceil( Log2( NumPartitionsInAtlasFrameAtt[ attrIdx ] ) ) bits.

**afati\_bottom\_right\_partition\_column\_offset**[ attrIdx ][ i ] specifies the offset between the column position of the attribute tile partition in the Attribute Video data with index attrIdx located at the bottom-right corner of the i-th attribute tile and the column position of the attribute tile partition with partition index equal to afati\_bottom\_right\_partition\_column\_offset[ attrIdx ][ i ]. When afati\_single\_partition\_per\_tile\_flag[ attrIdx ] is equal to 1, the value of afati\_bottom\_right\_partition\_column\_offset[ attrIdx ][ i ] is inferred to be equal to 0.

**afati\_bottom\_right\_partition\_row\_offset**[ attrIdx ][ i ] specifies the offset between the row position of the attribute tile partition in the Attribute Video Data with index attridx located at the bottom-right corner of the i-th attribute tile and the row position of the attribute tile partition with partition index equal to afati\_top\_left\_partition\_idx[ attrIdx ][ i ]. When afati\_single\_partition\_per\_tile\_flag[ attrIdx ] is equal to 1, the value of afati\_bottom\_right\_partition\_row\_offset[ attrIdx ][ i ] is inferred to be equal to 0.

The variables topLeftColumnAtt[ attrIdx ][ i ], topLeftRowAtt[ attrIdx ][ i ], bottomRightColumnAtt[ attrIdx ][ i ], and bottomRightRowAtt[ attrIdx ][ i ], which specify the corresponding tile column and row positions for the top left and bottom right attribute tiles in an attribute tile for the attribute signalled in the Attribute Video Data unit with index attrIdx are computed as follows:

topLeftColumnAtt[ attrIdx ][ i ] =  
 afati\_top\_left\_partition\_idx[ attrIdx ][ i ] % NumPartitionColumnsAtt[ attrIdx ]  
 topLeftRowAtt[ attrIdx ][ i ] =  
 afati\_top\_left\_partition\_idx[ attrIdx ][ i ] / NumPartitionColumnsAtt[ attrIdx ]  
 bottomRightColumnAtt[ attrIdx ][ i ] = topLeftColumnAtt[ attrIdx ][ i ] +  
 afati\_bottom\_right\_partition\_column\_offset[ attrIdx ][ i ]  
 bottomRightRowAtt[ attrIdx ][ i ] = topLeftRowAtt[ attrIdx ][ i ] +  
 afati\_bottom\_right\_partition\_row\_offset[ attrIdx ][ i ]

It is a requirement of bitstream conformance that the values of bottomRightColumnAtt[ attrIdx ][ i ] and bottomRightRowAtt[ attrIdx ][ i ] shall be smaller or equal to ( asve\_attribute\_frame\_width[ attrIdx ] + 63 ) / 64 – 1 and ( asve\_attribute\_frame\_height[ attrIdx ] + 63 ) / 64 – 1, respectively.

It is also a requirement of bitstream conformance that there shall not be a value of j, where j != i, that satisfies either one of these properties:

topLeftColumnAtt[ attrIdx ][ i ] <=  
 topLeftColumnAtt[ attrIdx ][ j ] <= bottomRightColumnAtt[ attrIdx ][ i ]  
 topLeftRowAtt[ attrIdx ][ i ] <=   
 topLeftRowAtt[ attrIdx ][ j ] <= bottomRightRowAtt[ attrIdx ][ i ]

The variables TileOffsetXAtt[ attrIdx ][ i ], TileOffsetYAtt[ attrIdx ][ i ], TileWidthAtt[ attrIdx ][ i ], and TileHeightAtt[ attrIdx ][ i ], which specify the horizontal position, vertical position, width, and height of an attribute tile respectively, are then computed, using the lists PartitionWidthAtt[ attrIdx ][ i ] and PartitionHeightAtt[ attrIdx ][ j ] that are computed as specified in subclause 7.6, as follows:

TileOffsetXAtt[ attrIdx ][ i ] =  
 PartitionPosXAtt[ attrIdx ][ topLeftColumn[ attrIdx ][ i ] ]  
 TileOffsetYAtt[ attrIdx ][ i ] =  
 PartitionPosYAtt[ attrIdx ][ topLeftColumn[ attrIdx ][ i ] ]  
 TileWidthAtt[ attrIdx ][ i ] = 0   
 TileHeightAtt[ attrIdx ][ i ] = 0  
 jStart = topLeftColumnAtt[ attrIdx ][ i ]  
 jEnd = bottomRightColumnAtt[ attrIdx ][ i ]  
 for( j = jStart; j <= jEnd; j++) {  
 TileWidthAtt[ attrIdx ][ i ] += PartitionWidthAtt[ attrIdx ][ j ]  
 }  
 jStart = topLeftRowAtt[ attrIdx ][ i ]  
 jEnd = bottomRightRowAtt[ attrIdx ][ i ]  
 for( j = jStart; j <= jEnd; j++) {  
 TileHeightAtt[ attrIdx ][ i ] += PartitionHeightAtt[ attrIdx ][ j ]  
 }

**afati\_signalled\_tile\_id\_flag**[ attrIdx ] equal to 1 specifies that the attribute tile ID for each attribute tile in the Attribute Video Data with index attrIdx is signalled. afati\_signalled\_tile\_id\_flag equal to 0 specifies that attribute tile IDs are not signalled.

**afati\_signalled\_tile\_id\_length\_minus1**[ attrIdx ] plus 1 specifies the number of bits used to represent the syntax element afati\_tile\_id[ attrIdx ][ i ] when present. The value of afati\_signalled\_tile\_id\_length\_minus1[ attrIdx ] shall be in the range of 0 to 15, inclusive. When not present, the value of afati\_signalled\_tile\_id\_length\_minus1[ attrIdx ] is inferred to be equal to Ceil( Log2( afati\_num\_tiles\_in\_atlas\_frame\_minus1[ attrIdx ] + 1 ) ) – 1.

**afati\_tile\_id**[ attrIdx ][ i ] specifies the attribute tile ID of the i-th attribute tile i the Attribute Video Data with index attridx. When not present, the value of afati\_tile\_id[ attrIdx ][ i ] is inferred to be equal to i, for each i in the range of 0 to afati\_num\_tiles\_in\_atlas\_frame\_minus1[ attrIdx ], inclusive. It is a requirement of bitstream conformance that afati\_tile\_id[ attrIdx ][ i ] shall not be equal to afati\_tile\_id[ attrIdx ][ j ] for all i != j. The length of the afati\_tile\_id[ attrIdx ][ i ] syntax element is afati\_signalled\_tile\_id\_length\_minus1[ attrIdx ] + 1 bits.

The variable FirstTileIDAtt[ attrIdx ] is computed as follows:

FirstTileIDAtt[ attrIdx ] = afati\_tile\_id[ attrIdx ][ 0 ]  
 for ( i = 1; i < afati\_num\_tiles\_in\_atlas\_frame\_minus1[ attrIdx ] + 1; i++ )  
 FirstTileIDAtt[ attrIdx ] =   
 Min( FirstTileID[ attrIdx ], afati\_tile\_id[ attrIdx ][ i ]

The arrays TileIDToIndexAtt[ attrIdx ] and TileIndexToIDAtt[ attrIdx ] provide a forward and inverse mapping, respectively, of the ID associated with each attribute tile and the order index of how each attribute tile was specified in the atlas frame attribute tile information syntax.

##### Atlas frame mesh information semantics

**afmi\_use\_single\_mesh\_flag** equal to 1 specifies that there is only one submesh referred by mesh patches in each atlas frame referring to the AFPS. afmi\_use\_single\_mesh\_falg equal to 0 specifies that there may be more than one submeshes that are referred to by mesh patches in each atlas frame referring to the AFPS.

**afmi\_num\_submeshes\_minus2** plus 2 specifies the number of submeshes referred by mesh patches in each atlas frame referring to the AFPS. The value of afmi\_num\_submeshes\_minus2 shall be in the range of 0 to 62, inclusive.

The variable NumSubMeshes is computed as follows:

NumSubMeshes = afmi\_num\_submeshes\_minus2 + 2

When afmi\_num\_submeshes\_minus2 is not present and afmi\_use\_single\_mesh\_flag is equal to 1, NumSubMeshes value is inferred to be equal to 1.

**afmi\_signalled\_submesh\_id\_flag** equal to 1 specifies that the submesh ID referred by mesh patches in each atlas frame is signalled. afmi\_signalled\_tile\_id\_flag equal to 0 specifies that submesh IDs are not signalled.

**afmi\_signalled\_submesh\_id\_length\_minus1** plus 1 specifies the number of bits used to represent the syntax element afmi\_submesh\_id[ i ] when present, and the syntax element mdu\_submesh\_id[ tileID ][ patchIdx ] in a meshpatch data unit with index patchIdx, in the current atlas tile with tile ID equal to tileID. The value of afmi\_signalled\_submesh\_id\_length\_minus1 shall be in the range of 0 to 15, inclusive. When not present, its value is inferred to be equal to Ceil( Log2( NumSubMeshes ) ) – 1.

**afmi\_submesh\_id**[ i ] specifies the submesh ID of the i-th submesh. When not present, the value of afmi\_submesh\_id[ i ] is inferred to be equal to i, for each i in the range of 0 to NumSubMeshes - 1, inclusive. It is a requirement of bitstream conformance that afmi\_submesh\_id[ i ] shall not be equal to afmi\_submesh\_id[ j ] for all i != j. The length of the afmi\_submesh\_id[ i ] syntax element is afmi\_signalled\_submesh\_id\_length\_minus1 + 1 bits.

The variable FirstSubmeshID is computed as follows:

FirstSubmeshID = afmi\_submesh\_id[0 ]  
 for ( i = 1; i < NumSubMeshes; i++ )  
 FirstSubmeshID = Min(FirstSubmeshID, afmi\_submesh\_id[ i ] )

#### Atlas adaptation parameter set RBSP semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.6.3 apply.

#### Supplemental enhancement information RBSP semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.6.4 apply.

#### Access unit delimiter RBSP semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.6.5 apply.

#### End of sequence RBSP semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.6.6 apply.

#### End of bitstream RBSP semantics

The specifications in ISO/IEC DS 23090-5(2E):2023 subclause 8.4.6.7 apply.

#### Filler data RBSP semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.6.8 apply.

#### Atlas tile layer RBSP semantics

The specifications in ISO/IEC DS 23090-5(2E):2023 subclause 8.4.6.9 apply.

#### RBSP trailing bit semantics

The specifications in ISO/IEC IS 23090-5(2E):2023 subclause 8.4.6.10 apply.

#### Atlas tile header semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.6.11 apply.

#### Reference list structure semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.6.12 apply.

#### Common atlas sequence parameter set RBSP semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.6.13 apply.

#### Common atlas frame RBSP semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.6.14 apply.

### Atlas tile data unit semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.7 apply.

### Supplemental enhancement information message semantics

The specifications in ISO/IEC 23090-5(2E):2023 subclause 8.4.8 apply.

### VDMC atlas tile data unit semantics

#### General VDMC atlas tile data unit semantics

**atdu\_meshpatch\_mode**[ tileID ][ p ] indicates the meshpatch mode for the meshpatch with patch index p in the current atlas tile with tile ID equal to tileID. The permitted values for atdu\_meshpatch\_mode[ tileID ][ p ] are specified in Table 4 for atlas tiles with ath\_type equal to I\_TILE, in Table 5 for atlas tiles with ath\_type equal to P\_TILE, and in Table 6 for atlas tiles with ath\_type equal to SKIP\_TILE. When not present, the value of atdu\_meshpatch\_mode[ tileID ][ p ] is inferred to be equal to P\_SKIP. Modes indicated as reserved shall not be present in bitstreams conforming to this version of this document.

Table 4 – Meshpatch modes for I\_TILE type atlas tiles

|  |  |  |
| --- | --- | --- |
| **atdu\_meshpatch\_mode[ tileID ][ p ]** | **Identifier** | **Description** |
| 0 | I\_INTRA | Non-predicted meshpatch mode |
| 1-13 | I\_RESERVED | Reserved modes for future use by ISO/IEC |
| 14 | I\_END | Meshpatch termination mode |

Table 5 – Meshpatch modes for P\_TILE type atlas tiles

|  |  |  |
| --- | --- | --- |
| **atdu\_meshpatch\_mode[ tileID ][ p ]** | **Identifier** | **Description** |
| 0 | P\_SKIP | Meshpatch Skip mode |
| 1 | P\_MERGE | Meshpatch Merge mode |
| 2 | P\_INTER | Inter predicted Meshpatch mode |
| 3 | P\_INTRA | Non-predicted Meshpatch mode |
| 4-13 | P\_RESERVED | Reserved modes for future use by ISO/IEC |
| 14 | P\_END | Patch termination mode |

Table 6 – Meshpatch modes for SKIP\_TILE type atlas tiles

|  |  |  |
| --- | --- | --- |
| **atdu\_meshpatch\_mode[ tileID ][ p ]** | **Identifier** | **Description** |
| 0 | P\_SKIP | Meshpatch Skip mode |

#### Meshpatch information data semantics

None

#### Meshpatch data unit semantics

**mdu\_submesh\_id**[ tileID ][ patchIdx ] indicates the associated submesh ID specified in the current meshpatch with index patchIdx, in the current atlas tile with tile ID equal to tileID. The value of mdu\_submesh\_id[ tileID ][ patchIdx ] shall be one of afmi\_submesh\_id[ i ] where i is in the range 0 to 65535, inclusive.

**mdu\_displ\_id**[ tileID ][ patchIdx ] indicates the associated displacement ID specified in the current meshpatch with index patchIdx, in the current atlas tile with tile ID equal to tileID. The value of mdu\_disp\_id[ tileID ][ patchIdx ] in the range 0 to 65535, inclusive.

**mdu\_face\_count\_minus1**[ tileID ][ patchIdx ] specifies the number of faces associated with the current meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID.

**mdu\_2d\_pos\_x**[ tileID ][ patchIdx ] specifies the x-coordinate of the top-left corner of the meshpatch’s bounding box for the current meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID, expressed as a multiple of PatchPackingBlockSize.

**mdu\_2d\_pos\_y**[ tileID ][ patchIdx ] specifies the y-coordinate of the top-left corner of the meshpatch’s bounding box for the current meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID, expressed as a multiple of PatchPackingBlockSize.

**mdu\_2d\_size\_x\_minus1**[ tileID ][ patchIdx ] plus 1 specifies the width value of the current meshpatch’s bounding box for the meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID.

**mdu\_2d\_size\_y\_minus1**[ tileID ][ patchIdx ] plus 1 specifies the height value of the current meshpatch’s bounding box for the meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID.

**mdu\_parameters\_override\_flag**[ tileID ][ patchIdx ] equal to 1 indicates the parameters mdu\_subdivision\_override\_flag, mdu\_quantization\_override\_flag, mdu\_transform\_method\_override\_flag, and mdu\_transform\_parameters\_override\_flag are present in a meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID.

**mdu\_subdivision\_override\_flag**[ tileID ][ patchIdx ] equal to 1 indicates mdu\_subdivision\_method and mdu\_subdivision\_iteration\_count\_minus1 are present in a meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID. When mdu\_subdivision\_override\_flag[ tileID ][ patchIdx ] is not present, its value is inferred to be equal to 0.

**mdu\_quantization\_override\_flag**[ tileID ][ patchIdx ] equal to 1 indicates vdmc\_quantization\_parameters(qpIndex, subdivisionCount) syntax structure is present in a meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID. When mdu\_quantization\_override\_flag[ tileID ][ patchIdx ] is not present, its value is inferred to be equal to 0.

The variable QpIndex for the current patch is derived as follows:

QpIndex = mdu\_quantization\_override\_flag[ tileID ][ patchIdx ] ? 2 :  
 afve\_quantization\_parameters\_enable\_flag ? 1: 0

**mdu\_transform\_method\_override\_flag**[ tileID ][ patchIdx ] equal to 1 indicates mdu\_transform\_method is present in a meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID. When mdu\_transform\_method\_override\_flag[ tileID ][ patchIdx ] is not present, its value is inferred to be equal to 0.

**mdu\_transform\_parameters\_override\_flag**[ tileID ][ patchIdx ] equal to 1 indicates vdmc\_lifting\_transform\_parameters(lptIndex, subdivisionCount) syntax structure is present in a meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID. When mdu\_transform\_parameters\_override\_flag[ tileID ][ patchIdx ] is not present, its value is inferred to be equal to 0.

The variable LtpIndex for the current patch is derived as follows:

LtpIndex = mdu\_transform\_parameters\_override\_flag[ tileID ][ patchIdx ] ? 2 :  
 afve\_transform\_parameters\_enable\_flag ? 1: 0

**mdu\_subdivision\_iteration\_count**[ tileID ][ patchIdx ] indicates the number of iterations used for the subdivision in a meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID. When mdu\_subdivision\_iteration\_count[ tileID ][ patchIdx ] is not present its values is inferred to be equal to afve\_subdivision\_iteration\_count.

**mdu\_lod\_adaptive\_subdivision\_flag**[ tileID ][ patchIdx ]equal to 1 indicates that subdivision method is signaled for each subdivision iteration in a meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID. mdu\_lod\_adaptive\_subdivision\_flag equal to 0 indicates that the same subdivision method is applied for each subdivision iterations in a meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID.

**mdu\_subdivision\_method**[ tileID ][ patchIdx ][ i ] indicates the identifier of the method to subdivide the subparts of the mesh associated with a meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID at the subdivision with subdivision index equal to i. When mdu\_subdivision\_method[ tileID ][ patchIdx ][ i ] is not present, its value is inferred to be equal to afve\_subdivision\_method[ i ]. Table 2 describes the list of supported subdivision methods and their relationship with mdu\_subdivision\_method.

**mdu\_displacement\_coordinate\_system**[ tileID ][ patchIdx ] indicates the identifier of the coordinate system of the subparts of the mesh associated with a meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID. Table 7 describes the list of supported displacement coordinate system and their relationship with mdu\_displacement\_coordinate\_system[ tileID ][ patchIdx ].

Table 7 – Displacement coordinate system types

|  |  |
| --- | --- |
| mdu\_displacement\_coordinate\_system[ tileID ][ patchIdx ] | Name of displacement coordinate system |
| 0 | CANNONICAL |
| 1 | LOCAL |

**mdu\_transform\_method**[ tileID ][ patchIdx ] indicates the identifier of the transform applied to the displacement associated with a meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID. When mdu\_transform\_method[ tileID ][ patchIdx ] is not present, its values is inferred to be equal to afve\_transform\_method. Table 3 describes the list of supported transforms and their relationship with mdu\_transform\_method[ tileID ][ patchIdx ].

**mdu\_block\_countblock\_count\_minus1**[ tileID ][ patchIdx ][ i ] plus 1 specifies the quotient of the number of vertices associated with the i-th subdivision iteration for the current meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID divided by PatchPackingBlockSize ´ PatchPackingBlockSize. The length of the mdu\_block\_countblock\_count\_minus1[ tileID ][ patchIdx ][ i ] syntax element is blockCountBitCount bits where blockCountBitCount is computed as follows:

maxBlockCount = (mdu\_2d\_size\_x\_minus1[ tileID ][ patchIdx ] + 1) \*  
 \*  
 (mdu\_2d\_size\_x\_minus1[ tileID ][ patchIdx ] + 1) +  
 +  
 PatchPackingBlockSize \*\* PatchPackingBlockSize - 1 ) /  
 ( PatchPackingBlockSize \*\* PatchPackingBlockSize )  
 blockCountBitCount = max ( 1, ceil( Log2( maxBlockCount ) ) )

**mdu\_last\_pos\_in\_blockin\_block**[ tileID ][ patchIdx ][ i ] indicates the remainder of the number of vertices associated with the i-th subdivision iteration for the current meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID divided by PatchPackingBlockSize ´ PatchPackingBlockSize. mdu\_last\_pos\_in\_blockin\_block[ tileID ][ patchIdx ][ i ] syntax element is 2 ´ asps\_log2\_patch\_packing\_block\_size bits.

**mdu\_attributes\_2d\_pos\_x**[ tileID ][ patchIdx ][ i ] specifies the x-coordinate of the top-left corner of the meshpatch’s attribute bounding box for the current meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID, for the attribute signalled in the Attribute Video Data unit with index i, expressed as a multiple of PatchPackingBlockSize. When mdu\_attributes\_2d\_pos\_x[ tileID ][ patchIdx ][ i ]is not present, its values is inferred to be equal to 0.

**mdu\_attributes\_2d\_pos\_y**[ tileID ][ patchIdx ][ i ] specifies the y-coordinate of the top-left corner of the meshpatch’s attribute bounding box for the current meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID, for the attribute signalled in the Attribute Video Data unit with index i, expressed as a multiple of PatchPackingBlockSize. When mdu\_attributes\_2d\_pos\_y[ tileID ][ patchIdx ][ i ]is not present, its values is inferred to be equal to 0.

**mdu\_attributes\_2d\_size\_x\_minus1**[ tileID ][ patchIdx ][ i ] plus 1 specifies the width value of the meshpatch’s attribute bounding box for the meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID, for the attribute signalled in the Attribute Video Data unit with index i. When mdu\_attributes\_2d\_size\_x\_minus1[ tileID ][ patchIdx ][ i ] is not present, its values is inferred to be equal to asve\_attribute\_frame\_width[ i ] – 1.

**mdu\_attributes\_2d\_size\_y\_minus1**[ tileID ][ patchIdx ][ i ] plus 1 specifies the height value of the meshpatch’s attribute bounding box for the meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID, for the attribute signalled in the Attribute Video Data unit with index i. When mdu\_attributes\_2d\_size\_y\_minus1[ tileID ][ patchIdx ][ i ]is not present, its values is inferred to be equal to asve\_attribute\_frame\_height[ i ] – 1.

#### Skip meshpatch data unit semantics

None.

#### Merge meshpatch data unit semantics

**mmdu\_ref\_index**[ tileID ][ p ] specifies the atlas reference frame index, refIdx, for the current meshpatch with index p, in the current atlas tile, with tile ID equal to tileID. The value of mmdu\_ref\_index[ tileID ][ p ] shall be in the range of 0 to NumRefIdxActive – 1, inclusive. When mmdu\_ref\_index[ tileID ][ p ] is not present, it is inferred to be equal to 0.

**mmdu\_patch\_index**[ tileID ][ p ] specifies the index, RefPatchIdx, of the meshpatch with index p, in the atlas tile, with tile ID equal to tileID, with the same ID as the current tile address in the atlas frame that corresponds to index refIdx in the current reference atlas frame list.

**mmdu\_texture\_projection\_present\_flag**[ tileID ][ p ] equal to 1 indicates that texture\_projection\_merge\_information( tileID, p ) syntax structure is present in the current meshpatch with index p, in the current atlas tile, with tile ID equal to tileID and the texture coordinates for the submesh indicated by the reference meshpatch will be derived using geometry projection.. If mmdu\_texture\_projection\_present\_flag is equal to 0, then texture\_projection\_merge\_information( tileID, p ) syntax structure is not present and the texture coordinates are present in the submesh indicated by the reference meshpatch and will not be derived by geometry projection..

#### Inter meshpatch data unit semantics

**imdu\_ref\_index**[ tileID ][ p ] specifies the atlas reference frame index, refIdx , for the current meshpatch with index p, in the current atlas tile, with tile ID equal to tileID. The value of imdu\_ref\_index[ tileID ][ p ] shall be in the range of 0 to NumRefIdxActive – 1, inclusive. When imdu\_ref\_index[ tileID ][ p ] is not present, it is inferred to be equal to 0.

**imdu\_patch\_index**[ tileID ][ p ] specifies the index, RefPatchIdx, of the meshpatch with index p, in the atlas tile, with tile ID equal to tileID, with the same ID as the current tile address in the atlas frame that corresponds to index refIdx in the current reference atlas frame list.

**imdu\_delta\_face\_count\_minus1**[ tileID ][ p ] specifies the difference of the number of faces of the meshpatch with index p, in the current atlas tile, with tile ID equal to tileID, and the number of faces of the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that is associated with the reference refIdx.

**imdu\_2d\_delta\_pos\_x**[ tileID ][ p ] specifies the difference of the x‑coordinate of the top-left corner of the meshpatch bounding box of the meshpatch with index p in the current atlas tile, with tile ID equal to tileID, and of the x‑coordinate of the top-left corner of the meshpatch bounding box of the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that is associated with the reference refIdx, expressed as a multiple of PatchPackingBlockSize.

**imdu\_2d\_delta\_pos\_y**[ tileID ][ p ] specifies the difference of the y‑coordinate of the top-left corner of the meshpatch bounding box of the meshpatch with index p, in the current atlas tile, with tile ID equal to tileID, and of the y‑coordinate of the top-left corner of the meshpatch bounding box of the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that is associated with the reference refIdx, expressed as a multiple of PatchPackingBlockSize.

**imdu\_2d\_delta\_size\_x**[ tileID ][ p ] specifies the difference of the width values of the meshpatch with index p, in the current atlas tile, with tile ID equal to tileID, and the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that is associated with the reference refIdx.

**imdu\_2d\_delta\_size\_y**[ tileID ][ p ] specifies the difference of the height values of the meshpatch with index p, in the current atlas tile, with tile ID equal to tileID, and the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that corresponds to the reference refIdx.

**imdu\_subdivision\_override\_flag**[ tileID ][ patchIdx ] equal to 1 indicates imdu\_subdivision\_iteration\_count\_minus1 are present in a meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID.

**imdu\_subdivision\_iteration\_coun**t[ tileID ][ patchIdx ] indicates the number of iterations used for the subdivision in a meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID. When imdu\_subdivision\_iteration\_count[ tileID ][ patchIdx ] is not present its values is inferred to be equal to afve\_subdivision\_iteration\_count.

**imdu\_delta\_block\_count**[ tileID ][ patchIdx ][ i ] specifies the difference of the quotient of the number of vertices associated with the i-th subdivision iteration for the current meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID and the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that is associated with the reference refIdx.

**imdu\_delta\_last\_pos\_in\_blockin\_block**[ tileID ][ patchIdx ][ i ] specifies the difference of the remainder of the number of vertices associated with the i-th subdivision iteration for the current meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID and the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that is associated with the reference refIdx.

**imdu\_attributes\_2d\_delta\_pos\_x**[ tileID ][ p ][ i ] specifies the difference of the x‑coordinate of the top-left corner of the i-th attribute meshpatch bounding box of the meshpatch with index p, in the current atlas tile, with tile ID equal to tileID, and of the x‑coordinate of the top-left corner of the i-th attribute meshpatch bounding box of the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that is associated with the reference refIdx, expressed as a multiple of PatchPackingBlockSize.

**imdu\_attributes\_2d\_delta\_pos\_y**[ tileID ][ p ][ i ] specifies the difference of the y‑coordinate of the top-left corner of the i-th attribute meshpatch bounding box of meshpatch with index p, in the current atlas tile, with tile ID equal to tileID, and of the y‑coordinate of the top-left corner of the i-th attribute meshpatch bounding box of the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that is associated with the reference refIdx, expressed as a multiple of PatchPackingBlockSize.

**imdu\_attributes\_2d\_delta\_size\_x**[ tileID ][ p ][ i ] specifies the difference of the width values of the i-th attribute meshpatch with index p, in the current atlas tile, with tile ID equal to tileID, and the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that is associated with the reference refIdx.

**imdu\_attributes\_2d\_delta\_size\_y**[ tileID ][ p ][ i ] specifies the difference of the height values of the i-th attribute meshpatch with index p, in the current atlas tile, with tile ID equal to tileID, and the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that corresponds to the reference refIdx.

**imdu\_texture\_projection\_present\_flag**[ tileID ][ p ] equal to 1 indicates that texture\_projection\_inter\_information( tileID, p ) syntax structure is present in the current meshpatch with index p, in the current atlas tile, with tile ID equal to tileID and the texture coordinates for the submesh indicated by the reference meshpatch will be derived using geometry projection. If imdu\_texture\_projection\_present\_flag is equal to 0, then texture\_projection\_inter\_information( tileID, p ) syntax structure is not present and the texture coordinates are present in the submesh indicated by the reference meshpatch and will not be derived by geometry projection.

#### Texture projection information semantics

**tpi\_face\_id\_present\_flag**[ tileID ][ patchIdx ] equal to 1 specifies that the syntax element si\_face\_id[ tileId ][ patchIdx ][ i ] will be present in the sub-patch with index i, in the meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID. If tpi\_face\_id\_present\_flag[ tileID ][ patchIdx ] is equal to 0, the si\_face\_id[ tileId ][ patchIdx ][ i ] will not be present. If tpi\_face\_id\_present\_flag[ tileID ][ patchIdx ] is not present, it is assumed to be 0.

**tpi\_frame\_scale**[ tileID ][ patchIdx ] indicates the value of the frame scale, for the meshpatch with index patchIdx in the current atlas tile, with tile ID equal to tileID. The variable TexCoordProjectionFrameScale[ tileID ][ patchIdx ] is used for texture coordinate derivation from geometry projection, and is defined as:

TexCoordProjectionFrameScale[ tileID ][ patchIdx ] =  
 tpi\_frame\_scale[ tileID ][ patchIdx ]

**tpi\_subpatch\_count\_minus1**[ tileID ][ patchIdx ] plus 1 indicates the number of sub-patches present in the meshpatch, for the meshpatch with index patchIdx in the current atlas tile, with tile ID equal to tileID.

#### Sub-patch information semantics

**si\_face\_id**[ tileID ][ patchIdx ][ i ] specifies the face ID for the sub-patch with index i, in the meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID. If si\_face\_id[ tileID ][ patchIdx ][ i ] is not present, it is assumed to be equal to i.

**si\_projection\_id**[ tileID ][ patchIdx ][ i ] specifies the values of the projection mode and of the index of the normal to the projection plane for the sub-patch present in the meshpatch with index patchIdx in the current atlas tile, with tile ID equal to tileID, for the sub-patch with index i. The value of si\_projection\_id[ tileID ][ patchIdx ][ i ] shall be in range of 0 to asps\_max\_number\_projections\_minus1, inclusive. The number of bits used to represent si\_projection\_id[ tileID ][ patchIdx ][ i ] is Ceil( Log2( asps\_max\_number\_projections\_minus1 + 1) ). When si\_projection\_id[ tileID ][ patchIdx ][ i ] is not present, its value is inferred to be equal to 0.

**si\_orientation\_id**[ tileID ][ patchIdx ][ i ] specifies the orientation index for the sub-patch present in the meshpatch with index patchIdx in the current atlas tile, with tile ID equal to tileID, for the sub-patch index i, used to determine the sub-patch rotation homography transform, as indicated in Table 10 that are to be used to transform the vertex’s 3D space coordinates into texture coordinates ( u, v ). When si\_orientation\_id[ tileID ][ patchIdx ][ i ] is not present, its value is inferred to be equal to 0.

**si\_2d\_pos\_x**[ tileID ][ patchIdx ][ i ] specifies the x-coordinate of the top-left corner of the sub-patch bounding box size for the sub-patch present in the meshpatch with index patchIdx in the current atlas tile, with tile ID equal to tileID, for the sub-patch with index i. When si\_2d\_pos\_x[ tileID ][ patchIdx ][ i ] is not present, its value is inferred to be equal to 0.

**si\_2d\_pos\_y**[ tileID ][ patchIdx ][ i ] specifies the y-coordinate of the top-left corner of the sub-patch bounding box size for sub-patch present in the meshpatch with index patchIdx in the current atlas tile, with tile ID equal to tileID, for the sub-patch with index i. When si\_2d\_pos\_y[ tileID ][ patchIdx ][ i ] is not present, its value is inferred to be equal to 0.

**si\_2d\_size\_x\_minus1\_diff**[ tileID ][ patchIdx ][ i ] specifies the difference between the width values of the sub-patch bounding box size for the sub-patch present in the meshpatch with index patchIdx and in the current atlas tile, with tile ID equal to tileID, and with submesh index equal to smIdx, for the sub-patches with index i and index i – 1. When i = 0, si\_2d\_size\_x\_minus1\_diff[ tileID ][ patchIdx ][ i ] plus 1 specifies the width of the sub-patch bounding box size. When si\_2d\_size\_x\_minus1[ tileID ][ patchIdx ][ i ] is not present, its value is inferred to be equal to TileMeshpatchTexcoordProjectionWidth[ tileID ][ patchIdx ] – 1, for a submesh with index smIdx.

**si\_2d\_size\_y\_minus1\_diff**[tileID ][ patchIdx ][ i ] specifies the difference between the height values of the sub-patch bounding box size for the sub-patch present in the meshpatch with index patchIdx in the current atlas tile, with tile ID equal to tileID, for the sub-patchs associated with index i and index i – 1. When i = 0, si\_2d\_size\_y\_minus1\_diff[ tileID ][ patchIdx ][ i ] plus 1 specifies the height of the sub-patch bounding box size. When si\_2d\_size\_y\_minus1[ tileID ][ patchIdx ][ i ]is not present, its value is inferred to be equal to TileTexcoordProjectionHeight[ tileID ][ patchIdx ] – 1, for a submesh with index smIdx.

**si\_scale\_present\_flag**[ tileID ][ patchIdx ][ i ] equal to 1 specifies that the sub-patch scale is present for the sub-patch present in the meshpatch with index patchIdx in the current atlas tile, with tile ID equal to tileID, for the sub-patch with index i. If si\_scale\_present\_flag[ tileID ][ patchIdx ][ i ] is equal to 0, no sub-patch scale information is present. If si\_scale\_present\_flag[ tileID ][ patchIdx ][ i ] is not present, it is assumed to be 0.

**si\_scale\_power\_factor**[ tileID ][ patchIdx ][ i ] if present, specifies the scaling power factor for the sub-patch present in the meshpatch with index patchIdx in the current atlas tile, with tile ID equal to tileID, for the sub-patch with index i, SubpatchScalingFactor[ tileID ][ patchIdx ][ i ], as follows:

SubpatchScalingFactor[ tileID ][ patchIdx ][ i ] = FrameScale[ tileID ][ patchIdx ]  
 for( i = 0; i <= si\_scale\_power\_factor[ tileID ][ patchIdx ][ i ]; i++)  
 SubpatchScalingFactor[ tileID ][ patchIdx ][ i ] \*= TexcoordProjectionScaleFactor

If si\_scale\_power\_factor[ tileID ][ patchIdx ][ i ] is not present, then the SubpatchScalingFactor[ tileID ][ patchIdx ][ i ] is equal to the FrameScale[ tileID ][ patchIdx ]

#### Texture projection inter information semantics

**tpii\_face\_id\_present\_flag**[ tileID ][ patchIdx ] equal to 1 specifies that the syntax element si\_face\_id[ tileId ][ patchIdx ][ i ] will be present in the sub-patch with index i, in the meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID. If tpii\_face\_id\_present\_flag[ tileID ][ patchIdx ] is equal to 0, the si\_face\_id[ tileId ][ patchIdx ][ i ] will not be present. If tpii\_face\_id\_present\_flag[ tileID ][ patchIdx ] is not present, it is assumed to be 0.

**tpii\_frame\_scale**[ tileID ][ patchIdx ] indicates the value of the frame scale, for the meshpatch with index patchIdx in the current atlas tile, with tile ID equal to tileID. The variable TexCoordProjectionFrameScale[ tileID ][ patchIdx ] is used for texture coordinate derivation from geometry projection, and is defined as:

TexCoordProjectionFrameScale[ tileID ][ patchIdx ] =  
 tpi\_frame\_scale[ tileID ][ patchIdx ]

**tpii\_subpatch\_count\_minus1**[ tileID ][ patchIdx ] plus 1 indicates the number of sub-patches present in the meshpatch, for the meshpatch with index patchIdx in the current atlas tile, with tile ID equal to tileID.

**tpii\_subpatch\_inter\_enable\_flag**[ tileID ][ patchIdx ] equal to 1 indicates that the syntax element tpii\_subpatch\_inter\_present\_flag[ tileID ][ patchIdx ][ subIdx ] is present for the subpatch with index subIdx, in the current meshpatch with index p, in the current atlas tile, with tile ID equal to tileID. If tpii\_subpatch\_inter\_enable\_flag[ tileID ][ patchIdx ] is equal to 0, then the syntax element tpii\_subpatch\_inter\_present\_flag[ tileID ][ patchIdx ] is not present.

**tpii\_subpatch\_inter\_present\_flag**[ tileID ][ patchIdx ][ subIdx ] equal to 1 indicates that the subpatch information is provided using the subpatch\_inter\_information( tileID, patchIdx, subIdx ) syntax structure for the subpatch with index subIdx, in the current meshpatch with index p, in the current atlas tile, with tile ID equal to tileID. If tpii\_subpatch\_inter\_present\_flag[ tileID ][ patchIdx ][ subIdx ] is equal to 0, then the subpatch information is provided using the subpatch\_information( tileID, patchIdx, subIdx ) syntax structure. If tpii\_subpatch\_inter\_present\_flag[ tileID ][ patchIdx ][ subIdx ] is not present, it is assumed to be equal to 0.

#### Sub-patch inter information semantics

**sii\_subpatch\_index\_diff**[ tileID ][ patchIdx ][ i ] plus i specifies the index, RefSubPatchIdx, of the sub-patch with index i, in the meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID.

**sii\_2d\_pos\_x\_delta**[ tileID ][ patchIdx ][ i ] specifies the difference of the x-coordinate of the top-left corner of the sub-patch bounding box of the sub-patch with index i, in the meshpatch with index patchIdx in the current atlas tile, with tile ID equal to tileID, and of the x-coordinate of the top-left corner of the sub-patch bounding box of the sub-patch with index RefSubPatchIdx, in the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that is associated with the reference refIdx.

**sii\_2d\_pos\_y\_delta**[ tileID ][ patchIdx ][ i ] specifies the difference of the y-coordinate of the top-left corner of the sub-patch bounding box of the sub-patch with index i, in the meshpatch with index patchIdx in the current atlas tile, with tile ID equal to tileID, and of the y-coordinate of the top-left corner of the sub-patch bounding box of the sub-patch with index RefSubPatchIdx, in the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that is associated with the reference refIdx.

**sii\_2d\_size\_x\_delta**[ tileID ][ patchIdx ][ i ] specifies the difference of the width values of the sub-patch bounding box size for the sub-patch with index i, in the meshpatch with index patchIdx and in the current atlas tile, with tile ID equal to tileID, and the sub-patch with index RefSubPatchIdx, in the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that is associated with the reference refIdx.

**sii\_2d\_size\_y\_delta**[tileID ][ patchIdx ][ i ] specifies the difference of the height values of the sub-patch bounding box size for the sub-patch with index i, in the meshpatch with index patchIdx and in the current atlas tile, with tile ID equal to tileID, and the sub-patch with index RefSubPatchIdx, in the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that is associated with the reference refIdx.

#### Texture projection information

**tpmi\_subpatch\_merge\_present\_flag**[ tileID ][ patchIdx ] equal to 1 indicates that some sub-patches will be updated using the subpatch\_merge\_information( tileID, patchIdx, subIdx ) syntax structure. If tpmi\_subpatch\_merge\_present\_flag[ tileID ][ patchIdx ] is equal to 0, the sub-patches are not updated. In both cases, the list of sub-patches from the current meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID, is initialized with the list of sub-patches from meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that is associated with the reference refIdx.

**tpmi\_update\_subpatch\_count\_minus1**[ tileID ][ patchIdx ] plus 1 indicates the number of sub-patches to be updated in the meshpatch, for the meshpatch with index patchIdx in the current atlas tile, with tile ID equal to tileID.

#### Sub-patch information semantics

**smi\_subpatch\_index**[ tileID ][ patchIdx ][ i ] specifies the index, RefSubPatchIdx, of the sub-patch with index i, in the meshpatch with index patchIdx, in the current atlas tile, with tile ID equal to tileID.

**smi\_2d\_pos\_x\_delta**[ tileID ][ patchIdx ][ i ] specifies the difference of the x-coordinate of the top-left corner of the sub-patch bounding box of the sub-patch with index i, in the meshpatch with index patchIdx in the current atlas tile, with tile ID equal to tileID, and of the x-coordinate of the top-left corner of the sub-patch bounding box of the sub-patch with index RefSubPatchIdx, in the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that is associated with the reference refIdx.

**smi\_2d\_pos\_y\_delta**[ tileID ][ patchIdx ][ i ] specifies the difference of the y-coordinate of the top-left corner of the sub-patch bounding box of the sub-patch with index i, in the meshpatch with index patchIdx in the current atlas tile, with tile ID equal to tileID, and of the y-coordinate of the top-left corner of the sub-patch bounding box of the sub-patch with index RefSubPatchIdx, in the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that is associated with the reference refIdx.

**smi\_2d\_size\_x\_delta**[ tileID ][ patchIdx ][ i ] specifies the difference of the width values of the sub-patch bounding box size for the sub-patch with index i, in the meshpatch with index patchIdx and in the current atlas tile, with tile ID equal to tileID, and the sub-patch with index RefSubPatchIdx, in the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that is associated with the reference refIdx.

**smi\_2d\_size\_y\_delta**[tileID ][ patchIdx ][ i ] specifies the difference of the height values of the sub-patch bounding box size for the sub-patch with index i, in the meshpatch with index patchIdx and in the current atlas tile, with tile ID equal to tileID, and the sub-patch with index RefSubPatchIdx, in the meshpatch with index RefPatchIdx, in the atlas tile with the same ID as the current tile in the atlas frame that is associated with the reference refIdx.

# Decoding process

## General decoding process

The specifications in ISO/IEC 23090-5(2E):2023 subclause 9. 1 apply with the following additions.

The following additional decoding process is invoked on each of its associated V3C sub-bitstream component:

* The decoding process for the V3C sub-bitstream component corresponding to the basemesh component, which is determined through either examining if vuh\_unit\_type is equal to V3C\_BMD or through external means if the V3C unit header is unavailable, is as specified in subclause 9.9.

The decoding process may include additional processing steps required to convert the decoded basemesh stream to a nominal format, i.e. a nominal coordinate system, bit depth, etc. Such conversion process is outside the scope of this document. However, for purposes of conformance to point B, as defined in Annex A, an example process is described in Annex B .

Outputs defined in subclause 9.9 may have an assigned composition time and output order index. The composition time and the output order index may be provided either in the bitstream or by external means not specified in this document. The composition time and the output order index may be utilized to determine timing relationships of and synchronize V3C components, and to perform operations as defined in Annex B, if required.

## Atlas data decoding process

### General atlas decoding process

The specifications in ISO/IEC 23090-5(2E):2023 subclause 9.2.1 apply.

### Decoding process for a coded atlas frame

The specifications in ISO/IEC 23090-5(2E):2023 subclause 9.2.2 apply with the following addition.

The processes in subclause 9.2.8 specify the meshpatch decoding processes according to the patch mode as follows:

* Decoding of intra coded meshpatches is specified in subclause 9.2.8.2.
* Decoding of skip mode coded meshpatches is specified in subclause 9.2.8.4.
* Decoding of merge mode coded meshpatches is specified in subclause 9.2.8.5.
* Decoding of inter coded meshpatches is specified in subclause 9.2.8.6.

### Atlas NAL unit decoding process

The specifications in ISO/IEC 23090-5(2E):2023 subclause 9.2.3 apply with the following additions.

### Atlas tile header decoding process

#### Atlas frame order count derivation process

The specifications in ISO/IEC 23090-5(2E):2023 subclause 9.2.4.1 apply.

#### Decoding process for generating unavailable reference atlas frames

##### General

The specifications in ISO/IEC 23090-5(2E):2023 subclause 9.2.4.2.1 apply.

##### Generation of one unavailable atlas frame

When this process is invoked, an unavailable atlas frame is generated as follows:

* For all tiles that are associated with this atlas frame the following applies:

for( t = 0; t<= afti\_num\_tiles\_in\_atlas\_frame\_minus1; t++ ) {  
 tileID = TileIndexToID( t )  
 AtduTotalNumMeshpatches[ tileID ] = MaxNumMeshpatches for( p = 0; p<= AtduTotalNumMeshpatches[tileID ]; p++ ) {  
 TileMeshpatch2dPosX[ tileID ][ p ] = 0  
 TileMeshpatch2dPosY[ tileID ][ p ] = 0  
 TileMeshpatch2dSizeX[ tileID ][ p ] = 1  
 TileMeshpatch2dSizeY[ tileID ][ p ] = 1  
 for( attrIdx = 0; attrIdx < asve\_num\_attribute\_video; attrIdx){   
 TileMeshpatchAttributes2dPosX[ tileID ][ p ][ attrIdx ] = 0  
 TileMeshpatchAttributes2dPosY[ tileID ][ p ][ attrIdx ] = 0   
 TileMeshpatchAttributes2dSizeX[ tileID ][ p ][ attrIdx ] = 1  
 TileMeshpatchAttributes2dSizeY[ tileID ][ p ][ attrIdx ] = 1  
 }  
 TileMeshpatchSubmeshID[ tileID ][ p ] = 0  
 TileMeshpatchDisplID[ tileID ][ p ] = 0  
 TileMeshpatchVertexCount[ tileID ][ p ] = 0  
 TileMeshpatchFaceCount[ tileID ][ p ] = 0  
 TileMeshpatchSubdivMethod[ tileID ][ p ] = 0  
 TileMeshpatchSubdivCount[ tileID ][ p ] = 0  
 TileMeshpatchDispCoordSys[ tileID ][ p ] = 0  
 TileMeshpatchTransformMethod[ tileID ][ p ] = 0  
 TileMeshpatchFrameScale[ tileID ][ p ] = 1  
 TileMeshpatchNumSubpatches[ tileID ][ p ] = 0  
 }  
}

#### Reference atlas frame list construction process

The specifications in ISO/IEC 23090-5(2E):2023 subclause 9.2.4.3 apply.

#### Reference atlas frame marking process

The specifications in ISO/IEC 23090-5(2E):2023 subclause 9.2.4.4 apply.

### Decoding process for patch data units

The specifications in ISO/IEC 23090-5(2E):2023 subclause 9.2.5 apply.

### Decoding process of the block to patch map

The specifications in ISO/IEC 23090-5(2E):2023 subclause 9.2.6 apply.

### Conversion of tile level patch information to atlas level patch information

The specifications in ISO/IEC 23090-5(2E):2023 subclause 9.2.7 apply.

### Decoding process for meshpatch data units

#### General decoding process for meshpatch data units

Inputs to this process are the current patch index, p, of the current tile with tile ID equal to tileID, and the current meshpatch mode, atdu\_meshpatch\_mode[ tileID ][ p ].

Outputs of this process are several parameters associated with the current meshpatch with patch index p, including its 2D and corresponding 3D location information.

More specifically the following variables are derived:

Let the variables horLimit and verLimit be computed as follows:

horLimit = TileWidth[ TileIDToIndex[ tileID ] ]  
 verLimit = TileHeight[ TileIDToIndex[ tileID ] ]

TileMeshpatch2dPosX[ tileID ][ p ] specifies the x-coordinate of the top-left corner of the meshpatch bounding box for the current meshpatch with patch index p, of the current tile with tile ID equal to tileID, relative to the top left corner of the current tile, in units of atlas samples. The value of TileMeshpatch2dPosX[ tileID ][ p ] shall be in the range of 0 to horLimit – 1, inclusive.

TileMeshpatch2dPosY[ tileID ][ p ] specifies the y-coordinate of the top-left corner of the meshpatch bounding box for the current meshpatch with patch index p, of the current tile with tile ID equal to tileID, relative to the top left corner of the current tile, in units of atlas samples. The value of TileMeshpatch2dPosY[ tileID ][ p ] shall be in the range of 0 to verLimit – 1, inclusive.

TileMeshpatch2dSizeX[ tileID ][ p ] specifies the width of the bounding box of the current meshpatch with patch index p, of the current tile with tile ID equal to tileID, in units of atlas samples. The value of TileMeshpatch2dSizeX[ tileID ][ p ] shall be in the range of 1 to horLimit, inclusive.

TileMeshpatch2dSizeY[ tileID ][ p ] specifies the height of the bounding box of the current meshpatch with patch index p, of the current tile with tile ID equal to tileID, in units of atlas samples. The value of TileMeshpatch2dSizeY[ tileID ][ p ] shall be in the range of 1 to verLimit, inclusive.

If asve\_num\_attribute\_video is greater than 0, the following additional variables are defined:

Let the variables horLimitAtt[ attrIdx ] and verLimitAtt[ attrIdx ] for the attrIdx-th attribute be computed as follows:

[Ed.Note (LK,DB) It is assumed that number of tiles and their IDs are the same in AFATI and in AFTI. See comment in 8.3.6.2.4. TileIDAttr is equal to tileID]

horLimitAtt[ attrIdx ] =  
 TileWidthAtt[ attrIdx ][ TileIDToIndexAtt[ attrIdx ][ tileIDAttr ] ]  
 verLimit[ attrIdx ] =  
 TileHeightAtt[ attrIdx ][ TileIDToIndexAtt[ attrIdx ][ tileIDAttr ] ]

TileMeshpatchAttributes2dPosX[ tileID ][ p ][ attrIdx ] specifies the x-coordinate of the top-left corner of the attribute frame associated with the current meshpatch with patch index p, in the current tile with tile ID equal to tileID, for the attribute signalled in the Attribute Video Data unit with index attrIdx, in units of atlas samples. The value of TileMeshpatchAttributes2dPosX[ tileID ][ p ][ attrIdx ] shall be in the range of 0 to horLimitAtt[ attrIdx ] – 1, inclusive.

TileMeshpatchAttributes2dPosY[ tileID ][ p ][ attrIdx ] specifies the y-coordinate of the top-left corner of the attribute frame associated with the current meshpatch with patch index p, in the current tile with tile ID equal to tileID, for the attribute signalled in the Attribute Video Data unit with index attrIdx, in units of atlas samples. The value of TileMeshpatchAttributes2dPosY[ tileID ][ p ][ attrIdx ] shall be in the range of 0 to verLimitAtt[ attrIdx ] – 1, inclusive.

TileMeshpatchAttributes2dSizeX[ tileID ][ p ][ attrIdx ] specifies the width of the attribute frame of the current meshpatch with patch index p, in the current tile with tile ID equal to tileID, for the attribute signalled in the Attribute Video Data unit with index attrIdx, in units of atlas samples. The value of TileMeshpatchAttributes2dSizeX[ tileID ][ p ] shall be in the range of 1 to horLimitAtt[ attrIdx ], inclusive.

TileMeshpatchAttributes2dSizeY[ tileID ][ p ][ attrIdx ] specifies the height of the attribute frame of the current meshpatch with patch index p, in the current tile with tile ID equal to tileID, for the attribute signalled in the Attribute Video Data unit with index attrIdx, in units of atlas samples. The value of TileMeshpatchAttributes2dSizeY[ tileID ][ p ] shall be in the range of 1 to verLimitAtt[ attrIdx ], inclusive.

It is a requirement of bitstream conformance that the following conditions apply:

if( !asve\_displacement\_id\_present\_flag ){  
 TileMeshpatch2dPosX[ tileID ][ p ] + TileMeshpatch2dSizeX[ tileID ][ p ] <=  
 horLimit (1)  
 TileMeshpatch2dPosY[ tileID ][ p ] + TileMeshpatch2dSizeY[ tileID ][ p ] <=  
 verLimit (2)  
 }  
 for( attrIdx = 0; attrIdx < asve\_num\_attribute\_video; attrIdx){  
 TileMeshpatchAttributes2dPosX[ tileID ][ p ] +  
 TileMeshpatchAttributes2dSizeX[ tileID ][ p ] <= horLimitAtt[ attrIdx ]  
 TileMeshpatchAttributes2dPosY[ tileID ][ p ] +  
 TileMeshpatchAttributes2dSizeY[ tileID ][ p ]  <= verLimitAtt[ attrIdx ]  
 }

The additional meshpatch related variables are derived:

TileMeshpatchTexcoordProjectionFlag[ tileID ][ p ] specifies that the texture coordinates for the meshpatch will be derived using geometry projection.

TileMeshpatchTexcoordProjectionWidthNormalization[ tileID ][ p ] specifies the width of the projection surface used for texture coordinate derivation.

TileMeshpatchTexcoordProjectionHeightNormalization[ tileID ][ p ] specifies the height of the projection surface used for texture coordinate derivation.

TileMeshpatchTexcoordProjectionGutter[ tileID ][ p ] specifies the gutter used for texture coordinate derivation.

TileMeshpatchSubMeshID[ tileID ][ p ] specifies the ID of the submesh for the current meshpatch with patch index p, in the current tile with tile ID equal to tileID. The value of TileMeshpatchSubMeshID[ tileID ][ p ] shall be one of afmi\_submesh\_id[ i ] where i is in the range 0 to 65535, inclusive.

TileMeshpatchDisplID[ tileID ][ p ] specifies the ID of the displacement for the current meshpatch with patch index p, in the current tile with tile ID equal to tileID.

TileMeshpatchVertexCount[ tileID ][ p ] specifies the number of vertices for the current meshpatch with patch index p, in the current tile with tile ID equal to tileID.

TileMeshpatchFaceCount[ tileID ][ p ] specifies the number of faces for the current meshpatch with patch index p, in the current tile with tile ID equal to tileID.

TileMeshpatchSubdivMethod[ tileID ][ p ][ i ] specifies the method of subdivision for the current meshpatch with patch index p, in the current tile with tile ID equal to tileID at the subdivision with subdivision index equal to i.

TileMeshpatchSubdivMethod[ tileID ][ p ] specifies the method of subdivision for the current meshpatch with patch index p, in the current tile with tile ID equal to tileID.

TileMeshpatchSubdivCount[ tileID ][ p ] specifies the number of subdivisions for the current meshpatch with patch index p, in the current tile with tile ID equal to tileID.

TileMeshpatchDispCoordSys[ tileID ][ p ] specifies the displacement coordinate system for the current meshpatch with patch index p, in the current tile with tile ID equal to tileID.

TileMeshpatchTransformMethod[ tileID ][ p ] specifies the method of transform for the current meshpatch with patch index p, in the current tile with tile ID equal to tileID.

If afve\_projection\_texcoord\_present\_flag[ SubMeshIDToIndex[ TileMeshpatchSubMeshID[ tileID ][ p ] ] ] is equal to 1, the additional variables are derived:

TileMeshpatchFrameScale[ tileID ][ p ] specifies the frame scale for the current meshpatch with patch index p, in the current tile with tile ID equal to tileID.

TileMeshpatchNumSubpatches[ tileID ][ p ] specifies the number of subpatches for the current meshpatch with patch index p, in the current tile with tile ID equal to tileID.

TileMeshpatchSubpatchFaceId[ tileID ][ p ][ i ] specifies the face ID for the subpatch with index i, of the current meshpatch with patch index p, in the current tile with tile ID equal to tileID.

TileMeshpatchSubpatchProjectionID[ tileID ][ p ][ i ] specifies the projection ID for the subpatch with index i, of the current meshpatch with patch index p, in the current tile with tile ID equal to tileID.

TileMeshpatchSubpatchOrientationID[ tileID ][ p ][ i ] specifies the orientation ID for the subpatch with index i, of the current meshpatch with patch index p, in the current tile with tile ID equal to tileID.

TileMeshpatchSubpatch2dPosX[ tileID ][ p ][ i ] specifies the x-coordinate of the top-left corner for the subpatch with index i, of the current meshpatch with patch index p, in the current tile with tile ID equal to tileID.

TileMeshpatchSubpatch2dPosY[ tileID ][ p ][ i ] specifies the y-coordinate of the top-left corner. for the subpatch with index i, of the current meshpatch with patch index p, in the current tile with tile ID equal to tileID.

TileMeshpatchSubpatch2dSizeX[ tileID ][ p ][ i ] specifies the width. for the subpatch with index i, of the current meshpatch with patch index p, in the current tile with tile ID equal to tileID.

TileMeshpatchSubpatch2dSizeY[ tileID ][ p ][ i ] specifies the height for the subpatch with index i, of the current meshpatch with patch index p, in the current tile with tile ID equal to tileID.

TileMeshpatchSubpatchScale[ tileID ][ p ][ i ] specifies the scale for the subpatch with index i, of the current meshpatch with patch index p, in the current tile with tile ID equal to tileID.

These variables are initially set as follows:

TileMeshpatch2dPosX[ tileID ][ p ] = 0  
 TileMeshpatch2dPosY[ tileID ][ p ] = 0  
 TileMeshpatch2dSizeX[ tileID ][ p ] = 1  
 TileMeshpatch2dSizeY[ tileID ][ p ] = 1  
 for( attrIdx = 0; attrIdx < asve\_num\_attribute\_video; attrIdx){   
 TileMeshpatchAttributes2dPosX[ tileID ][ p ] = 0  
 TileMeshpatchAttributes2dPosY[ tileID ][ p ] = 0   
 TileMeshpatchAttributes2dSizeX[ tileID ][ p ] = 1  
 TileMeshpatchAttributes2dSizeY[ tileID ][ p ] = 1  
 }  
 TileMeshpatchSubmeshID[ tileID ][ p ] = 0  
 TileMeshpatchDisplID[ tileID ][ p ] = 0  
 TileMeshpatchVertexCount[ tileID ][ p ] = 0  
 TileMeshpatchFaceCount[ tileID ][ p ] = 0  
 TileMeshpatchSubdivCount[ tileID ][ p ] = 0  
 TileMeshpatchDispCoordSys[ tileID ][ p ] = 0  
 TileMeshpatchTransformMethod[ tileID ][ p ] = 0  
 TileMeshpatchFrameScale[ tileID ][ p ] = 1  
 TileMeshpatchNumSubpatches[ tileID ][ p ] = 0

If atdu\_meshpatch\_mode[ tileID ][ p ] is equal to I\_INTRA or P\_INTRA, then the process for decoding intra coded meshpatches in subclause 9.2.8.2 is used, with p and tileID as the inputs to that process, and the outputs of that process are used as the output.

If atdu\_meshpatch\_mode[ tileID ][ p ] is equal to P\_SKIP, then the process for decoding skip coded meshpatches in subclause 9.2.8.4 is used, with p and tileID as the inputs to that process, and the outputs of that process are used as the output.

If atdu\_meshpatch\_mode[ tileID ][ p ] is equal to P\_MERGE, then the process for decoding merge coded meshpatches in subclause 9.2.8.5 is used, with p and tileID as the inputs to that process, and the outputs of that process are used as the output.

If atdu\_meshpatch\_mode[ tileID ][ p ] is equal to P\_INTER, then the process for decoding inter coded meshpatches in subclause 9.2.8.6 is used, with p and tileID as the inputs to that process, and the outputs of that process are used as the output.

#### Decoding process for meshpatch data units coded in intra mode

Inputs to this process are the current patch index, p, and the current tile ID, tileID.

The following meshpatch related variables are assigned given the parsed elements in the meshpatch data unit:

TileMeshpatchSubmeshID[ tileID ][ p ] = mdu\_submesh\_id[ tileID ][ p ] TileMeshpatchSubdivCount[ tileID ][ p ] = PatchSubdivisionCount[ tileID ][ p ]  
 for( i = 0; i < TileMeshpatchSubdivCount[ tileID ][ p ] ; i++ ){  
 TileMeshpatchSubdivMethod[ tileID ][ p ][ i ] =  
 PatchSubdivisionMethod[ tileID ][ patchIdx ][ i ]  
 }  
 TileMeshpatchDispCoordSys[ tileID ][ p ] =  
 mdu\_displacement\_coordinate\_system[ tileID ][ p ]  
 TileMeshpatchTransformMethod[ tileID ][ p ]= mdu\_transform\_method[ tileID ][ p ]

If asve\_displacement\_id\_present\_flag is 0, the following applies:

TileMeshpatchFaceCount[ tileID ][ p ] =  
 mdu\_face\_count\_minus1[ tileID ][ p ] + 1  
 TileMeshpatch2dPosX[ tileID ][ p ] =  
 mdu\_2d\_pos\_x[ tileID ][ p ] \* PatchPackingBlockSize (3)  
 TileMeshpatch2dPosY[ tileID ][ p ] =  
 mdu\_2d\_pos\_y[ tileID ][ p ] \* PatchPackingBlockSize (4)  
 TileMeshpatch2dSizeX[ tileID ][ p ] =  
 (mdu\_2d\_size\_x\_minus1[ tileID ][ p ] + 1) \* PatchSizeXQuantizer (5)  
 TileMeshpatch2dSizeY[ tileID ][ p ] =  
 (mdu\_2d\_size\_y\_minus1[ tileID ][ p ] + 1) \* PatchSizeYQuantizer (6)  
 for( i=0; i<= TileSubdivisionCount[ tileID ][ p ]; i++){  
 TilePatchVertexBlockCount[ tileID ][ p ][ i ] =  
 mpdumpdu\_vertex\_count\_inlastblock[ tileID ][ patchIdx ][ i ] == 0 ?  
 (mpdu\_vertex\_block\_count\_minus1[ tileID ][ patchIdx ][ i ] + 1) :  
 mpdu\_vertex\_block\_count\_minus1[ tileID ][ patchIdx ][ i ]  
 TilePatchVertexCountLast[ tileID ][ p ][ i ] =  
 mpdumpdu\_vertex\_count\_inlastblock[ tileID ][ patchIdx ][ i ]  
 TilePatchVertexCount[ tileID ][ p ][ i ] =  
 TilePatchVertexBlockCount[ tileID ][ p ][ i ] \*  
 PatchPackingBlockSize \* PatchPackingBlockSize +  
 mpdumpdu\_vertex\_count\_inlastblock[ tileID ][ patchIdx ][ i ]  
 TilePatchTotalVertexCount[ tileID ][ p ] +=  
 TilePatchVertexCount[ tileID ][ p ][ i ]  
 }

If asve\_displacement\_id\_present\_flag is 1, the following applies:

TileMeshpatchDisplID[ tileID ][ p ] = mdu\_displ\_id[ tileID ][ p ]

If asve\_num\_attribute\_video is greater than 0, the following applies:

[Ed.Note (LK,DB) It is assumed that number of tiles and their IDs are the same in AFATI and in AFTI. See comment in 8.3.6.2.4]

for( i = 0; i< asve\_num\_attribute\_video; i++ ){  
 if(asve\_attribute\_subtexture\_enabled\_flag[ i ]){  
 TileMeshpatchAttributes2dPosX[ tileID ][ p ][ i ] =   
 mdu\_attributes\_2d\_pos\_x[ tileID ][ p ][ i ]  
 TileMeshpatchAttributes2dPosY[ tileID ][ p ][ i ] =   
 mdu\_attributes\_2d\_pos\_y[ tileID ][ p ][ i ]  
 TileMeshpatchAttributes2dSizeX[ tileID ][ p ][ i ] =  
 (mdu\_attributes\_2d\_size\_x\_minus1[ tileID ][ p ][ i ] + 1) \*PatchSizeYQuantizer  
 TileMeshpatchAttributes2dSizeY[ tileID ][ p ][ i ] =   
 (mdu\_attributes\_2d\_size\_y\_minus1[ tileID ][ p ][ i ] + 1) \* PatchSizeYQuantizer  
 }else{  
 TileMeshpatchAttributes2dPosX[ tileID ][ p ][ i ] = 0  
 TileMeshpatchAttributes2dPosY[ tileID ][ p ][ i ] = 0  
 TileMeshpatchAttributes2dSizeX[ tileID ][ p ][ i ] =  
 TileWidthAtt[ attrIdx ][ TileIDToIndexAtt[ attrIdx ][ tileID ] ]  
 TileMeshpatchAttributes2dSizeY[ tileID ][ p ][ i ] =  
 TileHeightAtt[ attrIdx ][ TileIDToIndexAtt[ attrIdx ][ tileID ] ]  
 }  
 }

smIdx = SubmeshIDToIndex[ TileMeshpatchSubmeshID[ tileID ][ p ] ]  
 TileMeshpatchTexcoordProjectionFlag[ tileID ][ p ] = TexcoordProjectionFlag[ smIdx ]  
 TileMeshpatchTexcoordProjectionWidthNormalization[ tileID ][ p ] =  
 TexcoordProjectionWidthNormalization[ smIdx ]  
 TileMeshpatchTexcoordProjectionHeightNormalization[ tileID ][ p ] =  
 TexcoordProjectionHeightNormalization[ smIdx ]  
 TileMeshpatchTexcoordProjectionGutter[ tileID ][ p ] =  
 TexcoordProjectionGutter[ smIdx ]  
  
If TileMeshpatchTexcoordProjectionFlag[ tileID ][ p ] is equal to 1, decoding of the texture projection information specified in subclause 9.2.8.3 is invoked with p and tileID as inputs.

#### Decoding process for texture projection information

Inputs to this process are the patch index, p, and the current tile ID, tileID.

TileMeshpatchFrameScale[ tileID ][ p ] = tpi\_frame\_scale[ tileID ][ p ]  
 TileMeshpatchNumSubpatches[ tileID ][ p ] =   
 tpi\_subpatch\_count\_minus1[ tileID ][ p ] + 1 (7)  
 for( i = 0; i < TileMeshpatchNumSubpatches[ tileID ][ p ]; i++ ){  
 if( tpi\_face\_id\_present\_flag[ tileID ][ p ] )  
 TileMeshpatchSubpatchIdxToFaceId[ tileID ][ p ][ i ] =  
 si\_face\_id[ tileID ][ p ][ i ]   
 else  
 TileMeshpatchSubpatchIdxToFaceId[ tileID ][ p ][ i ] = i  
 TileMeshpatchSubpatchProjectionID[ tileID ][ p ][ i ] =  
 si\_projection\_id[ tileID ][ p ][ i ] (8)  
 TileMeshpatchSubpatchOrientationID[ tileID ][ p ][ i ] =  
 si\_orientation\_id[ tileID ][ p ][ i ] (9)  
 TileMeshpatchSubpatch2dPosX[ tileID ][ i ][ p ] =   
 si\_2d\_pos\_x[ tileID ][ p ][ i ] (10)  
 TileMeshpatchSubpatch2dPosY[ tileID ][ p ][ i ] =  
 si\_2d\_pos\_y[ tileID ][ p ][ i ] (11)  
 if( i > 0 ) {  
 TileMeshpatchSubpatch2dSizeX[ tileID ][ p ][ i ] =  
 ( si\_2d\_size\_x\_minus1\_diff[ tileID ][ p ][ i ] +  
 TileMeshpatchSubpatch2dSizeX[ tileID ][ p ][ i - 1 ] ) (12)  
 TileMeshpatchSubpatch2dSizeY[ tileID ][ p ][ i ] =  
 ( si\_2d\_size\_y\_minus1\_diff[ tileID ][ p ][ i ] +  
 TileMeshpatchSubpatch2dSizeY[ tileID ][ p ][ i - 1 ] ) (13)  
 } else {  
 TileMeshpatchSubpatch2dSizeX[ tileID ][ p ][ i ] =  
 ( si\_2d\_size\_x\_minus1\_diff[ tileID ][ p ][ i ] + 1 ) (14)  
 TileMeshpatchSubpatch2dSizeY[ tileID ][ p ][ i ] =  
 ( si\_2d\_size\_y\_minus1\_diff[ tileID ][ p ][ i ] + 1 ) (15)  
 }  
 TileMeshpatchSubpatchScale[ tileID ][ p ][ i ] =   
 SubpatchScalingFactor[ tileID ][ p ][ i ] (16)  
 }

#### Decoding process for meshpatch data units coded in skip prediction mode

Inputs to this process are the current patch index, p, and current tile ID, tileID.

First, refIdx is set to 0.

Then the reference atlas frame, refAtlasFrm, is selected as the atlas frame corresponding to the ( refIdx + 1 )-th entry in the reference atlas frame list RefAtlasFrmList, RefAtlasFrmList[ refIdx ].

If p is equal to 0, then PredictorIdx is set to 0.

The variable RefPatchIdx, which corresponds to the predictor patch index in the tile with ID equal to tileID, in the reference atlas frame, refAtlasFrm, is computed as:

RefPatchIdx = PredictorIdx (17)

and PredictorIdx is set to RefPatchIdx + 1.

The process described in subclause 9.2.8.6.2.1 is invoked with the variables refIdx, RefPatchIdx, and tileID as inputs, and the outputs are the variables refMeshpatch2dPosX, refMeshpatch2dPosY, refMeshpatch2dSizeX, refMeshpatch2dSizeY, refMeshpatchSubmeshID, refMeshpatchVertexCount, refMeshpatchFaceCount, refMeshpatchSubdivMethod, refMeshpatchSubdivCount, refMeshpatchDispCoordSys, refMeshpatchTransformMethod, the 1D arrays refMeshpatchVertexblockCount, refMeshpatchVertexCountLast, refMeshpatchAttributes2dPosX, refMeshpatchAttributes2dPosY, refMeshpatchAttributes2dSizeX, and refMeshpatchAttributes2dSizeY.

The following parameters are derived:

TileMeshpatchSubmeshID[ tileID ][ p ] = refMeshpatchSubmeshID (18)  
 TileMeshpatchSubdivCount[ tileID ][ p ] = refMeshpatchSubdivCount   
 for( i = 0; i < TileMeshpatchSubdivCount[ tileID ][ p ] ; i++ ){  
 TileMeshpatchSubdivMethod[ tileID ][ p ][ i ] = refMeshpatchSubdivMethod[ i ] (19)  
 }  
 TileMeshpatchDispCoordSys[ tileID ][ p ] = refMeshpatchDispCoordSys (20)  
 TileMeshpatchTransformMethod[ tileID ][ p ] = refMeshpatchTransformMethod (21)

If asve\_displacement\_id\_present\_flag is 0, the following applies:

TileMeshpatchVertexCount[ tileID ][ p ] = refMeshpatchVertexCount (22)  
 TileMeshpatchFaceCount[ tileID ][ p ] = refMeshpatchFaceCount (23)  
 TileMeshpatch2dPosX[ tileID ][ p ] = refMeshpatch2dPosX (24)  
 TileMeshpatch2dPosY[ tileID ][ p ] = refMeshpatch2dPosY (25)  
 TileMeshpatch2dSizeX[ tileID ][ p ] = refMeshpatch2dSizeX (26)  
 TileMeshpatch2dSizeY[ tileID ][ p ] = refMeshpatch2dSizeY (27)  
 for( i=0; i<= TileSubdivisionCount[ tileID ][ p ]; i++){  
 TilePatchVertexBlockCount[ tileID ][ p ][ i ] =  
 refMeshatchVertexBlockCount[ i ]  
 TilePatchVertexCountLast[ tileID ][ p ][ i ] = refMeshpatchVertexCountLast[ i ]  
 TilePatchVertexCount[ tileID ][ p ][ i ] =  
 TilePatchVertexBlockCount[ tileID ][ p ][ i ] \*  
 PatchPackingBlockSize \* PatchPackingBlockSize +  
 mpdu\_vertex\_count\_inlastblock[ tileID ][ patchIdx ][ i ]  
 TilePatchTotalVertexCount[ tileID ][ p ] +=  
 TilePatchVertexCount[ tileID ][ p ][ i ]  
 }

If asve\_displacement\_id\_present\_flag is 1, the following applies:

TileMeshpatchDisplID[ tileID ][ p ] = refMeshpatchDisplID (28)

The associated attribute 2D parameters are derived as follows:

for( i = 0; i< asve\_num\_attribute\_video; i++ ){  
 TileMeshpatchAttributes2dPosX[ tileID ][ p ][ i ] =  
 refMeshpatchAttributes2dPosX[ i ]  
 TileMeshpatchAttributes2dPosY[ tileID ][ p ][ i ] =  
 refMeshpatchAttributes2dPosY[ i ]  
 TileMeshpatchAttributes2dSizeX[ tileID ][ p ][ i ] =  
 refMeshpatchAttributes2dSizeX[ i ]   
 TileMeshpatchAttributes2dSizeY[ tileID ][ p ][ i ] =  
 refMeshpatchAttributes2dSizeY[ i ]   
 }

smIdx = SubmeshIDToIndex[ TileMeshpatchSubmeshID[ tileID ][ p ] ]  
 TileMeshpatchTexcoordProjectionFlag[ tileID ][ p ] = TexcoordProjectionFlag[ smIdx ]  
 TileMeshpatchTexcoordProjectionWidthNormalization[ tileID ][ p ] =  
 TexcoordProjectionWidthNormalization[ smIdx ]  
 TileMeshpatchTexcoordProjectionHeightNormalization[ tileID ][ p ] =  
 TexcoordProjectionHeightNormalization[ smIdx ]  
 TileMeshpatchTexcoordProjectionGutter[ tileID ][ p ] =  
 TexcoordProjectionGutter[ smIdx ]

If TileMeshpatchTexcoordProjectionFlag[ tileID ][ p ] is equal to 1, the process in subclause 9.2.8.6.2.2 is invoked with p and tileID as inputs, and the outputs are the variables refMeshpatchFrameScale and refMeshpatchNumSubpatches, the 1D arrays refMeshpatchSubpatchIdxToFaceId, refMeshpatchSubpatchProjectionID, refMeshpatchSubpatchOrientationID, refMeshpatchSubpatch2dPosX, refMeshpatchSubpatch2dPosY, refMeshpatchSubpatch2dSizeX, refMeshpatchSubpatch2dSizeY, and refMeshpatchSubpatchScale.

Then texture projection information is derived as follows:

TileMeshpatchFrameScale[ tileID ][ p ] = refMeshpatchFrameScale  
 TileMeshpatchNumSubpatches[ tileID ][ p ] = refMeshpatchNumSubpatches  
 for( i = 0; i < TileMeshpatchNumSubpatches[ tileID ][ p ]; i++ ){  
 TileMeshpatchSubpatchIdxToFaceId[ tileID ][ p ][ i ] =   
 refMeshpatchSubpatchIdxToFaceId,[ i ]  
 TileMeshpatchSubpatchProjectionID[ tileID ][ p ][ i ] =   
 refMeshpatchSubpatchProjectionID[ i ]  
 TileMeshpatchSubpatchOrientationID[ tileID ][ p ][ i ] =   
 refMeshpatchSubpatchOrientationID[ i ]   
 TileMeshpatchSubpatch2dPosX[ tileID ][ p ][ i ] =   
 refMeshpatchSubpatch2dPosX[ i ]  
 TileMeshpatchSubpatch2dPosY[ tileID ][ p ][ i ] =   
 refMeshpatchSubpatch2dPosY[ i ]   
 TileMeshpatchSubpatch2dSizeX[ tileID ][ p ][ i ] =   
 refMeshpatchSubpatch2dSizeX[ i ]  
 TileMeshpatchSubpatch2dSizeY[ tileID ][ p ][ i ] =   
 refMeshpatchSubpatch2dSizeY [ i ]   
 TileMeshpatchSubpatchScale[ tileID ][ p ][ i ] =   
 refMeshpatchSubpatchScale[ i ]   
 }

#### Decoding process for meshpatch data units coded in merge prediction mode

Inputs to this process are the current patch index, p, and the current tile ID, tileID.

First, the reference atlas frame index, refIdx, is derived as mmdu\_ref\_index[ tileID ][ p ].

Then the reference atlas frame, refAtlasFrm, is selected as the atlas frame corresponding to the ( refIdx + 1 )-th entry in the reference atlas frame list RefAtlasFrmList, RefAtlasFrmList[ refIdx ].

If p is equal to 0, then PredictorIdx is set to 0.

Then the predictor patch index, RefPatchIdx, in the tile with ID equal to tile ID, in the reference atlas frame, refAtlasFrm, is computed as:

RefPatchIdx = PredictorIdx  + mmdu\_patch\_index[ tileID ][ p ] (29)

and PredictorIdx is set to RefPatchIdx + 1.

The process described in subclause 9.2.8.6.2.1 is invoked with the variables refIdx, RefPatchIdx, and tileID as inputs, and the outputs are the variables refMeshpatch2dPosX, refMeshpatch2dPosY, refMeshpatch2dSizeX, refMeshpatch2dSizeY, refMeshpatchSubmeshID, refMeshpatchVertexCount, refMeshpatchFaceCount, refMeshpatchSubdivMethod, refMeshpatchSubdivCount, refMeshpatchDispCoordSys, refMeshpatchTransformMethod, the 1D arrays refMeshatchVertexBlockCount, refMeshpatchVertexCountLast, refMeshpatchAttributes2dPosX, refMeshpatchAttributes2dPosY, refMeshpatchAttributes2dSizeX, and refMeshpatchAttributes2dSizeY.

The following parameters are derived:

TileMeshpatchSubmeshID[ tileID ][ p ] = refMeshpatchSubmeshID (30)  
 TileMeshpatchSubdivCount[ tileID ][ p ] = refMeshpatchSubdivCount (31)  
 for( i = 0; i < TileMeshpatchSubdivCount[ tileID ][ p ] ; i++ ){  
 TileMeshpatchSubdivMethod[ tileID ][ p ][ i ] = refMeshpatchSubdivMethod[ i ] (32)  
 }  
 TileMeshpatchDispCoordSys[ tileID ][ p ] = refMeshpatchDispCoordSys (33)  
 TileMeshpatchTransformMethod[ tileID ][ p ] = refMeshpatchTransformMethod (34)

If asve\_displacement\_id\_present\_flag is 0, the following applies:35

TileMeshpatchFaceCount[ tileID ][ p ] = refMeshpatchFaceCount (36)  
 TileMeshpatch2dPosX[ tileID ][ p ] = refMeshpatch2dPosX (37)  
 TileMeshpatch2dPosY[ tileID ][ p ] = refMeshpatch2dPosY (38)  
 TileMeshpatch2dSizeX[ tileID ][ p ] = refMeshpatch2dSizeX (39)  
 TileMeshpatch2dSizeY[ tileID ][ p ] = refMeshpatch2dSizeY (40)

for( i=0; i<= TileSubdivisionCount[ tileID ][ p ]; i++){  
 TilePatchVertexBlockCount[ tileID ][ p ][ i ] = refMeshatchVertexBlockCount[ i ] TilePatchVertexCountLast[ tileID ][ p ][ i ] = refMeshpatchVertexCountLast[ i ]  
 TilePatchVertexCount[ tileID ][ p ][ i ] =  
 TilePatchVertexBlockCount[ tileID ][ p ][ i ] \*  
 PatchPackingBlockSize \* PatchPackingBlockSize +  
 mpdu\_vertex\_count\_inlastblock[ tileID ][ p ][ i ]  
 TilePatchTotalVertexCount[ tileID ][ p ] +=  
 TilePatchVertexCount[ tileID ][ p ][ i ]  
 }

If asve\_displacement\_id\_present\_flag is 1, the following applies:

TileMeshpatchDisplID[ tileID ][ p ] = refMeshpatchDisplID (41)

The associated attribute 2D parameters are derived as follows:

for( i = 0; i< asve\_num\_attribute\_video; i++ ){  
 TileMeshpatchAttributes2dPosX[ tileID ][ p ][ i ] =  
 refMeshpatchAttributes2dPosX[ i ]  
 TileMeshpatchAttributes2dPosY[ tileID ][ p ][ i ] =  
 refMeshpatchAttributes2dPosY[ i ]  
 TileMeshpatchAttributes2dSizeX[ tileID ][ p ][ i ] =  
 refMeshpatchAttributes2dSizeX[ i ]  
 TileMeshpatchAttributes2dSizeY[ tileID ][ p ][ i ] =  
 refMeshpatchAttributes2dSizeY[ i ]  
 }

smIdx = SubmeshIDToIndex[ TileMeshpatchSubmeshID[ tileID ][ p ] ]  
 TileMeshpatchTexcoordProjectionFlag[ tileID ][ p ] = TexcoordProjectionFlag[ smIdx ]  
 TileMeshpatchTexcoordProjectionWidthNormalization[ tileID ][ p ] =  
 TexcoordProjectionWidthNormalization[ smIdx ]  
 TileMeshpatchTexcoordProjectionHeightNormalization[ tileID ][ p ] =  
 TexcoordProjectionHeightNormalization[ smIdx ]  
 TileMeshpatchTexcoordProjectionGutter[ tileID ][ p ] =  
 TexcoordProjectionGutter[ smIdx ]

If mmdu\_texture\_projection\_present\_flag[ tileID ][ p ] is equal to 1, the process in subclause 9.2.8.6.2.2 is invoked with p and tileID as inputs, and the outputs are the variables refMeshpatchFrameScale and refMeshpatchNumSubpatches, the 1D arrays refMeshpatchSubpatchIdxToFaceId,, refMeshpatchSubpatchProjectionID, refMeshpatchSubpatchOrientationID, refMeshpatchSubpatch2dPosX, refMeshpatchSubpatch2dPosY, refMeshpatchSubpatch2dSizeX, refMeshpatchSubpatch2dSizeY, and refMeshpatchSubpatchScale.

Then texture projection information is derived as follows:

TileMeshpatchFrameScale[ tileID ][ p ] = refMeshpatchFrameScale  
TileMeshpatchNumSubpatches[ tileID ][ p ] = refMeshpatchNumSubpatches  
 for( i = 0; i < TileMeshpatchNumSubpatches[ tileID ][ p ]; i++ ){  
 TileMeshpatchSubpatchIdxToFaceId[ tileID ][ p ][ i ] =   
 refMeshpatchSubpatchIdxToFaceId,[ i ]  
 TileMeshpatchSubpatchProjectionID[ tileID ][ p ][ i ] =   
 refMeshpatchSubpatchProjectionID[ i ]  
 TileMeshpatchSubpatchOrientationID[ tileID ][ p ][ i ] =   
 refMeshpatchSubpatchOrientationID[ i ]   
 TileMeshpatchSubpatch2dPosX[ tileID ][ p ][ i ] =  
 refMeshpatchSubpatch2dPosX[ i ]  
 TileMeshpatchSubpatch2dPosY[ tileID ][ p ][ i ] =  
 refMeshpatchSubpatch2dPosY[ i ]   
 TileMeshpatchSubpatch2dSizeX[ tileID ][ p ][ i ] =  
 refMeshpatchSubpatch2dSizeX[ i ]  
 TileMeshpatchSubpatch2dSizeY[ tileID ][ p ][ i ] =  
 refMeshpatchSubpatch2dSizeY[ i ]   
 TileMeshpatchSubpatchScale[ tileID ][ p ][ i ] =  
 refMeshpatchSubpatchScale[ i ]   
 }  
 numSubPatchesUpdate = tpmi\_update\_subpatch\_count\_minus1[ tileID ][ p ]  
 for( i = 0; i < numSubPatchesUpdate; i++ ){  
 subPatchIdx = smi\_subpatch\_index[ tileID ][ p ][ i ]  
 TileMeshpatchSubpatch2dPosX[ tileID ][ p ][ subPatchIdx ] =  
 TileMeshpatchSubpatch2dPosX[ tileID ][ p ][ subPatchIdx ] +  
 smi\_2d\_pos\_x\_delta[ tileID ][ p ][ i ]   
 TileMeshpatchSubpatch2dPosY[ tileID ][ p ][ subPatchIdx ] =  
 TileMeshpatchSubpatch2dPosY[ tileID ][ p ][ subPatchIdx ] +  
 smi\_2d\_pos\_y\_delta[ tileID ][ p ][ i ]  
 TileMeshpatchSubpatch2dSizeX[ tileID ][ p ][ subPatchIdx ] =  
 TileMeshpatchSubpatch2dSizeX[ tileID ][ p ][ subPatchIdx ] +  
 smi\_2d\_size\_x\_delta[ tileID ][ p ][ i ]  
 TileMeshpatchSubpatch2dSizeY[ tileID ][ p ][ subPatchIdx ] =  
 TileMeshpatchSubpatch2dSizeY[ tileID ][ p ][ subPatchIdx ] +   
 smi\_2d\_size\_y\_delta[ tileID ][ p ][ i ]  
 }

#### Decoding process for meshpatch data units coded in inter prediction mode

##### General decoding process for meshpatch data units coded in inter prediction mode

Inputs to this process are the current patch index, p, and the current tile ID, tileID.

First, the reference atlas frame index, refIdx, is derived as imdu\_ref\_index[ tileID ][ p ].

Then the reference atlas frame, refAtlasFrm, is selected as the atlas frame corresponding to the ( refIdx + 1 )-th entry in the reference atlas frame list RefAtlasFrmList, RefAtlasFrmList[ refIdx ].

If p is equal to 0, then PredictorIdx is set to 0.

Then the corresponding patch index, RefPatchIdx, in the tile with ID equal to tile ID, in the reference atlas frame, refAtlasFrm, is computed as:

RefPatchIdx = PredictorIdx + imdu\_patch\_index[ tileID ][ p ] (42)

and PredictorIdx is set to RefPatchIdx + 1.

The process described in subclause 9.2.8.6.2.1 is invoked with the variables refIdx, RefPatchIdx, and tileID as inputs, and the outputs are the variables refMeshpatch2dPosX, refMeshpatch2dPosY, refMeshpatch2dSizeX, refMeshpatch2dSizeY, refMeshpatchSubmeshID, refMeshpatchVertexCount, refMeshpatchFaceCount, refMeshpatchSubdivMethod, refMeshpatchSubdivCount, refMeshpatchDispCoordSys, refMeshpatchTransformMethod,, the 1D arrays refMeshpatchBlockCount, refMeshpatchVertexCountLast, refMeshpatchAttributes2dPosX, refMeshpatchAttributes2dPosY, refMeshpatchAttributes2dSizeX, and refMeshpatchAttributes2dSizeY.

The following parameters are derived:

TileMeshpatchSubmeshID[ tileID ][ p ] = refMeshpatchSubmeshID   
 for( i = 0; i < TileMeshpatchSubdivCount[ tileID ][ p ] ; i++ ){  
 TileMeshpatchSubdivMethod[ tileID ][ p ][ i ] = refMeshpatchSubdivMethod[ i ]  
 }  
 TileMeshpatchDispCoordSys[ tileID ][ p ] = refMeshpatchDispCoordSys  
 TileMeshpatchTransformMethod[ tileID ][ p ] = refMeshpatchTransformMethod

If asve\_displacement\_id\_present\_flag is 0, the following applies:

TileMeshpatchFaceCount[ tileID ][ p ] = refMeshpatchFaceCount  
 TileMeshpatch2dPosX[ tileID ][ p ] =  
 refMeshpatch2dPosX + imdu\_2d\_delta\_pos\_x[ tileID ][ p ] \* PatchPackingBlockSize  
 TileMeshpatch2dPosY[ tileID ][ p ] =  
 refMeshpatch2dPosY + imdu\_2d\_delta\_pos\_y[ tileID ][ p ] \* PatchPackingBlockSize  
 TileMeshpatch2dSizeX[ tileID ][ p ] = refMeshpatch2dSizeX +  
 imdu\_2d\_delta\_size\_x[ tileID ][ p ] \* PatchSizeXQuantizer  
 TileMeshpatch2dSizeY[ tileID ][ p ] = refMeshpatch2dSizeY +  
 imdu\_2d\_delta\_size\_y[ tileID ][ p ] \* PatchSizeYQuantizer  
 TileMeshpatchSubdivCount[ tileID ][ p ] = InterPatchSubdivisionCount[ tileID ][ p ]  
 for( i=0; i<= TileMeshpatchSubdivCount[ tileID ][ p ]; i++){  
 TilePatchVertexBlockCount[ tileID ][ p ][ i ] =  
 (i < refMeshpatchSubdivCount ? refMeshpatchTotalVertexBlockCount[ i ] : 0 )  
 + imdu\_delta\_block\_count[ tileID ][ p ][ i ] + 1  
 TilePatchVertexCountLastTilePatchVertexCountLast[ tileID ][ p ][ i ] =  
 (i < refMeshpatchSubdivCount ? refMeshpatchVertexCountLast[ i ] : 0 )  
 + imdu\_delta\_last\_pos\_in\_blockin\_block[ tileID ][ p ][ i ]  
 TilePatchVertexCount[ tileID ][ p ][ i ] =  
 TilePatchVertexBlockCount[ tileID ][ p ][ i ] \*  
 PatchPackingBlockSize\*PatchPackingBlockSize +  
 ipdu\_vertex\_count\_inlastblock[ tileID ][ p ][ i ]  
 TilePatchTotalVertexCount[ tileID ][ p ] +=  
 TilePatchVertexCount[ tileID ][ p ][ i ]  
 }

If asve\_displacement\_id\_present\_flag is 1, the following applies:

TileMeshpatchDisplID[ tileID ][ p ] = refMeshpatchDisplID

The associated attribute 2D parameters are derived as follows:

for( i = 0; i< asve\_num\_attribute\_video; i++ ){  
 TileMeshpatchAttributes2dPosX[ tileID ][ p ][ i ] =  
 refMeshpatchAttributes2dPosX[ i ] +  
 imdu\_attribute\_2d\_delta\_pos\_x[ tileID ][ p ] \* PatchPackingBlockSize  
 TileMeshpatchAttributes2dPosY[ tileID ][ p ][ i ] =  
 refMeshpatchAttributes2dPosY[ i ] +  
 imdu\_attribute\_2d\_delta\_pos\_y[ tileID ][ p ] \* PatchPackingBlockSize  
 TileMeshpatchAttributes2dSizeX[ tileID ][ p ][ i ] =  
 refMeshpatchAttributes2dSizeX[ i ] +  
 imdu\_attribute\_2d\_delta\_size\_x[ tileID ][ p ] \* PatchSizeXQuantizer  
 TileMeshpatchAttributes2dSizeY[ tileID ][ p ][ i ] =  
 refMeshpatchAttributes2dSizeY[ i ] +  
 imdu\_attribute\_2d\_delta\_size\_y[ tileID ][ p ] \* PatchSizeXQuantizer  
 }

smIdx = SubmeshIDToIndex[ TileMeshpatchSubmeshID[ tileID ][ p ] ]  
 TileMeshpatchTexcoordProjectionFlag[ tileID ][ p ] = TexcoordProjectionFlag[ smIdx ]  
 TileMeshpatchTexcoordProjectionWidthNormalization[ tileID ][ p ] =  
 TexcoordProjectionWidthNormalization[ smIdx ]  
 TileMeshpatchTexcoordProjectionHeightNormalization[ tileID ][ p ] =  
 TexcoordProjectionHeightNormalization[ smIdx ]  
 TileMeshpatchTexcoordProjectionGutter[ tileID ][ p ] =  
 TexcoordProjectionGutter[ smIdx ]

If imdu\_texture\_projection\_present\_flag[ tileID ][ p ] is equal to 1, the process in subclause 9.2.8.6.2.2 is invoked with p and tileID as inputs, and the outputs are the variables refMeshpatchFrameScale and refMeshpatchNumSubpatches, the 1D arrays refMeshpatchSubpatchIdxToFaceId,, refMeshpatchSubpatchProjectionID, refMeshpatchSubpatchOrientationID, refMeshpatchSubpatch2dPosX, refMeshpatchSubpatch2dPosY, refMeshpatchSubpatch2dSizeX, refMeshpatchSubpatch2dSizeY, and refMeshpatchSubpatchScale.

Then texture projection information is derived as follows:

TileMeshpatchFrameScale[ tileID ][ p ] = tpii\_frame\_scale[ tileID ][ p ]  
TileMeshpatchNumSubpatches[ tileID ][ p ] = tpii\_subpatch\_count\_minus1[ tileId ][ p ] + 1  
 for( i = 0; i < TileMeshpatchNumSubpatches[ tileID ][ p ]; i++ ){  
 if( tpii\_subpatch\_inter\_present\_flag[ tileID ][ p ][ i ] ){  
 sIdx = sii\_subpatch\_index\_diff[ tileID ][ p ][ i ] + i  
 TileMeshpatchSubpatchIdxToFaceId[ tileID ][ p ][ i ] =   
 refMeshpatchSubpatchIdxToFaceId[ sIdx ]  
 TileMeshpatchSubpatchProjectionID[ tileID ][ p ][ i ] =   
 refMeshpatchSubpatchProjectionID[ sIdx ]  
 TileMeshpatchSubpatchOrientationID[ tileID ][ p ][ i ] =   
 refMeshpatchSubpatchOrientationID[ sIdx ]   
 TileMeshpatchSubpatch2dPosX[ tileID ][ p ][ i ] =  
 refMeshpatchSubpatch2dPosX[ sIdx ] +  
 sii\_2d\_pos\_x\_delta[ tileID ][ p ][ i ]  
 TileMeshpatchSubpatch2dPosY[ tileID ][ p ][ i ] =  
 refMeshpatchSubpatch2dPosY[ sIdx ] +  
 sii\_2d\_pos\_y\_delta[ tileID ][ p ][ i ]  
 TileMeshpatchSubpatch2dSizeX[ tileID ][ p ][ i ] =  
 refMeshpatchSubpatch2dSizeX[ sIdx ] +  
 sii\_2d\_size\_x\_delta[ tileID ][ p ][ i ]  
 TileMeshpatchSubpatch2dSizeY[ tileID ][ p ][ i ] =  
 refMeshpatchSubpatch2dSizeY[ sIdx ] +  
 sii\_2d\_size\_y\_delta[ tileID ][ p ][ i ]  
 TileMeshpatchSubpatchScale[ tileID ][ p ][ i ] =  
 refMeshpatchSubpatchScale[ i ]   
 } else {  
 if( tpii\_face\_id\_present\_flag[ tileID ][ p ] )  
 TileMeshpatchSubpatchIdxToFaceId[ tileID ][ p ][ i ] =  
 si\_face\_id[ tileID ][ p ][ i ]   
 else  
 TileMeshpatchSubpatchIdxToFaceId[ tileID ][ p ][ i ] = i  
 TileMeshpatchSubpatchProjectionID[ tileID ][ p ][ i ] =  
 si\_projection\_id[ tileID ][ p ][ i ]  
 TileMeshpatchSubpatchOrientationID[ tileID ][ p ][ i ] =  
 si\_orientation\_id[ tileID ][ p ][ i ]  
 TileMeshpatchSubpatch2dPosX[ tileID ][ i ][ p ] =   
 si\_2d\_pos\_x[ tileID ][ p ][ i ]  
 TileMeshpatchSubpatch2dPosY[ tileID ][ p ][ i ] =  
 si\_2d\_pos\_y[ tileID ][ p ][ i ]  
 if( i > 0 ) {  
 TileMeshpatchSubpatch2dSizeX[ tileID ][ p ][ i ] =  
 ( si\_2d\_size\_x\_minus1\_diff[ tileID ][ p ][ i ] +  
 TileMeshpatchSubpatch2dSizeX[ tileID ][ p ][ i - 1 ] )  
 TileMeshpatchSubpatch2dSizeY[ tileID ][ p ][ i ] =  
 ( si\_2d\_size\_y\_minus1\_diff[ tileID ][ p ][ i ] +  
 TileMeshpatchSubpatch2dSizeY[ tileID ][ p ][ i - 1 ] )  
 } else {  
 TileMeshpatchSubpatch2dSizeX[ tileID ][ p ][ i ] =  
 ( si\_2d\_size\_x\_minus1\_diff[ tileID ][ p ][ i ] + 1 )  
 TileMeshpatchSubpatch2dSizeY[ tileID ][ p ][ i ] =  
 ( si\_2d\_size\_y\_minus1\_diff[ tileID ][ p ][ i ] + 1 )  
 }  
 TileMeshpatchSubpatchScale[ tileID ][ p ][ i ] =   
 SubpatchScalingFactor[ tileID ][ p ][ i ]  
 }  
 }

##### Derivation of inter reference patch parameters

###### General derivation of inter reference patch parameters

Inputs to this process are the atlas frame reference index, refIdx, the patch reference index, refPatchIdx, and the current tile ID, tileID.

Outputs to this process are the variables refMeshpatch2dPosX, refMeshpatch2dPosY, refMeshpatch2dSizeX, refMeshpatch2dSizeY, refMeshpatchSubmeshID, refMeshpatchDisplID, refMeshpatchVertexCount, refMeshpatchFaceCount, refMeshpatchSubdivMethod, refMeshpatchSubdivCount, refMeshpatchDispCoordSys, refMeshpatchTransformMethod, the 1D arrays refMeshpatchAttributes2dPosX, refMeshpatchAttributes2dPosY, refMeshpatchAttributes2dSizeX, and refMeshpatchAttributes2dSizeY.

First, the meshpatch, refMeshpatch, is determined that has an index equal to refPatchIdx in the tile with tile ID equal to tileID of the ( refIdx + 1 )-th entry of the reference atlas frame list RefAtlasFrmList, RefAtlasFrmList[ refIdx ].

Then, the outputs of this process are derived based on the associated parameters of the meshpatch, refMeshpatch as follows:

refMeshpatch2dPosX = TileMeshpatch2dPosX[ tileID ][ refPatchIdx ]  
 refMeshpatch2dPosY = TileMeshpatch2dPosY[ tileID ][ refPatchIdx ]  
 refMeshpatch2dSizeX = TileMeshpatch2dSizeX[ tileID ][ refPatchIdx ]  
 refMeshpatch2dSizeY = TileMeshpatch2dSizeY[ tileID ][ refPatchIdx ]  
 refMeshpatchSubmeshID = TileMeshpatchSubmeshID[ tileID ][ RefPatchIdx ]  
 refMeshpatchDisplID = TileMeshpatchDisplID[ tileID ][ RefPatchIdx ]  
 refMeshpatchVertexCount = TileMeshpatchVertexCount[ tileID ][ refPatchIdx ]  
 refMeshpatchFaceCount = TileMeshpatchFaceCount[ tileID ][ refPatchIdx ]  
 refMeshpatchSubdivCount = TileMeshpatchSubdivCount[ tileID ][ refPatchIdx ]  
 for( i = 0; i < refMeshpatchSubdivCount ; i++ ){  
 refMeshpatchSubdivMethod[ i ] =  
 TileMeshpatchSubdivMethod[ tileID ][ refPatchIdx  ][ i ]  
 } refMeshpatchDispCoordSys = TileMeshpatchDispCoordSys[ tileID ][ refPatchIdx ]  
 refMeshpatchTransformMethod = TileMeshpatchTransformMethod[ tileID ][ refPatchIdx ]

The associated attribute 2D parameters are derived as follows:

for( i = 0; i< asve\_num\_attribute\_video; i++ ){  
 refMeshpatchAttributes2dPosX[ i ] =  
 TileMeshpatchAttributes2dPosX[ tileID ][ refPatchIdx ][ i ]  
 refMeshpatchAttributes2dPosY[ i ] =  
 TileMeshpatchAttributes2dPosY[ tileID ][ refPatchIdx ][ i ]  
 refMeshpatchAttributes2dSizeX[ i ] =  
 TileMeshpatchAttributes2dSizeX[ tileID ][ refPatchIdx ][ i ]  
 refMeshpatchAttributes2dSizeY[ i ] =  
 TileMeshpatchAttributes2dSizeY[ tileID ][ refPatchIdx ][ i ]  
 }

where TileMeshpatch2dPosX , TileMeshpatch2dPosY, TileMeshpatch2dSizeX, TileMeshpatch2dSizeY, TileMeshpatchSubmeshID, TileMeshpatchDisplID, TileMeshpatchVertexCount, TileMeshpatchFaceCount, TileMeshpatchSubdivMethod, TileMeshpatchSubdivCount, TileMeshpatchDispCoordSys, TileMeshpatchTransformMethod, TileMeshpatchAttributes2dPosX, TileMeshpatchAttributes2dPosY, TileMeshpatchAttributes2dSizeX, and TileMeshpatchAttributes2dSizeY are the associated tile meshpatch parameters from the meshpatch with index refPatchIdx in the tile ID equal to tileID of the refIdx-th entry of the reference atlas frame list struct, RefAtlasFrmList.

###### Texture projection information derivation of inter reference patch parameters

Inputs to this process are the atlas frame reference index, refIdx, the patch reference index, refPatchIdx, and the current tile ID, tileID.

First, the meshpatch, refMeshpatch, is determined that has an index equal to refPatchIdx in the tile with tile ID equal to tileID of the ( refIdx + 1 )-th entry of the reference atlas frame list RefAtlasFrmList, RefAtlasFrmList[ refIdx ].

Outputs to this process are the variables refMeshpatchFrameScale and refMeshpatchNumSubpatches, the 1D arrays refMeshpatchSubpatchIdxToFaceId,, refMeshpatchSubpatchProjectionID, refMeshpatchSubpatchOrientationID, refMeshpatchSubpatch2dPosX, refMeshpatchSubpatch2dPosY, refMeshpatchSubpatch2dSizeX, refMeshpatchSubpatch2dSizeY, and refMeshpatchSubpatchScale.

Then, the outputs of this process are derived based on the associated parameters of the meshpatch, refMeshpatch as follows:

refMeshpatchFrameScale = TileMeshpatchFrameScale[ tileID ][ refPatchIdx ]  
 refMeshpatchNumSubpatches = TileMeshpatchNumSubpatches[ tileID ][ refPatchIdx ]  
 for( i = 0; i < TileMeshpatchNumSubpatches[ tileID ][ p ]; i++ ){  
 refMeshpatchSubpatchIdxToFaceId[ i ] =   
 TileMeshpatchSubpatchIdxToFaceId[ tileID ][ refPatchIdx ][ i ]  
 refMeshpatchSubpatchProjectionID[ i ] =   
 TileMeshpatchSubpatchProjectionID[ tileID ][ refPatchIdx ][ i ]  
 refMeshpatchSubpatchOrientationID [ i ] =   
 TileMeshpatchSubpatchOrientationID[ tileID ][ refPatchIdx ][ i ]   
 refMeshpatchSubpatch2dPosX[ i ] =   
 TileMeshpatchSubpatch2dPosX[ tileID ][ refPatchIdx ][ i ]  
 refMeshpatchSubpatch2dPosY[ i ] =   
 TileMeshpatchSubpatch2dPosY[ tileID ][ refPatchIdx ][ i ]   
 refMeshpatchSubpatch2dSizeX[ i ] =   
 TileMeshpatchSubpatch2dSizeX[ tileID ][ refPatchIdx ][ i ]  
 refMeshpatchSubpatch2dSizeY[ i ] =   
 TileMeshpatchSubpatch2dSizeY[ tileID ][ refPatchIdx ][ i ]  
 refMeshpatchSubpatchScale[ i ] =   
 TileMeshpatchSubpatchScale[ tileID ][ refPatchIdx  ][ i ]   
 }

### Conversion of tile level meshpatch information to atlas level meshpatch information

#### General

In some implementations it is possible that the reconstruction process may be performed using tile information directly, i.e. the reconstruction process is tile based. In other implementations the reconstruction process may be performed using all tile information, i.e. is atlas based. In that particular case, a conversion of tile level patch information to atlas level patch information may be necessary. This conversion is performed in this subclause.

#### Conversion of tile level meshpatch information to atlas level meshpatch information

##### General

In this subclause tile level patch information is converted to atlas level meshpatch information, using also the processes AttributeTileMeshpatchParamsToAtlas( atlasPatchIdx, t, p ) and SubpatchTileParamsToAtlas( atlasPatchIdx, tileID, p ) defined in subclauses 9.2.9.2.2 and 9.2.9.2.3, respectively, as follows:

AtlasTotalNumMeshpatches = 0  
 atlasPatchIdx = 0  
 for( t = 0; t <= afti\_num\_tiles\_in\_atlas\_frame\_minus1; t++ ) {  
 tileID = TileIndexToID[ t ]  
 tileOffsetX = TileOffsetX[ t ]  
 tileOffsetY = TileOffsetY[ t ]  
 for( p = 0; p < AtduTotalNumMeshpatches[ tileID ]; p++ ) {  
 AtlasMeshpatch2dSizeX[ atlasPatchIdx ] =  
 TileMeshpatch2dSizeX[ tileID ][ p ]  
 AtlasMeshpatch2dSizeY[ atlasPatchIdx ] =  
 TileMeshpatch2dSizeY[ tileID ][ p ]  
 AtlasMeshpatch2dPosX[ atlasPatchIdx ] =  
 TileMeshpatch2dPosX[ tileID ][ p ] + tileOffsetX  
 AtlasMeshpatch2dPosY[ atlasPatchIdx ] =  
 TileMeshpatch2dPosY[ tileID ][ p ] + tileOffsetY  
 AtlasMeshpatchSubmeshID[ atlasPatchIdx ] =  
 TileMeshpatchSubmeshID[ tileID ][ p ]  
 AtlasMeshpatchDisplID[ atlasPatchIdx ] =  
 TileMeshpatchDisplID[ tileID ][ p ]  
 AtlasMeshpatchVertexCount[ atlasPatchIdx ] =  
 TileMeshpatchVertexCount[ tileID ][ p ]  
 AtlasMeshpatchFaceCount[ [ atlasPatchIdx ] =  
 TileMeshpatchFaceCount[ tileID ][ p ] + 1  
 AtlasMeshpatchSubdivCount[ atlasPatchIdx ] =  
 TileMeshpatchSubdivCount[ tileID ][ p ]  
 for( i = 0; i < AtlasMeshpatchSubdivCount[ atlasPatchIdx ]; i++ ){  
 AtlasMeshpatchSubdivMethod[ atlasPatchIdx ][ i ] =  
 TileMeshpatchSubdivMethod[ tileID ][ p ][ i ]  
 }  
 AtlasMeshpatchDispCoordSys[ atlasPatchIdx ] =  
 TileMeshpatchDispCoordSys[ tileID ][ p ]  
 AtlasMeshpatchTransformMethod[ atlasPatchIdx ] =  
 TileMeshpatchTransformMethod[ tileID ][ p ]  
 for( i=0; i<= AtlasPatchSubdivisionCount[ atlasPatchIdx ]; i++){  
 AtlasPatchVertexBlockCount[ atlasPatchIdx ][ i ] =  
 TilePatchVertexBlockCount[ tileID ][ p ][ i ]   
 AtlasPatchVertexCountLast[ atlasPatchIdx ][ i ] =  
 TilePatchVertexCountLast[ tileID ][ p ][ i ]  
 AtlasPatchVertexCount[ atlasPatchIdx ][ i ] =  
 TileVertexCount[ tileID ][ p ][ i ]  
 AtlasPatchTotalVertexCount[ atlasPatchIdx ] =  
 TilePatchTotalVertexCount[ tileID ][ p ]  
 }  
 if( asve\_num\_attribute\_video > 0 ){  
 AttributeTileMeshpatchParamsToAtlas( atlasPatchIdx, t, p )  
 }  
 AtlasMeshpatchTexcoordProjectionFlag[ atlasPatchIdx ] =  
 TileMeshpatchTexcoordProjectionFlag[ tileID ][ p ]  
 AtlasMeshpatchTexcoordProjectionWidthNormalization[ atlasPatchIdx ] =   
 TileMeshpatchTexcoordProjectionWidthNormalization[ tileID ][ p ]  
 AtlasMeshpatchTexcoordProjectionHeightNormalization[ atlasPatchIdx ] =  
 TileMeshpatchTexcoordProjectionHeightNormalization[ tileID ][ p ]  
 AtlasMeshpatchTexcoordProjectionGutter[ atlasPatchIdx ] =  
 TileMeshpatchTexcoordProjectionGutter[ tileID ][ p ]  
 if( AtlasMeshpatchTexcoordProjectionFlag[ atlasPatchIdx ] )  
 SubpatchTileParamToAtlas( atlasPatchIdx, tileID, p )  
 atlasPatchIdx += 1  
 }  
 }  
 AtlasTotalNumMeshpatches = atlasPatchIdx

In this version of the document AtlasPatchVertexCount[ atlasPatchIdx ][ i ] shall be greater than 2 for each patch with index atlasPatchIdx and for each subdivision with index i.

##### Process of copying attribute meshpatch parameters from a tile to an atlas representation

The process AttributeTileMeshpatchParamsToAtlas( atlasPatchIdx, t, p ) is defined as follows:

AttributeTileMeshpatchParamsToAtlas( atlasPatchIdx, t, p ) { for( ai = 0; ai< asve\_num\_attribute\_video; ai++ ) {  
 tileIDAtt = TileIndexToIDAtt[ ai ][ t ]  
 tileOffsetAttX = TileOffsetXAtt[ ai ][ t ]  
 tileOffsetAttY = TileOffsetYAtt[ ai ][ t ]   
 if( asve\_attribute\_subtexture\_enabled\_flag[ ai ] ) {  
 AtlasMeshpatchAttributes2dPosX[ atlasPatchIdx ][ ai ] =   
 TileMeshpatchAttributes2dPosX[ tileIDAtt ][ p ][ ai ] + tileOffsetAttX  
 AtlasMeshpatchAttributes2dPosY[ atlasPatchIdx ][ ai ] =   
 TileMeshpatchAttributes2dPosY[ tileIDAtt ][ p ][ ai ] + tileOffsetAttY  
 AtlasMeshpatchAttributes2dSizeX[ atlasPatchIdx ][ ai ] =  
 TileMeshpatchAttributes2dSizeX[ tileIDAtt ][ p ][ ai ]  
 AtlasMeshpatchAttributes2dSizeY[ atlasPatchIdx ][ ai ] =   
 TileMeshpatchAttributes2dSizeY[ tileIDAtt ][ p ][ ai ]  
 }else{  
 AtlasMeshpatchAttributes2dPosX[ atlasPatchIdx ][ ai ] = 0  
 AtlasMeshpatchAttributes2dPosY[ atlasPatchIdx ][ ai ] = 0  
 AtlasMeshpatchAttributes2dSizeX[ atlasPatchIdx ][ ai ] =  
 TileWidthAtt[ ai ][ t ]  
 AtlasMeshpatchAttributes2dSizeY]atlasPatchIdx ][ i ] =  
 TileHeightAtt[ ai ][ t ]  
 }  
 }  
 }

##### Process of copying subpatch parameters from a tile to an atlas representation

The process SubpatchTileParamToAtlas( atlasPatchIdx, tileID, p ) is defined as follows:

SubpatchTileParamToAtlas( atlasPatchIdx, tileID, p ) {  
 AtlasMeshpatchFrameScale[ atlasPatchIdx ] =  
 TileMeshpatchFrameScale[ tileID ][ p ]  
 AtlasMeshpatchNumSubpatches[ atlasPatchIdx ] =  
 TileMeshpatchNumSubpatches[ tileID ][ p ]  
 for( i = 0; i < AtlasMeshpatchNumSubpatches[ atlasPatchIdx ]; i++ ){  
 AtlasMeshpatchSubpatchFaceId[ atlasPatchIdx ][ i ] =  
 TileMeshpatchSubpatchFaceId[ tileID ][ p ][ i ]  
 AtlasMeshpatchSubpatchProjectionID[ atlasPatchIdx ][ i ] =  
 TileMeshpatchSubpatchProjectionID[ tileID ][ p ][ i ]  
 AtlasMeshpatchSubpatchOrientationID[ atlasPatchIdx ][ i ] =  
 TileMeshpatchSubpatchOrientationID[ tileID ][ p ][ i ])  
 AtlasMeshpatchSubpatch2dPosX[ atlasPatchIdx ][ i ] =  
 TileMeshpatchSubpatch2dPosX[ tileID ][ i ][ p ]  
 AtlasMeshpatchSubpatch2dPosY[ atlasPatchIdx ][ i ] =  
 TileMeshpatchSubpatch2dPosY[ tileID ][ p ][ i ]  
 AtlasMeshpatchSubpatch2dSizeX[ atlasPatchIdx ][ i ] =  
 TileMeshpatchSubpatch2dSizeX[ tileID ][ p ][ i ]  
 AtlasMeshpatchSubpatch2dSizeY[ atlasPatchIdx ][ i ] =  
 TileMeshpatchSubpatch2dSizeY[ tileID ][ p ][ i ]   
 AtlasMeshpatchSubpatchScale[ atlasPatchIdx ][ i ] =  
 TileMeshpatchSubpatchScale[ tileID ][ p ][ i ]   
 }  
 }

## Occupancy video decoding process

The specifications in ISO/IEC 23090-5(2E):2023 subclause 9.3 apply.

## Geometry video decoding process

The specifications in ISO/IEC 23090-5(2E):2023 subclause 9.4 apply with following additions.

Otherwise, if ptl\_profile\_codec\_group\_idc is equal to X, subclause J.8 apply.

[Ed. Note: profile X will be defined in the future release of the document]

## Attribute video decoding process

The specifications in ISO/IEC 23090-5(2E):2023 subclause 9.5 apply.

## Packed video decoding process

The specifications in ISO/IEC 23090-5(2E):2023 subclause 9.6 apply.

## Common atlas decoding process

The specifications in ISO/IEC 23090-5(2E):2023 subclause 9.7 apply.

## Sub-bitstream extraction process

The specifications in ISO/IEC 23090-5(2E):2023 subclause 9.8 apply.

## Basemesh decoding process

The decoding process of a basemesh component associated with the atlas, with atlas ID DecAtlasID, is performed as follows.

For a basemesh component, the codec is first determined through external means outside of this document or as follows:

* If ptl\_profile\_codec\_group\_idc is equal to 127 and the component codec mapping SEI message is present, the codec can be determined as specified in subclause ISO/IEC 23090-5:2023:F.2.4 with vps\_ext\_bmesh\_data\_substream\_codec\_id[ DecAtlasID] as the codec mapping index.
* Otherwise, for values of ptl\_profile\_codec\_group\_idc in the range of 0 .. 4, the codec can be determined to be the CodecGroup entry corresponding to the value of ptl\_profile\_codec\_group\_idc in Annex A.

Then, the basemesh decoding process, according to the corresponding coding specification, is invoked using the basemesh sub-bitstreams present in the V3C bitstream as the input.

Outputs of this process are:

* DecBasemeshFrameCount, indicating the number of decoded basemesh frames,
* a 1D array DecSubmeshesPerFrameCount, indicating the number of submeshes, where the dimension corresponds to the decoded basemesh frame index,
* a 2D array of structures DecBasemeshFrames, indicating the decoded submesh frames, where the dimensions correspond to the decoded basemesh frame index and the decoded submesh index, respectively. Each element of the DecBasemeshFrames has the following members:
* submeshId, indicating the ID of a given submesh of the basemesh,
* verCoordCount, indicating the number of vertex coordinates,
* verCoordBitDepth, indicating the vertex coordinate bit depth,
* faceCount, indicating the number of faces,
* attributeCount, indicating the number of attributes and
* the following 1D arrays of size attributeCount:
  + attrTypeArray, indicating the type of attribute, according to Table 1,
  + attrValueDimensionArray, indicating the dimension of attribute values,
  + attrValueCountArray, indicating the number of attribute values,
  + attrValueIsPerVertexArray, indicating if the attribute values are defined per face or per vertex,
  + attrValueBitDepthArray indicating the attribute value bit depth,

where the dimensions correspond to the attribute index,

* a 2D array verCoordsArray, indicating vertex coordinate values, where the dimensions correspond to the vertex coordinate value index and the vertex coordinate value dimension index, respectively,
* 2D array verCoordConnArray, indicating vertex coordinates connectivity, where the dimensions correspond to the face index and the face dimension index, 0..2, respectively,
* a 3D array attrValuesArray, indicating attribute values, where the dimensions correspond to the attribute index, the attribute value index, and the attribute value dimension index, respectively,
* a 3D array attrConnArray, indicating attribute connectivity, where the dimensions correspond to the attribute index, the attribute face index, and the attribute face dimension index, 0..2, respectively,
* the following 1D arrays of size DecBaseMeshFrameCount:
  + DecBasemeshOutOrdIdx, indicating the basemesh output order index, and
  + DecBasemeshCompTime, indicating the basemesh composition time, where the dimensions correspond to the decoded basemesh frame index.

It is requirement that DecSubmeshesPerFrameCount of DecBasemeshFrames at composition time x shall be equal to NumSubMeshes in atlas frame at the same composition time x.

# Pre-reconstruction process

The specifications in ISO/IEC 23090-5(2E):2023 clause 10 do not apply.

# Reconstruction process

## General

The reconstruction process depends on the profiles defined in Annex A

The processes described in this subclause are invoked for decoded atlas frames, decoded basemesh frames, decoded video frames, and syntax elements associated with the same atlas ID, identified by a variable RecAtlasID.

The reconstruction process for the current mesh frame with a composition time index compTimeIdx, takes as inputs the syntax elements and upper-case variables from subclauses 8 and 9, and the following arrays:

* a 1D array of submeshesPerFrameCountNF[ compTimeIdx ], specifying the number of decoded submeshes per frame, in nominal format,
* a 2D array of structures basemeshFramesNF[ compTimeIdx ][ submeshIdx ], specifying the decoded basemesh in nominal format, where submeshIdx is in the range of 0 to submeshesPerFrameCountNF[ compTimeIndx ] – 1
* when vps\_geometry\_video\_present\_flag[ RecAtlasID ] is equal to 1, and if the profile or geo codec ID indicates the usage of a 2D video codec, then a 5D array geoFramesNF[ 0 ][ compTimeIdx ][ 0 ][ y ][ x ] specifying the decoded geometry frames in the nominal format, where y is in the range of 0 to asps\_frame\_height – 1, inclusive, and x is in the range of 0 to asps\_frame\_width – 1, inclusive, otherwise, if the profile or geo codec indicates an arithmetic codec, then a 3D array dispFramesNF[ compTimeIdx ][ dispIdx ][ dispDimIdx ] specifying the decoded displacement in nominal format, where dispIdx is in the range of 0 to DispCountPerFrame[ compTimeIdx ]  – 1, inclusive, and dispDimIdx is in the range of 0 to DispDimension – 1.
* when ai\_attribute\_count[ RecAtlasID ] is greater than 0, a 6D array attrFramesNF[ attrIdx ][ 0 ][ compTimeIdx ][ compIdx ][ y ][ x ] specifying the decoded attribute frames in the nominal format, where attrIdx is in the range of 0 to ai\_attribute\_count[ RecAtlasID ] − 1, inclusive, compIdx is in the range of 0 to ai\_attribute\_dimension\_minus1[ RecAtlasID ][ attrIdx ], inclusive, y is in the range of 0 to vps\_ext\_attribute\_frame\_height[ attrIdx ] – 1, inclusive, and x is in the range of 0 to vps\_ext\_attribute\_frame\_width[ attrIdx ] – 1, inclusive.

Output of this process is a sequence of reconstructed mesh frames before the post-reconstruction step with their respective attribute maps.

A struct RecMeshFrame that represents a reconstructed mesh frame has the following members:

* a variable verCoordCount, specifying the number of vertex coordinates in the reconstructed mesh frame,
* a 2D array verCoordsArray, of size verCoordCount × 3, specifying the reconstructed vertex coordinates,
* a variable faceCount, specifying the number of vertex coordinates faces,
* a 2D array verCoordFaces, of size faceCount × 3, specifying the reconstructed connectivity indices associated with the vertex coordinates,
* The following 1D arrays of size vps\_ext\_bmesh\_data\_attribute\_count[ RecAttlasID ]:
  + a variable attrValueCountArray, specifying the number of values for an attribute in the reconstructed mesh frame,
  + a variable attrValueDimensionArray, specifying the dimension of an attribute,
* a 3D array attrValuesArray, indicating attribute values, where the dimensions correspond to the attribute index, the attribute value index, and the attribute value dimension index, respectively,
* a 3D array attrFacesArray, indicating attribute connectivity, where the dimensions correspond to the attribute index, the attribute face index, and the attribute face dimension index, 0..2, respectively,
* a variable submeshCount, specifying the number of submeshes in the reconstructed mesh frame,
* a 1D array submeshVerCoordCount, of size submeshCount, specifying the number of reconstructed vertices associated with the submesh,
* a 1D array submeshLodCount, of size submeshCount, specifying the number of LoDs associated with the submesh,
* a 2D array submeshVerCoordPerLodCount, of size submeshCount × submeshLodCount[ smIdx ], specifying the number of reconstructed vertices associated with the i-th LoD of the submesh with index smIdx.

The output attribute maps are represented as:

* a 4D array aFrame, of size attrCount × attrDim × height × width, where attrCount is equal to ai\_attribute\_count[ RecAtlasID ], attrDim is equal to ai\_attribute\_dimension\_minus1[ RecAtlasID ][ attrIdx ] + 1, height is equal to vps\_ext\_attribute\_frame\_height[ attrIdx ], and width is equal to vps\_ext\_attribute\_frame\_width[ attrIdx ], specifying the reconstructed attribute maps associated with the reconstructed mesh.

The conversion of variable is performed as follows:

DisplacementDim = Min( DisplacementDim, DispDimension )

The reconstruction of mesh is performed as follows:

* The output 4D array aFrame[ attrIdx ][ compIdx ][ y ][ x ] is obtained as follows:

for( a = 0; a < ai\_attribute\_count[ RecAtlasID ]; a++ ){  
 for( j = 0; j < vps\_ext\_attribute\_frame\_height[ a ]; j++ ){  
 for( i = 0; i < vps\_ext\_attribute\_frame\_width[ a ]; i++ ){  
 for( c = 0;  
 c < ai\_attribute\_dimension\_minus1[ RecAtlasID ][ a ] + 1; c++ ){  
 aFrame[ a ][ c ][ j ][ i ] =  
 attrFramesNF[ a ][ 0 ][ compTimeIdx ][ c ][ j ][ i ]  
 }  
 }  
 }  
 }

* An array gFrame[ compIdx ][ y ][ x ] is derived as follows:

for( j = 0; j < asps\_frame\_height; j++ )  
 for( i = 0; i < asps\_frame\_width; i++ )  
 for( c = 0; c < DisplacementDim; c++ )  
 gFrame[ c ][ j ][ i ] = geoFramesNF[ 0 ][ compTimeIdx ][ c ][ j ][ i ]

* For p = 0..AtlasTotalNumMeshpatches - 1, the following applies:
  + An array dFrame[ dispIdx ][ dispDimIdx ] is derived with dispIdx equals to DisplIDToIndex[ AtlasMeshpatchDisplID[ p ] ] as as follows:

for( i = 0; i < 3; i++ )  
 if( ( DispDimension == 1 ) && ( i > 1 ) )  
 dFrame[ dispIdx ][ i ] = 0  
 else  
 dFrame[ dispIdx ][ i ] = dispFramesNF[ compTimeIdx ][ j ][ i ]

* For p = 0..AtlasTotalNumMeshpatches – 1, the following applies:
  + a struct, baseSubmeshFrame is derived with submeshId equals to BaseMeshSubmeshIDToIndex[ AtlasMeshpatchSubmeshID[ p ] ] as follows:

baseSubmeshFrame.verCoordCount =  
 basemeshFramesNF[ compTimeIdx ][ submeshIdx ].verCoordCount  
 baseSubmeshFrame.faceCount =  
 basemeshFramesNF[ compTimeIdx ][ submeshIdx ].faceCount  
 baseSubmeshFrame.verCoordsArray =  
 basemeshFramesNF[ compTimeIdx ][ submeshIdx ].verCoordsArray  
 baseSubmeshFrame.verCoordFacesArray =  
 basemeshFramesNF[ compTimeIdx ][ submeshIdx ].verCoordFacesArray  
 basemeshFrame.attrIdxTexCoords = -1  
 basemeshFrame.attrIdxFacegroupIds = -1  
 for( i =0; i < vps\_ext\_bmesh\_data\_attribute\_count[ RecAtlasID ]; i++ ){  
 basemeshFrame.attrValueCountArray[ i ] =  
 basemeshFramesNF[ compTimeIdx ][ submeshIdx ].attrValueCount[ i ]  
 basemeshFrame.attrValueDimensionArray[ i ] =  
 basemeshFramesNF[ compTimeIdx ][ submeshIdx ].attrValueDimensionArray[ i ]  
 basemeshFrame.attrValuesArray[ i ] =  
 basemeshFramesNF[ compTimeIdx ][ submeshIdx ].attrValuesArray[ i ]  
 basemeshFrame.attrFacesArray[ i ] =  
 basemeshFramesNF[ compTimeIdx ][ submeshIdx ].attrFacesArray[ i ]  
 if( vps\_ext\_bmesh\_attribute\_type[ RecAtlasID ][ i ] == ATTR\_TEXCOORD )  
 basemeshFrame.attrIdxTexCoords = i  
 if( vps\_ext\_bmesh\_attribute\_type[ RecAtlasID ][ i ] == ATTR\_FACEGROUP\_ID )  
 basemeshFrame.attrIdxFacegroupIds = i  
 }

* the submesh reconstruction process described in subclause 11.2 is invoked with the structure baseSubmeshFrame, the array gFrame or the array dFrame and the patch index p as inputs. The output is recSubmeshFrames[ p ], the p-th element of 1D array of structs.

Then, the submesh appending process described in subclause is invoked 11.3 with a variable AtlasTotalNumMeshpatches and the 1D array recSubmeshFrames, of size AtlasTotalNumMeshpatches, as inputs and a struct recMeshFrame as an output.

The additional variables are calculated and used for post-reconstruction:

recMeshFrame.submeshCount = AtlasTotalNumMeshpatches  
 for( i = 0; i < AtlasTotalNumMeshpatches; i++ ) {  
 recMeshFrame.submeshVerCoordCount[ i ] = recSubmeshFrames[ p ].verCoordCount  
 recMeshFrame.submeshLodCount = AtlasMeshpatchSubdivCount[ p ]  
 for( j = 0; j < AtlasMeshpatchSubdivCount[ p ]; j++ ) {  
 recMeshFrame.submeshVerCoordPerLodCount[ i ][ j ] =   
 AtlasMeshpatchVertexCountPerSubdiv[ p ][ j ]  
 }  
 }

[Ed. Note DBG: we assume one-to-one relationship between submesh and patch]

## Submesh Reconstruction Process

### General

Inputs to this process are a struct baseSubmeshFrame, a 3D array gFrame, if the profile or geo codec ID indicates the usage of a 2D video codec, or a 2D array dFrame, if the profile or geo codec indicates an arithmetic codec, and atlas patch index p.

An output of this process is a struct recSubmeshFrame.

The submesh reconstruction process proceeds by invoking the various processes described below.

* First, If the variable AtlasMeshpatchTexcoordProjectionFlag[ p ] is equal to 1, then the texture coordinate generation process described in subclause 11.2.2 is invoked to generate the texture coordinates with the struct baseSubmeshFrame and index p as inputs, and the struct baseSubmeshFrame with updated texture coordinates as an output.
* Then, the subdivision process described in subclause 11.2.3 is invoked with the struct baseSubmeshFrame, AtlasMeshpatchSubdivMethod[ p ], and AtlasMeshpatchSubdivCount[ p ] as inputs, and a struct subdivSubmeshFrame, a 2D array verCoordEdges and a 1D array levelOfDetailCounts as outputs.
* If the syntax element vps\_geometry\_video\_present\_flag[ RecAtlasID ] is equal to 1, then the following steps are executed:
  + if the profile or geo codec ID indicates the usage of a 2D video codec, then the inverse coefficient packing process described in subclause 11.2.4 is invoked with the array gFrame, AtlasMeshpatch2dSizeX[ p ], AtlasMeshpatch2dSizeY[ p ], AtlasMeshpatch2dPosX[ p ], AtlasMeshpatch2dPosY[ p ], gi\_geometry\_2d\_bit\_depth\_minus1[ RecAtlasID ] + 1, AtlasMeshpatchSubdivCount[ p ], AtlasPatchTotalVertexCount[ atlasPatchIdx ],  and the 1D array AtlasPatchVertexCount[ atlasPatchIdx ] as inputs, and a 2D array dispQuantCoeffArray as an output. Otherwise, if the profile or geo codec indicates an arithmetic codec, then the output dispQuantCoeffArray assumes the value of the disparity frame dFrame.
  + the inverse quantization process described in subclause 11.2.5 is invoked with the variables AtlasPatchTotalVertexCount[ atlasPatchIdx ], AtlasMeshpatchSubdivCount[ p ], the 2D array dispQuantCoeffArray, and the 1D array AtlasPatchVertexCount[ atlasPatchIdx ] as inputs and a 2D array dispCoeffArray as an output.
  + if the profile or geo codec ID indicates the usage of a 2D video codec, then the inverse transform process described in subclause 11.2.6 is invoked with the variables AtlasPatchTotalVertexCount[ atlasPatchIdx ], AtlasMeshpatchSubdivCount[ p ], AtlasMeshpatchTransformMethod[ p ], the 1D array AtlasPatchVertexCount[ atlasPatchIdx ] and the 2D arrays dispCoeffArray, and verCoordEdges as inputs and a 2D array dispArray as output. Otherwise, if the profile or geo codec indicates an arithmetic codec, then the output dispCoeffArray assumes dispQuantCoeffArray.
  + the normal, tangent and bitangent vectors generation process described in subclause 11.2.7 is invoked with the struct subdivSubmeshFrame and AtlasMeshpatchDispCoordSys[ p ] as inputs, and parameters normals, tangents, and bitangents as outputs.
  + The vertex coordinates reconstruction process described in subclause 11.2.8 is invoked with the struct subdivSubmeshFrame, the arrays dispArray, normals, tangents, and bitangents as inputs and a struct vertexRefinedSubmeshFrame as an output.
* Otherwise, if vps\_geometry\_video\_present\_flag[ RecAtlasID ] is equal to 0, then the vertexRefinedSubmeshFrame struct is set equal to subdivSubmeshFrame.
* If the syntax element asve\_attribute\_subtexture\_enabled\_flag[ i ], for the Attribute Video data unit with index i, is equal to 1, then the texture coordinates adaptation process described in subclause 11.2.9 is invoked with the struct vertexRefinedSubmeshFrame, AtlasPatchAttribute2dSizeX[ p ][ i ], AtlasMeshpatchAttributes2dSizeY[ p ][ i ], AtlasPatchAttribute2dPosX[ p ][ i ], AtlasPatchAttribute2dPosY[ p ][ i ], asve\_attribute\_frame\_width[ i ] and asve\_attribute\_frame\_height[ i ] as inputs, and a struct recSubmeshFrames[ p ] as an output. Otherwise, recSubmeshFrames[ p ] is set as vertexRefinedSubmeshFrame.

### Texture coordinate derivation

#### General

Inputs to this process a struct baseSubmeshFrame and a variable p, indicating the index of the atlas meshpatch.

Outputs to this process is a struct genBasemeshFrame.

submeshVerCoordCount = baseSubmeshFrame.verCoordCount submeshFaceCount = baseSubmeshFrame.faceCount  
 submeshVerCoords = baseSubmeshFrame.verCoordsArray submeshVerCoordFaces = baseSubmeshFrame.verCoordFacesArray

First, the face to sub-patch mapping process described in subclause 11.2.2.2, is invoked with the parameters baseSubmeshFrame and meshpatch index p as inputs, and the parameter faceToSubPatch as output.

Then, for each sub-patch, the homography transform process described in subclause 11.2.2.3, is invoked with the parameters submeshVerCoordCount, submeshFaceCount, submeshVerCoordFaces, submeshVerCoords, meshindex p, and faceToSubPatch as inputs, and the parameters faceHomography as output.

Next, the vertex projection process described in subclause 11.2.2.4 is invoked with parameters submeshVerCoordCount, submeshFaceCount, submeshVerCoords, submeshVerCoordFaces, meshindex p, and faceHomography as inputs, and the parameters genSubmeshTexCoordCount, genSubmeshTexCoords and genSubmeshTexCoordFaces as outputs.

The output struct genBasemeshFrame is derived as follows:

genBasemeshFrame.verCoordCount = submeshVerCoordCount  
 genBasemeshFrame.faceCount = submeshFaceCount  
 genBasemeshFrame.verCoordsArray = submeshVerCoords  
 genBasemeshFrame.verCoordFacesArray = submeshVerCoordFaces  
 for( i =0; i < vps\_ext\_bmesh\_data\_attribute\_count[ RecAtlasID ]; i++ ){  
 genBasemeshFrame.attrValueCountArray[ i ] =  
 baseSubmeshFrame.attrValueCount[ i ]  
 genBasemeshFrame.attrValueDimensionArray[ i ] =  
 baseSubmeshFrame.attrValueDimensionArray[ i ]  
 genBasemeshFrame.attrValuesArray[ i ] =   
 baseSubmeshFrame.attrValues[ i ]  
 genBasemeshFrame.attrFacesArray[ i ] =  
 baseSubmeshFrame.attrFacesArray[ i ]  
 }  
 texIdx = asve\_projection\_texcoord\_output\_attribute\_index  
 genBasemeshFrame.attrIdxTexCoords = texIdx  
 genBasemeshFrame.attrValueCountArray[ texIdx ] = submeshTexCoordCount  
 genBasemeshFrame.attrValuesArray[ texIdx ] = submeshTexCoords genBasemeshFrame.attrFacesArray[ texIdx ] = submeshTexCoordFaces

The derived texture coordinates are normalized, that is, the values are between 0 and 1. To generate quantized texture coordinates using the bit depth indicated in ASPS, the following operation is done:

scale = ( 1 << ( asve\_projection\_texcoord\_output\_bit\_depth\_minus1 + 1 ) ) – 1  
 for( i = 0; i < submeshVerCoordCount; i++ ) {  
 uv = genBasemeshFrame.attrValuesArray[ texIdx ][ i ]  
 genBasemeshFrame.attrValuesArray[ texIdx ][ i ][ 0 ] =  
 Floor( uv[ 0 ] \* scale )  
 genBasemeshFrame.attrValuesArray[ texIdx ][ i ][ 1 ] =  
 Floor( uv[ 1 ] \* scale )  
 }

#### Face to sub-patch mapping

##### General

Inputs to this process are:

* a struct baseSubmeshFrame, indicating the submesh from decoded basemesh sub-bitstream,
* a variable p, indicating the index of the atlas meshpatch.

The outputs of this process are:

* faceToSubPatch, which is a 1D array of size submeshFaceCount indicating the sub-patch index associated with each face of the submesh

The 1D array faceIdxToFaceId is determined in the following way:

If asve\_projection\_texcoord\_mapping\_attribute\_index\_present\_flag is equal to 0, then the face ID is derived implicitly using connected components, and the array submeshFaceIds is obtained by invoking the process described in subclause 11.2.2.2.2 with parameters baseSubmeshFrame.verCoordCount, baseSubmeshFrame.faceCount, baseSubmeshFrame.verCoordFacesArray, and baseSubmeshFrame.VerCoords as inputs, and the parameter faceIdxToFaceId as output.

Otherwise, consider attrIdx is equal to asve\_projection\_texcoord\_mapping\_attribute\_index. If vps\_ext\_bmesh\_attrbute\_type[ RecAtlasID ][ attrIdx ] is equal to ATTR\_TEXCOORD, then the face ID is transmitted using the first coordinate of the base submesh texture coordinates, and the faceIdxToFaceId array is determined as follows:

for( i =0; i < baseSubmeshFrame.faceCount; i++ ){  
 attrValueIdx = baseSubmeshFrame.attrFacesArray[ attrIdx ][ i ][ 0 ]  
 faceId = baseSubmeshFrame.attrValuesArray[ attrValueIdx ][ 0 ]  
 faceIdxToFaceId[ i ] = faceId  
 }

else, if vps\_ext\_bmesh\_attrbute\_type[ RecAtlasID ][ attrIdx ] is equal to ATTR\_FACEGROUP\_ID, then face ID is transmitted using the face group ID attribute, and the faceIdxToFaceId array is determined as follows:

for( i =0; i < baseSubmeshFrame.faceCount; i++ ){  
 attrValueIdx = baseSubmeshFrame.attrFacesArray[ attrIdx ][ i ][ 0 ]  
 faceId = baseSubmeshFrae.attrValuesArray[ attrValueIdx ]  
 faceIdxToFaceId[ i ] = faceId  
 }

The mapping between the elements in the array faceToSubPatch and the sub-patch indices is obtained in the following way:

for( i =0; i < submeshFaceCount; i++ ){  
 subPatchIdx = -1  
 for( idx = 0; idx < AtlasMeshpatchNumSubpatches[ p ]; idx++ ) {  
 if( AtlasMeshpatchSubpatchFaceId[ p ][ idx ] == faceIdxToFaceId[ i ] ) {  
 subPatchIdx = idx  
 }  
 }  
 faceToSubPatch[ i ] = subPatchIdx  
 }

##### Face ID creation using connected components

Inputs to this process are:

* a variable submeshVerCoordCount, indicating the number of vertex coordinates in the submesh,
* a variable submeshFaceCount, indicating the number of faces in the submesh,
* a 2D array submeshVerCoords, of size submeshVerCoordCount × 3, indicating the vertex coordinates of the submesh,
* a 2D array submeshVerCoordFaces, of size submeshFaceCount × 3, indicating the connectivity indices associated with the vertex coordinates of the submesh.

The output of this process is:

* a 1D array faceIdxToFaceId, of size submeshFaceCount, indicating the face IDs of the submesh.

First, a 2D array of vertex to faces for each vertex is created as follows:

for( a = 0; a < submeshVerCoordCount; a++ ) {  
 vertex2faceCounts[ a ] = 0  
 }  
 for( t = 0; t < submeshFaceCount; t++ ) {  
 for( i = 0; i < 3; i++ ) {  
 a = submeshVerCoordFaces[ t ][ i ]  
 vertex2face[ a ][ vertex2faceCounts[ a ]++ ] = t   
 }  
 }

Now lists of connected components are created, if the faces share a vertex, as follows:

for( t = 0; t < submeshFaceCount; t++ )  
 faceIdxToFaceId[ t ] = ‑1  
 faceIdCount = 0  
 for( t = 0; t < submeshFaceCount; t++ ){  
 if( faceIdxToFaceId[ t ] == ‑1 ){  
 faceId = faceIdCount  
 faceIdCount = faceIdCount + 1  
 submeshFaceIds[ t ] = faceId  
 lifoFaces[ 0 ] = t  
 numFaces = 1  
 while(numFaces > 0){  
 numFaces = numFaces – 1  
 tIndex = lifoFaces[ numFaces ]  
 for( vIdx = 0;vIdx < 3; vIdx++){  
 v = submeshVerCoordFace[ tIndex ][ vIdx ]  
 for( n = 0;n < vertex2faceCounts[ v ]; n++){  
 neighbourFaceIdx = vertex2face[ v ][ n ]  
 if( faceIdxToFaceId[ neighbourFaceIdx ] == ‑1 ){  
 faceIdxToFaceId[ neighbourFaceIdx ] = faceId  
 lifoFaces[ numFaces ] = neighbourFaceIdx  
 numFaces = numFaces + 1  
 }  
 }  
 }  
 }  
 }  
 }

Next the connected components are ordered according to the sum of the areas of the faces that compose each component.

for( s = 0; s < faceIdCount; s++ )  
 ccArea[ s ] = 0  
 for( t = 0; t < submeshFaceCount; t++ ){  
 sIdx = submeshFaceIds[ t ]  
 face = submeshVerCoordFaces[ t ]  
 v0 = submeshVerCoords[ face[ 0 ] ]  
 v1 = submeshVerCoords[ face[ 1 ] ]  
 v2 = submeshVerCoords[ face[ 2 ] ]  
 normal = Cross( (v1 – v0 ), ( v2 – v0 ) )  
 faceArea = 0.5 \* sqrt(normal \* normal)  
 ccArea[ sIdx ] = ccArea[ sIdx ] + faceArea  
 }  
 for( s = 0; s < faceIdCount; s++ )  
 mapping[ s ] = -1  
 for( s = 0; s < faceIdCount; s++ )  
 maxArea = -1  
 for( searchIdx = 0; searchIdx < faceIdCount; searchIdx++ ){  
 if( maxArea < ccArea[ searchIdx ] ){  
 maxArea = ccArea[ searchIdx ]  
 mapping[ s ] = searchIndex  
 }  
 }  
 ccArea[ mapping[ s ] ] = -1  
 }  
 for( t = 0; t < faceCount0; t++ ){  
 faceIdxToFaceId[ t ] = mapping[ faceIdxToFaceId[ t ] ]

#### Homography Transform

Inputs to this process are:

* a variable submeshVerCoordCount, indicating the number of vertex coordinates in the submesh,
* a variable submeshFaceCount, indicating the number of faces in the submesh,
* a 2D array submeshVerCoords, of size submeshVerCoordCount × 3, indicating the vertex coordinates of the submesh,
* a 2D array submeshVerCoordFaces, of size submeshFaceCount × 3, indicating the connectivity indices associated with the vertex coordinates of the submesh,
* a variable p, indicating the index of the atlas meshpatch,
* a 1D array faceToSubPatch, of size submeshFaceCount, indicating the sub-patch index of each face of the submesh.

The outputs of this process are:

* a 3D array faceHomography, of size submeshFaceCount × 3 × 4, indicating the homography transform of each face of the submesh.

Let the variables widthNormalization, heightNormalization, and gutter be set by the variables AtlasMeshpatchTexcoordProjectionWidthNormalization[ p ], AtlasMeshpatchTexcoordProjectionHeightNormalization[ p ], and AtlasMeshpatchTexcoordProjectionGutter[ p ], respectively.

The variables AtlasMeshpatchSubpatchAxisU[ p ][ i ], and AtlasMeshpatchSubpatchAxisV[ p ][ i ], AtlasMeshpatchSubpatchAxisD[ p ][ i ], for meshpatch with index p, and sub-patch with index i, are derived as follows:

projectionPlane = AtlasMeshpatchSubpatchProjectionID[ p ][ i ]   
 rotationAxisMode = Max( 0, ( projectionPlane - 2)/4 )

If rotationAxisMode is equal to 0, then the following applies:

if( rotationAxisMode == 0 )  
 AtlasMeshpatchSubpatchAxisD[ p ][ i ] = projectionPlane % 3 (43)  
 else  
 if( rotationAxisMode == 1 )  
 AtlasMeshpatchSubpatchAxisD[ p ][ i ] = 2 \* ( projectionPlane % 2 ) (44)  
 else  
 if( rotationAxisMode == 2 )  
 AtlasMeshpatchSubpatchAxisD[ p ][ i ] = 2 - ( projectionPlane % 2 ) (45)  
 else  
 AtlasMeshpatchSubpatchAxisD[ p ][ i ] =1 - ( projectionPlane % 2 ) (46)  
 AtlasMeshpatchSubpatchAxisU[ p ][ i ] =  
 ( AtlasMeshpatchSubpatchAxisD[ p ] == 2 ) ? 0 : 2 (47)  
 AtlasMeshpatchSubpatchAxisV[ p ][ i ] =  
 ( AtlasMeshpatchSubpatchAxisD[ p ] == 1 ) ? 0 : 1 (48)

The values of the variables AtlasMeshpatchSubpatchAxisU[ p ][ i ], AtlasMeshpatchSubpatchAxisV[ p ][ i ], and AtlasMeshpatchSubpatchAxisD[ p ][ i ] that are derived from Equations 42 to 47 are also presented in tabular form in Table 8.

Table 8 – Derivation of sub-patch projection variables based on AtlasMeshpatchSubpatchProjectionID[ p ][ i ] values

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | AtlasMeshpatchSubpatchProjectionID[ p ][ i ] | | | | | | | | | | | | | | | | | |
| **Projection Variables** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** |
| AtlasMeshpatchAxisU[ p ][ i ] | 2 | 2 | 0 | 2 | 2 | 0 | 2 | 0 | 2 | 0 | 0 | 2 | 0 | 2 | 2 | 2 | 2 | 2 |
| AtlasMeshpatchAxisV[ p ][ i ] | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| AtlasMeshpatchAxisD[ p ][ i ] | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 2 | 0 | 2 | 2 | 1 | 2 | 1 | 1 | 0 | 1 | 0 |

First, rotation3DHomography, a 2D array of size 4 × 4 indicating the rotation of position in 3D space is determined according to the axis by which the points are being rotate, is determined by the variable rotationAxisMode. The rotation3DHomography is provided by Table 9:

Table 9 – Derivation of rotation homography transform based on sub-patch projection variables

|  |  |  |
| --- | --- | --- |
| rotationAxisMode | rotation3DHomography | Description |
| 0 |  | No rotation |
| 1 |  | Rotation around the y-axis |
| 2 |  | Rotation around the x-axis |
| 3 |  | Rotation around the z-axis |

Next, orthographic3DHomography, a 2D array of size 3 × 4 indicating the orthographic projection of a rotated 3D point into a 2D surface, is initialized with zeros, then some positions are altered according to the variables AtlasMeshpatchAxisU[ p ][ i ] and AtlasMeshpatchAxisV[ p ][ i ]:

orthographic3DHomography[ 0 ][ AtlasMeshpatchAxisU[ p ][ i ] ] = 1

orthographic3DHomography[ 0 ][ AtlasMeshpatchAxisV[ p ][ i ] ] = 1

orthographic3DHomography[ 2 ][ 3 ] = 1

For all the vertices that belong to the sub-patch with index i, we first determine the projected bounding box as follows:

subPatchProjBoundingBoxMin[ 0 ] = subPatchProjBoundingBoxMin[ 1 ] = MAX\_VAL  
 subPatchProjBoundingBoxMax[ 0 ] = subPatchProjBoundingBoxMax[ 1 ] = MIN\_VAL  
 for( trIdx =0; trIdx <= submeshFaceCount; trIdx++ )  
 if( faceToSubpatch[ trIdx ] == i ){  
 for( j = 0; j < 3; j++ ){  
 vIdx = submeshVerCoordFaces[ trIdx ][ j ]   
 verCoord = submeshVerCoords[ vIdx ]  
 homographyCoord = [ verCoord[ 0 ] ][ verCoord[ 1 ] ][ verCoord[ 2 ] ][ 1 ]  
 projectedVertex =  
 orthographic3DHomography \* rotation3DHomography \* homographyCoord  
 if( projectedVertex[ 0 ] > subPatchProjBoundingBoxMax[ 0 ] )  
 subPatchProjBoundingBoxMax[ 0 ] = projectedVertex[ 0 ]  
 if( projectedVertex[ 1 ] > subPatchProjBoundingBoxMax[ 1 ] )  
 subPatchProjBoundingBoxMax[ 1 ] = projectedVertex[ 1 ]  
 if( projectedVertex[ 0 ] < subPatchProjBoundingBoxMin[ 0 ] )  
 subPatchProjBoundingBoxMin[ 0 ] = projectedVertex[ 0 ]  
 if( projectedVertex[ 1 ] < subPatchProjBoundingBoxMin[ 1 ] )  
 subPatchProjBoundingBoxMin[ 1 ] = projectedVertex[ 1 ]  
 }  
 }  
 }

Then the biasRemovalHomography, a 2D array of size 3 × 3 indicating the bounding box limits for the sub-patch with index i, is used to remove the offset from the projected points, is initialized to all 0s and then obtained as follows:

biasRemovalHomography[ 0 ][ 0 ] = 1  
 biasRemovalHomography[ 0 ][ 2 ] = −1 \* subPatchProjBoundingBoxMin[ 0 ]  
 biasRemovalHomography[ 1 ][ 1 ] = 1  
 biasRemovalHomography[ 1 ][ 2 ] = −1 \* subPatchProjBoundingBoxMin[ 1 ]  
 biasRemovalHomography[ 2 ][ 2 ] = 1

Considering that the mesh was defined in 3D space using a right-hand coordinate system, when projected into the surface defined by the axis from Table 8, it might lead to the inversion of the winding of the face, which could result in a culling by the face in some rendering systems. In case one of the projection directions is equal to 0, 4, 5, 6, 9, 11, 12, 14, or 15, the winding of the projected face must be inverted to avoid the unwanted culling, that is, the projected x-coordinate must be inverted, which can be accomplished by applying the windingReversalHomography, a 2D array of size 3 × 3, which is initialized to all 0s and modified then as follows:

windingReversalHomography[ 0 ][ 0 ] = −1  
 windingReversalHomography[ 0 ][ 2 ] =  
 subPatchProjBoundingBoxMax[ 0 ] - subPatchProjBoundingBoxMin[ 0 ]  
 windingReversalHomography[ 1 ][ 1 ] = 1  
 windingReversalHomography[ 2 ][ 2 ] = 1

Otherwise, windingReversalHomography is just the identity matrix.

Next, scaleHomography, a 2D array of size 3 × 3 indicating the scaling of the projected vertices, is initialized to all 0s and modified as follows:

scaleHomography[ 0 ][ 0 ] = AtlasMeshpatchSubpatchScale[ p ][ i ]  
 scaleHomography[ 1 ][ 1 ] = AtlasMeshpatchSubpatchScale[ p ][ i ]  
 scaleHomography[ 2 ][ 2 ] = 1

Then the gutterHomography, a 2D array of size 3 × 3 indicating the offset defined by the gutter, is initialized to all 0s and modified as follows:

gutterHomography[ 0 ][ 0 ] = 1  
 gutterHomography[ 0 ][ 2 ] = gutter  
 gutterHomography[ 1 ][ 1 ] = 1   
 gutterHomography[ 1 ][ 2 ] = gutter  
 gutterHomography[ 2 ][ 2 ] = 1

When placing the sub-patch in the texture map image, it may be rotated, which is indicated by the variable AtlasMeshpatchSubpatchOrientationID[ p ][ i ]. Let the variables oIdx, sizeU, and sizeV be defined as AtlasMeshpatchSubpatchOrientationID[ p ][ i ], AtlasMeshpatchSubpatch2dSizeX[ p ][ i ], and AtlasMeshpatchSubpatch2dSizeY[ p ][ i ] . The subpatchRotationHomography, a 2D array of size 3 × 3, is obtained as indicated in Table 10.

Table 10 – Patch rotation homography according to an indicated sub-patch orientation index, oIdx

|  |  |  |
| --- | --- | --- |
| **oIdx** | subpatchRotationHomography | **Description** |
| 0 |  | No transformation |
| 1 |  | The y axis is inverted first and then x and y axes are swapped |
| 2 |  | The x and y axes are inverted |
| 3 |  | The x axis is inverted first and then x and y axes are swapped |

Next, the subpatchTranslationHomography, a 2D array of size 3 × 3 indicating the position of the patch in the texture map, is initialized to all 0s and modified as follows:

subpatchTranslationHomography[ 0 ][ 0 ] = 1  
 subpatchTranslationHomography[ 0 ][ 2 ] = AtlasMeshpatchSubpatch2dPosX[ p ][ i ]  
 subpatchTranslationHomography[ 1 ][ 1 ] = 1   
 subpatchTranslationHomography[ 1 ][ 2 ] = AtlasMeshpatchSubpatch2dPosY[ p ][ i ]  
 subpatchTranslationHomography[ 2 ][ 2 ] = 1

And the normalizationHomography, a 2D array of size 3 × 3 indicating the normalization of the texture coordinates, is initialized to all 0s and modified as follows:

normalizationHomography[ 0 ][ 0 ] = 1/widthNormalization  
 normalizationHomography[ 1 ][ 1 ] = 1/heightNormalization   
 normalizationHomography[ 2 ][ 2 ] = 1

The subPatchHomography, a 3D array of size AtlasMeshpatchNumSubpatches[ p ] × 3 × 3 indicating the homography transform per sub-patch, is obtained by multiplying the homography transforms defined previously for each sub-patch with index i, that is:

subPatchHomography[ i ] = rotation3DHomography  
 subPatchHomography[ i ] = orthographic3DHomography \* subPatchHomography  
 subPatchHomography[ i ] = biasRemovalHomography \* subPatchHomography  
 subPatchHomography[ i ] = windingReversalHomography \* subPatchHomography  
 subPatchHomography[ i ] = scaleHomography \* subPatchHomography  
 subPatchHomography[ i ] = gutterHomography \* subPatchHomography  
 subPatchHomography[ i ] = subpatchRotationHomography \* subPatchHomography  
 subPatchHomography[ i ] = subpatchTranslationHomography \* subPatchHomography  
 subPatchHomography[ i ] = normalizationHomography \* subPatchHomography

The output is then obtained by copying the corresponding sub-patch homography, as follows:

for( trIdx =0; trIdx <= faceCount0; trIdx++ )  
 if( faceToSubpatch[ trIdx ] != -1 )  
 faceHomography[ trIdx ] = subPatchHomography[ faceToSubpatch[ trIdx ] ]

#### Vertex projection

Inputs to this process are:

* a variable submeshVerCoordCount, indicating the number of vertex coordinates in the submesh,
* a variable submeshFaceCount, indicating the number of faces in the submesh,
* a 2D array submeshVerCoords, of size submeshVerCoordCount × 3, indicating the vertex coordinates of the submesh,
* a 2D array submeshVerCoordFaces, of size submeshFaceCount × 3, indicating the connectivity indices associated with the vertex coordinates of the submesh,
* a 3D array faceHomography, of size submeshFaceCount × 4 × 3, indicating the homography transform of each face of the submesh.

The outputs of this process are:

* a variable genSubmeshTexCoordCount, indicating the number of texture coordinates generated for the submesh,
* a 2D array genSubmeshTexCoords, of size genSubmeshTexCoordCount × 2, indicating the texture coordinates generated for the submesh,
* a 2D array genSubmeshTexCoordFaces, of size submeshFaceCount × 3, indicating the connectivity indices associated with the texture coordinates generated for the submesh.

The outputs are obtained in the following manner:

genSubmeshTexCoordCount= 0  
 for( trIdx =0; trIdx <= submeshFaceCount; trIdx++ ){  
 if( faceToSubPatch[ trIdx ] != -1 ) {  
 for( j = 0; j < 3; j++ ){  
 vIdx = submehVerCoordFaces[ trIdx ][ j ]  
 verCoord = submeshVerCoords[ vIdx ]  
 homographyCoord = [ verCoord[ 0 ] ][ verCoord[ 1 ] ][ verCoord[ 2 ] ][ 1 ]  
 uvCoord = faceHomography[ trIdx ] \* homographyCoord  
 found = false  
 for( n = 0; n < genSubmeshTexCoordCount && !found; n++ ){  
 if( ( genSubmeshTexCoords[ n ][ 0 ] == uvCoord[ 0 ] ) &&  
 ( genSubmeshTexCoords[ n ][ 1 ] == uvCoord[ 1 ] ) ) {  
 uvFace[ j ] = n  
 found = true  
 }  
 }  
 if( !found){  
 uvFace[ j ] = genSubmeshTexCoordCount  
 genSubmeshTexCoords[ genSubmeshTexCoordCount ][ 0 ] = uvCoord[ 0 ]  
 genSubmeshTexCoords[ genSubmeshTexCoordCount ][ 1 ] = uvCoord[ 1 ]  
 genSubmeshTexCoordCount = genSubmeshTexCoordCount + 1  
 }  
 }  
 genSubmeshTexCoordFaces[ trIdx ][ 0 ]= uvFace[ 0 ]  
 genSubmeshTexCoordFaces[ trIdx ][ 1 ]= uvFace[ 1 ]  
 genSubmeshTexCoordFaces[ trIdx ][ 2 ]= uvFace[ 2 ]  
 }  
 }

### Subdivision process

#### General

Inputs to this process are a struct baseSubmeshFrame, and array subdivisionMethod and variable subdivisionIterationCount.

An output of this process is a struct subdivMeshFrame, a 2D array verCoordEdges of size basemeshFrame.verCoordCount × 2 and an array levelOfDetailVertexCounts of size subdivisionIterationCount + 1.

A struct subdivMeshFrame is initialized as follows:

First, subdivMeshFrame is set equal to baseSubmeshFrame.

Then, verCoordEdges,texCoordEdges, and levelOfDetailVertexCounts are initialized as follows:

for( i = 0; i < baseSubmeshFrame.verCoordCount; i++ ) {  
 verCoordEdges[ i ][ 0 ] = verCoordEdges[ i ][ 1 ] = 0  
 }  
 for( i = 0; i < baseSubmeshFrame.texCoorCount; i++ ) {  
 texCoordEdges[ i ][ 0 ] = texCoordEdges[ i ][ 1 ] = 0  
 }  
 for( i = 0; i < subdivisionIterationCount + 1; i++ ) {  
 levelOfDetailVertexCounts[ i ] = 0  
 }  
 levelOfDetailVertexCounts[ 0 ] = baseSubmeshFrame.verCoordCount

If subdivisionIterationCount is greater than 0, for each subdivisionIterationIndex = 0..(subdivisionIterationCount – 1) following applies:

* First inputSubmeshMeshFrame is equal to subdivMeshFrame
* Then subdivision process for the vertex coordinates is invoke. The inputs to the process are variables positionDimension set as 3, inputSubmeshMeshFrame.verCoordCount, inputSubmeshMeshFrame.faceCount, and the array inputSubmeshMeshFrame.verCoordsArray, inputSubmeshMeshFrame.verCoordFacesArray. The outputs of the process are variables subdivMeshFrame.verCoordCount, subdivMeshFrame.faceCount., and arrays subdivMeshFrame.verCoordsArray, subdivMeshFrame.verCoordFacesArray, verCoordEdges, and levelOfDetailVertexCounts[subdivisionIterationIndex + 1].
  + If subdivisionMethod[ subdivisionIterationIndex ] is equal to 1, then the midpoint subdivision process described in subclause 11.2.3.2 is invoked
  + If subdivisionMethod[ subdivisionIterationIndex ] is equal to 2, then the loop subdivision process described in subclause 11.2.3.3 is invoked
  + If subdivisionMethod[ subdivisionIterationIndex ] is equal to 3, then the LS3 subdivision process described in subclause 11.2.3.4 is invoked
* Then subdivision process for the texture coordinates is invoke. Consider the texture coordinate index texIdx equal to inputSubmeshMeshFrame.attIdxTexCoords. The inputs to the process are variables positionDimension set as 2, inputSubmeshMeshFrame.attrValueCountArray[ texIdx ], inputSubmeshMeshFrame.attrValueArray[ texIdx ], inputSubmeshMeshFrame.faceCount and inputSubmeshMeshFrame.attrFaceArray[ texIdx ]. The outputs of the process are subdivMeshFrame.attrValueCountArray[ texIdx ], subdivMeshFrame.attrValueArray[ texIdx ], subdivMeshFrame.faceCount, subdivMeshFrame.attrFaceArray[ texIdx ], texCoordEdges and texLevelOfDetailCount[ subdivisionIterationIndex + 1 ].
  + If subdivisionMethod[ subdivisionIterationIndex ] is equal to 1, then the midpoint subdivision process described in subclause 11.2.3.2 is invoked
  + If subdivisionMethod[ subdivisionIterationIndex ] is equal to 2, then the loop subdivision process described in subclause 11.2.3.3 is invoked
  + If subdivisionMethod[ subdivisionIterationIndex ] is equal to 3, then the LS3 subdivision process described in subclause 11.2.3.4 is invoked

#### Midpoint subdivision process

Inputs to this process are:

* positionDimension, indicating the dimension of the positions to be subdivided,
* subdivisionIterationIndex, indicating the index of the current subdivision iteration ,
* positionCountIn, indicating the number of positions to be subdivided,
* faceCountIn, indicating the number of faces created with the positions,
* positionArrayIn, of size positionCountIn × positionDimension indicating the positions to be subdivided,
* facesArrayIn, of size faceCountIn × 3 indicating the connectivity indices associated with the positions to be subdivided.

The outputs to the process are:

* positionCountOut, indicating the number of positions,
* faceCountOut, indicating the number of faces,
* positionArrayOut, of size positionCountOut × positionDimension, indicating the positions,
* facesArrayOut, of size faceCountOut × 3, indicating the connectivity indices associated with the positions
* edges, of size positionCountOut × 2, indicating the indices of two positions used to generate the position of the current index,
* levelOfDetailCounts[ subdivisionIterationIndex ], indicating the number of positions associated with the current subdivision iteration, and
* valences, of size positionCountOut, indicating the number of connected edges as each vertex.

First, 2D array edges is calculated as follows:

for( a = 0; a < positionCountIn; a++ ) {  
 neighbourCounts[ a ] = 0  
 }  
 positionCounter = positionCountIn  
 faceCounter = faceCountIn  
 ComputeNeighbours( faceCount, faces, neighbours, neighbourCounts)  
 for( a = 0; a < positionCount; a++ ) {  
 valences[ a ] = neighbourCounts[ a ]  
 for( n = 0; n < neighbourCounts[ a ]; n++ ) {  
 b = neighbours[ a ][ n ]  
 if ( a > b ) {  
 edges[ positionCounter ][ 0 ] = a  
 edges[ positionCounter ][ 1 ] = b  
 edgeToPosition[ a ][ b ] = positionCounter  
 positionCounter += 1  
 }  
 }  
 }

Then subdivision process for the current subdivision iteration is applied as follows:

* First, output arrays are initialized.

positionArrayOut = positionArrayIn  
 faceArrayOut = faceArrayIn

* Then, output arrays are calculated as follows.

for( e = positionCount; e < positionCounter; e++) {  
 for( d = 0; d < positionDimension; d++ ) {  
 positionArrayOut[ e ][ d ] = ( positionArrayOut[ edges[ e ][ 0 ] ][ d ] +   
 positionArrayOut[ edges[ e ][ 1 ] ][ d ] ) / 2  
 }  
 }  
 for( t = 0; t < faceCountIn; t++ ) {  
 for( i = 0; i < 3; i++ ) {  
 tri0[ i ] = facesOut[ t ][ i ]  
 vindex0 = facesOut[ t ][ i ]  
 vindex1 = facesOut[ t ][ ( i + 1 ) %3 ]  
 tri1[i] = edgeToPosition[ vindex0 ][ vindex1 ]  
 facesOut[ t ][ i ] = tri1[ i ]  
 }  
 facesOut[ faceCounter ][ 0 ] = tri0[ 0 ]  
 facesOut[ faceCounter ][ 1 ] = tri1[ 0 ]  
 facesOut[ faceCounter ][ 2 ] = tri1[ 2 ]  
 faceCounter += 1  
 facesOut[ faceCounter ][ 0 ] = tri1[ 0 ]  
 facesOut[ faceCounter ][ 1 ] = tri0[ 1 ]  
 facesOut[ faceCounter ][ 2 ] = tri1[ 1 ]  
 faceCounter += 1  
 facesOut[ faceCounter ][ 0 ] = tri1[ 2 ]  
 facesOut[ faceCounter ][ 1 ] = tri1[ 1 ]  
 facesOut[ faceCounter ][ 2 ] = tri0[ 2 ]  
 faceCounter += 1  
 }  
 if( it < subdivisionIterationCount – 1 ) {  
 for( e = positionCountIn; e < positionCounter; e++) {  
 id = 2 \* ( e – positionCountIn) + positionCounter  
 for( i = 0; i < 2; i++ ) {  
 edges[ id + i ][ 0 ] = edges[ e ][ i ]  
 edges[ id + i ][ 1 ] = e  
 edgeToPosition[ edges[ e ][ i ] ][ e ] = id + i  
 }  
 }  
 for( t = 0; t < faceCountIn; t++ ) {  
 id = 3 \* t + 3 \* positionCounter – 2 \* positionCountIn  
 for( i = 0; i < 3; i++ ) {  
 vindex0 = subdivSubmeshFaces[ t ][ i ]  
 vindex1 = subdivSubmeshFaces[ t ][ ( i + 1 ) % 3 ]  
 edges[ id + i ][ 0 ] = min( vindex0, vindex1 )  
 edges[ id + i ][ 1 ] = max( vindex0, vindex1 )  
 edgeToPosition[ vindex0 ][ vindex1 ] = id + i  
 }  
 }  
 }  
 positionOfPrevLoD = positionCounter - positionCountIn  
 positionCounter += 2 \* positionOfPrevLoD + 3 \* triangleCount  
 positionCount = positionCounter  
 faceCount = faceCounter  
 levelOfDetailCounts[ subdivisionIterationIndex ] = positionCounter positionCountOut = positionCountIn  
 faceCountOut = faceCountIn

#### Loop subdivision process

Inputs to this process are:

* positionDimension, indicating the dimension of the positions to be subdivided,
* subdivisionIterationIndex, indicating the index of the current subdivision iteration ,
* positionCountIn, indicating the number of positions to be subdivided,
* faceCountIn, indicating the number of faces created with the positions,
* positionArrayIn, of size positionCountIn × positionDimension indicating the positions to be subdivided,
* facesArrayIn, of size faceCountIn × 3 indicating the connectivity indices associated with the positions to be subdivided.

The outputs of this process are:

* positionCountOut, indicating the number of positions,
* faceCountOut, indicating the number of faces,
* positionArrayOut, of size positionCountOut × positionDimension, indicating the positions,
* facesArrayOut, of size faceCountOut × 3, indicating the connectivity indices associated with the positions,
* edges, of size positionCountOut × 2, indicating the indices of two positions used to generate the position of the current index,
* levelOfDetailCounts[ subdivisionIterationIndex ], indicating the number of positions associated with the current subdivision iteration.

First, 2D array edges is calculated as follows:

for( a = 0; a < positionCountIn; a++ ) {  
 neighbourCounts[ a ] = 0  
 }  
 positionCounter = positionCountIn  
 faceCounter = faceCountIn  
 ComputeNeighbours( faceCount, faces, neighbours, neighbourCounts)  
 for( a = 0; a < positionCount; a++ ) {  
 for( n = 0; n < neighbourCounts[ a ]; n++ ) {  
 b = neighbours[ a ][ n ]  
 if ( a > b ) {  
 edges[ positionCounter ][ 0 ] = a  
 edges[ positionCounter ][ 1 ] = b  
 edgeToPosition[ a ][ b ] = positionCounter  
 positionCounter += 1  
 }  
 }  
 }

Then subdivision process for the current subdivision iteration is applied as follows:

* First, output arrays are initialized.

positionArrayOut = positionArrayIn  
 faceArrayOut = faceArrayIn

* Then, output arrays are calculated as follows.

ComputeNeighbours( faceCount, recVerCoordFaces, verNeighbours, verNeighbourCounts)  
 ComputeTriangleNeighbours( faceCount, faceArrayOut, triNeighbours, triNeighbourCounts) for( e = positionCount; e < positionCounter; e++) {  
 a = edges[ e ][ 0 ]  
 b = edges[ e ][ 1 ]  
 adjVertexCount, adjVertices, adjWeights =   
 computeLoopOddMask(a, b, faceArrayOut, triNeighbours, triNeighbourCounts)   
 for(j = 0; j < adjVertexCount; j++){  
 vj = adjVertices[j]  
 wj = adjWeights[j]  
 for( d = 0; d < positionDimension; d++ ) {  
 positionArrayOut[ e ][ d ] += wj \* positionArrayOut [ vj ][ d ]  
 }  
 }  
 }  
 for( v = 0; v < positionCount; v ++) {  
 for( d = 0; d < positionDimension; d++ ) {  
 oldPositionArrayOut[v][d] = positionArrayOut  
 }  
 }  
 for( v = 0; v < positionCount; v ++){  
 adjVertexCount, adjVertices, adjWeights =   
 computeLoopEvenMask(v, faceArrayOut, verNeighbours, verNeighbourCounts,  
 triNeighbours, triNeighbourCounts)  
 for(j = 0; j < adjVertexCount; j++){  
 vj = adjVertices[j]  
 wj = adjWeights [j]  
 for( d = 0; d < positionDimension; d++ ) {  
 positionArrayOut[ v ][ d ] += wj \* oldPositionsArrayOut[vj][d]  
 }  
 }  
 }  
 for( t = 0; t < faceCountIn; t++ ) {  
 for( i = 0; i < 3; i++ ) {  
 tri0[ i ] = facesOut[ t ][ i ]  
 vindex0 = facesOut[ t ][ i ]  
 vindex1 = facesOut[ t ][ ( i + 1 ) %3 ]  
 tri1[i] = edgeToPosition[ vindex0 ][ vindex1 ]  
 facesOut[ t ][ i ] = tri1[ i ]  
 }  
 facesOut[ faceCounter ][ 0 ] = tri0[ 0 ]  
 facesOut[ faceCounter ][ 1 ] = tri1[ 0 ]  
 facesOut[ faceCounter ][ 2 ] = tri1[ 2 ]  
 faceCounter += 1  
 facesOut[ faceCounter ][ 0 ] = tri1[ 0 ]  
 facesOut[ faceCounter ][ 1 ] = tri0[ 1 ]  
 facesOut[ faceCounter ][ 2 ] = tri1[ 1 ]  
 faceCounter += 1  
 facesOut[ faceCounter ][ 0 ] = tri1[ 2 ]  
 facesOut[ faceCounter ][ 1 ] = tri1[ 1 ]  
 facesOut[ faceCounter ][ 2 ] = tri0[ 2 ]  
 faceCounter += 1  
 }  
 if( it < subdivisionIterationCount – 1 ) {  
 for( e = positionCountIn; e < positionCounter; e++) {  
 id = 2 \* ( e – positionCountIn) + positionCounter  
 for( i = 0; i < 2; i++ ) {  
 edges[ id + i ][ 0 ] = edges[ e ][ i ]  
 edges[ id + i ][ 1 ] = e  
 edgeToPosition[ edges[ e ][ i ] ][ e ] = id + i  
 }  
 }  
 for( t = 0; t < faceCountIn; t++ ) {  
 id = 3 \* t + 3 \* positionCounter – 2 \* positionCountIn  
 for( i = 0; i < 3; i++ ) {  
 vindex0 = subdivSubmeshFaces[ t ][ i ]  
 vindex1 = subdivSubmeshFaces[ t ][ ( i + 1 ) % 3 ]  
 edges[ id + i ][ 0 ] = min( vindex0, vindex1 )  
 edges[ id + i ][ 1 ] = max( vindex0, vindex1 )  
 edgeToPosition[ vindex0 ][ vindex1 ] = id + i  
 }  
 }  
 }  
 positionOfPrevLoD = positionCounter - positionCountIn  
 positionCounter += 2 \* positionOfPrevLoD + 3 \* triangleCount  
 positionCount = positionCounter  
 faceCount = faceCounter  
 levelOfDetailCounts[ subdivisionIterationIndex ] = positionCounter positionCountOut = positionCountIn  
 faceCountOut = faceCountIn

The function computeLoopOddMask(a, b, faces, triNeighbours, triNeighbourCounts) computes a vertex count, adjacent vertex indices, and adjacent vertex weights used to calculate new odd vertex v, see Figure 2, is as follows:

adjVertexCount, adjVertices, adjWeights =   
 computeLoopOddMask(a, b, faces, triNeighbours, triNeighbourCounts){  
 edgeTriNeighbours, edgeTriNeighbourCount =  
 ComputeTriangleNeighboursOfEdge( a, b, triNeighbours, triNeighbourCounts)  
 isBoundaryOrAbnormal = false  
 if ( edgeTriNeighbourCount == 2 ) {  
 c = findFaceThirdVertex( a, b, faces[ edgeTriNeighbours[0] ] )  
 d = findFaceThirdVertex( a, b, faces[ edgeTriNeighbours[1] ] )  
 if ((c >= 0 && d >= 0) && (c != d)){   
 adjVertexCount = 4  
 adjVertices[0] = a  
 adjVertices[1] = b  
 adjVertices[2] = c  
 adjVertices[3] = d  
 adjWeights[0] = 0.375  
 adjWeights[1] = 0.375  
 adjWeights[2] = 0.125  
 adjWeights[3] = 0.125  
 } else {  
 isBoundaryOrAbnormal = true  
 }  
 } else {  
 isBoundaryOrAbnormal = true  
 }  
 if (isBoundaryOrAbnormal){  
 adjVertexCount = 2  
 adjVertices[0] = a  
 adjVertices[1] = b  
 adjWeights[0] = 0.5  
 adjWeights[1] = 0.5  
 }  
 }

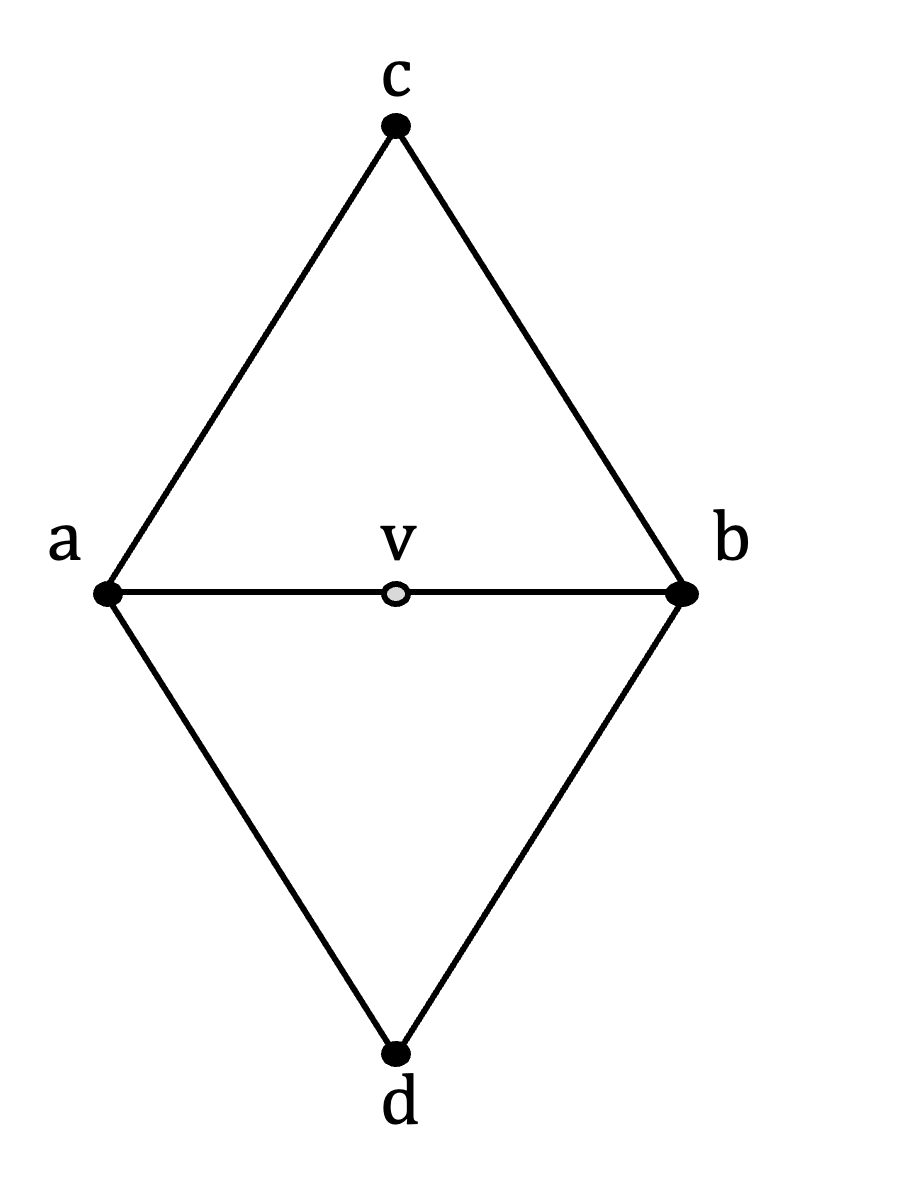


Figure 2 An example of new odd vertex v

The function computeLoopEvenMask(v, faces, verNeighbours, verNeighbourCounts, triNeighbours, triNeighbourCounts) computes a vertex count, adjacent vertex indices, and adjacent vertex weights to update existing even vertices v, see Figure 3, is as follows :

adjVertexCount, adjVertices, adjWeights =  
 computeLoopEvenMask(v, faces , verNeighbours, verNeighbourCounts,  
 triNeighbours, triNeighbourCounts){  
 isAbnormal = false  
 if(verNeighbourCounts[ v ] >= 2){  
 for (i = 0; i < verNeighbourCounts[ v ]; i++){  
 edgeTriNeighbours, edgeTriNeighbourCount =  
 ComputeTriangleNeighboursOfEdge( v, verNeighbours[ v ][ i ],  
 triNeighbours, triNeighbourCounts)  
 if (edgeTriNeighbourCount == 1) {  
 adjBoundaryVertices[ adjBoundaryVerticesCount++ ] =   
 verNeighbours[ v ][ i ]  
 }  
 }  
 if(adjBoundaryVerticesCount != 2){  
 adjVertexCount = verNeighbourCounts[ v ] + 1  
 beta = calculateBata(verNeighbourCounts[ v ])  
 adjVertices[ 0 ] = v  
 adjWeights [ 0 ] = 1 – verNeighbourCounts[ v ] \* beta  
 for (i = 0; i < verNeighbourCounts[ v ]; i++){  
 adjVertices[ i + 1 ] = verNeighbours[ v ][ i ]  
 adjWeights[ i + 1 ] = beta  
 }  
 } else if (abBoundaryVertices[0] != abBoundaryVertices[1]) {  
 adjVertexCount = 3  
 adjVertices[0] = v  
 adjVertices[1] = abBoundaryVertices[0]  
 adjVertices[2] = abBoundaryVertices[1]  
 adjWeights [0] = 0.75  
 adjWeights [1] = 0.125  
 adjWeights [2] = 0.125  
 } else {  
 isAbnormal = true  
 }  
 } else {  
 isAbnormal = true  
 }  
 if ( isAbnormal ) {   
 adjVertexCount = 0  
 }  
 }

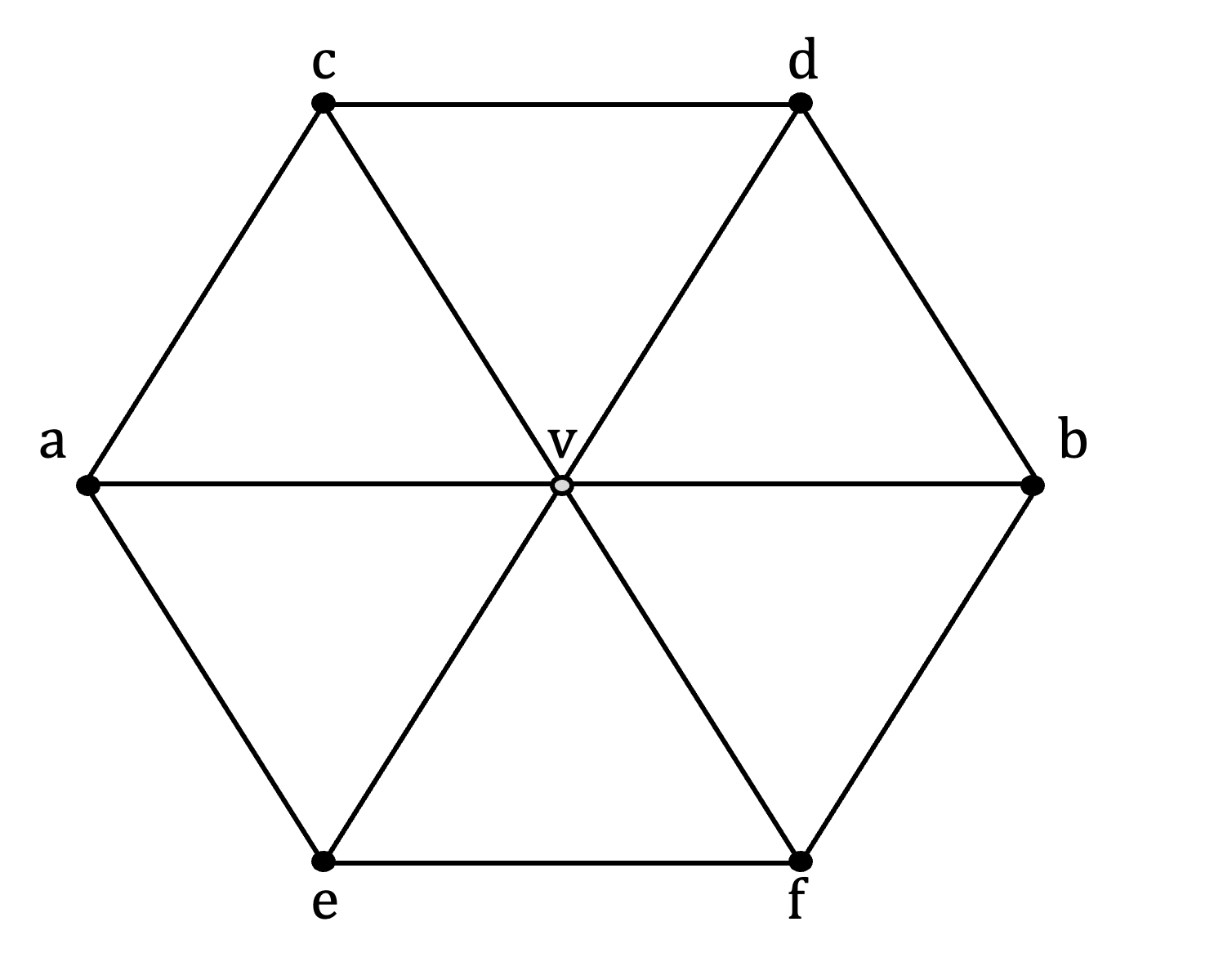


Figure 3 An example of a even vertex v

The function calculateBata is as follows:

beta = calculateBata (neighbourCount){  
 betaListSize = 20  
 betaList = {  
 0.0000000000000000000  
 0.1875000000000000000  
 0.1875000000000000000  
 0.1210937500000000000  
 0.0840932189257828921  
 0.0625000000000000000  
 0.0490249201910889013  
 0.0400678098159402973  
 0.0337850179292333436  
 0.0291777532480480203  
 0.0256734766761676954  
 0.0229266863992014756  
 0.0207192286042057368  
 0.0189078057828722147  
 0.0173949930162198702  
 0.0161125312963783796  
 0.0150113193131492777  
 0.0140552080419541169  
 0.0132170329611838808  
 0.0124760061060220778  
 }  
 if (betaListSize > neighbourCount){  
 beta = betaListSize[neighbourCount]  
 } else {  
 beta = -1  
 }  
 }

#### LS3 subdivision process

[Ed. Note (LK, ETRI) we cam add Note with non-normative reference to Reference paper: Boyé, Simon, Gael Guennebaud, and Christophe Schlick, “Least squares subdivision surfaces”, Computer Graphics Forum, vol. 29. no. 7. Oxford, UK: Blackwell Publishing Ltd, 2010]

Inputs to this process are:

* positionDimension, indicating the dimension of the positions to be subdivided,
* subdivisionIterationIndex, indicating the index of the current subdivision iteration ,
* positionCountIn, indicating the number of positions to be subdivided,
* faceCountIn, indicating the number of faces created with the positions,
* positionArrayIn, of size positionCountIn × positionDimension indicating the positions to be subdivided,
* facesArrayIn, of size faceCountIn × 3 indicating the connectivity indices associated with the positions to be subdivided,
* normalArrayIn, of size positionCountIn × positionDimension indicating the normals at the positions to be subdivided.

The outputs of this process are:

* positionCountOut, indicating the number of positions,
* faceCountOut, indicating the number of faces,
* positionArrayOut, of size positionCountOut × positionDimension, indicating the positions,
* facesArrayOut, of size faceCountOut × 3, indicating the connectivity indices associated with the positions
* normalArrayOut, of size positionCountOut × positionDimension, indicating the normals of the positions.
* edges, of size positionCountOut × 2, indicating the indices of two positions used to generate the position of the current index,
* levelOfDetailCounts[ subdivisionIterationIndex ], indicating the number of positions associated with the current subdivision iteration.

First, 2D array edges is calculated as follows:

for( a = 0; a < positionCountIn; a++ ) {  
 neighbourCounts[ a ] = 0  
 }  
 positionCounter = positionCountIn  
 faceCounter = faceCountIn  
 ComputeNeighbours( faceCount, faces, neighbours, neighbourCounts)  
 for( a = 0; a < positionCount; a++ ) {  
 for( n = 0; n < neighbourCounts[ a ]; n++ ) {  
 b = neighbours[ a ][ n ]  
 if ( a > b ) {  
 edges[ positionCounter ][ 0 ] = a  
 edges[ positionCounter ][ 1 ] = b  
 edgeToPosition[ a ][ b ] = positionCounter  
 positionCounter += 1  
 }  
 }  
 }

Then subdivision process for the current subdivision iteration is applied as follows:

* First, output arrays are initialized.

positionArrayOut = positionArrayIn  
 normalArrayOut = normalArrayIn  
 faceArrayOut = faceArrayIn

* Then, output arrays are calculated as follows.

ComputeNeighbours( faceCount, recVerCoordFaces, verNeighbours, verNeighbourCounts)  
 ComputeTriangleNeighbours( faceCount, faces, triNeighbours, triNeighbourCounts)  
 for( e = positionCount; e < positionCounter; e++) {  
 a = edges[ e ][ 0 ]  
 b = edges[ e ][ 1 ]  
 adjVertexCount, adjVertices, adjWeights =   
 computeLoopOddMask( a, b, faceArrayOut, triNeighbours, triNeighbourCounts )  
 if (positionDimension == 3){  
 validity, newPosition, newNormal = LS3Projection(e, positionArrayOut,  
 normalArrayOut, adjVertexCount, adjVertices, adjWeights)  
 if (validity){  
 for( d = 0; d < positionDimension; d++ ) {   
 positionArrayOut[ e ][ d ] = newPosition[ d ]  
 normalArrayOut[ e ][ d ] = newNormal[ d ]   
 }  
 }  
 }  
 if (positionDimension != 3 || !validity){  
 for(j = 0; j < adjVertexCount; j++){  
 vj = adjVertices[ j ]  
 wj = adjWeights[ j ]  
 for( d = 0; d < positionDimension; d++ ) {  
 positionArrayOut[ e ][ d ] += wj \* positionArrayOut[ vj ][ d ]  
 if (positionDimension == 3)  
 normalArrayOut [ e ][ d ] += wj \* normalArrayOut[ vj ][ d ]  
 }  
 }  
 }  
 }  
 for( v = 0; v < positionCount; v ++){  
 for( d = 0; d < positionDimension; d++ ){  
 oldPositionArrayOut[ v ][ d ] = positionArrayOut  
 if (positionDimension == 3)  
 oldNormalArrayOut[ v ][ d ] = normalArrayOut   
 }  
 }  
 for( v = 0; v < positionCount; v ++) {  
 adjVertexCount, adjVertices, adjWeights =   
 computeLoopEvenMask(v, faceArrayOut, verNeighbours, verNeighbourCounts,  
 triNeighbours, triNeighbourCounts )  
 if (positionDimension == 3){  
 validity, newPosition, newNormal = LS3Projection(v, oldPositionArrayOut,  
 oldNormalArrayOut, adjVertexCount, adjVertices, adjWeights)  
 if (validity){  
 for( d = 0; d < positionDimension; d++ ) {  
 positionArrayOut[ e ][ d ] = newPosition[ d ]  
 normalArrayOut[ e ][ d ] = newNormal[ d ]  
 }  
 }  
 }  
 if (positionDimension != 3 || !validity){  
 for(j = 0; j < adjVertexCount; j++){  
 vj = adjVertices[j]  
 wj = adjWeights[j]  
 for( d = 0; d < positionDimension; d++ ) {  
 positionArrayOut[ v ][ d ] += wj \* oldPositionArrayOut [ vj ][ d ]  
 if (positionDimension == 3) {  
 normalArrayOut[ v ][ d ] += wj \* oldnormalArrayOut [ vj ][ d ]  
 }  
 }  
 }  
 }  
 }  
 for( t = 0; t < faceCountIn; t++ ) {  
 for( i = 0; i < 3; i++ ) {  
 tri0[ i ] = facesOut[ t ][ i ]  
 vindex0 = facesOut[ t ][ i ]  
 vindex1 = facesOut[ t ][ ( i + 1 ) %3 ]  
 tri1[i] = edgeToPosition[ vindex0 ][ vindex1 ]  
 facesOut[ t ][ i ] = tri1[ i ]  
 }  
 facesOut[ faceCounter ][ 0 ] = tri0[ 0 ]  
 facesOut[ faceCounter ][ 1 ] = tri1[ 0 ]  
 facesOut[ faceCounter ][ 2 ] = tri1[ 2 ]  
 faceCounter += 1  
 facesOut[ faceCounter ][ 0 ] = tri1[ 0 ]  
 facesOut[ faceCounter ][ 1 ] = tri0[ 1 ]  
 facesOut[ faceCounter ][ 2 ] = tri1[ 1 ]  
 faceCounter += 1  
 facesOut[ faceCounter ][ 0 ] = tri1[ 2 ]  
 facesOut[ faceCounter ][ 1 ] = tri1[ 1 ]  
 facesOut[ faceCounter ][ 2 ] = tri0[ 2 ]  
 faceCounter += 1  
 }  
 if( it < subdivisionIterationCount – 1 ) {  
 for( e = positionCountIn; e < positionCounter; e++) {  
 id = 2 \* ( e – positionCountIn) + positionCounter  
 for( i = 0; i < 2; i++ ) {  
 edges[ id + i ][ 0 ] = edges[ e ][ i ]  
 edges[ id + i ][ 1 ] = e  
 edgeToPosition[ edges[ e ][ i ] ][ e ] = id + i  
 }  
 }  
 for( t = 0; t < faceCountIn; t++ ) {  
 id = 3 \* t + 3 \* positionCounter – 2 \* positionCountIn  
 for( i = 0; i < 3; i++ ) {  
 vindex0 = subdivSubmeshFaces[ t ][ i ]  
 vindex1 = subdivSubmeshFaces[ t ][ ( i + 1 ) % 3 ]  
 edges[ id + i ][ 0 ] = min( vindex0, vindex1 )  
 edges[ id + i ][ 1 ] = max( vindex0, vindex1 )  
 edgeToPosition[ vindex0 ][ vindex1 ] = id + i  
 }  
 }  
 }  
 positionOfPrevLoD = positionCounter – positionCountIn  
 positionCounter += 2 \* positionOfPrevLoD + 3 \* triangleCount  
 positionCount = positionCounter  
 faceCount = faceCounter  
 levelOfDetailCounts[ subdivisionIterationIndex ] = positionCounter  
 positionCountOut = positionCountIn  
 faceCountOut = faceCountIn

The function LS3Projection(vIndex, positions, normals, adjVertexCount, adjVertices, adjWeights) is defined to apply the LS3 projection algorithm using a smoothing mask (e.g. Loop Mask). This refinement method takes normals into account.

validity, newPosition, newNormal = LS3Projection( vIndex, positions, normals,  
 adjVertexCount, adjVertices, abjWeights ) {  
 validity = 1  
 p = positionArray[currentVertexIdx]  
 n = normalArray[currentVertexIdx]  
 beta = 1  
 sumP = 0  
 sumN = 0   
 sumDotPN = 0  
 sumDotPP = 0  
 sumW = 0  
 for( v = 0; v < adjVertexCount; v++){  
 sumP += positions[ adjVertices[ v ] ] \* adjWeights[ v ]  
 sumN += normals[ adjVertices[ v ] ] \* adjWeights[ v ]  
 sumDotPN +=   
 adjWeights[ v ] \*  
 Dot( positions[ adjVertices[ v ] ], normals[adjVertices[ v ] ] )  
 sumDotPP += adjWeights[ v ] \* SquaredNorm( positions[ adjVertices[ v ] ] )  
 sumW += adjWeights[ v ]  
 }  
 invSumW = 1 / sumW  
 aux4 = beta \* 0.5 \* (sumDotPN - invSumW \* Dot( sumP, sumN )) /  
 (sumDotPP - invSumW \* SquaredNorm( sump ))  
 uLinear = (sumN - sumP \* (2 \* aux4)) \* invSumW  
 uConstant = -invSumW \* (Dot(uLinear, sumP) + sumDotPP \* aux4)  
 uQuad = aux4  
 orig = sumP \* invSumW  
 if (abs(uQuad) > 0.0000001){  
 b = 1 / uQuad  
 center = uLinear \* (-0.5 \* b)  
 radius = sqrt(SquaredNorm(center) - b \* uConstant)  
 normal = orig - center  
 normalize(normal)  
 position = center + normal \* radius  
 normal = uLinear + position \* (2 \* uQuad)  
 normalize(normal)  
 } else if (uQuad == 0){  
 s = 1 / Norm(uLinear)  
 uLinear \*= s  
 uConstant \*= s  
 normal = uLinear  
 position = orig - uLinear \* (Dot(orig, uLinear) + uConstant)  
 } else {  
 f = 1 / sqrt(SquaredNorm(uLinear) - 4 \* uConstant \* uQuad)  
 uConstant \*= f  
 uQuad \*= f  
 dir = uLinear + orig \* (2 \* uQuad)  
 ilg = 1. / Norm(dir)  
 dir \*= ilg  
 ad = uConstant + Dot(uLinear, orig) + uQuad \* SquaredNorm(orig)  
 delta = -ad \* min(ilg, 1)  
 p = orig + dir \* delta  
 for (int i = 0; i < 2; ++i){  
 grad = uLinear + p \* (2 \* uQuad)  
 ilg = 1. / Norm(grad)  
 delta = -(uConstant + Dot(uLinear, p) + uQuad \* SquaredNorm(p)) \*  
 min(ilg, 1)  
 p += dir \* delta  
 }  
 position = p  
 normal = uLinear + position \* (2 \* uQuad)  
 normalize(normal)  
 }  
 newPosition = position  
 newNormal = normal  
 zeroNormal = 1  
 for( d = 0; d < positionDimension; d++){  
 if (isNaN(position[ d ]) || isInf(position[ d ]))  
 validity = 0  
 if (isNaN(normal[ d ]) || isInf(normal[ d ]))  
 validity = 0  
 if (normal[ d ] != 0)  
 zeroNormal = 0  
 }  
 if (zeroNormal){  
 validity = 0  
 }  
 }

### Inverse image packing of transform coefficients

Inputs to this process are a 3D array dispQuantCoeffFrame, of size asps\_frame\_width × asps\_frame\_height × 3, the variables patch2dSizeX, patch2dSizeY, patch2dPosX, patch2dPosY, bitDepth, subdivisionIterationCount, and verCoordCount.

The output of this process is a 2D array dispQuantCoeffArray, of size verCoordCount × 3, indicating the quantized displacement wavelet coefficients.

The 2D array dispQuantCoeffArray is initialized to 0. The wavelet coefficients inverse packing process proceeds as follows:

First, patchWidthInblocks, pixelsPerBlock and shift are derived as follows:

patchWidthInBlocks = (patch2dSizeX + PatchPackingBlockSize - 1)/ PatchPackingBlockSize  
 pixelsPerBlock = PatchPackingBlockSize \* PatchPackingBlockSize  
 shift = (1 << bitDepth) >> 1

Then, all the elements of 1D arrays of size subdivisionIterationCount + 1, vStart, vEnd, and startBlock are initialized as 0 and a variable blockCount is set as 0.

The variable dispPackingOrder is set as the syntax element asve\_packing\_method, and the variable videoChromaFormat is set as the variable DecGeoChromaFormat from the decoded geometry video component.

vStart, vEnd, startBlock and blockCount are calculated as follows:

if ( subdivisionIterationCount = 0 ){  
 vEnd[ 0 ] =   AtlasPatchTotalVertexCount[ atlasPatchIdx ]  
 blockCount = (  AtlasPatchTotalVertexCount[ atlasPatchIdx ]  
 + PatchPackingBlockSize - 1 )/PatchPackingBlockSize   
 }else {  
 for( i= 0; i < subdivisionIterationCount +1; i++ ){  
 vStart[ i ] = i == 0? 0 : AtlasPatchVertexCount[ atlasPatchIdx ][ i - 1 ]  
 vEnd[ i ] = vStart[ i ] + i ] + AtlasPatchVertexCount[ atlasPatchIdx ][ i ]  
 blockCountLevel [ i ] = AtlasPatchVertexBlockCount[ atlasPatchIdx ][ i ] +  
 ( AtlasPatchVertexCountLast[ atlasPatchIdx ][ i ] == 0 ? 0 : 1 )startBlock[ i ] = i == 0 ? 0 : (startBlock[ i - 1 ] + blockCount [ i ])  
 blockCount += blockCountLevel[ i ]  
 }  
 }

Then dispQuantCoeffArray is acquired as follows:

heightInBlocks = (blockCount + widthInBlocks – 1) / patchWidthInBlocks   
 origHeight = heightInBlocks \* PatchPackingBlockSize  
 totalBlocksInPatch = ( patch2dSizeX \* origHeight ) / pixelsPerBlock  
 for( lodIdx = 0; lodIdx < subdivisionIterationCount + 1; lodIdx++ ) {  
 for( v = vStart[ lodIdx ]; v < vEnd[ lodIdx ]; v++ ) { blockIndex = ( v – vStart[ lodIdx ] ) / pixelsPerBlock+ startBlock[ lodIdx ]  
 indexWithinBlock = ( v – vStart[ lodIdx ] ) % pixelsPerBlock  
 if( dispPackingOrder ){  
 blockIndex = totalBlocksInVideoFrame – 1 - blockIndex  
 indexWithinBlock = pixelsPerBlock – 1 - indexWithinBlock  
 }  
 x0 = ( blockIndex % widthInBlocks ) \* PatchPackingBlockSize    
 y0 = ( blockIndex / widthInBlocks ) \* PatchPackingBlockSize    
 ( x, y ) = computeMorton2D( indexWithinBlock )  
 x1 = x0 + x + patch2dPosX  
 y1 = y0 + y + patch2dPosX  
 for( d = 0; d < DisplacementDim; d++ ) {  
 if ( videoChromaFormat == 4:2:0 || videoChromaFormat == 4:2:2 ||  
  videoChromaFormat == 4:0:0 ) {  
 if( dispPackingOrder )  
 dispQuantCoeffArray[ v ][ d ] =  
 dispQuantCoeffFrame[ x1 ][ d \* origHeight + y1 ][ 0 ] – shift  
 } else {  
 dispQuantCoeffArray[ v ][ d ] =  
 dispQuantCoeffFrame[ x1 ][ y1 ][ d ] – shift  
 }  
 }  
 }  
 }

### Inverse quantization

Inputs to this process are the variables verCoordCount, subdivisionIterationCount, the 2D array dispQuantCoeffArray, of size verCoordCount × 3, and the 1D array levelOfDetailCounts, of size (subdivisionIterationCount + 1).

The output of this process is a 2D array dispCoeffArray, of size verCoordCount × 3.

First, a 2D array iscale, of size (subdivisionIterationCount + 1)× 3, indicating inverse scale factor is derived as follows:

lodQuantizationFlag = vqp\_lod\_quantization\_flag[ QpIndex ]  
 directQuantizationEnableFlag = vqp\_direct\_quantization\_enabled\_flag[ QpIndex ]  
 if ( lodQuantizationFlag ) {  
 for( lodIdx = 0; lodIdx < subdivisionIterationCount + 1; lodIdx++ ) {  
 for( dimIdx = 0; dimIdx < DisplacementDim; dimIdx ++ ) {  
 iscale[ lodIdx ][ dimIdx ] = InverseScale[ QpIdx ][ lodIdx ][ dimIdx ]  
 }  
 }  
 } else {  
 for( dimIdx = 0; dimIdx < DisplacementDim; dimIdx++ ) { iscale[ 0 ][ dimIdx ] = InverseScale[ QpIdx ][ 0 ][ dimIdx ]  
 levelOfDetailInverseScale[ dimIdx ] =   
 1 << vqp\_log2\_lod\_inverse\_scale[ QpIdx ][ dimIdx ]  
 }  
 for( lodIdx = 1; lodIdx < lodCount; lodIdx++ ) {  
 for( dimIdx = 0; dimIdx < DisplacementDim; dimIdx++ ) {  
 iscale[ lodIdx ][ dimIdx ] = iscale[ lodIdx - 1 ][ dimIdx ] \*   
 levelOfDetailInverseScale[ dimIdx ]  
 }  
 }  
 }

Then, all the elements of the 2D array dispCoeffArray, of size verCoordCount × 3, are initialized as 0.

The inverse quantization process proceeds as follows:

vcount0 = 0  
 for( i = 0; i < subdivisionIterationCount; i++ ) {  
 vcount1 = levelOfDetailCounts[ i ]  
 for( v = vcount0; v < vcount1; v++ ) {  
 for( d = 0; d < DisplacementDim; d++ ) {  
 dispCoeffArray[ v ][ d ] = dispQuantCoeffArray[ v ][ d ] \* iscale[ i ][ d ]  
 }  
 }  
 vcount0 = vcount1  
 }

Then, if vqp\_inverse\_quantization\_offset\_enable\_flag is 1, all the elements of the inverse quantized 2D array dispCoeffArray, of size verCoordCount × 3, are adjusted by quantization offset values as follows:

vcount0 = 0  
 for( i = 0; i < subdivisionIterationCount; i++ ) {  
 vcount1 = levelOfDetailCounts[ i ]  
 for( v = vcount0; v < vcount1; v++ ) {  
 for( d = 0; d < DisplacementDim; d++ ) {  
 k = dispCoeffArray[ v ][ d ] == 0 ? 0 :  
 (dispCoeffArray[ v ][ d ] > 0 ? 1 ： 2)  
 inverseQuantizationOffsetSign[ QpIndex ][ i ][ d ][ k ] =  
 ( i ==0 ? inverseQuantizationOffsetSign[ QpIndex ][ i ][ d ][ k ]:  
 (inverseQuantizationOffsetSign[ QpIndex ][ i ][ d ][ k ] +  
 inverseQuantizationOffsetSign[ QpIndex ][ i - 1 ][ d ][ k ]))  
 inverseQuantizationOffsetValuePrec1[ QpIndex ][ i ][ d ][ k ] = ( i == 0 ?  
 inverseQuantizationOffsetValuePrec1[ QpIndex ][ i ][ d ][ k ] :  
 (inverseQuantizationOffsetValuePrec1[ QpIndex ][ i ][ d ][ k ] +  
 inverseQuantizationOffsetValuePrec1[ QpIndex ][ i - 1 ][ d ][ k ]))  
 inverseQuantizationOffsetValuePrec2[ QpIndex ][ i ][ d ][ k ] = ( i == 0 ?  
 inverseQuantizationOffsetValuePrec2[ QpIndex ][ i ][ d ][ k ] :  
 (inverseQuantizationOffsetValuePrec2[ QpIndex ][ i ][ d ][ k ] +  
 inverseQuantizationOffsetValuePrec2[ QpIndex ][ i - 1 ][ d ][ k ]))  
 dispCoeffArray[ v ][ d ] +=  
 pow(2,-inverseQuantizationOffsetValuePrec1[ QpIndex ][ i ][ d ][ k ]) +  
 pow(2,-inverseQuantizationOffsetValuePrec2[ QpIndex ][ i ][ d ][ k ]) \*  
 (inverseQuantizationOffsetSign[ QpIndex ][ i ][ d ][ k ] == 0 ? :1:-1) }  
 }  
 vcount0 = vcount1  
 }

### Inverse transform

#### General

Inputs to this process are the variables verCoordCount, subdivisionIterationCount, transformMethod, the 1D array levelOfDetailCounts, of size (subdivisionIterationCount + 1), and the 2D arrays dispCoeffArray, of size verCoordCount × 3, and edges, of size verCoordCount × 2.

The output of this process is dispArray, which is a 2D array of size verCoordCount × 3 indicating the displacements to be applied to the mesh positions.

If transformMethod is equal to LINEAR\_LIFTING, then process described in subclause 11.2.6.2 is invoked with the parameters variables verCoordCount, dispCoeffArray, subdivisionIterationCount, levelOfDetailCounts, edges, valenceUpdate, and valences as inputs, dispArray as outputs

Otherwise, if transformMethod is equal to NONE, the output 2D array dispArray is derived as follows:

dispArray = dispCoeffArray

#### Inverse linear lifting transform

Inputs to this process are:

* verCoordCount, which is a variable indicating the number of vertex coordinates in the subdivided submesh.
* dispCoeffArray, which is a 2D array of size verCoordCount × 3 indicating the displacement wavelet coefficients.
* subdivisionIterationCount, which is a variable indicating the number of subdivision iterations.
* levelOfDetailCounts, a 1D array of size (subdivisionIterationCount + 1) indicating the number of vertex coordinates associated with each subdivision iteration.
* edges, which is a 2D array of size verCoordCount × 2 which indicates for each vertex v produced by the subdivision process described in subclause 11.2.3 the two indices (a, b) of the two vertices used to generated it (i.e., v generated as the middle of the edge (a, b)).
* valenceUpdate, which is a variable indicating whether the update operation use valence-based weight (when 1) or not (when 0), and
* valences, which is a 1D array of size levelOfDetailCounts[0], indicating the number of connected edges at each vertex of the submesh before any subdivisions are applied.

The output of this process is:

* a 2D array dispArray, of size verCoordCount × 3, indicating the displacements to be applied to the mesh position.

First, variables are derived as follows:

for(i = 0; i < subdivisionIterationCount; i++ ){   
 updateWeights[ i ] = UpdateWeight[ LtpIndex ][ i ]  
 predWeights[ i ] = PredictionWeight[ LtpIndex ][ i ]  
 }  
 skipUpdate = vltp\_skip\_update\_flag[ 0 ][ LtpIndex ]

The inverse wavelet transform process proceeds as follows:

for( i = 0; i < subdivisionIterationCount; i++ ) {  
 vcount0 = levelOfDetailVertexCounts[ i ]  
 vcount1 = levelOfDetailVertexCounts[ i + 1 ]  
 if ( asve\_lifting\_offset\_present\_flag ) {  
 for ( v = vcount0; skipUpdate == 0 && v < vcount1; ++v ) {  
 dispCoeffArray[ v ][ 0 ] += VltpLiftingOffset[ i ]  
 }  
 }  
 for ( v = vcount0; skipUpdate == 0 && v < vcount1; ++v ) {  
 a = verCoordEdges[ v ][ 0 ]  
 b = verCoordEdges[ v ][ 1 ]  
 for( d = 0; d < DisplacementDim; d++ ) {  
 if (valenceUpdate) {  
 valenceA = ( a < levlofDetailVertexCounts[0] ) ? valences[a] : 6  
 valenceB = ( b < levlofDetailVertexCounts[0] ) ? valences[b] : 6  
 disp0 = updateWeights[ i ] \* dispCoeffArray[ v ][ d ] / valenceA  
 disp1 = updateWeights[ i ] \* dispCoeffArray[ v ][ d ] / valenceB   
 dispCoeffArray [ a ][ d ] -= disp0  
 dispCoeffArray [ b ][ d ] -= disp1  
 }   
 else {  
 disp = updateWeights[ i ] \* dispCoeffArray[ v ][ d ]  
 dispCoeffArray[ a ][ d ] -= disp  
 dispCoeffArray[ b ][ d ] -= disp  
 }  
 }  
 }  
 for ( v = vcount0; v < vcount1; ++v ) {  
 a = verCoordEdges[ v ][ 0 ]  
 b = verCoordEdges[ v ][ 1 ]  
 for( d = 0; d < DisplacementDim; d++ ) {  
 dispCoeffArray[ v ][ d ] +=  
 predWeights \* ( dispCoeffArray[ a ][ d ] + dispCoeffArray[ b ][ d ] )  
 }  
 }  
 }  
for ( v = 0; v < verCoordCount; ++v ) {  
 for( d = 0; d < DisplacementDim; d++ ) {  
 dispArray[ v ][ d ] = dispCoeffArray[ v ][ d ]  
 }  
}

### Normal, tangent, and bitangent vector generation

#### General

Inputs to this process are a struct subdivSubmeshFrame and a variable dispCoordinateSystem.

The outputs of this process are 2D arrays normal, tangents, and bitangents, of size subdivSubmeshFrame.verCoordCount × 3.

First, normal, tangents and bitangents are initialized as follows:

for( i = 0; i < subdivSubmeshFrame.verCoordCount; i++ ){  
 normals[ i ][ 0 ] = 1  
 normals[ i ][ 1 ] = 0  
 normals[ i ][ 2 ] = 0  
 tangents[ i ][ 0 ] = 0  
 tangents[ i ][ 1 ] = 1  
 tangents[ i ][ 2 ] = 0  
 bitangents[ i ][ 0 ] = 0  
 bitangents[ i ][ 1 ] = 0  
 bitangents[ i ][ 2 ] = 1  
 }

Then, the subdivided mesh normal vector generation process subclause 11.2.7.2 is invoked with the parameters subdivSubmeshFrame.verCoordCount, subdivSubmeshFrame.faceCount, and an array subdivSubmeshFrame.verCoordFaceArray as inputs, and the array normals as output.

If dispCoordinateSystem is equal to 1, the tangent and bitangent vector generation process described in subclause 11.2.7.3 is invoked with the variable subdivSubmeshFrame.verCoordCount and the array normal as inputs, and the arrays tangents, and bitangents as outputs.

#### Subdivided submesh normal vector generation process

The inputs of this process are:

* a variable verCoordCount, indicating the number of vertex coordinates in the subdivided submesh,
* a variable faceCount , indicating the number of faces in the subdivided submesh,
* a 2D array verCoordFacesArray , of size faceCount × 3, indicating the connectivity indices associated with the vertex coordinates of the subdivided submesh,
* a 2D array verCoordsArray, of size verCoordCount × 3, indicating the positions of the subdivided submesh.

The output of this process is:

* a 2D array normals, of size verCoordCount × 3 indicating the normals to be used when applying the displacements to the subdivided submesh positions.

for ( v = 0; v < verCoordCount; ++v ) {  
 for( d = 0; d < 3; d++ ) {  
 normals[ v ][ d ] = 0  
 }  
 }  
 for( t = 0; t < faceCount; t++ ) {  
 for( i = 0; i < 3; i++ ) {  
 vindex = verCoordFacesArray[ t ][ i ]  
 for( d = 0; d < 3; d++ ) {  
 pos[ i ][ d ] = verCoordsArray[ vindex ][ d ]  
 }  
 }  
 for( d = 0; d < 3; d++ ) {  
 u[ d ] = pos[ 1 ][ d ] - pos[ 0 ][ d ]  
 v[ d ] = pos[ 2 ][ d ] - pos[ 0 ][ d ]  
 }  
 n[ 0 ] = u[ 1 ] \* v[ 2 ] - u[ 2 ] \* v[ 1 ]  
 n[ 1 ] = u[ 2 ] \* v[ 0 ] - u[ 0 ] \* v[ 2 ]  
 n[ 2 ] = u[ 0 ] \* v[ 1 ] - u[ 1 ] \* v[ 2 ]  
 for( i = 0; i < 3; i++ ) {  
 v = verCoordFacesArray[ t ][ i ]  
 for( d = 0; d < 3; d++ ) {  
 normals[ v ][ d ] += n[ d ]  
 }  
 }  
 }  
 for ( v = 0; v < verCoordCount; ++v ) {  
 norm = sqrt( normals[ v ][ 0 ] \* normals[ v ][ 0 ] +  
 normals[ v ][ 1 ] \* normals[ v ][ 1 ] +  
 normals[ v ][ 2 ] \* normals[ v ][ 2 ] )  
 if ( norm > 0 ) {  
 for( d = 0; d < 3; d++ ) {  
 normals[ v ][ d ] /= norm  
 }  
 }  
 }

#### Tangent and bitangent vector generation process

Inputs to this process are:

* a variable verCoordCount, indicating the number of vertex coordinates in the subdivided submesh,
* a 2D array normal, of size verCoordCount × 3 indicating the normal to be used when applying the displacements to the subdivided submesh positions.

The outputs of this process are:

* a 2D array tangents, of size verCoordCount × 3, indicating the tangents to be used when applying the displacements to the subdivided submesh positions,
* a 2D array bitangents, of size verCoordCount × 3, indicating the tangents to be used when applying the displacements to the subdivided submesh positions.

The tangent and bitangent vector generation process are as follows:

for ( v = 0; v < verCoordCount ; ++v ) {  
 e0[ 0 ] = 1  
 e0[ 1 ] = 0  
 e0[ 2 ] = 0  
 e1[ 0 ] = 0  
 e1[ 1 ] = 1  
 e1[ 2 ] = 0  
 e2[ 0 ] = 0  
 e2[ 1 ] = 0  
 e2[ 2 ] = 1  
 d0 = normals[ v ][ 0 ]  
 d1 = normals[ v ][ 1 ]  
 d2 = normals[ v ][ 2 ]  
 a0 = abs( d0 )  
 a1 = abs( d1 )  
 a2 = abs( d2 )  
 if ( a0 <= a1 && a0 <= a2 ) {  
 for( d = 0; d < 3; d++ ) {  
 t[ d ] = e0[ d ] – d0 \* normals[ v ][ d ]  
 }  
 } else if ( a1 <= a2 ) {  
 for( d = 0; d < 3; d++ ) {  
 t[ d ] = e1[ d ] – d1 \* normals[ v ][ d ]  
 }  
 } else {  
 for( d = 0; d < 3; d++ ) {  
 t[ d ] = e2[ d ] – d2 \* normals[ v ][ d ]  
 }  
 }  
 norm = sqrt( t[ 0 ] \* t[ 0 ] + t[ 1 ] \* t[ 1 ] + t[ 2 ] \* t[ 2 ] )  
 if ( norm > 0 ) {  
 for( d = 0; d < 3; d++ ) {  
 t[ v ][ d ] /= norm  
 }  
 }  
 b[ 0 ] = n[ 1 ] \* t[ 2 ] - n[ 2 ] \* t[ 1 ]  
 b[ 1 ] = n[ 2 ] \* t[ 0 ] - n[ 0 ] \* t[ 2 ]  
 b[ 2 ] = n[ 0 ] \* t[ 1 ] - n[ 1 ] \* t[ 0 ]  
 for( d = 0; d < 3; d++ ) {  
 tangents[ v ][ d ] = t[ d ]  
 bitangents[ v ][ d ] = b[ d ]  
 }  
 }

### Vertex coordinates reconstruction

The inputs of this process are a struct subdivSubmeshFrame and the 2D arrays array dispArray, normals, tangents and bitangents, all of size subdivMeshFrame.verCoordCount × 3.

The output of this process a struct vertexRefinedSubmeshFrame.

First, the struct vertexRefinedSubmeshFrame is set equal to the input subdivSubmeshFrame.

Then, the vertex coordinates displacement process proceeds as follows:

for ( v = 0; v < vertexRefinedSubmeshFrame.verCoordCount ; ++v ) {  
 for( d = 0; d < 3; d++ ) {  
 vertexRefinedSubmeshFrame.verCoordsArray[ v ][ d ] =  
 subdivSubmeshFrame.verCoordsArray[ v ][ d ] +  
 dispArray[ v ][ 0 ] \* normals[ v ][ d ] +  
 dispArray[ v ][ 1 ] \* tangents[ v ][ d ] +  
 dispArray[ v ][ 2 ] \* bitangents[ v ][ d ]  
 if( vertexRefinedSubmeshFrame.verCoordsArray[ v ][ d ] < 0 ß) {  
 vertexRefinedSubmeshFrame.verCoordsArray[ v ][ d ] = 0  
 } else if(vertexRefinedSubmeshFrame.verCoordsArray[ v ][ d ] >=   
 (1 << bitDepthPosition)){  
 vertexRefinedSubmeshFrame.verCoordsArray[ v ][ d ] =   
 (1 << bitDepthPosition) - 1)  
 }  
 }  
 }

It is a requirement of displacement bitstream conformance that the positions of the reconstructed subdivided submesh recSubmeshVerCoords[ v ][ d ] shall be in the range 0 .. ( ( 1 << bitDepthPosition) – 1 ), whereby bitDepthPosition is equal to the syntax element asps\_geometry\_3d\_bit\_depth\_minus1 + 1.

### Texture coordinate adaptation process

Inputs to this process are a struct vertexRefinedSubmeshFrame, the variables subTextureWidth, subTextureHeight, subTexturePosX, subTexturePosY, textureVideoWidth, and textureVideoHeigh.

The output of this process is a struct recSubmeshFrame with the updated texture coordinates.

First, a struct recSubmeshFrame is initializedwith an input of vertexRefinedSubmeshFrame.

Then, the scale factors are derived as follows:

resolutionRatioX = ( subTextureWidth – 1 ) / ( textureVideoWidth – 1 )  
 resolutionRatioY = ( subTextureHeight – 1 ) / ( textureVideoHeight – 1 )  
 shiftPosX = ( subTexturePosX ) / ( textureVideoWidth – 1 )  
 shiftPosY =  
 ( textureVideoHeight - subTextureHeight – subTexturePosY ) /   
 ( textureVideoHeight – 1 )

The adaptation process is as followings:

for( i =0; i <= recSubmeshFrame.attrValueCountArray[ attrIdxTexCoords ] ; i++ ){  
 recSubmeshFrame.attrValueArray[ attrIdxTexCoords ][ i ][ 0 ] =   
 vertexRefinedSubmeshFrame.attrValueArray[ attrIdxTexCoords ][ i ][ 0 ]   
 \* resolutionRatioX + shiftPosX  
 recSubmeshFrame.attrValueArray[ attrIdxTexCoords ][ i ][ 1 ] =   
 vertexRefinedSubmeshFrame.attrValueArray[ attrIdxTexCoords ][ i ][ 1 ]   
 \* resolutionRatioY + shiftPosY  
 }

## Append submesh to a mesh process

Inputs to this process are submeshCount and a 1D array refinedSubmeshFrames, of size submeshCount.

Output to this process is a struct recMeshFrame.

First, recMeshFrame is set equal to refinedSubmeshFrames[ 0 ].

Each variable member of refinedSubmeshFrame[ i ] for i = 1..submeshCount – 1 that indicate the size of an array member is accumulated into the corresponding variable member of recMeshFrame. Array members of refinedSubmeshFrame[ i ] for i = 1..submeshCount – 1 that indicate the vertex coordinates and the attributes are concatenated to the corresponding array member of recMeshFrame. Array members of refinedSubmeshFrame[ i ] for i = 1..submeshCount – 1 that indicate the connectivities are concatenated with adjustment to the corresponding array member of recMeshFrame. The process is as follows:

for( i = 1; i < submeshCount; i++){  
 for( v = 0; v < refinedSubmeshFrame[ i ].verCoordCount; v++ ){  
 recMeshFrame.verCoordsArray[ recMeshFrame.verCoordCount + v ] =  
 refinedSubmeshFrame[ i ].verCoordsArray[ v ]  
 }  
 for( v = 0; v < refinedSubmeshFrame[ i ].texCoordCount; v++ ){  
 recMeshFrame.texCoordsArray[ recMeshFrame.texCoordCount + v ] =  
 refinedSubmeshFrame[ i ].texCoordsArray[ v ]  
 }  
 recMeshFrame.verCoordCount += refinedSubmeshFrame[ i ].verCoordCount  
 recMeshFrame.texCoordCount += refinedSubmeshFrame[ i ].texCoordCount  
 for( v = 0; v < refinedSubmeshFrame[ i ].faceCount; v++ ){  
 for( k = 0; k < 3; k++ ){  
 recMeshFrame.verCoordFacesArray[ recMeshFrame.faceCount + v ][ k ]=  
 refinedSubmeshFrame[ i ].verCoordFacesArray[ v ][ k ] +  
 recMeshFrame.verCoordCount   
 }  
 for( v = 0; v < refinedSubmeshFrame[ i ].texCoordFaceCount; v++ ){  
 for( k = 0; k < 3; k++ ){  
 recMeshFrame.texCoordFacesArray[ recMeshFrame.texCoordFaceCount + v ][ k ] =  
 refinedSubmeshFrame[ i ].texCoordFacesArray[ v ][ k ] +  
 recMeshFrame.verCoordCount  
 }  
 }  
 }  
 recMeshFrame.faceCount += refinedSubmeshFrame[ i ].faceCount recMeshFrame.texCoordFaceCount += refinedSubmeshFrame[ i ].texCoordFaceCount  
 }

# Post-reconstruction process

## General

The post-reconstruction process depends on the profiles defined in Annex A.

The processes described in this subclause are invoked for reconstructed mesh frames and syntax elements associated with the same atlas ID, identified by a variable postRecAtlasID.

This process is invoked after reconstructing the geometry and attributes for the current mesh frame with a composition time index compTimeIdx. This process takes as inputs the syntax elements and upper-case variables from subclauses 8, 9, 10, and 11, and the following variables and arrays:

* a struct RecMeshFrame, which represents a reconstructed mesh frame and has the following members:
  + a variable verCoordCount, specifying the number of vertex coordinates in the reconstructed mesh frame,
  + a 2D array verCoordsArray, of size verCoordCount × 3, specifying the reconstructed vertex coordinates,
  + a variable faceCount, specifying the number of vertex coordinates faces,
  + a 2D array verCoordFaces, of size faceCount × 3, specifying the reconstructed connectivity indices associated with the vertex coordinates,
  + a variable attrCount, specifying the number of attributes,
  + a 1D array attrValueCountArray, of size attrCount, specifying the number of values for an attribute in the reconstructed mesh frame,
  + a 1D array attrValueDimensionArray, of size attrCount, specifying the dimension of an attribute,
  + a 3D array attrValuesArray, indicating attribute values, where the dimensions correspond to the attribute index, the attribute value index, and the attribute value dimension index, respectively,
  + a 3D array attrFacesArray, indicating attribute connectivity, where the dimensions correspond to the attribute index, the attribute face index, and the attribute face dimension index, 0..2, respectively,
  + a variable submeshCount, specifying the number of submeshes in the reconstructed mesh frame, and
  + a 1D array submeshVerCoordCount, of size submeshCount, specifying the number of reconstructed vertices associated with the submesh.
  + a 1D array submeshLodCount, of size submeshCount, specifying the number of LoDs associated with the submesh.
  + a 2D array submeshVerCoordPerLodCount, of size submeshCount × submeshLodCount[ smIdx ], specifying the number of reconstructed vertices per LoD associated with the submesh with index smIdx.

The output of this process is:

* a struct PostRecMeshFrame, which represents a reconstructed mesh frame after applying the post-reconstruction.

The following ordered steps apply:

* If the mesh post-reconstruction system has selected to use the m-th zippering instance for performing zippering operations during mesh reconstruction and the bitstream contains a zippering SEI message at the time instant compTimeIdx, subclause 12.2 is invoked with variable m and the struct RecMeshFrame as inputs,the struct PostRecMeshFrame, which contains the updated positions of the vertices, and possible updated values of attributes, together with connectivities associated with the vertices, and the connectivities associated with the attributes, in the current reconstructed mesh frame after zippering.

NOTE – A bitstream may contain multiple zippering filter instances. The mesh reconstruction system can choose an appropriate filter instance for zippering based on considerations such as computational complexity, power, and visual quality requirements, etc. These considerations are outside the scope of this document.

## Zippering process

This process is invoked when a bitstream contains a zippering SEI message, and the mesh reconstruction system has selected to use this m-th zippering filter instance for performing zippering during mesh reconstruction.

Inputs to this process are:

* a variable m, specifying the zippering filter instance,
* a struct RecMeshFrame that represents a reconstructed mesh frame.

The output of this process is:

* a struct PostRecMeshFrame, which represents a reconstructed mesh frame after applying the post-reconstruction.

The following applies:

* The output structure PostRecMeshFrame is initialized with the input structure RecMeshFrame.
* A variable zipMethod is set to zp\_method\_type[ m ].
* If zipMethod is equal to 1, then subclause 12.2.1 is invoked with variable m and struct RecMeshFrame as inputs and struct Post RecMeshFrame as output.
* Otherwise, if zipMethod is equal to 2, subclause 12.2.2 is invoked with variable m and struct RecMeshFrame as inputs and struct Post RecMeshFrame as output.

### Zippering method by distance

This process is invoked when a bitstream contains a zippering SEI message with zippering method equals to 1. This method describes a zippering procedure based on the distance between vertices to be matched.

Inputs to this process are:

* a variable m, specifying the zippering filter instance,
* a struct RecMeshFrame that represents a reconstructed mesh frame.

Output of this process is:

* a struct PostRecMeshFrame that represents a reconstructed mesh frame after applying the post-reconstruction.

The following applies:

* First, the boundary vertices are identified by invoking subclause 12.2.3 with variables RecMeshFrame.verCoordCount, RecMeshFrame.faceCount, and arrays RecMeshFrame.verCoordsArray and RecMeshFrame.verCoordFaces as inputs, and array isBoundaryVertex as output.
* Then, the matching between boundary vertices is determined by invoking subclause 12.2.4 with variables m, RecMeshFrame.verCoordCount, RecMeshFrame.submeshCount, a 2D array RecMeshFrame.verCoordsArray, and 1D arrays RecMeshFrame.submeshVerCoordCount and isBoundaryVertex as inputs, and 1D array borderCount, and 2D arrays zipMatchSubmeshIdx and zipMatchBorderIdx as outputs.
* Finally, the border vertices are fused invoking subclause 12.2.6 with struct RecMeshFrame, 1D arrays isBoundaryVertex, borderCount, 2D arrays zipMatchSubmeshIdx and zipMatchBorderIdx as inputs, and the struct PostRecMeshFrame and the arrays submeshAdjacencyCounts, submeshAdjacency, and matchedBorderPointFlag as outputs.

### Zippering method by index matching

This process is invoked when a bitstream contains a zippering SEI message with zippering method equals to 2. This method describes a zippering procedure based on signalling the index of vertices that are matched.

Inputs to this process are:

* a variable m, specifying the zippering filter instance,
* a struct RecMeshFrame that represents a reconstructed mesh frame.

Output of this process is:

* a struct PostRecMeshFrame that represents a reconstructed mesh frame after applying the post-reconstruction.

The following applies:

* First, the boundary vertices are identified by invoking subclause 12.2.3 with variables RecMeshFrame.verCoordCount, RecMeshFrame.faceCount, and arrays RecMeshFrame.verCoordsArray and RecMeshFrame.verCoordFaces as inputs, and array isBoundaryVertex as output.
* Then, the matching between boundary vertices is identified by invoking subclause 12.2.5 with variable m as input, and 1D array borderCount, and 2D arrays zipMatchSubmeshIdx and zipMatchBorderIdx as outputs.
* Next, the border vertices are fused invoking subclause 12.2.6 with struct RecMeshFrame, a 1D array isBoundaryVertex, and 1D array borderCount, and 2D arrays zipMatchSubmeshIdx and zipMatchBorderIdx as inputs, and the struct PostRecMeshFrameNoLodMatch and the arrays submeshAdjacencyCounts, submeshAdjacency, and matchedBorderPointFlag as outputs.
* Finally, unmatched LoD vertices are handled invoking subclause 12.2.7 with struct PostRecMeshFrameNoLodMatch and arrays isBoundaryVertex, borderCount, zipMatchSubmeshIdx, zipMatchBorderIdx, matchedBorderPointFlag submeshAdjacencyCounts, and submeshAdjacency as input, and PostRecMeshFrame as output.

### Identification of boundary vertices

This process identifies a vertex on a mesh as a boundary vertex if it lies on an edge being used by a single triangle. Zippering should be applied only to vertices which correspond to such boundary vertices.

Inputs to this process are:

* a variable verCoordCount, specifying the number of vertex coordinates in the reconstructed mesh frame,
* a variable faceCount, specifying the number of faces in the reconstructed mesh frame,
* a 2D array recVerCoords, of size verCoordCount × 3, specifying the reconstructed vertex coordinates,
* a 2D array recVerCoordFaces, of size faceCount × 3, specifying the reconstructed connectivity indices associated with the vertex coordinates,

Outputs of this process are:

* a 1D array isBoundaryVertex, of size verCoordCount, specifying whether the n-th reconstructed vertex is a boundary vertex of the reconstructed mesh.

First, the output array isBoundaryVertex[ n ], n =0..verCoordCount – 1, is initialized to 0.

Then, the following applies:

ComputeNeighbours( faceCount, recVerCoordFaces, verNeighbours, verNeighbourCounts)  
 ComputeTriangleNeighbours( faceCount, recVerCoordFaces,  
 triNeighbours, triNeighbourCounts)  
 for( vIdx = 0; vIdx < verCoordCount; vIdx++ )  
 isBoundaryVertex[ vIdx ] = 0  
 for( fIdx = 0; fIdx < faceCount; fIdx++ )  
 faceTag[ fIdx ] = -1  
 for( vIdx = 0; vIdx < verCoordCount; vIdx++ )  
 for( m = 0; m < neighbourCounts[ n ]; m++ )  
 adj = neighbours[ vIdx ][ m ]  
 if( adj > vIdx ){  
 //mark all triangles connected to vIdx as 0  
 for( n = 0; n < triNeighbourCounts[ vfdx ]; m++ ){  
 fIdx = neighbours[ vIdx ][ n ]  
 faceTag[ fIdx ] = 0  
 }  
 //mark all triangles connected to adj as 1  
 for( n = 0; n < triNeighbourCounts[ adj ]; m++ ){  
 fIdx = neighbours[ adj ][ n ]  
 faceTag[ fIdx ] = 1  
 }  
 //now for all triangles connected to vIdx, add the tags different than 0  
 counter = 0  
 for( n = 0; n < triNeighbourCounts[ vfdx ]; m++ ){  
 if( faceTag[ fIdx ] != 0 )  
 counter = counter + 1  
 }  
 //if the counter is equal to 1, then this is a boundary vertex  
 if( counter == 1 ) {  
 isBoundaryVertex[ vIdx ] = 1  
 isBoundaryVertex[ adj ] = 1  
 }  
 }  
 }  
 }

### Identification of matched boundary vertices by distance

Inputs to this process are:

* a variable m, specifying the zippering filter instance,
* a variable verCoordCount, specifying the number of vertex coordinates in the reconstructed mesh frame,
* a variable submeshCount, specifying the number of submeshes in the reconstructed mesh frame,
* a 2D array recVerCoords, of size verCoordCount × 3, specifying the reconstructed vertex coordinates,
* a 1D array submeshVerCoordCount, of size submeshCount, specifying number of reconstructed vertices associated with the submesh,
* a 1D array isBoundaryVertex, of size verCoordCount, specifying whether the n-th reconstructed vertex is a boundary vertex of the reconstructed mesh.

Outputs of this process are:

* a 1D array borderCount, of size submeshCount, specifying number of border vertices in the submesh,
* a 2D array zipMatchSubmeshIdx, of size submeshCount × borderCount[ i ], specifying the submesh index of the matched border point,
* a 2D array zipMatchBorderIdx, of size submeshCount × borderCount[ i ], specifying the border index of the matches border point.

The following applies:

pointIdx = 0  
 for( smIdx = 0; smIdx < submeshCount; smIdx++ ) {  
 bIdx = 0  
 for( vIdx = 0; vIdx < submeshVerCoordCount[ smIdx ]; vIdx++ ) {  
 if( isBoundaryVertex[ pointIdx ] ){  
 matchedBorderPointFlag[ smIdx ][ bIdx ] = 0  
 submeshIdx[ pointIdx ] = smIdx  
 submeshVertexIdx[ pointIdx ] = bIdx  
 bIdx = bIdx + 1  
 } else {  
 submeshIdx[ pointIdx ] = -1  
 submeshVertexIdx[ pointIdx ] = -1  
 }  
 pointIdx = pointIdx + 1  
 }  
 borderCount[ smIdx ] = bIdx  
 }  
 maxDistance = zp\_max\_match\_distance[ m ]

If the maxDistance is equal to 0, it indicates that the distance between border points is zero, indicating that there are no gaps between the submeshes, and zippering is not necessary, so load the output indicating no matches in the following way:

for( smIdx = 0; smIdx < submeshCount; smIdx++ ) {  
 for( bIdx = 0; bIdx < borderCount[ smIdx ]; bIdx++ ) {  
 zipMatchSubmeshIdx[ smIdx ][ bIdx ] = submeshCount  
 zipMatchBorderIdx[ smIdx ][ bIdx ] = -1  
 }  
 }

Otherwise, calculate the array of distance between border points in the following way:

if( zp\_send\_distance\_per\_submesh[ m ] ){  
 numSubmesh = zp\_number\_of\_submeshes\_minus1[ m ] + 1  
 for( smIdx = 0; smIdx < numSubmesh; smIdx++ ) {  
 smMaxDistance = zp\_max\_match\_distance\_per\_submesh[ m ][ smIdx ]  
 if( smMaxDistance == 0 ){  
 for( bIdx = 0; bIdx < borderCount[ smIdx ]; bIdx++ ) {  
 zipBoundaryDistance[ smIdx ][ bIdx ] = smMaxDistance  
 }  
 } else {  
 if( zp\_send\_distance\_per\_border\_point[ m ][ smIdx ] ){  
 bCount = zp\_number\_of\_border\_points[ m ][ smIdx ]  
 for( bIdx = 0; bIdx < bCount; bIdx++ ) {  
 zipBoundaryDistance[ smIdx ][ bIdx ] =  
 zp\_border\_point\_distance[ m ][ smIdx ][ bIdx ]  
 }  
 } else {  
 for( bIdx = 0; bIdx < borderCount[ smIdx ]; bIdx++ ) {  
 zipBoundaryDistance[ smIdx ][ bIdx ] = smMaxDistance  
 }  
 }  
 }  
 } } else {  
 for( smIdx = 0; smIdx < submeshCount; smIdx++ ) {  
 for( bIdx = 0; bIdx < borderCount[ smIdx ]; bIdx++ ) {  
 zipBoundaryDistance[ smIdx ][ bIdx ] = maxDistance  
 }  
 }  
 }

Now use the distance array to find the best match between border points, as follows:

for( pIdx = 0; pIdx < verCoordCount; pIdx++ ) {  
 if( isBoundaryVertex[ pIdx ] ){  
 smIdx = submeshIdx[ pIdx ]   
 bIdx= submeshVertexIdx[ pIdx ]  
 vertex = recVerCoords[ pIdx ]  
 if( matchedBorderPointFlag[ smIdx ][ bIdx ] == 0 ) {  
 // initialize the structure   
 zipMatchSubmeshIdx[ smIdx ][ bIdx ] = submeshCount  
 zipMatchBorderIdx[ smIdx ][ bIdx ] = -1  
 // find all nearest points from other submeshes   
 minDist = zipBoundaryDistance[ smIdx ][ bIdx ]  
 for( pMatchIdx = 0; pMatchIdx < verCoordCount; pMatchIdx++ ) {  
 if( isBoundaryVertex[ pMatchIdx ] ){  
 smMatchedIdx = submeshIdx[ pIdx ]  
 bMatchedIdx= submeshVertexIdx[ pIdx ]  
 matchedVertex = recVerCoords[ pMatchIdx ]  
 if( smIdx != smMatchedIdx ){  
 distance = 0  
 for( k =0; k < 3; k++ ){  
 diff = vertex[ k ] – matchedVertex[ k ]  
 distance = distance + diff \* diff  
 }  
 if( distance < minDist ){  
 minDist = distance  
 zipMatchSubmeshIdx[ smIdx ][ bIdx ] = smMatchedIdx  
 zipMatchBorderIdx[ smIdx ][ bIdx ] = bMatchedIdx  
 }  
 }  
 }  
 }  
 if( zipMatchSubmeshIdx[ smIdx ][ bIdx ] != submeshCount ){  
 matchedBorderPointFlag[ smIdx ][ bIdx ] = 1  
 smMatchedIdx = zipMatchSubmeshIdx[ smIdx ][ bIdx ]  
 bMatchedIdx = zipMatchBorderIdx[ smIdx ][ bIdx ]  
 if( matchedBorderPointFlag[ smMatchedIdx ][ bMatchedIdx ] == 0 ){  
 matchedBorderPointFlag[ smMatchedIdx ][ bMatchedIdx ] = 1  
 zipMatchSubmeshIdx[ smMatchedIdx ][ bMatchedIdx ] = smIdx  
 zipMatchBorderIdx[ smMatchedIdx ][ bMatchedIdx ] = bIdx  
 }  
 }  
 }  
 }  
 }

### Identification of matched boundary vertices by explicit signaling

Input to this process is:

* a variable m, specifying the zippering filter instance,

Outputs of this process are:

* a 1D array borderCount, of size submeshCount, specifying number of border vertices in the submesh,
* a 2D array zipMatchSubmeshIdx, of size submeshCount × borderCount[ i ], specifying the submesh index of the matched border point,
* a 2D array zipMatchBorderIdx, of size submeshCount × borderCount[ i ], specifying the border index of the matches border point.

The following applies:

numSubmeshes = zp\_number\_of\_submeshes\_minus1[ m ] + 1  
 for( smIdx = 0; smIdx < numSubmeshes; smIdx++ ) {  
 borderCount[ smIdx ] = zp\_number\_of\_border\_points[ k ][ smIdx ]  
 for( bIdx = 0; bIdx < borderCount[ smIdx ]; bIdx++ ) {  
 zipMatchSubmeshIdx[ smIdx ][ bIdx ] =  
 zp\_border\_point\_match\_submesh\_index[ m ][ smIdx ][ bIdx ]  
 zipMatchBorderIdx[ smIdx ][ bIdx ] =  
 zp\_border\_point\_match\_border\_point\_index[ m ][ smIdx ][ bIdx ]  
 }  
 }

### Border vertex fusion

Inputs to this process are:

* a struct RecMeshFrame that represents a reconstructed mesh frame,
* a 1D array isBoundaryVertex, of size RecMeshFrame.verCoordCount, specifying whether the n-th reconstructed vertex is a boundary vertex of the reconstructed mesh,
* a 1D array borderCount, of size RecMeshFrame.submeshCount, specifying number of border vertices in the submesh,
* a 2D array zipMatchSubmeshIdx, of size RecMeshFrame.submeshCount × RecMeshFrame.submeshVerCoordCount[ i ], specifying the submesh index of the matched border point,
* a 2D array zipMatchBorderIdx, of size RecMeshFrame.submeshCount × RecMeshFrame.submeshVerCoordCount[ i ], specifying the border index of the matches border point.

Outputs of this process are:

* a struct PostRecMeshFrame that represents a reconstructed mesh frame after applying the post-reconstruction,
* a 1D array submeshAdjacencyCounts, of size RecMeshFrame.submeshCount, specifying number of adjacent submeshes for the i-th submesh,
* a 2D array submeshAdjacency, of size RecMeshFrame.submeshCount × RecMeshFrame.submeshAdjacencyCounts[ i ], specifying the submesh index of adjacent submesh
* a 2D array matchedBorderPointFlag, of size RecMeshFrame.submeshCount × borderCount[ i ], specifying if the j-th border vertex in the i-th submesh is matched or not.

The following process applies:

pointIdx = 0  
 for( smIdx = 0; smIdx < RecMeshFrame.submeshCount; smIdx++ ) {  
 bIdx = 0  
 for( vIdx = 0; vIdx < RecMeshFrame.submeshVerCoordCount[ smIdx ]; vIdx++ ) {  
 if( isBoundaryVertex[ pointIdx ] ){  
 matchedBorderPointFlag[ smIdx ][ bIdx ] = 0  
 submeshIdx[ pointIdx ] = smIdx  
 submeshVertexIdx[ pointIdx ] = bIdx  
 bIdx = bIdx + 1  
 }  
 pointIdx = pointIdx + 1  
 }   
 submeshAdjacencyCounts[ smIdx ] = 0  
 }  
 for( idx = 0; idx < RecMeshFrame.verCoordCount; idx++ ) {  
 if( isBoundaryVertex[ idx ] ){  
 curVerCoord = RecMeshFrame.verCoordsArray[ idx ]  
 curSubmeshIdx = submeshIdx[ idx ]  
 curBorderIdx = submeshVertexIdx[ idx ]  
 if( !matchedBorderPointFlag[ smIdx ][ bIdx ] ){  
 matchedSubmeshIdx = zipMatchSubmeshIdx[ curSubmeshIdx ][ curBorderIdx ]  
 matchedBorderIdx = zipMatchBorderIdx[ curSubmeshIdx ][ curBorderIdx ]  
 // transform the match vertex from border indexing to mesh indexing  
 matchedIdx = 0  
 for( i = 0; i < matchedSubmeshIdx; i++)  
 matchedIdx = matchedIdx + RecMeshFrame.submeshVerCoordCount[ i ]  
 borderIdx = 0  
 borderFoundFlag = 0  
 for( i = 0; i < RecMeshFrame.submeshVerCoordCount[ smIdx ]; i++)  
 if( isBoundaryVertex[ matchedIdx +i ] ){  
 if( borderIdx == matchedBorderIdx ){  
 matchedIdx = matchedIdx + i  
 borderFoundFlag = 1  
 }  
 else  
 borderIdx = borderIdx + 1  
 }  
 }  
 if( borderFoundFlag == 1 ){  
 matchedVerCoord = PostRecMeshFrame.verCoordsArray[ matchedIdx ]  
 if( matchedBorderPointFlag[ matchedSubmeshIdx ][ matchedBorderIdx ] ){  
 // matched vertex has been used before, so just copy the value  
 for( k = 0; n < 3; k++ )  
 PostRecMeshFrame.verCoordsArray[ idx ][ k ] =  
 matchedVerCoord[ k ]  
 } else {  
 // first match between bot vertices, use the average value  
 for( k = 0; n < 3; k++ )  
 avgVerCoord[ k ] =  
 ( curVerCoord[ k ] + matchedVerCoord[ k ] ) / 2  
 for( k = 0; k < 3; k++ )  
 PostRecMeshFrame.verCoordsArray[ idx ][ k ] = avgVerCoord[ k ]  
 for( k = 0; k < 3; k++ )  
 PostRecMeshFrame.verCoordsArray[ matchedIdx ][ k ] =  
 avgVerCoord[ k ]  
 // updating the lists of adjacent submeshes  
 if( findIndexInArray( matchedSubmeshIdx,  
 submeshAdjacency[ curSubmeshIdx ],  
 submeshAdjacencyCounts[ curSubmeshIdx ] ) == -1 ){  
 adjIdx = submeshAdjacencyCounts[ curSubmeshIdx ]  
 submeshAdjacency[ curSubmeshIdx ][ adjIdx ] = matchedSubmeshIdx  
 submeshAdjacencyCounts[ curSubmeshIdx ] =  
 submeshAdjacencyCounts[ curSubmeshIdx ] + 1  
 }  
 if( findIndexInArray( curSubmeshIdx,  
 submeshAdjacency[ matchedSubmeshIdx ],  
 submeshAdjacencyCounts[ matchedSubmeshIdx ] ) == -1 ){  
 adjIdx = submeshAdjacencyCounts[ curSubmeshIdx ]  
 submeshAdjacency[ matchedSubmeshIdx ][ adjIdx ] = curSubmeshIdx  
 submeshAdjacencyCounts[ curSubmeshIdx ] =  
 submeshAdjacencyCounts[ curSubmeshIdx ] + 1  
 }  
 }  
 }  
 }  
 }  
 }

### Handling of unmatched LoD vertices

This process is invoked when a bitstream contains a zippering SEI message with zippering method equals to 2. This method describes a zippering procedure for handling vertices generated as the result of unequal number of LoD vertices between adjacent submeshes.

Inputs to this process are:

* a variable m, specifying the zippering filter instance,
* a struct PostRecMeshFrame that represents a reconstructed mesh frame after applying the post-reconstruction
* a 1D array isBoundaryVertex, of size PostRecMeshFrame .verCoordCount, specifying whether the n-th reconstructed vertex is a boundary vertex of the reconstructed mesh,
* a 1D array borderCount, of size PostRecMeshFrame.submeshCount, specifying number of border vertices in the submesh,
* a 2D array zipMatchSubmeshIdx, of size PostRecMeshFrame.submeshCount × borderCount[ i ], specifying the submesh index of the matched border point,
* a 2D array zipMatchBorderIdx, of size PostRecMeshFrame.submeshCount × borderCount[ i ], specifying the border index of the matches border point,
* a 2D array matchedBorderPointFlag, of size PostRecMeshFrame.submeshCount × borderPointCounts[ i ], specifying if the j-th border vertex in the i-th submesh is matched or not
* a 1D array submeshAdjacencyCounts, of size PostRecMeshFrame.submeshCount, specifying number of adjacent submeshes for the i-th submesh,
* a 2D array submeshAdjacency, of size PostRecMeshFrame.submeshCount × submeshAdjacencyCounts[ i ], specifying the submesh index of adjacent submesh.

Outputs of this process are:

* a struct PostRecMeshFrameFull that represents a reconstructed mesh frame after applying the handling process of unmatched LoD vertices.

The following applies:

* The output structure PostRecMeshFrameFull is initialized with the input structure PostRecMeshFrame.
* A variable zipUnmatchedLODVerticesMethod is set equal to zp\_unmatched\_lod\_vertices\_method\_type[ m ].
* A 2D array frameSubmeshLodMaps, of size submeshCount × submeshVerCoordCount[ i ], specifying LoD index for n-th vertex of i-th submesh, is derived in the following way:

for( smIdx = 0; smIdx < PostRecMeshFrame.submeshCount; smIdx++ ) {  
 vIdx = 0  
 for( lodIdx = 0;  
 lodIdx < PostRecMeshFrame.submeshLodCount[ smIdx ];  
 lodIdx++ ) {  
 for( e = 0;  
 e < PostRecMeshFrame.submeshVerCoordCountPerLoD[ smIdx ][ lodIdx ];  
 e++) {  
 frameSubmeshLodMaps[ smIdx ][ vIdx ] = lodIdx  
 vIdx = vIdx + 1  
 }  
 }  
 }

* The arrays accSubmeshVerCoordCount, of size submeshCount, indicating the accumulated number of vertices per submesh, and the array submeshVertexId, of size PostRecMeshFrame.verCoordCount, which is used to change from border indexing to mesh vertex indexing, are derived in the following way:

accSubmeshVerCoordCount[ 0 ] = 0  
 pointIdx = 0  
 for( smIdx = 0; smIdx < PostRecMeshFrame.submeshCount; smIdx++ ) {  
 accSubmeshVerCoordCount[ smIdx + 1 ] = accSubmeshVerCoordCount[ smIdx ] +  
 PostRecMeshFrame.submeshVerCoordCount[ smIdx ]  
 bIdx = 0  
 for( vIdx = 0; vIdx < PostRecMeshFrame.submeshVerCoordCount[ smIdx ]; vIdx++ ) {  
 if( isBoundaryVertex[ pointIdx ] ){  
 submeshVertexIdx[ pointIdx ] = bIdx  
 bIdx = bIdx + 1  
 } else {  
 submeshVertexIdx[ pointIdx ] = -1  
 }  
 pointIdx = pointIdx + 1  
 }  
 }

* If zipUnmatchedLODVerticesMethod is equal to 1, then the subclause 12.2.7.1 is invoked with struct PostRecMeshFrame and arrays isBoundaryVertex, borderCount, matchedBorderPointFlag, submeshAdjacencyCounts, submeshAdjacency, frameSubmeshLodMaps, accSubmeshVerCoordCount and submeshVertexIdx as inputs, and PostRecMeshFrameFull as output.
* Otherwise, if zipUnmatchedLODVerticesMethod is equal to 2, then the subclause 0 is invoked with struct PostRecMeshFrame and arrays isBoundaryVertex, borderCount, zipMatchSubmeshIdx, zipMatchBorderIdx, matchedBorderPointFlag, submeshAdjacencyCounts, submeshAdjacency, frameSubmeshLodMaps, accSubmeshVerCoordCount and submeshVertexIdx as inputs, and PostRecMeshFrameFull as output.

#### Removing edges of unmatched LoD vertices

This process is invoked when a bitstream contains a zippering SEI message with zippering method equals to 2 and with zippering method for unmatched LoD vertices set to 1. This method describes a zippering procedure for removing edges asthe result of unmatched LoD vertices.

Inputs to this process are:

* a struct PostRecMeshFrame that represents a reconstructed mesh frame after applying the post-reconstruction ,
* a 1D array isBoundaryVertex, of size PostRecMeshFrame.verCoordCount, specifying whether the n-th reconstructed vertex is a boundary vertex of the reconstructed mesh,
* a 1D array borderCount, of size PostRecMeshFrame.submeshCount, specifying the number of border points in the submesh,
* a 2D array matchedBorderPointFlag, of size PostRecMeshFrame.submeshCount × borderCount[ i ], specifying if the j-th border vertex in the i-th submesh is matched or not
* a 1D array submeshAdjacencyCounts, of size PostRecMeshFrame.submeshCount, specifying number of adjacent submeshes for the i-th submesh,
* a 2D array submeshAdjacency, of size PostRecMeshFrame.submeshCount × submeshAdjacencyCounts[ i ], specifying the submesh index of adjacent submesh,
* a 2D array frameSubmeshLodMaps, of size PostRecMeshFrame.submeshCount × PostRecMeshFrame.submeshVerCoordCount[ i ], specifying LoD index for n-th vertex of i-th submesh
* a 1D array accSubmeshVerCoordCount, of size submeshCount, indicating the accumulated number of vertices per submesh,
* a 1D array submeshVertexId, of size PostRecMeshFrame.verCoordCount, used to change from border indexing to mesh vertex indexing.

Output of this process is:

* a struct PostRecMeshFrameFull that represents a reconstructed mesh frame after applying the handling process of unmatched LoD vertices.

First, the arrays verNeighbors and verNeighborsCount, which indicate the list of adjacent vertices to a vertex, and the arrays triNeighbors and triNeighborsCoutn, which indicate the list of triangles connected to a vertex, are obtained as follows:

ComputeNeighbours( PostRecMeshFrame.faceCount,  
 PostRecMeshFrame.verCoordFaces,  
 verNeighbours, verNeighbourCounts)  
 ComputeTriangleNeighbours( PostRecMeshFrame.faceCount,  
 PostRecMeshFrame.recVerCoordFaces,  
 triNeighbours, triNeighbourCounts)

Then, the following applies:

for( smIdx = 0; smIdx < PostRecMeshFrame.submeshCount; smIdx++ ) {  
 for( j = 0; j < submeshAdjacencyCounts[ smIdx ]; j++ ) {  
 neighSmIdx = submeshAdjacency[ smIdx ][ j ]  
 lodDiff = PostRecMeshFrame.submeshLodCount[ smIdx ] -  
 PostRecMeshFrame.submeshLodCount[ neighborIdx ]  
 if( lodDiff > 0 ) {  
 // getting the first vertex index of the unmatched lod  
 for( s = 0; s < PostRecMeshFrame.submeshVerCoordCount[ smIdx ] – 1; s++ ) {  
 if( frameSubmeshLodMaps[ smIdx ][ s ] ==  
 frameSubmeshLodMaps[ smIdx ][ posLast ] – lodDiff + 1 ) {  
 firstUnmatchedLodVertexIdx = s  
 break  
 }  
 }  
 // unmatched vertex loop (in reverse order to go from higher to lower lod)   
 for( i = PostRecMeshFrame.submeshVerCoordCount[ smIdx ] - 1;  
  i >= firstUnmatchedLodVertexIdx; i-- ) {  
 i\_frm = accSubmeshVerCoordCount[ smIdx ] + i  
 i\_brd = submeshVertexIdx[ i\_frm ]  
 if ( isBoundaryVertex[ i\_frm ] &&  
 !matchedBorderPointFlag[ smIdx ][ i\_brd ] ) {  
 //search for the two vertices that share border edge  
 brdCount = 0  
 for( m = 0; m < verNeighbourCounts[ i\_frm ]; m++ ) {  
 adj = verNeighbours[ i\_frm ][ m ]  
 for( n = 0; n < triNeighbourCounts[ i\_frm ]; n++ ) {  
 fIdx = triNeighbours[ i\_frm ][ n ]  
 faceTag[ fIdx ] = 0  
 }  
 for( n = 0; n < triNeighbourCounts[ adj ]; n++ ) {  
 fIdx = triNeighbours[ adj ][ n ]  
 faceTag[ fIdx ] = 1  
 }  
 counter = 0  
 for( n = 0; n < triNeighbourCounts[ i\_frm ]; n++ ) {  
 fIdx = triNeighbours[ i\_frm ][ n ]  
 if( faceTag[ fIdx ] != 0 ){  
 counter = counter + 1  
 brdFaceIdx = fIdx  
 }  
 }  
 if( counter == 1 ) {  
 brdFace[ brdCount ] = brdFaceIdx  
 brdVertex[ brdCount ] = adj  
 brdCount = brdCount + 1  
 }  
 }  
 // all triangles connected to i\_frm will go to brVertex[0]   
 for( n = 0; n < triNeighbourCounts[ i\_frm ]; n++ ) {  
 fIdx = triNeighbours[ i\_frm ][ n ]  
 for( k = 0; k < 3; k++)  
 if(PostRecMeshFrameFull.verCoordFaces[ fIdx ][ k ] ==  
 i\_frm)  
 PostRecMeshFrameFull.verCoordFaces[ fIdx ][ k ] =  
 brdVertex[ 0 ]  
 }  
 }  
 // create degenerate triangles  
 PostRecMeshFrame.verCoordFaces[ brdFace[ 0 ] ] =  
 ( i\_frm, i\_frm, i\_frm )  
 // do the same for the attributes as well  
 for( a = 0; a < PostRecMeshFrameFull.attrCount; a++){  
 attrFaceArray = PostRecMeshFrameFull.attrFacesArray[ a ]  
 for( k = 0; k < 3; k++ ) {  
 if(PostRecMeshFrameFull.verCoordFaces[ brdFace[ 0 ] ][ k ] ==  
 i\_frm )  
 i\_attr\_frm = attrFaceArray[ brdFace[ 0 ] ][ k ]  
 if(PostRecMeshFrameFull.verCoordFaces[ brdFace[ 0 ] ][ k ] ==  
 brdVertex[ 0 ] )  
 i\_attr\_neigh\_frm = attrFaceArray[ brdFace[ 0 ] ][ k ]  
 }  
 // all triangles connected to i\_attr\_frm go to i\_attr\_neigh\_frm   
 for( n = 0; n < triNeighbourCounts[ i\_frm ]; n++ ) {  
 fIdx = triNeighbours[ i\_frm ][ n ]  
 for( k = 0; k < 3; k++)  
 if(attrFaceArray[ fIdx ][ k ] ==  
 i\_attr\_frm)  
 attrFaceArray[ fIdx ][ k ] = i\_a\_neigh\_frm  
 }  
 }  
 // create degenerate triangles  
 attrFaceArray[ brdFace[ 0 ] ] =  
 ( i\_attr\_frm, i\_attr\_frm, i\_attr\_frm )  
 }  
 // and consider the vertex matched to avoid processing it again  
 matchedBorderPointFlag[ smIdx ][ i\_brd ] = 1  
 }  
 //remove degenerate triangles and unreferenced vertices   
 RemoveDegeneratedTriangles( PostRecMeshFrameFull )  
 }  
 }  
 }

#### Adding edges to unmatched LoD vertices

This process is invoked when when a bitstream contains a zippering SEI message with zippering method equals to 2 and with zippering method for unmatched LoD vertices set to 2. This method describes a zippering procedure for adding edges as the result of unmatched LoD vertices

Inputs to this process are:

* a struct PostRecMeshFrame that represents a reconstructed mesh frame after applying the post-reconstruction
* a 1D array isBoundaryVertex, of size PostRecMeshFrame.verCoordCount, specifying whether the n-th reconstructed vertex is a boundary vertex of the reconstructed mesh,
* a 1D array borderCount, of size PostRecMeshFrame.submeshCount, specifying the number of border points in the submesh,
* a 2D array zipMatchSubmeshIdx, of size PostRecMeshFrame.submeshCount × borderCount[ i ], specifying the submesh index of the matched border point,
* a 2D array zipMatchBorderIdx, of size PostRecMeshFrame.submeshCount × borderCount[ i ], specifying the border index of the matches border point,
* a 2D array matchedBorderPointFlag, of size PostRecMeshFrame.submeshCount × borderCount[ i ], specifying if the j-th border vertex in the i-th submesh is matched or not
* a 1D array submeshAdjacencyCounts, of size PostRecMeshFrame.submeshCount, specifying number of adjacent submeshes for the i-th submesh,
* a 2D array submeshAdjacency, of size PostRecMeshFrame.submeshCount × submeshAdjacencyCounts[ i ], specifying the submesh index of adjacent submesh,
* a 2D array frameSubmeshLodMaps, of size PostRecMeshFrame.submeshCount × PostRecMeshFrame.submeshVerCoordCount[ i ], specifying LoD index for n-th vertex of i-th submesh
* a 1D array accSubmeshVerCoordCount, of size submeshCount, indicating the accumulated number of vertices per submesh,

a 1D array submeshVertexId, of size PostRecMeshFrame.verCoordCount, used to change from border indexing to mesh vertex indexing.

Output of this process is:

* a struct PostRecMeshFrameFull that represents a reconstructed mesh frame after applying the handling process of unmatched LoD vertices.

First, the arrays verNeighbors and verNeighborsCount, which indicate the list of adjacent vertices to a vertex, and the arrays triNeighbors and triNeighborsCoutn, which indicate the list of triangles connected to a vertex, are obtained as follows:

ComputeNeighbours( PostRecMeshFrame.faceCount,  
 PostRecMeshFrame.verCoordFaces,  
 verNeighbours, verNeighbourCounts)  
 ComputeTriangleNeighbours( PostRecMeshFrame.faceCount,  
 PostRecMeshFrame.recVerCoordFaces,  
 triNeighbours, triNeighbourCounts)

Then, the following applies:

for( smIdx = 0; smIdx < PostRecMeshFrame.submeshCount; smIdx++ ) {  
 for( j = 0; j < submeshAdjacencyCounts[ smIdx ]; j++ ) {  
 neighSmIdx = submeshAdjacency[ smIdx ][ j ]  
 lodDiff = PostRecMeshFrame.submeshLodCount[ smIdx ] -  
 PostRecMeshFrame.submeshLodCount[ neighborIdx ]  
 if( lodDiff > 0 ) {  
 // getting the first vertex index of the unmatched lod  
 for( s = 0; s < PostRecMeshFrame.submeshVerCoordCount[ smIdx ] – 1; s++ ) {  
 if( frameSubmeshLodMaps[ smIdx ][ s ] ==  
 frameSubmeshLodMaps[ smIdx ][ posLast ] – lodDiff + 1 ) {  
 firstUnmatchedLodVertexIdx = s  
 break  
 }  
 }  
 // unmatched vertex loop  
 for( i = firstUnmatchedLodVertexIdx;  
 i <= PostRecMeshFrame.submeshVerCoordCount[ smIdx ]; i++) {  
 i\_frm = accSubmeshVerCoordCOunt[ smIdx ] + i  
 i\_brd = submeshVertexIdx[ i\_frm ]  
 if( isBoundaryVertex[ i\_frm ] &&  
 !matchedBorderPointFlag[ smIdx ][ i\_brd ] ) {  
 //search for the two vertices that share border edge  
 brdNeighbor = 0  
 for( m = 0; m < verNeighbourCounts[ i\_frm ]; m++ ) {  
 adj = verNeighbours[ i\_frm ][ m ]  
 for( n = 0; n < triNeighbourCounts[ i\_frm ]; n++ ) {  
 fIdx = triNeighbours[ i\_frm ][ n ]  
 faceTag[ fIdx ] = 0  
 }  
 for( n = 0; n < triNeighbourCounts[ adj ]; n++ ) {  
 fIdx = triNeighbours[ adj ][ n ]  
 faceTag[ fIdx ] = 1  
 }  
 counter = 0  
 for( n = 0; n < triNeighbourCounts[ i\_frm ]; n++ ) {  
 fIdx = triNeighbours[ i\_frm ][ n ]  
 if( faceTag[ fIdx ] != 0 )  
 counter = counter + 1  
 }  
 if( counter == 1 ) {  
 brdNeighbors[ brdNeighbor ] = submeshVertexIdx[ adj ]  
 brdNeighbor = brdNeighbor + 1  
 }  
 }  
 // get the matched vertices to the two adjacent neighbors  
 v1MatchedSmIdx = zipMatchSubmeshIdx[ smIdx ][ brdNeighbors[ 0 ] ]  
 v1MatchedBrdIdx = zipMatchBorderIdx[ smIdx ][ brdNeighbors[ 0 ] ]  
 v2MatchedSmIdx = zipMatchSubmeshIdx[ smIdx ][ brdNeighbors[ 1 ] ]  
 v2MatchedBrdIdx = zipMatchBorderIdx[ smIdx ][ brdNeighbors[ 1 ] ]  
 if( v1MatchedSmIdx == v2MatchedSmIdx &&  
 v1MatchedSmIdx == neighSmIdx ) {  
 // transforming to mesh vertices indexing  
 v1MatchedIdx = accSubmeshVerCoordCount[ v1MatchedSmIdx ]  
 borderIdx = 0  
 for( k = 0; k <  
 PostRecMeshFrame.submeshVerCoordCount[ v1MatchedSmIdx ];  
 k++ ) {  
 if( isBoundaryVertex[ v1MatchedIdx + k ] ) {  
 if( borderIdx == v1MatchedBrdIdx )  
 v1MatchedIdx = v1MatchedIdx + k  
 else  
 borderIdx = borderIdx + 1  
 }  
 }  
 v2MatchedIdx = accSubmeshVerCoordCount[ v2MatchedSmIdx ]  
 borderIdx = 0  
 for( k = 0; k <  
 PostRecMeshFrame.submeshVerCoordCount[ v2MatchedSmIdx ];  
 k++ ) {  
 if( isBoundaryVertex[ v2MatchedIdx + k ] ) {  
 if( borderIdx == v2MatchedBrdIdx )  
 v2MatchedIdx = v2MatchedIdx + k  
 else  
 borderIdx = borderIdx + 1  
 }  
 }  
 // now get the triangle that lies on the edge   
 for( n = 0; n < triNeighbourCounts[ v1MatchedIdx ]; n++ ) {  
 fIdx = triNeighbours[ v1MatchedIdx ][ n ]  
 faceTag[ fIdx ] = 0  
 }  
 for( n = 0; n < triNeighbourCounts[ v2MatchedIdx ]; n++ ) {  
 fIdx = triNeighbours[ v2MatchedIdx ][ n ]  
 faceTag[ fIdx ] = 1  
 }  
 counter = 0  
 for( n = 0; n < triNeighbourCounts[ v1MatchedIdx ]; n++ ) {  
 fIdx = triNeighbours[ v1MatchedIdx ][ n ]  
 if( faceTag[ fIdx ] != 0 ){  
 fUpdIdx = fIdx  
 counter = counter + 1  
 }  
 }  
 if( counter == 1 ) {  
 // getting the vertex on the opposite corner of the edge  
 uncV = 0  
 for( idx = 0; idx < 3; idx++ ) {  
 if( PostRecMeshFrame.verCoordFaces[ fIdx ][ idx ] !=  
 v1MatchedIdx &&  
 PostRecMeshFrame.verCoordFaces[ fIdx ][ idx ] !=  
 v2MatchedIdx )  
 uncV =  
 PostRecMeshFrame.verCoordFaces[ fUpdIdx ][ idx ]  
 }  
 //add new vertex in the middle of the edge  
 //and average with unmatched vertex  
 vIdx = PostRecMeshFrameFull.verCoordCount  
 for( k = 0; k < 3; k++ ) {  
 PostRecMeshFrameFull.verCoordsArray[ vIdx ][ k ] =  
 ( PostRecMeshFrame.verCoordsArray[ i\_frm ][ k ] +  
 ( PostRecMeshFrame.verCoordsArray[ v1MatchedIdx ][ k ] +  
 PostRecMeshFrame.verCoordsArray[ v2MatchedIdx ][ k ] ) /  
 2 ) / 2  
 PostRecMeshFrameFull.verCoordsArray[ i\_frm ][ k ] =  
 PostRecMeshFrameFull.verCoordsArray[ vIdx ][ k ]  
 }  
 PostRecMeshFrameFull.verCoordCount =  
 PostRecMeshFrameFull.verCoordCount + 1  
 //create a new triangle and update old one  
 fNewIdx = PostRecMeshFrameFull.faceCount  
 cIdx = ( idx + 2 ) % 3  
 PostRecMeshFrameFull.verCoordFaces[ fNewIdx ] =  
 ( uncV,  
 PostRecMeshFrameFull.verCoordsArray[ fNewIdx ],  
 PostRecMeshFrameFull.verCoordFaces[ fUpdIdx ][ cIdx ] )  
 cIdx = ( idx + 1 ) % 3  
 PostRecMeshFrameFull.verCoordFaces[ fUpdIdx ] =  
 ( uncV,  
 PostRecMeshFrameFull.verCoordFaces[ fUpdIdx ][ cIdx ],  
 PostRecMeshFrameFull.verCoordsArray[ fNewIdx ]  
 PostRecMeshFrameFull.faceCount =  
 PostRecMeshFrameFull.faceCount + 1  
 for( a = 0; a < RecMeshFrameFull.attrCount; a++ ) {  
 attrValues = PostRecMeshFrameFull.attrValuesArray[ a ]  
 attrFaces = PostRecMeshFrameFull.attrFacesArray[ a ]  
 attrNewValIdx =  
 PostRecMeshFrameFull.attrValueCountArray[ a ]  
 for( k = 0; k < attrValueDimensionArray[ a ]; k++ ) {  
 l = ( idx + 1 ) % 3  
 r = ( idx + 2 ) % 3  
 attrValues[ attrNewValIdx ][ k ] =  
 ( attrValues[ attrFaces[ fUpdIdx ][ l ] ][ k ] +  
 attrValues[ attrFaces[ fUpdIdx ][ r ] ][ k ] ) / 2  
 }  
 PostRecMeshFrameFull.attrValueCountArray[ a ] =  
 PostRecMeshFrameFull.attrValueCountArray[ a ] + 1  
 attrFaceNewIdx = PostRecMeshFrameFull.faceCount  
 attrFaces[ attrFaceNewIdx ] =  
 ( attrFaces[ fUpdIdx ][ idx ],  
 attrNewValIdx,  
 attrFaces[ fUpdIdx ][ ( idx + 2 ) % 3 ] )  
 attrFaces[ fUpdIdx ] =  
 ( attrFaces[ fUpdIdx ][ idx ],  
 attrFaces[ fUpdIdx ][ ( idx + 1 ) % 3 ],  
 attrNewValIdx )  
 }  
 )  
 }  
 }  
 }  
 }  
 }  
 }

# Adaptation process

The specifications in ISO/IEC 23090-5(2E):2023 clause 13 do not apply.

# Parsing process

The specifications in ISO/IEC 23090-5(2E):2023 clause 14 and its subclauses apply.

1. (normative)  
     
   Profile, tier, and levels
   1. Overview of profiles, tiers, and levels

The specifications in ISO/IEC 23090-5(2E):2023 Annex A.1 apply.

* 1. Profile, tier and level structure

The specifications in ISO/IEC 23090-5(2E):2023 Annex A.2 apply.

* 1. CodecGroup profile components

The specifications in ISO/IEC 23090-5(2E):2023 Annex A.3 apply.

* 1. Toolset profile components

The specifications in ISO/IEC 23090-5(2E):2023 Annex A.4 apply.

* 1. Reconstruction profile components

The specifications in ISO/IEC 23090-5(2E):2023 Annex A.5 apply.

* 1. Tiers and Levels

The specifications in ISO/IEC 23090-5(2E):2023 Annex A.6 apply.

* 1. Decoder instantiations

The specifications in ISO/IEC 23090-5(2E):2023 Annex A.7 apply.

1. (informative)  
     
   Post-decoding conversion to nominal video formats
   1. General

The specifications in ISO/IEC 23090-5(2E):2023 Annex B.1 apply with the following additions.

The basemesh frames provided by the decoder may require additional processing steps before being input to the reconstruction process. Such processing steps may include conversion of the basemesh data to a nominal format (e.g. a nominal bit depth, etc.), as described in subclause B.2, an alignment operation (e.g. output composition, atlas composition alignment, etc.), as described in subclause B.3.

The processes described in subclause B.2.7 are invoked for decoded basemesh frames associated with the same atlas ID, identified by variable ConvAtlasID, which is set equal to vuh\_atlas\_id or determined through external means if the V3C unit header is unavailable.

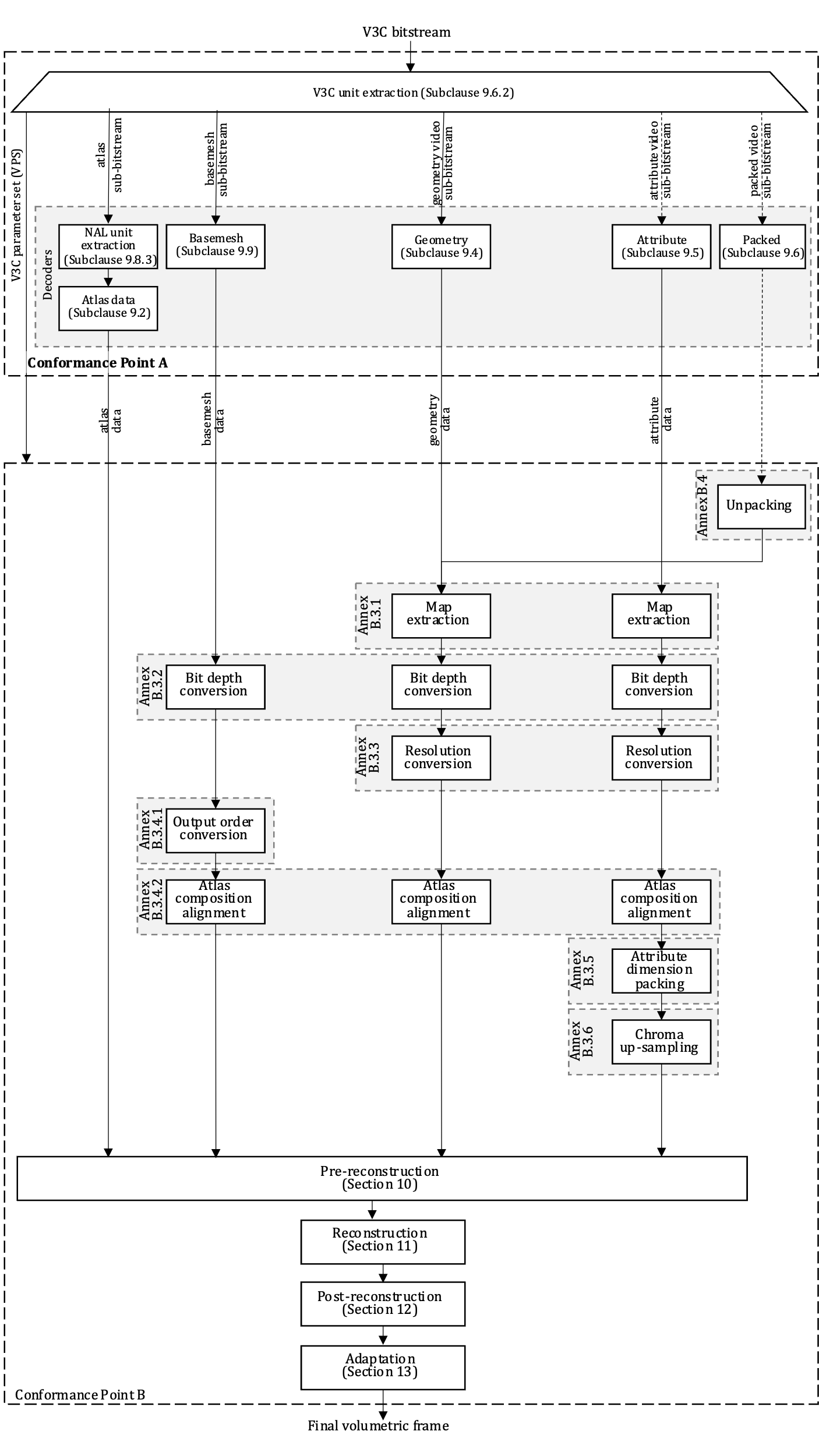


Figure B-1 – Post-processing conversion for a V3C bitstream with a single atlas

* 1. Nominal format conversion
     1. General

The specifications in ISO/IEC 23090-5(2E):2023 Annex B.2.1 applies with the following addition.

The nominal format refers collectively to the nominal bit depth and composition time index that the decoded basemesh frames and arithmetic coded displacement frames should be converted to.

The nominal frame resolution for non-auxiliary geometry video components is defined by the nominal width, VideoWidthNF, set equal to vps\_frame\_width[ ConvAtlasID ], and the nominal height, VideoHeightNF, set equal to vps\_frame\_height[ ConvAtlasID ]. The nominal frame resolution for non-auxiliary attribute video components is defined by the nominal width, VideoWidthAttNF[ attrIdx ], set equal to vps\_ext\_attribute\_frame\_width[ ConvAtlasID ][ attrIdx ], and the nominal height, VideoHeightAttNF[ attrIdx ], set equal to vps\_ext\_attribute\_frame\_height[ ConvAtlasID ][ attrIdx ]. VideoWidthAttNF[ attrIdx ] and VideoHeightAttNF[ attrIdx ] will be used instead of VideoWidthNF and VideoHeightNF for the frame resolution conversion of the attribute video with attribute index attrIdx.

Each basemesh sub-bitstream is associated with a nominal bit depth, which is the target bit depth that all operations for reconstruction are expected to be performed in. The nominal bit depth of a basemesh component for its vertex positions, VerCoordsBitDepthNF, is set equal to vps\_ext\_bmesh\_geometry\_3d\_bit\_depth\_minus1[ ConvAtlasID ] + 1, and for its attribute with index i, AttrBitDepthNF[ i ], is set equal to vps\_ext\_bmesh\_attribute\_bit\_depth\_minus1[ ConvAtlasID ][ i ] + 1.

Each arithmetic coded displacement sub-bitstream is associated with a nominal bit depth, which is the target bit depth that all operations for reconstruction are expected to be performed in. The nominal bit depth of a displacement component, DispBitDepthNF, is set equal to GeoBitDepthNF, that is equal to gi\_geometry\_2d\_bit\_depth\_minus1[ ConvAtlasID ] + 1.

The conversion processes to the nominal format are defined as follows:

* for the basemesh component subclause B.2.7 is invoked.
* for the arithmetic coded displacement component subclause B.2.8 is invoked.
  + 1. Occupancy nominal format conversion

The specifications in ISO/IEC 23090-5(2E):2023 Annex B.2.2 applies.

* + 1. Geometry nominal format conversion

The specifications in ISO/IEC 23090-5(2E):2023 Annex B.2.3 applies.

* + 1. Auxiliary geometry nominal format conversion

The specifications in ISO/IEC 23090-5(2E):2023 Annex B.2.4 applies.

* + 1. Attribute nominal format conversion

The specifications in ISO/IEC 23090-5(2E):2023 Annex B.2.5 applies.

* + 1. Auxiliary attribute nominal format conversion

The specifications in ISO/IEC 23090-5(2E):2023 Annex B.2.6 applies.

* + 1. Basemesh nominal format conversion

This process converts the decoded basemesh frames, DecBasemeshFrames, to the nominal format.

Let the variables basemeshNumComp, basemeshHeight, basemeshWidth, verNumComp and verMSBAlignFlag be set as follows:

basemeshNumComp = 1  
 basemeshHeight = 1  
 basemeshWidth = 1  
 verNumComp = 3  
 verMSBAlignFlag = vps\_ext\_bmesh\_geometry\_msb\_align\_flag[ ConvAtlasID ]

and the array attrMSBAlignFlag be set as follows:

for( i = 0; i < vps\_ext\_bmesh\_data\_attribute\_count[ ConvAtlasID ]; i++ ) {  
 attrMSBAlignFlag[ i ] = vps\_ext\_bmesh\_attribute\_msb\_align\_flag [ ConvAtlasID ][ i ]  
 }

Outputs of this process are:

* the 1D array SubmeshesPerFrameCountNF, indicating the number of submeshes, where the dimension corresponds to the basemesh composition time index, and
* the 2D array of structures BasemeshFramesNF, indicating the decoded basemesh frames in the nominal format, where the dimensions correspond to the basemesh composition time index and the submesh index, respectively. Each element of the BasemeshFramesNF has the following members:
  + submeshId, indicating the submesh ID,
  + verCoordCount, indicating the number of vertex coordinates,
  + verCoordBitDepth, indicating the vertex coordinate bit depth,
  + faceCount, indicating the number of vertex coordinates faces,
  + attributeCount, indicating the number of attributes, and
  + The following 1D arrays of size attributeCount:
    - attrTypeArray, indicating the type of attribute, according to Table 1,
    - attrValueDimensionArray, indicating the dimension of an attribute,
    - attrValueCountArray, indicating the number of values of an attribute,
    - attrValueBitDepthArray indicating the attribute value bit depth, and
  + 2D array verCoordValuesArray indicating vertex coordinate values, where the dimensions correspond to the vertex coordinate value index and the vertex coordinate value dimension index, respectively,
  + 2D array verCoordFacesArray, indicating vertex coordinates connectivity, where the dimensions correspond to the vertex coordinates face index and the vertex coordinates face dimension index respectively,
  + a 3D array attrValuesArray, indicating attribute values, where the dimensions correspond to the attribute index, the attribute value index, and the attribute value dimension index, respectively,
  + a 3D array attrFacesArray, indicating attribute connectivity, where the dimensions correspond to the attribute index, the attribute face index, and the attribute face dimension index, respectively,

To convert the decoded basemesh to the nominal format, several processing steps may need to be performed depending on the original format of the decoded basemesh. In particular:

* For each frame index frameIdx, for each submesh index smIdx, consider the following conversions:
* A temporary struct DecTriangularBasemeshFrame can be derived from the struct DecBasemeshFrames[ frameIdx ][ smIdx ] by transforming the linear index structure for the connectivities to an array with triangle connectivities and by selecting the attributes as indicated in the VPS structure in the following way:

DecTriangularBasemeshFrame.submeshId =  
 DecBasemeshFrames[ frameIdx ][ smIdx ].submeshId DecTriangularBasemeshFrame.verCoordCount =  
 DecBasemeshFrames[ frameIdx ][ smIdx ].verCoordCount  
 DecTriangularBasemeshFrame.verCoordBitDepth =  
 DecBasemeshFrames[ frameIdx ][ smIdx ].verCoordBitDepth  
 DecTriangularBasemeshFrame.faceCount =  
 DecBasemeshFrames[ frameIdx ][ smIdx ].faceCount   
 DecTriangularBasemeshFrame.verCoordsArray =   
 DecBasemeshFrames[ frameIdx ][ smIdx ].verCoordsArray  
 faceCount = DecBasemeshFrames[ frameIdx ][ smIdx ].faceCount  
 for( i = 0; i < faceCount; i++ ){  
 for( k = 0; k < 3; k++ ){  
 DecTriangularBasemeshFrame.verCoordFacesArray[ i ][ k ] =  
 DecBasemeshFrames[ frameIdx ][ smIdx ].verCoordConnArray[ i ][ k ]  
 }  
 }  
 DecTriangularBasemeshFrames.attributeCount =  
 vps\_ext\_bmesh\_data\_attribute\_count[ ConvAtlasID ]  
 for( i = 0; i < DecTriangularBasemeshFrames.attributeCount; i++ ){  
 refIdx = vps\_ext\_bmesh\_attribute\_index[ ConvAtlasID ][ i ]  
 DecTriangularBasemeshFrame.attrTypeArray[ i ] =  
 vps\_ext\_bmesh\_attribute\_type[ ConvAtlasID ][ i ]  
 DecTriangularBasemeshFrame.attrValueDimensionArray[ i ] =  
 DecBasemeshFrames[ frameIdx ][ smIdx ].attrValueDimensionArray[ refIdx ]  
 DecTriangularBasemeshFrame.attrValueCountArray[ i ] =  
 DecBasemeshFrames[ frameIdx ][ smIdx ].attrValueCountArray[ refIdx ]  
 DecTriangularBasemeshFrame.attrValueBitDepthArray[ i ] =  
 DecBasemeshFrames[ frameIdx ][ smIdx ].attrValueBitDepthArray[ refIdx ]   
 DecTriangularBasemeshFrame.attrValuesArray[ i ] =  
 DecBasemeshFrames[ frameIdx ][ smIdx ].attrValuesArray[ refIdx ]  
 apv =  
 DecBasemeshFrames[ frameIdx ][ smIdx ].attrValueIsPerVertexArray[ refIdx ] )  
 attrConn = DecBasemeshFrames[ frameIdx ][ smIdx ].attrConnArray[ refIdx ]  
 for( j = 0; j < DecBasemeshFrames[ frameIdx ][ smIdx ].faceCount; j++ ) {  
 for( k = 0; k < 3; k++ ) {  
 if( apv )  
 DecTriangularBasemeshFrame.attrFacesArray[ i ][ j ][ k ] =  
 attrConn[ j ][ k ]  
 else  
 DecTriangularBasemeshFrame.attrFacesArray[ i ][ j ][ k ] =  
 attrConn[ j ][ 0 ]  
 }  
 }  
 }

Next, the temporary struct DecTriangularBasemeshFrame is stored in the struct BasemeshFramesNBD[ frameIdx ][ smIdx ] for further processing.

* To convert the bit depth of the decoded vertex coordinates of the basemesh frames to the required nominal bit depth, subclause ISO/IEC 23090-5(2E):2023 Annex B.3.2 is invoked with variables BasemeshFramesNBD[ frameIdx ][ smIdx ].verCoordBitDepth, BasemeshFramesNBD[ frameIdx ][ smIdx ].verCoordCount, verNumComp, VerCoordsBitDepthNF, and vertMSBAlignFlag, and array BasemeshFramesNBD[ frameIdx ][ smIdx ].verCoordsArray as inputs. The output of this process is the decoded vertex coordinate of the basemesh frame at the nominal bit depth, BasemeshFramesNBD[ frameIdx ][ smIdx ].verCoordArray. BasemeshFameNBD[ frameIdx ][ smIdx ].verCoordBitDepth is also updated to VerCoordsBitDepthNF.
* To convert the bit depth of the decoded attributes of the basemesh frames to the required nominal bit depth, subclause ISO/IEC 23090-5(2E):2023 Annex B.3.2 is invoked for each attribute index attrIdx with variables BasemeshFameNBD[ frameIdx ][ smIdx ].attrValueBitDepthArray[ attrIdx ], BasemeshFameNBD[ frameIdx ][ smIdx ].attrValueCountArray[ attrIdx ], BasemeshFameNBD[ frameIdx ][ smIdx ].attrValueDimensionArray[ attrIdx ], BasemeshFameNBD[ frameIdx ][ smIdx ].attrValueArray[ attrIdx ], AttrBitDepthNF[ attrIdx ], and attrMSBAlignFlag[ attrIdx ], as inputs. The output of this process is the decoded attributes of the basemesh frame at the nominal bit depth, BasemeshFameNBD[ frameIdx ][ smIdx ].attrValueArray[ attrIdx ]. The variable BasemeshFameNBD[ frameIdx ][ smIdx ].DecAttrValueBitDepth[ attrIdx ] is also updated to AttrBitDepthNF[ attrIdx ].
* For the frame alignment operation, to use the functions defined in subclause ISO/IEC 23090-5(2E):2023 Annex B used for video conversion. The input to the function needs to be a 4D array where the dimensions indicate the frame index, the component index, the index for the height and the index for the width of the image respectively, that is, a 4D array that has the following structure: INPUT\_ARRAY[ frameIdx ][ compIdx ][ y ][ x ], where compIdx is the range of 0 to numComp, y in the range of 0 to height, and x in the range of 0 to width. Therefore, the function can be used for aligning frames of a basemesh structure, if we perform a simple conversion that assumes that each decoded basemesh frame is an element of a 4D array, which contains just a single element, that is, the variables numComp, height and width are all equal to 1.
* For each submesh index smIdx, consider the following conversions:

for( frIdx = 0; frIdx < DecBasemeshFrameCount; frIdx++)  
 submeshFrames[ frIdx ][ 0 ][ 0 ][ 0 ] = BasemeshFramesNBD[ frIdx ][ smIdx ]

* To identify missing frames in the sequence and re-order the input, subclause ISO/IEC 23090-5(2E):2023 Annex B.3.4.1 is invoked with variables DecBasemeshFrameCount, numComp, height, and width, and arrays submeshFrames, DecBasemeshOutOrdIdx, and DecBasemeshCompTime as inputs. The outputs of this process are the number of frames being output, numOutOrdBasemeshFrames, the decoded basemesh frames ordered according to the output order index, submeshFramesONF, a 1D array, framePresent, indicating the presence of a frame, and a 1D array, decBasemeshCompTimeONF, indicating the composition time of each frame ordered according to the output order index.
* To align the composition times of the decoded basemesh frames in output order and the decoded atlas frames, subclause ISO/IEC 23090-5(2E):2023 Annex B.3.4.2 is invoked with variables NumDecAtlasFrames, numOutOrdBasemeshFrames, numComp, height, and width, and arrays DecAtlasOutOrdIdx, DecAtlasCompTime, submeshFramesONF, decBasemeshCompTimeONF, and framePresent as inputs. The outputs of this process are the number of frames being output, rangeCompTimeIdx, the output composition time of each frame, basemeshCompTime, and a sequence of frames in the nominal format aligned with the decoded atlas composition time, submeshFramesNF.

for( compTimeIdx = 0; compTimeIdx < rangeCompTimeIdx; compTimeIdx++)  
 BasemeshFramesNF[ compTimeIdx ][ smIdx ] =  
 submeshFramesNF[ compTimeIdx ][ 0 ][ 0 ][ 0 ]

* For the 2D array DecSubmeshesPerFrameCount, consider the following conversions:

for( frIdx = 0; frIdx < DecBasemeshFrameCount; frIdx++)  
 decSubmeshesPerFrameCount[ frIdx ][ 0 ][ 0 ][ 0 ] =  
 DecSubmeshesPerFrameCount[ frIdx ]

* To identify missing frames in the sequence and re-order the input, subclause ISO/IEC 23090-5(2E):2023 Annex B.3.4.1 is invoked with variables DecBasemeshFrameCount, numComp, height, and width, and arrays decSubmeshesPerFrameCount, DecBasemeshOutOrdIdx, and DecBasemeshCompTime as inputs. The outputs of this process are the number of frames being output, numOutOrdBasemeshFrames, the decoded number of submeshes per frame ordered according to the output order index, decSubmeshesPerFrameCountONF, a 1D array, framePresent, indicating the presence of a frame, and a 1D array, decBasemeshCompTimeONF, indicating the composition time of each frame ordered according to the output order index.
* To align the composition times of the decoded number of submeshes per frame in output order and the decoded atlas frames, subclause ISO/IEC 23090-5(2E):2023 Annex B.3.4.2 is invoked with variables NumDecAtlasFrames, numOutOrdBasemeshFrames, numComp, height, and width, and arrays DecAtlasOutOrdIdx, DecAtlasCompTime, decSubmeshesPerFrameCountONF, decBasemeshCompTimeONF, and framePresent as inputs. The outputs of this process are the number of frames being output, rangeCompTimeIdx, the output composition time of each frame, basemeshCompTime, and a sequence of number of submeshes per frame in the nominal format aligned with the decoded atlas composition time, submeshesPerFrameCountNF. The additional conversion from 4D array to a 2D array can be performed:

for( compTimeIdx = 0; compTimeIdx < rangeCompTimeIdx; compTimeIdx++)  
 SubmeshesPerFrameCountNF[ compTimeIdx ] =   
 submeshesPerFrameCountNF[ compTimeIdx ][ 0 ][ 0 ][ 0 ]

* + 1. Arithmetic coded displacement nominal format conversion

This process converts the decoded displacement frames, DecDispFrames, to the nominal format.

Let the variables dispNumComp, dispHeight, dispWidth, vertNumComp and vertMSBAlignFlag be set as follows:

dispNumComp = DispDimension  
 dispHeight = 1  
 dispWidth = DispCountPerFrame[ frameIdx ]  
 vertNumComp = 3  
 vertMSBAlignFlag = dsps\_msb\_align\_flag

Output of this process is:

* the 2D array of structures DispFramesNF, indicating the decoded displacement frames in the nominal format, where the dimensions correspond to the displacement composition time index.

To convert the decoded displacement to the nominal format, several processing steps may need to be performed depending on the original format of the decoded displacement. In particular:

* Initialize decDispFramesNBD[ frameIdx ] with DecDispFrames[ frameIdx ].
* To convert the bit depth of the decoded displacement frames to the required nominal bit depth, subclause B.3 is invoked for each frame index frameIdx, with variables DecDispBitDepth, totalVertCount, vertNumComp, VerCoordsBitDepthNF, and vertMSBAlignFlag of the decoded displacement frame DecDispFrames[ frameIdx ], and array DecDispFrame of the DecDispFrames[ frameIdx ] as inputs. The output of this process is the displacement frame at the nominal bit depth, DecDispFrame of the decDispFramesNBD[ frameIdx ]. The variable DecDispBitDepth is also updated to VerCoordsBitDepthNF.
* To identify missing frames in the sequence and re-order the input, subclause B.3 is invoked with variables DecDispFrameCount, dispNumComp, dispHeight, and dispWidth, and arrays decDispFramesNBD, DecDispOutOrdIdx, and DecDispCompTime as inputs. The outputs of this process are the number of frames being output, numOutOrdDispFrames, the decoded displacement frames ordered according to the output order index, decDispFramesONF, a 1D array, framePresent, indicating the presence of a frame, and a 1D array, decDispCompTimeONF, indicating the composition time of each frame ordered according to the output order index.
* To align the composition times of the decoded displacement frames in output order and the decoded atlas frames, subclause B.3 is invoked with variables NumDecAtlasFrames, numOutOrdDispFrames, dispNumComp, dispHeight, and dispWidth, and arrays DecAtlasOutOrdIdx, DecAtlasCompTime, decDispFramesONF, decDispCompTimeONF, and framePresent as inputs. The outputs of this process are the number of frames being output, rangeCompTimeIdx, the output composition time of each frame, dispCompTime, and a sequence of frames in the nominal format aligned with the decoded atlas composition time, DispFramesNF.
  1. Conversion operations

The specifications in ISO/IEC 23090-5(2E):2023 Annex B.3 applies.

* 1. Unpacking process of a decoded packed video

The specifications in ISO/IEC 23090-5(2E):2023 Annex B.4 applies.

1. (informative)  
     
   V3C sample stream format

Annex C of ISO/IEC 23090-5(2E):2023 applies.

1. (normative)  
     
   NAL sample stream format

Annex C of ISO/IEC 23090-5(2E):2023 applies.

1. (normative)  
     
   Atlas hypothetical reference decoder

Annex E of ISO/IEC 23090-5(2E):2023 applies.

1. (normative)  
     
   Supplemental enhancement information
   1. General

The specifications in ISO/IEC 23090-5(2E):2023 Annex F.1 apply with the following additions to Table F-1:

Table F-1 – The essential and non-essential SEI messages

|  |  |  |
| --- | --- | --- |
| **SEI message** | **NAL Type** | **Conformance Type** |
| Zippering | NAL\_PREFIX\_ESEI | Type-B |
| Submesh SOI relationship indication | NAL\_PREFIX\_NSEI/  NAL\_SUFFIX\_NSEI | N/A |
| Submesh distortion indication | NAL\_PREFIX\_NSEI/  NAL\_SUFFIX\_NSEI | N/A |

* 1. SEI payload syntax

The specifications in ISO/IEC 23090-5(2E):2023 Annex F.2 and its subclauses apply, with the following additions:

* + 1. V-DMC registered SEI message syntax

|  |  |
| --- | --- |
| vdmc\_registered\_sei\_message( payloadSize ) { | **Descriptor** |
| vdmcPayloadType = 0 |  |
| headerSize = 0 |  |
| do { |  |
| **vrsm\_payload\_type\_byte** | u(8) |
| vmdcPayloadType += vrsm\_payload\_type\_byte |  |
| headerSize += 1 |  |
| } while( vrsm\_payload\_type\_byte == 0xFF ) |  |
| vdmcPayloadSize = payloadSize - headerSize |  |
| vdmc\_registered\_sei\_payload( vdmcPayloadType, vdmcPayloadSize ) |  |
| } |  |

* + 1. V-DMC registered SEI message syntax

|  |  |
| --- | --- |
| vdmc\_registered\_sei\_payload( vdmcPayloadType, vdmcPayloadSize ) { | **Descriptor** |
| if( vdmcPayloadType == 0 ) |  |
| zippering( vdmcPayloadSize ) |  |
| if( vdmcPayloadType == 1 ) |  |
| submesh\_soi\_relationship\_indication( vdmcPayloadSize ) |  |
| if( vdmcPayloadType == 2 ) |  |
| submesh\_distortion\_indication( vdmcPayloadSize ) |  |
| else |  |
| reserved\_message( vdmcPayloadSize ) |  |
| } |  |

* + 1. Zippering SEI payload syntax

|  |  |
| --- | --- |
| zippering( payloadSize ) { | **Descriptor** |
| **zp\_persistence\_flag** | u(1) |
| **zp\_reset\_flag** | u(1) |
| **zp\_instances\_updated** | u(8) |
| for( i = 0; i < zp\_instances\_updated; i++ ) { |  |
| **zp\_instance\_index**[ i ] | u(8) |
| k = zp\_instance\_index[ i ] |  |
| **zp\_instance\_cancel\_flag**[ k ] | u(1) |
| if( zp\_instance\_cancel\_flag[ k ] != 1 ) { |  |
| **zp\_method\_type**[ k ] | ue(v) |
| if( zp\_method\_type[ k ] == 1 ) { |  |
| **zp\_max\_match\_distance**[ k ] | ue(v) |
| if( zp\_max\_match\_distance\_per\_frame[ k ] != 0 ) { |  |
| **zp\_send\_distance\_per\_submesh**[ k ] | u(1) |
| if( zp\_send\_distance\_per\_submesh[ k ] ) { |  |
| **zp\_number\_of\_submeshes\_minus1**[ k ] | ue(v) |
| numSubmeshes = zp\_number\_of\_submeshes\_minus1[ k ] + 1 |  |
| for( p = 0; p < numSubmeshes ; p++ ) { |  |
| **zp\_max\_match\_distance\_per\_submesh**[ k ][ p ] | u(v) |
| if( zp\_max\_match\_distance\_per\_submesh[ k ][ p ] != 0 ) { |  |
| **zp\_send\_distance\_per\_border\_point**[ k ][ p ] | u(1) |
| if( zp\_send\_distance\_per\_border\_point[ k ][ p ] == 1 ) { |  |
| **zp\_number\_of\_border\_points**[ k ][ p ] | ue(v) |
| numBorderPoints =  zp\_number\_of\_border\_points[ k ][ p ] |  |
| for( b = 0; b < numBorderPoints ; b++ ) |  |
| **zp\_border\_point\_distance**[ k ][ p ][ b ] | u(v) |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |
| if( zp\_method\_type[ k ] == 2 ) { |  |
| **zp\_unmatched\_lod\_vertices\_method\_type**[ k ] | ue(v) |
| **zp\_number\_of\_submeshes\_minus1**[ k ] | ue(v) |
| numSubmeshes = zp\_number\_of\_submeshes\_minus1[ k ] + 1 | ue(v) |
| for( p = 0; p < numSubmeshes; p++ ) |  |
| **zp\_number\_of\_border\_points**[ k ][ p ] | ue(v) |
| for( p = 0; p < numSubmeshes; p++ ) { |  |
| numBorderPoints = zp\_number\_of\_border\_points[ k ][ p ] |  |
| for( b = 0; b < numBorderPoints ; b++ ) { |  |
| if( zipperingBorderPointMatchIndexFlag[ k ][ p ][ b ] == 0) { |  |
| **zp\_border\_point\_match\_submesh\_index**[ k ][ p ][ b ] | u(v) |
| submeshIndex =  zp\_border\_point\_match\_submesh\_index[ k ][ p ][ b ] |  |
| if( submeshIndex != numSubmeshes ) { |  |
| **zp\_border\_point\_match\_border\_point\_index**[ k ][ p ][ b ] | u(v) |
| borderIndex =  zp\_border\_point\_match\_border\_point\_index[ k ][ p ][ b ] |  |
| if( submeshIndex > p) |  |
| zipperingBorderPointMatchIndexFlag[ k ][ submeshIndex ][ borderIndex ] =1 |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |

* + 1. Submesh SOI relationship indication SEI payload syntax

|  |  |
| --- | --- |
| submesh\_soi\_relationship\_indication( payloadSize ) { | **Descriptor** |
| **ssr\_persistence\_association\_flag** | u(1) |
| **ssr\_number\_of\_active\_scene\_objects** | ue(v) |
| **ssr\_submesh\_id\_length\_minus1** | ue(v) |
| for( i = 0; i < ssr\_number\_of\_active\_scene\_object; i++ ) { |  |
| **ssr\_soi\_object\_idx**[ i ] | u(v) |
| **ssr\_number\_of\_submesh\_included**[ i ] | ue(v) |
| for( j = 0; j < ssr\_number\_of\_submesh\_included[ i ]; j++ ) { |  |
| **ssr\_submesh\_id**[ i ][ j ] | u(v) |
| **ssr\_completely\_included**[ i ][ j ] | u(1) |
| } |  |
| } |  |
| } |  |

* + 1. Submesh distortion indication SEI payload syntax

|  |  |
| --- | --- |
| submesh\_distortion\_indication( payloadSize ) { | **Descriptor** |
| **sdi\_number\_of\_submesh\_indicated** | ue(v) |
| **sdi\_submesh\_id\_length\_minus1** | ue(v) |
| for( i = 0; i < sdi\_number\_of\_submesh\_indicated; i++ ) { |  |
| **sdi\_submesh\_id**[ i ] | u(v) |
| **sdi\_number\_of\_vertices\_of\_original\_submesh**[ i ] | ue(v) |
| **sdi\_subdivision\_iteration\_count**[ i ] | ue(v) |
| **sdi\_number\_of\_distortion\_indicated\_minus1**[ i ] | ue(v) |
| for( j = 0; j < sdi\_number\_of\_distortion\_indicated\_minus1 + 1; j++ ) { |  |
| **sdi\_distortion\_metrics\_type**[ i ][ j ] | u(8) |
| for( k = 0; j < sdi\_subdivision\_iteration\_count; k++ ) |  |
| **sdi\_distortion**[ i ][ j ][ k ] | ue(v) |
| } |  |
| } |  |
| } |  |

* 1. SEI payload semantics

The specifications in ISO/IEC 23090-5(2E):2023 Annex F.3 and its subclauses apply, with the following additions:

* + 1. V-DMC registered SEI message semantics

Each V-DMC registered SEI message consists of the variable specifying the type vdmcPayloadType. The derived SEI message payload size vdmcPayloadSize is specified in bytes and shall be equal to the number of bytes in the V-DMC registered SEI message payload.

**vrsm\_payload\_type\_byte** is a byte of the payload type of an SEI message.

* + 1. V-DMC registered SEI payload semantics

The V-DMC registered SEI payload provides a mechanism to include SEI messages that are specific to V-DMC toolsets.

* + 1. Zippering SEI payload semantics

This SEI message specifies the recommended zippering methods and their associated parameters that could be used to process the vertices of the current mesh frame after it is reconstructed, so as to obtain improved reconstructed geometry quality.

Up to 256 different zippering instances could be specified for use with each mesh frame. These instances are indicated using an array ZipperingMethod. The zippering instance that a decoder may select to operate in, is outside the scope of this document.

At the start of each sequence, let ZipperingMethod[ i ] be set equal to 0, where i corresponds to the zippering instance index and is in the range of 0 to 255, inclusive. When ZipperingMethod[ i ] is equal to 0 it means that no zippering filter is indicated for the zippering instance with index i.

**zp\_persistence\_flag** specifies the persistence of the zippering SEI message for the current layer. zp\_persistence\_flag equal to 0 specifies that the zippering SEI message applies to the current decoded atlas frame only.

Let aFrmA be the current atlas frame. zp\_persistence\_flag equal to 1 specifies that the zippering SEI message persists for the current layer in output order until any of the following conditions are true:

* A new CAS begins.
* The bitstream ends.
* An atlas frame aFrmB in the current layer in a coded atlas access unit containing a zippering SEI message with the same value of zp\_persistence\_flag and applicable to the current layer is output for which AtlasFrmOrderCnt( aFrmB ) is greater than AtlasFrmOrderCnt( aFrmA ), where AtlasFrmOrderCnt( aFrmB ) and AtlasFrmOrderCnt( aFrmA ) are the AtlasFrmOrderCntVal values of aFrmB and aFrmA, respectively, immediately after the invocation of the decoding process for atlas frame order count for aFrmB.

**zp\_reset\_flag** equal to 1 resets all entries in the array ZipperingMethod to 0 and all parameters associated with this SEI message are set to their default values.

**zp\_instances\_updated** specifies the number of zippering instances that will be updated in the current zippering SEI message.

**zp\_instance\_index**[ i ] indicates the i-th zippering instance index in the array ZipperingMethod that is to be updated by the current SEI message.

**zp\_instance\_cancel\_flag**[ k ]equal to 1 indicates that the value of ZipperingMethod[ k ] and that all parameters associated with the zippering instance with index k should be set to 0 and to their default values, respectively.

**zp\_method\_type**[ k ]indicates the zippering method, ZipperingMethod[ k ], that can be used for processing the current mesh frame as specified in **Table F-2** for zippering instance with index k.

**Table F-2 – Definition of zp\_method\_type[ k ]**

|  |  |
| --- | --- |
| **Value** | **Interpretation** |
| 0 | No zippering |
| 1 | Distance Zippering |
| 2 | Border Point Match Zippering |
| 3- | Reserved |

Values of zp\_method\_type[ k ] greater than 2 are reserved for future use by ISO/IEC. It is a requirement of bitstream conformance that bitstreams conforming to this version of this document shall not contain such values of zp\_method\_type[ k ]. Decoders shall ignore zippering SEI messages that contain reserved values of zp\_method\_type[ k ]. The default value of zp\_method\_type[ k ] is equal to 0.

**zp\_max\_match\_distance**[ k ]specifies the value of the variable zipperingMaxMatchDistance[ k ] used for processing the current mesh frame for zippering instance with index k when the zippering filtering process is used.

**zp\_send\_distance\_per\_submesh**[ k ]equal to 1 specifies that zippering by transmitting matching distance per submesh is applied to border points for the zippering instance with index k. zp\_send\_distance\_per\_submesh[ k ] equal to 0 specifies that zippering by matching distance per submesh is not applied to border points for the zippering instance with index k. The default value of zp\_send\_distance\_per\_submesh[ k ] is equal to 0.

**zp\_number\_of\_submeshes\_minus1**[ k ] plus 1indicates the number of submeshes that are to be zippered by the current SEI message. The value of zp\_number\_of\_submeshes\_minus1 shall be in the range from 0 to MaxNumSubmeshes[ frameIdx ] − 1, inclusive. The default value of zp\_number\_of\_submeshes\_minus1 is equal to 0

**zp\_max\_match\_distance\_per\_submesh**[ k ][ p ]specifies the value of the variable zipperingMaxMatchDistancePerPatch[ k ][ p ] used for processing the current submesh with index p in the current mesh frame for zippering instance with index k when the zippering process is used. The length of the zp\_max\_match\_distance\_per\_submesh[ k ][ p ] syntax element is Ceil( Log2( zp\_max\_match\_distance[ k ] ) ) bits.

**zp\_send\_distance\_per\_border\_point**[ k ]equal to 1 specifies that zippering by transmitting matching distance per border point is applied to border points for the zippering instance with index k. zp\_send\_distance\_per\_border\_point [ k ] equal to 0 specifies that zippering by matching distance per border point is not applied to border points for the zippering instance with index k. The default value of zp\_send\_distance\_per\_border\_point[ k ] is equal to 0.

**zp\_number\_of\_border\_points**[ k ][ p ] indicates the number of border points numBorderPoints[ p ] of a submesh with index p, in the current mesh frame for zippering instance with index k when the zippering process is used.

**zp\_border\_point\_distance**[ k ][ p ][ b ]specifies the value of the variable zipperingMaxMatchDistancePerBorderPoint[ k ][ p ][ b ] used for processing the current border point with index b, in the current submesh with index p, in the current mesh frame for zippering instance with index k when the zippering process is used**.** The length of the zp\_border\_point\_distance[ k ][ p ][ b ] syntax element is Ceil( Log2( zp\_max\_match\_distance\_per\_submesh[ k ][ p ] ) ) bits.

**zp\_unmatched\_lod\_vertices\_method\_type**[ k ]indicates the identifier of the rectification method that can be used for processing the vertices generated by unmatched LODs in the current mesh frame as specified in **Table F-3** for zippering instance with index k.

**Table F-3 – Rectification methods for unmatched LODs vertices**

|  |  |
| --- | --- |
| **zp\_unmatched\_lod\_vertices\_method\_type**[k] | **Method** |
| 0 | No rectification |
| 1 | Remove vertices |
| 2 | Insert new vertices |
| 3- | Reserved |

Values of zp\_unmatched\_lod\_vertices\_method\_type[k] greater than 2 are reserved for future use by ISO/IEC. It is a requirement of bitstream conformance that bitstreams conforming to this version of this document shall not contain such values of zp\_unmatched\_lod\_vertices\_method\_type[k]. Decoders shall ignore zippering SEI messages that contain reserved values of zp\_unmatched\_lod\_vertices\_method\_type[k]. The default value of zp\_unmatched\_lod\_vertices\_method\_type[k] is equal to 0.

**zp\_border\_point\_match\_submesh\_index**[ k ][ p ][ b ]specifies the value of the variable zipperingBorderPointMatchSubmeshIndex[ k ][ p ][ b ] used for processing the current border point with index b, in the current submesh with index p, in the current mesh frame for zippering instance with index k when the zippering process is used. The length of the zp\_border\_point\_match\_submesh\_index[ k ][ p ][ b ] syntax element is Ceil( Log2( zp\_number\_of\_submeshes\_minus1[ k ] + 2 ) ) bits.

**zp\_border\_point\_match\_border\_point\_index**[ k ][ p ][ b ] specifies the value of the variable zipperingBorderPointMatchBorderPointIndex[ k ][ p ][ b ] used for processing the current border point with index b, in the current patch with index p, in the current mesh frame for zippering instance with index k when the zippering filtering process is used The length of the zp\_border\_point\_match\_border\_point\_index[ k ][ p ][ b ] syntax element is Ceil( Log2( zp\_number\_of\_border\_points[ k ][ zp\_border\_point\_match\_submesh\_index[ k ][ p ][ b ] ] ) ) bits.

* + 1. Submesh SOI relationship indication SEI message semantics

This SEI message indicates the relationship between scene objects and submesh. One or more submesh can be associated to each scene objects and a submesh can be associated with more than one scene objects.

**ssr\_persistence\_association\_flag** indicates the relationship between the scene objects and submeshes are persistent. When the value of this flag is ‘0’ the relationship is only valid for the current frame.

**ssr\_number\_of\_active\_scene\_object** indicates the number of active scene object defined for a mesh at the time of SEI message is signalled.

**ssr\_submesh\_id\_length\_minus1** plus 1 specifies the number of bits used to represent the syntax element ssr\_submesh\_id[ i ][ j ].

**ssr\_soi\_object\_idx**[ i ] indicates the index of the i-th scene object to be associated with the submeshes.

**ssr\_number\_of\_submesh\_included**[ i ] indicates the number of submesh fully included in the i-th scene object.

**ssr\_submesh\_id**[ i ][ j ] indicates the identifier of a j-th submesh included in the i-th scene object. The number of bits used to represent ssr\_submesh\_id[ i ][ j ] is ssr\_submesh\_id\_length\_minus1 + 1.

**ssr\_completely\_included**[ i ][ j ] indicates the j-th submesh is completely included in the i-th scene object.

* + 1. Submesh distortion indication SEI message semantics

This SEI message indicates the number of vertices of original submesh and the similarity of the basemesh and original mesh at each subdivision iterations so that the decoder can estimate the loss of quality of decoded submesh. In some cases, reconstruction process can be stopped at certain number of iterations by using similarity information provided by this SEI message when estimated quality of reconstructed submesh is sufficient for intended use.

**sdi\_number\_of\_submesh\_indicated** indicates the number of submesh similarity information signalled by this SEI message.

**sdi\_submesh\_id\_length\_minus1** plus 1 specifies the number of bits used to represent the syntax element sdi\_submesh\_id[ i ].

**sdi\_submesh\_id**[ i ] indicates the identifier of the i-th submesh. The number of bits used to represent sdi\_submesh\_id[ i ] is sdi\_submesh\_id\_length\_minus1 + 1.

**sdi\_number\_of\_vertices\_of\_original\_submesh**[ i ] indicates the number of vertices of original submesh

**sdi\_subdivision\_iteration\_count**[ i ] indicates the number of subdivision iteration to be applied to generate reconstructed basemesh .

**sdi\_number\_of\_distortion\_indicated\_minus1**[ i ] plus 1 indicates the number of distortion associated with i-th submesh signalled by this SEI message.

**sdi\_distortion\_metrics\_type**[ i ][ j ] indicates the type of distortion metric for the j-th distortion of the i-th submesh. Available types are listed in **Table F-4**.

**Table F-4 – Metric distortion**

|  |  |
| --- | --- |
| **value** | **metric** |
| 0x00 | reserved |
| 0x01 | point cloud based D1 |
| 0x02 | point cloud based D2 |
| 0x03 | point cloud based PSNR |
| 0x04 | rendered image based PSNR |
| 0x05 | perceptual based distortion |
| 0x06 – 0xFF | reserved |

**sdi\_distortion**[ i ][ j ][ k ] indicates the j-th distortion metric between original mesh and reconstructed j-th submesh after k-th subdivision iteration is done.

1. (informative)  
     
   Volumetric usability information
   1. General

The specifications in ISO/IEC 23090-5(2E):2023 Annex G.1 apply.

* 1. VUI syntax
     1. VUI parameters syntax

The specifications in ISO/IEC 23090-5(2E):2023 Annex G.2.1 apply.

* + 1. HRD parameters syntax

The specifications in ISO/IEC 23090-5(2E):2023 Annex G.2.2 apply

* + 1. Sub-layer HRD parameters syntax

The specifications in ISO/IEC 23090-5(2E):2023 Annex G.2.3 apply.

* + 1. Maximum coded video resolution syntax

The specifications in ISO/IEC 23090-5(2E):2023 Annex G.2.4 apply.

* + 1. Coordinate system parameters syntax

The specifications in ISO/IEC 23090-5(2E):2023 Annex G.2.5 apply.

* + 1. V-DMC VUI extension

|  |  |
| --- | --- |
| vdmc\_vui\_parameters( ) { | **Descriptor** |
| for( i = 0; i < asve\_num\_attribute\_video; i++ ) |  |
| **vdmc\_vui\_one\_submesh\_per\_independent\_unit\_attribute\_flag**[ i ] | u(1) |
| **vdmc\_vui\_one\_submesh\_per\_independent\_unit\_geometry\_flag** | u(1) |
| } |  |

* 1. VUI semantics
     1. VUI parameters semantics

The specifications in ISO/IEC 23090-5(2E):2023 Annex G.3.1 apply.

* + 1. HRD parameters semantics

The specifications in ISO/IEC 23090-5(2E):2023 Annex G.3.2 apply.

* + 1. HRD sub-layer parameters semantics

The specifications in ISO/IEC 23090-5(2E):2023 Annex G.3.3 apply.

* + 1. Maximum coded video resolution semantics

The specifications in ISO/IEC 23090-5(2E):2023 Annex G.3.4 apply.

* + 1. Coordinate system parameters semantics

The specifications in ISO/IEC 23090-5(2E):2023 Annex G.3.5 apply.

* + 1. V-DMC VUI extension semantics

**vdmc\_vui\_one\_submesh\_per\_independent\_unit\_attribute\_flag**[ i ] equal to 1 indicates that the attribute video sub-bitstream with index i contains independently decodable units and each independently decodable unit contains coded data from at most one submesh. vdmc\_vui\_one\_submesh\_per\_independent\_unit\_attribute\_flag[ i ] equal to 0 indicates that the attribute video sub-bitstream with index i does not contain independently decodable units or that each independently decodable unit in the attribute video sub-bitstream may contain coded data corresponding to more than one submesh.

**vdmc\_vui\_one\_submesh\_per\_independent\_unit\_geometry\_**flag equal to 1 indicates that the geometry sub-bitstream contains independently decodable units and each independently decodable unit contains coded data from at most one submesh. vdmc\_vui\_one\_submesh\_per\_independent\_unit\_geometry\_flag equal to 0 indicates that the geometry sub-bitstream does not contain independently decodable units or that each independently decodable unit in the geometry sub-bitstream may contain coded data corresponding to more than one submesh.

1. (normative)  
     
   Basemesh sub-bistream
   1. Scope

This annex specifies the basemesh sub-bitstream. In this Annex, a bitstream refers to a basemesh sub-bitstream unless otherwise indicated.

* 1. Normative references

The list of normative references in Clause 2 applies.

* 1. Terms and definitions

For the purpose of this annex, the following definitions apply in addition to the definitions in Clause 3. These definitions are either not present in Clause 3or replace definitions in Clause 3.

coded basemesh access unit

set of basemesh NAL units that are associated with each other according to a specified classification rule, are consecutive in decoding order, and contain all basemesh NAL units pertaining to one particular output time



submesh

* 1. Abbreviated terms

The specifications in Clause 4 apply.

BMCL BaseMesh Coding Layer

BMFPS BaseMesh Frame Parameter Set

BMSPS BaseMesh Sequence Parameter Set

CBMB Coded BaseMesh Buffer

CBMS Coded BaseMesh Sequence

DBMB Decoded BaseMesh Buffer

* 1. Conventions

The specifications in Clause 5 apply.

* 1. Overall Basemesh Codec characteristics, decoding operations, and post-decoding processes
     1. Basemesh bitstream characteristics, decoding operations, and post-decoding processes

The decoded mesh frames may require the application of additional transformations, as described in [CrossReferenceXXX] before any reconstruction operations. The outputs [CrossReferenceXXX] are the following mesh struct in the nominal format: MeshNF.

* 1. Bitstream format
     1. NAL bitstream formats

This subclause specifies the relationship between the network abstraction layer (NAL) unit stream and the NAL sample stream, either of which are referred to as the NAL bitstream.

The bitstream can be in one of two formats: the NAL unit stream format or the sample stream format. The NAL unit stream format is conceptually the more "basic" type. It consists of a sequence of syntax structures called NAL units. This sequence is ordered in decoding order, as described in subclause H.8.4.2.3. There are constraints imposed on the decoding order (and contents) of the NAL units in the NAL unit stream.

NOTE – The NAL unit stream format is commonly not intended to be used in any applications on its own since it requires additional information, i.e., sub-bitstream size information, for decoding its associated sub-bitstreams. One method of achieving this is through the use of the NAL sample stream format.

The NAL sample stream format can be constructed from the NAL unit stream format by ordering the NAL units in decoding order and prefixing each NAL unit with a heading that specifies the exact size, in bytes, of the NAL unit. A sample stream header is included at the beginning of the sample stream bitstream that specifies the precision, in bytes, of the signalled NAL unit size. The NAL unit stream format can be extracted from the sample stream format by traversing through the sample stream format, reading the size information and appropriately extracting each NAL unit. Methods of framing NAL units in a manner other than the use of the sample stream format are outside the scope of this document. The sample stream format is specified in Annex D.

* 1. Syntax and semantics
     1. Method of specifying syntax in tabular form

The specifications in subclause 8.1 apply.

* + 1. Specification of syntax functions and descriptors

The specifications in subclause 8.2 and I.8.2 apply.

* + 1. Syntax in tabular form
       1. General NAL unit syntax

|  |  |
| --- | --- |
| bmesh\_nal\_unit( BmNumBytesInNalUnit ) { | **Descriptor** |
| bmesh\_nal\_unit\_header( ) |  |
| BmNumBytesInRbsp = 0 |  |
| for( i = 2; i < BmNumBytesInNalUnit; i++ ) |  |
| **rbsp\_byte**[ BmNumBytesInRbsp++ ] | b(8) |
| } |  |

* + - 1. NAL unit header syntax

|  |  |
| --- | --- |
| bmesh\_nal\_unit\_header( ) { | **Descriptor** |
| **bmesh\_nal\_forbidden\_zero\_bit** | f(1) |
| **bmesh\_nal\_unit\_type** | u(6) |
| **bmesh\_nal\_layer\_id** | u(6) |
| **bmesh\_nal\_temporal\_id\_plus1** | u(3) |
| } |  |

* + - 1. Raw byte sequence payloads, trailing bits, and byte alignment syntax
         1. Basemesh sequence parameter set RBSP syntax

General basemesh sequence parameter set RBSP syntax

|  |  |
| --- | --- |
| bmesh\_sequence\_parameter\_set\_rbsp( ) { | **Descriptor** |
| **bmsps\_sequence\_parameter\_set\_id** | u(4) |
| bmesh\_profile\_tier\_level( ) |  |
| **bmsps\_intra\_mesh\_codec\_id** | u(8) |
| **bmsps\_inter\_mesh\_codec\_id** | u(8) |
| **bmsps\_geometry\_3d\_bit\_depth\_minus1** | u(5) |
| **bmsps\_inter\_mesh\_max\_num\_mvp\_cand\_minus1** | u(2) |
| **bmsps\_mesh\_attribute\_count** | u(7) |
| for( i = 0; i < bmsps\_mesh\_attribute\_count; i++ ) { |  |
| **bmsps\_mesh\_attribute\_index**[ i ] | u(8) |
| **bmsps\_mesh\_attribute\_type\_id**[ i ] | u(4) |
| **bmsps\_mesh\_attribute\_dimension\_minus1**[ i ] | u(6) |
| **bmsps\_attribute\_bit\_depth\_minus1**[ i ] | u(5) |
| **bmsps\_attribute\_msb\_align\_flag**[ i ] | u(1) |
| } |  |
| **bmsps\_intra\_mesh\_post\_reindex\_method** | ue(v) |
| **bmsps\_log2\_max\_mesh\_frame\_order\_cnt\_lsb\_minus4** | ue(v) |
| **bmsps\_max\_dec\_mesh\_frame\_buffering\_minus1** | ue(v) |
| **bmsps\_max\_num\_reorder\_frames** | ue(v) |
| **bmsps\_max\_latency\_increase\_plus1** | ue(v) |
| **bmsps\_long\_term\_ref\_mesh\_frames\_flag** | u(1) |
| **bmsps\_num\_ref\_mesh\_frame\_lists\_in\_bmsps** | ue(v) |
| for( i = 0; i < bmsps\_num\_ref\_mesh\_frame\_lists\_in\_bmsps; i++ ) |  |
| bmesh\_ref\_list\_struct( i ) |  |
| **bmsps\_inter\_mesh\_motion\_group\_size\_minus1** | ue(v) |
| **bmsps\_inter\_mesh\_max\_num\_neighbours\_minus1** | u(3) |
| **bmsps\_codec\_specific\_parameters\_present\_flag** | u(1) |
| if( bmsps\_codec\_specific\_parameters\_present\_flag ) { |  |
| **bmsps\_mesh\_codec\_prefix\_length\_minus1** | u(8) |
| for( i = 0 ; i < bmsps\_mesh\_codec\_prefix\_length\_minus1 + 1; i++ ) { |  |
| **bmsps\_mesh\_codec\_prefix\_data\_byte**[ i ] | u(8) |
| } |  |
| } |  |
| **bmsps\_extension\_present\_flag** | u(1) |
| if( bmsps\_extension\_present\_flag ) { |  |
| **bmsps\_extension\_count** | u(8) |
| } |  |
| if( bmsps\_extension\_count ){ |  |
| **bmsps\_extensions\_length\_minus1** | ue(v) |
| for( i = 0; i < bmsps\_extension\_count; i++ ) { |  |
| **bmsps\_extension\_type**[ i ] | u(8) |
| **bmsps\_extension\_length**[ i ] | u(16) |
| bmsps\_extension( bmsps\_extension\_type[ i ], bmsps\_extension\_length[ i ] ) |  |
| } |  |
| } |  |
| rbsp\_trailing\_bits( ) |  |
| } |  |

Basemesh SPS extension syntax

|  |  |
| --- | --- |
| bmsps\_extension( extension\_type, extension\_length ) { | **Descriptor** |
| for( j = 0; j < extension\_length; j++ ) |  |
| **bmsps\_extension\_data\_byte** | u(8) |
| length\_alignment( ) |  |
| } |  |

Basemesh Profile, tier, and level syntax

|  |  |
| --- | --- |
| bmesh\_profile\_tier\_level( ) { | **Descriptor** |
| **bmptl\_tier\_flag** | u(1) |
| **bmptl\_profile\_codec\_group\_idc** | u(7) |
| **bmptl\_profile\_toolset\_idc** | u(8) |
| **bmptl\_reserved\_zero\_32bits** | u(32) |
| **bmptl\_level\_idc** | u(8) |
| **bmptl\_num\_sub\_profiles** | u(6) |
| **bmptl\_extended\_sub\_profile\_flag** | u(1) |
| for( i = 0; I < bmptl\_num\_sub\_profiles; i++ ) { |  |
| **bmptl\_sub\_profile\_idc**[ i ] | u(v) |
| } |  |
| **bmptl\_toolset\_constraints\_present\_flag** | u(1) |
| if( bmptl\_toolset\_constraints\_present\_flag ) { |  |
| bmesh\_profile\_toolset\_constraints\_information( ) |  |
| } |  |
| } |  |

Profile toolset constraints information syntax

|  |  |
| --- | --- |
| bmesh\_profile\_toolset\_constraints\_information( ) { | **Descriptor** |
| **bmptc\_one\_mesh\_frame\_only\_flag** | u(1) |
| **bmptc\_intra\_frames\_only\_flag** | u(1) |
| **bmptc\_motion\_vector\_derivation\_disable\_flag** | u(1) |
| **bmptc\_reserved\_zero\_5bits** | u(5) |
| **bmptc\_num\_reserved\_constraint\_bytes** | u(8) |
| for( i = 0; i < ptc\_num\_reserved\_constraint\_bytes; i++ ) |  |
| bm**ptc\_reserved\_constraint\_byte**[ i ] | u(8) |
| } |  |

* + - * 1. Basemesh frame parameter set RBSP syntax

General basemesh frame parameter set RBPS syntax

|  |  |
| --- | --- |
| bmesh\_frame\_parameter\_set\_rbsp( ) { | **Descriptor** |
| **bmfps\_sequence\_parameter\_set\_id** | u(4) |
| **bmfps\_frame\_parameter\_set\_id** | u(4) |
| bmesh\_submesh\_information( ) |  |
| **bmfps\_output\_flag\_present\_flag** | u(1) |
| **bmfps\_num\_ref\_idx\_default\_active\_minus1** | ue(v) |
| **bmfps\_additional\_lt\_mfoc\_lsb\_len** | ue(v) |
| **bmfps\_extension\_present\_flag** | u(1) |
| if( bmfps\_extension\_present\_flag ) |  |
| **bmfps\_extension\_8bits** | u(8) |
| if( bmfps\_extension\_8bits ) |  |
| while( more\_rbsp\_data( ) ) |  |
| **bmfps\_extension\_data\_flag** | u(1) |
| rbsp\_trailing\_bits( ) |  |
| } |  |

Basemesh submesh information

|  |  |
| --- | --- |
| bmesh\_submesh\_information( ) { | **Descriptor** |
| **bmsi\_use\_single\_mesh\_flag** | u(1) |
| if(!bmsi\_use\_single\_mesh\_flag){ |  |
| **bmsi\_num\_submeshes\_minus2** | ue(v) |
| NumBmeshSubMeshes = bmsi\_num\_submeshes\_minus2 + 2 |  |
| } |  |
| else |  |
| NumBmeshSubMeshes = 1 |  |
| **bmsi\_signalled\_submesh\_id\_flag** | u(1) |
| if( bmsi\_signalled\_submesh\_id\_flag ) { |  |
| **delta\_bmsi\_signalled\_submesh\_id\_delta\_length** | ue(v) |
| for( i = 0; < NumBmeshSubMeshes; i++ ) |  |
| **bmsi\_submesh\_id**[ i ] | u(v) |
| BaseMeshSubmeshIDToIndex[ bmsi\_submesh\_id[ i ] ] = i |  |
| BaseMeshSubmeshIndexToID[ i ] = bmsi\_submesh\_id[ i ] |  |
| } |  |
| } |  |
| else |  |
| for( i = 0; i < NumBmeshSubMeshes; i++ ) { |  |
| bmsi\_submesh\_id[ i ] = i |  |
| BaseMeshSubmeshIDToIndex[ i ] = i |  |
| BaseMeshSubmeshIndexToID[ i ] = i |  |
| } |  |
| } |  |

* + - * 1. Basmesh submesh layer RBSP syntax

|  |  |
| --- | --- |
| bmesh\_submesh\_layer\_rbsp( ) { | **Descriptor** |
| bmesh\_submesh\_header( ) |  |
| bmesh\_submesh\_unit( bmsh\_id, BmNumBytesInNalUnit – 2 ) |  |
| rbsp\_trailing\_bits( ) |  |
| } |  |

* + - * 1. Basemesh submesh header syntax

|  |  |
| --- | --- |
| bmesh\_submesh\_header ( ) { | **Descriptor** |
| if( bmesh\_nal\_unit\_type >= BNAL\_BLA\_W\_LP &&  bmesh\_nal\_unit\_type <= BNAL\_RSV\_IRAP\_BMCL\_29 ) |  |
| **bmsh\_no\_output\_of\_prior\_mesh\_frames\_flag** | u(1) |
| **bmsh\_basemesh\_frame\_parameter\_set\_id** | u(4) |
| **bmsh\_id** | u(v) |
| submeshID = bmsh\_id |  |
| **bmsh\_type** | ue(v) |
| if( bfps\_output\_flag\_present\_flag ) |  |
| **bmsh\_mesh\_output\_flag** | u(1) |
| **bmsh\_mesh\_frm\_order\_cnt\_lsb** | u(v) |
| if( bmsps\_num\_ref\_mesh\_frame\_lists\_in\_bmsps > 0 ) |  |
| **bmsh\_ref\_mesh\_frame\_list\_msps\_flag** | u(1) |
| if( bmsh\_ref\_basemesh\_frame\_list\_msps\_flag == 0 ) |  |
| bmesh\_ref\_list\_struct( bmsps\_num\_ref\_mesh\_frame\_lists\_in\_bmsps ) |  |
| else if( bmsps\_num\_ref\_mesh\_frame\_lists\_in\_bmsps > 1 ) |  |
| **bmsh\_ref\_mesh\_frame\_list\_idx** | u(v) |
| for( j = 0; j < NumLtrMeshFrmEntries[ BmRlsIdx ]; j++ ) { |  |
| **bmsh\_additional\_mfoc\_lsb\_present\_flag**[ j ] | u(1) |
| if( bmsh\_additional\_mfoc\_lsb\_present\_flag[ j ] ) |  |
| **bmsh\_additional\_mfoc\_lsb\_val**[ j ] | u(v) |
| } |  |
| if( bmsh\_type != SKIP\_SUBMESH ) { |  |
| if( bmsh\_type == P\_SUBMESH && num\_ref\_entries[ BmRlsIdx ] > 1 ) { |  |
| **bmsh\_num\_ref\_idx\_active\_override\_flag** | u(1) |
| if( bmsh\_num\_ref\_idx\_active\_override\_flag ) |  |
| **bmsh\_num\_ref\_idx\_active\_minus1** | ue(v) |
| } |  |
| } |  |
| byte\_alignment( ) |  |
| } |  |

* + - * 1. Reference list structure syntax

|  |  |
| --- | --- |
| bmesh\_ref\_list\_struct( rlsIdx ) { | **Descriptor** |
| **num\_ref\_entries**[ rlsIdx ] | ue(v) |
| for( i = 0; i < num\_ref\_entries[ rlsIdx ]; i++ ) { |  |
| if( bmsps\_long\_term\_ref\_mesh\_frames\_flag ) |  |
| **st\_ref\_mesh\_frame\_flag**[ rlsIdx ][ i ] | u(1) |
| if( st\_ref\_mesh\_frame\_flag[ rlsIdx ][ i ] ) { |  |
| **abs\_delta\_mfoc\_st**[ rlsIdx ][ i ] | ue(v) |
| if( abs\_delta\_mfoc\_st[ rlsIdx ][ i ] > 0 ) |  |
| **straf\_entry\_sign\_flag**[ rlsIdx ][ i ] | u(1) |
| } else { |  |
| **mfoc\_lsb\_lt**[ rlsIdx ][ i ] | u(v) |
| } |  |
| } |  |
| } |  |

* + - * 1. Basemesh submesh unit syntax

|  |  |
| --- | --- |
| bmesh\_submesh\_unit( submeshID, unitSize ) { | **Descriptor** |
| if( bmsh\_type == I\_SUBMESH ) { |  |
| bm\_intra\_submesh\_unit( submeshID, unitSize ) |  |
| } |  |
| else if( bmsh\_type == P\_SUBMESH ) { |  |
| bm\_inter\_submesh\_unit( submeshID, unitSize ) |  |
| } |  |
| else if( bmsh\_type == SKIP\_SUBMESH ) { |  |
| bm\_skip\_submesh\_unit( ) |  |
| } |  |
| } |  |

* + - * 1. Basemesh intra submesh unit syntax

|  |  |
| --- | --- |
| bm\_intra\_submesh\_unit( submeshID, unitSize ) { | **Descriptor** |
| embedded\_external\_data\_unit( ) |  |
| } |  |

bmidu\_intra\_submesh\_unit( submeshID, unitSize ) contains a portion of mesh data of size unitSize as an ordered stream of bytes or bits within which the locations of unit boundaries are identifiable from patterns in the data. The format of such basemesh data is identified by bmptl\_profile\_codec\_group\_idc or by a component codec mapping SEI message.

* + - * 1. Basemesh inter submesh unit syntax

|  |  |
| --- | --- |
| bm\_inter\_submesh\_unit( submeshID, unitSize ) { | **Descriptor** |
| embedded\_external\_data\_unit() |  |
| } |  |

bm\_inter\_submesh\_unit( submeshID, vertexCount ) contains a portion of motion data of size unitSize as an ordered stream of bytes or bits within which the locations of unit boundaries are identifiable from patterns in the data. The format of such basemesh data is identified by bmptl\_profile\_codec\_group\_idc or by a component codec mapping SEI message.

* + - * 1. Basemesh inter submesh data unit syntax

|  |  |
| --- | --- |
| bm\_inter\_submesh\_data\_unit\_default ( submeshID  ) { | **Descriptor** |
| **bm\_vertex\_count**[ submeshID ] | vu(v) |
| vertexCount = bm\_vertex\_count[ submeshID ] |  |
| **bmidu\_mv\_signalled\_flag\_count**[ subMeshID ] | ae(v) |
| for( d = 0; d < bmidu\_mv\_signalled\_flag\_count[ subMeshID ]; d++ ) { |  |
| **bmidu\_mv\_signalled\_flag**[ subMeshID ][ d ] | ae(v) |
| if( !bmidu\_mv\_signalled\_flag[ subMeshID ][ d ] ) |  |
| NoSignalledMvCount++ |  |
| } |  |
| vertexCount = vertexCount – NoSignalledMvCount |  |
| groupSize = bmsps\_inter\_mesh\_motion\_group\_size\_minus1 + 1 |  |
| groupCount = ( vertexCount – 1) / groupSize + 1 |  |
| vStart = 0 |  |
| for( g = 0; g < groupCount: g++ ) { |  |
| **bmidu\_skip\_group\_flag**[ submeshID ][ g ] | ae(v) |
| if( !bmidu\_skip\_group\_flag[ submeshID ][ g ] ) { |  |
| for ( k = 0;  k < 3; k++ ) { |  |
| if (k != 2 ||   (bmidu\_skip\_group\_comp\_flag[ subMeshID ][ g ][ 0 ] == 0||   bmidu\_skip\_group\_comp\_flag[ subMeshID ][ g ][ 1 ] == 0) ) { |  |
| **bmidu\_skip\_group\_comp\_flag**[ subMeshID ][ g ][ k ] | ae(v) |
| } |  |
| if ( !bmidu\_skip\_group\_comp\_flag[ subMeshID ][ g ][ k ] ) { |  |
| **bmidu\_mv\_pred\_mode\_group**[ subMeshID ][ g ][ k ] | ae(v) |
| } |  |
| } |  |
| if ( g == (groupCount – 1) ) |  |
| groupSize = vertexCount – \* (groupCount – 1) |  |
| for( v = vStart; v < (vStart+groupSize); v++ ) { |  |
| for( k = 0; k < 3; k++ ) { |  |
| if( !bmidu\_skip\_group\_flag[ subMeshID ][ g ] ) { |  |
| if( !bmidu\_skip\_group\_comp\_flag[ subMeshID ][ g ][ k ] ) { |  |
| **bmidu\_mv\_residual\_abs\_gt0**[ submeshID ][ v ][ k ] | ae(v) |
| if (bmidu\_mv\_residual\_abs\_gt0[ submeshID ][ v ][ k ]) { |  |
| **bmidu\_mv\_residual\_sign**[ submeshID ][ v ][ k ] | ae(v) |
| **bmidu\_mv\_residual\_abs\_gt1**[ submeshID ][ v ][ k ] | ae(v) |
| if (bmidu\_mv\_residual\_abs\_gt1[ submeshID ][ v ][ k ]) |  |
| **bmidu\_mv\_residual\_abs\_rem**[ submeshID ][ v ][ k ] | ae(v) |
| } |  |
| } |  |
| } |  |
| } |  |
| }//v |  |
| } |  |
| vStart += groupSize |  |
| } |  |
| } |  |

* + - * 1. Basemesh skip submesh data unit syntax

|  |  |
| --- | --- |
| sdu\_skip\_submesh\_unit( ) { | **Descriptor** |
| } |  |

* + - * 1. Embedded external data unit syntax

|  |  |
| --- | --- |
| embedded\_external\_data\_unit( ) { | **Descriptor** |
| while( more\_rbsp\_data() ) |  |
| **embedded\_external\_data\_bit** | u(1) |
| } |  |

* + 1. Semantics
       1. General

Semantics associated with the syntax structures and with the syntax elements within these structures are specified in this subclause. When the semantics of a syntax element are specified using a table or a set of tables, any values that are not specified in the table(s) shall not be present in the bitstream unless otherwise specified in this document.

* + - 1. NAL unit semantics
         1. General NAL unit semantics

BmNumBytesInNalUnit specifies the size of the NAL unit in bytes. This value is required for decoding of the NAL unit. Some form of demarcation of NAL unit boundaries is necessary to enable inference of BmNumBytesInNalUnit. One such demarcation method is specified in Annex D for the sample stream format. Other methods of demarcation can be specified outside this document.

NOTE 1 – The basemesh coding layer (BMCL) is specified to efficiently represent the content of the basemesh data. The NAL is specified to format that data and provide header information in a manner appropriate for conveyance on a variety of communication channels or storage media. All data are contained in NAL units, each of which contains an integer number of bytes. A NAL unit specifies a generic format for use in both packet-oriented and bitstream systems. The format of NAL units for both packet-oriented transport and sample streams is identical except that in the sample stream format specified in Annex [CrossRefeenceXXX] each NAL unit can be preceded by an additional element that specifies the size of the NAL unit.

**rbsp\_byte**[ i ] is the i-th byte of an RBSP. An RBSP is specified as an ordered sequence of bytes as follows:

The RBSP contains a string of data bits(SODB) as follows:

* If the SODB is empty (i.e., zero bits in length), the RBSP is also empty.
* Otherwise, the RBSP contains the SODB as follows:
  + The first byte of the RBSP contains the first (most significant, left-most) eight bits of the SODB; the next byte of the RBSP contains the next eight bits of the SODB, etc., until fewer than eight bits of the SODB remain.
  + The rbsp\_trailing\_bits( ) syntax structure is present after the SODB as follows:
    - The first (most significant, left-most) bits of the final RBSP byte contain the remaining bits of the SODB (if any).
    - The next bit consists of a single bit equal to 1 (i.e., rbsp\_stop\_one\_bit).
    - When the rbsp\_stop\_one\_bit is not the last bit of a byte-aligned byte, one or more bits equal to 0 (i.e. instances of rbsp\_alignment\_zero\_bit) are present to result in byte alignment.

Syntax structures having these RBSP properties are denoted in the syntax tables using an "\_rbsp" suffix. These structures are carried within NAL units as the content of the rbsp\_byte[ i ] data bytes. The association of the RBSP syntax structures to the NAL units is as specified in Table H-1.

NOTE 2 – When the boundaries of the RBSP are known, the decoder can extract the SODB from the RBSP by concatenating the bits of the bytes of the RBSP and discarding the rbsp\_stop\_one\_bit, which is the last (least significant, right-most) bit equal to 1, and discarding any following (less significant, farther to the right) bits that follow it, which are equal to 0. The data necessary for the decoding process is contained in the SODB part of the RBSP.

* + - * 1. NAL unit header semantics

**bmesh\_nal\_forbidden\_zero\_bit** shall be equal to 0.

**bmesh\_nal\_unit\_type** specifies the type of the RBSP data structure contained in the NAL unit as specified in Table H-1.

NAL units that have bmesh\_nal\_unit\_type in the range of BNAL\_UNSPEC\_45..BNAL\_UNSPEC\_63, inclusive, for which semantics are not specified, shall not affect the decoding process specified in this document.

NOTE 1 – NAL unit types in the range of BNAL\_UNSPEC\_45..BNAL\_UNSPEC\_63 can be used as determined by the application. No decoding process for these values of bmesh\_nal\_unit\_type is specified in this document. Since different applications can use these NAL unit types for different purposes, particular care needs to be exercised in the design of encoders that generate NAL units with these bmesh\_nal\_unit\_type values, and in the design of decoders that interpret the content of NAL units with these bmesh\_nal\_unit\_type values. This document does not define any management for these values. These bmesh\_nal\_unit\_type values could only be suitable for use in contexts in which "collisions" of usage (i.e., different definitions of the meaning of the NAL unit content for the same bmesh\_nal\_unit\_type value) are unimportant, or not possible, or are managed – e.g., defined or managed in the controlling application or transport specification, or by controlling the environment in which bitstreams are distributed.

For purposes other than determining the amount of data in the decoding units of the bitstream (as specified in Annex D), decoders shall ignore (i.e. remove from the bitstream and discard) the contents of all NAL units that use reserved values of bmesh\_nal\_unit\_type.

NOTE 2 – This requirement allows future definition of compatible extensions to this document.

Table H-1 – BNAL unit type codes and BNAL unit type classes

|  |  |  |  |
| --- | --- | --- | --- |
| bmesh\_nal\_unit\_type | Name of bmesh\_nal\_unit\_type | Content of basemesh NAL unit and RBSP syntax structure | NAL unit type class |
| 0 1 | NAL\_TRAIL\_N BNAL\_TRAIL\_R | Coded submesh of a non-TSA, non STSA trailing basemesh frame bmesh\_submesh\_layer\_rbsp( ) | BMCL |
| 2 3 | BNAL\_TSA\_N BNAL\_TSA\_R | Coded submesh of a TSA basemesh frame bmesh\_submesh\_layer\_rbsp( ) | BMCL |
| 4 5 | BNAL\_STSA\_N BNAL\_STSA\_R | Coded submesh of a STSA basemesh frame bmesh\_submesh\_layer\_rbsp( ) | BMCL |
| 6 7 | BNAL\_RADL\_N BNAL\_RADL\_R | Coded submesh of a RADL basemesh frame bmesh\_submesh\_layer\_rbsp( ) | BMCL |
| 8 9 | BNAL\_RASL\_N BNAL\_RASL\_R | Coded submesh of a RASL basemesh frame bmesh\_submesh\_layer\_rbsp( ) | BMCL |
| 10 11 | BNAL\_SKIP\_N BNAL\_SKIP\_R | Coded submesh of a skipped basemesh frame bmesh\_submesh\_layer\_rbsp( ) | BMCL |
| 12 14 | BNAL\_RSV\_BMCL\_N12 BNAL\_RSV\_BMCL\_N14 | Reserved non-IRAP sub-layer non-reference BMCL mesh NAL unit types | BMCL |
| 13 15 | BNAL\_RSV\_BMCL\_R13 BNAL\_RSV\_BMCL\_R15 | Reserved non-IRAP sub-layer reference BMCL mesh NAL unit types | BMCL |
| 16 17 18 | BNAL\_BLA\_W\_LP BNAL\_BLA\_W\_RADL BNAL\_BLA\_N\_LP | Coded submesh of a BLA basemesh frame  bmesh\_submesh\_layer\_rbsp( ) | BMCL |
| 19 20 | BNAL\_IDR\_W\_RADL BNAL\_IDR\_N\_LP | Coded submesh of an IDR basemesh frame  bmesh\_submesh\_layer\_rbsp( ) | BMCL |
| 21 | BNAL\_CRA | Coded submesh of a CRA basemesh frame  bmesh\_submesh\_layer\_rbsp( ) | BMCL |
| 22 23 | BNAL\_RSV\_IRAP\_BMCL\_22 BNAL\_RSV\_IRAP\_BMCL\_23 | Reserved IRAP BMCL NAL unit types | BMCL |
| 24..29 | BNAL\_RSV\_BMCL\_24.. BNAL\_RSV\_BMCL\_29 | Reserved non-IRAP BMCL NAL unit types | BMCL |
| 30 | BNAL\_BMSPS | Basemesh sequence parameter set bmesh\_sequence\_parameter\_set\_rbsp( ) | non-BMCL |
| 31 | BNAL\_BMFPS | Basemesh frame parameter set bmesh\_frame\_parameter\_set\_rbsp( ) | non-BMCL |
| 32 | BNAL\_AUD | Access unit delimiter access\_unit\_delimiter\_rbsp( ) | non-BMCL |
| 33 | BNAL\_EOS | End of sequence end\_of\_sequence\_rbsp( ) | non-BMCL |
| 34 | BNAL\_EOB | End of bitstream end\_of\_bmesh\_sub\_bitstream\_rbsp( ) | non-BMCL |
| 35 | BNAL\_FD | Filler filler\_data\_rbsp( ) | non-BMCL |
| 36 37 | BNAL\_PREFIX\_NSEI  BNAL\_SUFFIX\_NSEI | Non-essential supplemental enhancement information sei\_rbsp( ) | non-BMCL |
| 38 39 | BNAL\_PREFIX\_ESEI BNAL\_SUFFIX\_ESEI | Essential supplemental enhancement information sei\_rbsp( ) | non-BMCL |
| 40..44 | BNAL\_RSV\_NBMCL\_40 BNAL\_RSV\_NBMCL\_44 | Reserved non-BMCL NAL unit types | non-BMCL |
| 45..63 | BNAL\_UNSPEC\_45 BNAL\_UNSPEC\_63 | Unspecified non-BMCL NAL unit types | non-BMCL |

NOTE 3 – A clean random access (CRA) mesh frame can have associated random access skipped leading (RASL) or random access decodable leading (RADL) mesh frames present in the bitstream.

NOTE 4 – A broken link access (BLA) mesh frame having bmesh\_nal\_unit\_type equal to BNAL\_BLA\_W\_LP can have associated RASL or RADL mesh frames present in the bitstream. A BLA mesh frame having bmesh\_nal\_unit\_type equal to BNAL\_BLA\_W\_RADL does not have associated RASL mesh frames present in the bitstream, but can have associated RADL mesh frames in the bitstream. A BLA mesh frame having bmesh\_nal\_unit\_type equal to BNAL\_BLA\_N\_LP does not have associated leading mesh frames present in the bitstream.

NOTE 5 – An instantaneous decoding refresh (IDR) mesh frame having bmesh\_nal\_unit\_type equal to BNAL\_IDR\_N\_LP does not have associated leading mesh frames present in the bitstream. An IDR mesh frame having bmesh\_nal\_unit\_type equal to BNAL\_IDR\_W\_RADL does not have associated RASL mesh frames present in the bitstream, but can have associated RADL mesh frames in the bitstream.

All coded submesh NAL units of an access unit shall have the same value of bmesh\_nal\_unit\_type. A mesh frame or an access unit is also referred to as having a bmesh\_nal\_unit\_type equal to the bmesh\_nal\_unit\_type of the coded submesh NAL units of the mesh frame or the access unit.

If a mesh frame has bmesh\_nal\_unit\_type equal to BNAL\_TRAIL\_N, BNAL\_TSA\_N, BNAL\_STSA\_N, BNAL\_RADL\_N, BNAL\_RASL\_N, the mesh frame is an SLNR mesh frame. Otherwise, the mesh frame is a sub-layer reference mesh frame.

Each mesh frame, other than the first mesh frame in the bitstream in decoding order, is considered to be associated with the previous intra random access point (IRAP) mesh frame in decoding order.

When a mesh frame is a leading mesh frame, it shall be a RADL or RASL mesh frame.

When a mesh frame is a trailing mesh frame, it shall not be a RADL or RASL mesh frame.

When a mesh frame is a leading mesh frame, it shall precede, in decoding order, all trailing mesh frames that are associated with the same IRAP coded mesh frame.

No RASL mesh frames shall be present in the bitstream that are associated with a BLA mesh frame having bmesh\_nal\_unit\_type equal to BNAL\_BLA\_W\_RADL or BNAL\_BLA\_N\_LP.

No RASL mesh frames shall be present in the bitstream that are associated with an IDR mesh frame.

No RADL mesh frames shall be present in the bitstream that are associated with a BLA mesh frame having bmesh\_nal\_unit\_type equal to BNAL\_BLA\_N\_LP, an IDR mesh frame having bmesh\_nal\_unit\_type equal to BNAL\_IDR\_N\_LP.

NOTE 6 – It is possible to perform random access at the position of an IRAP coded mesh access unit by discarding all access units before the IRAP coded mesh access unit (and to correctly decode the IRAP coded mesh frame and all the subsequent non-RASL mesh frames in decoding order), provided each parameter set is available (either in the bitstream or by external means not specified in this document) when it needs to be activated.

Any RASL mesh frame associated with a CRA or BLA mesh frame shall precede any RADL mesh frame associated with the CRA or BLA mesh frame in output order.

Any RASL mesh frame associated with a CRA mesh frame shall follow, in output order, any IRAP coded mesh frame that precedes the CRA mesh frame in decoding order.

**bmesh\_nal\_layer\_id** specifies the identifier of the layer to which an BMCL NAL unit belongs or the identifier of a layer to which a non-BMCL NAL unit applies. The value of bmesh\_nal\_layer\_id shall be in the range of 0 to 62, inclusive. The value of 63 may be specified in the future by ISO/IEC. For purposes other than determining the amount of data in the decoding units of the bitstream, decoders shall ignore all data that follow the value 63 for bmesh\_nal\_layer\_id in a NAL unit, and decoders conforming to a profile specified in Clause H.11 shall ignore (i.e., remove from the bitstream and discard) all NAL units with values of bmesh\_nal\_layer\_id not equal to 0.

NOTE 7 – The value of 63 for bmesh\_nal\_layer\_id can be used to indicate an extended layer identifier in a future extension of this document.

The value of bmesh\_nal\_layer\_id shall be the same for all BMCL NAL units of a coded mesh frame. The value of bmesh\_nal\_layer\_id of a coded mesh frame is the value of the bmesh\_nal\_layer\_id of the BMCL NAL units of the coded mesh frame.

When bmesh\_nal\_unit\_type is equal to BNAL\_EOB, the value of bmesh\_nal\_layer\_id shall be equal to 0.

**bmesh\_nal\_temporal\_id\_plus1** minus 1 specifies a temporal identifier for the NAL unit. The value of bmesh\_nal\_temporal\_id\_plus1 shall not be equal to 0.

The variable BmTemporalID is specified as follows:

BmTemporalID = bmesh\_nal\_temporal\_id\_plus1 – 1 (49)

When bmesh\_nal\_unit\_type is in the range of BNAL\_BLA\_W\_LP to BNAL\_RSV\_IRAP\_BMCL29, inclusive, i.e., the coded submesh belongs to an IRAP coded mesh frame, BmTemporalID shall be equal to 0.

When bmesh\_nal\_unit\_type is equal to BNAL\_TSA\_R or BNAL\_TSA\_N, BmTemporalID shall not be equal to 0.

When bmesh\_nal\_layer\_id is equal to 0 and bmesh\_nal\_unit\_type is equal to BNAL\_STSA\_R or BNAL\_STSA\_N, BmTemporalID shall not be equal to 0.

The value of BmTemporalID shall be the same for all BMCL NAL units of an access unit. The value of BmTemporalID of a coded mesh frame or an access unit is the value of the BmTemporalID of the BMCL NAL units of the coded mesh frame or the access unit. The value of BmTemporalID of a sub-layer representation is the greatest value of BmTemporalID of all BMCL NAL units in the sub-layer representation.

The value of BmTemporalID for non-BMCL NAL units is constrained as follows:

* If bmesh\_nal\_unit\_type is equal to BNAL\_BMSPS, BmTemporalID shall be equal to 0 and the BmTemporalID of the access unit containing the NAL unit shall be equal to 0.
* Otherwise, if bmesh\_nal\_unit\_type is equal to BNAL\_EOS or BNAL\_EOB, BmTemporalID shall be equal to 0.
* Otherwise, if bmesh\_nal\_unit\_type is equal to BNAL\_AUD, or BNAL\_FD, BmTemporalID shall be equal to the BmTemporalID of the access unit containing the NAL unit.
* Otherwise, BmTemporalID shall be greater than or equal to the BmTemporalID of the access unit containing the NAL unit.

NOTE  – When the NAL unit is a non-BMCL NAL unit, the value of BmTemporalID is equal to the minimum value of the BmTemporalID values of all access units to which the non-BMCL NAL unit applies. When bmesh\_nal\_unit\_type is equal to BNAL\_BMFPS, BmTemporalID can be greater than or equal to the BmTemporalID of the containing access unit, as all basemesh frame parameter sets (BMFPSs) can be included in the beginning of a bitstream, wherein the first coded basemesh frame has BmTemporalID equal to 0. When bmesh\_nal\_unit\_type is equal to BNAL\_PREFIX\_NSEI, BNAL\_PREFIX\_ESEI, BNAL\_SUFFIX\_NSEI, or BNAL\_SUFFIX\_ESEI, BmTemporalID can be greater than or equal to the BmTemporalID of the containing access unit, as an SEI NAL unit can contain information, e.g., in a buffering period SEI message or a basemesh frame timing SEI message, that applies to a bitstream subset that includes access units for which the BmTemporalID values are greater than the BmTemporalID of the access unit containing the SEI NAL unit.

* + - * 1. Order of NAL units and basemesh frames, and association to coded basemesh frames, access units, and coded basemesh sequences

General

This subclause specifies constraints on the order of NAL units and basemesh frames in the basemesh sub-bitstream.

Any order of NAL units in the basemesh sub-bitstream obeying these constraints is referred to in the text as the decoding order of NAL units. Within a NAL unit, the syntax in subclause H.8.3, specifies the decoding order of syntax elements. Decoders shall be capable of receiving NAL units and their syntax elements in decoding order.

Order of BMSPS, and BMFPS RBSPs and their activation

This subclause specifies the activation process of basemesh sequence parameter sets (BMSPSs) and basemesh frame parameter sets (BMFPSs).

NOTE – The BMSPS and BMFPS mechanism decouples the transmission of infrequently changing information from the transmission of coded mesh data. BMSPSs and BMFPSs can, in some applications, be conveyed "out-of-band".

An BMFPS RBSP includes parameters that can be referred to by the coded submesh NAL units of one or more coded basemesh frames. Each BMFPS RBSP is initially considered not active for any basemesh at the start of the operation of the decoding process. At most one BMFPS RBSP is considered active for each basemesh at any given moment during the operation of the decoding process, and the activation of any particular BMFPS RBSP for a particular basemesh results in the deactivation of the previously active BMFPS RBSP for the particular mesh.

When an BMFPS RBSP (with a particular value of bmfps\_frame\_parameter\_set\_id) is not active for a particular basemesh and is referred to by a coded submesh NAL unit with bmesh\_nal\_layer\_id equal to 0 (using a value of bmsh\_basemesh\_frame\_parameter\_set\_id equal to the bmfps\_frame\_parameter\_set\_id value), it is then activated. This BMFPS RBSP is called the active BMFPS RBSP until it is deactivated by the activation of another BMFPS RBSP. An BMFPS RBSP, with that particular value of bmfps\_frame\_parameter\_set\_id, shall be available to the decoding process prior to its activation, included in at least one coded mesh access unit with BmTemporalID less than or equal to the BmTemporalID of the BMFPS NAL unit or provided through external means.

Any BMFPS NAL unit containing the value of bmfps\_frame\_parameter\_set\_id for the active BMFPS RBSP for a coded mesh frame shall have the same content as that of the active BMFPS RBSP for the coded basemesh frame, unless it follows the last BMCL NAL unit of the coded basemesh frame and precedes the first BMCL NAL unit of another coded basemesh frame.

An BMSPS RBSP includes parameters that can be referred to by one or more BMFPS RBSPs. Each BMSPS RBSP is initially considered not active at the start of the operation of the decoding process. At most one BMSPS RBSP is considered active at any given moment during the operation of the decoding process, and the activation of any particular BMSPS RBSP results in the deactivation of the previously active BMSPS RBSP.

When an BMSPS RBSP (with a particular value of bmsps\_sequence\_parameter\_set\_id) is not already active for a particular basemesh and it is referred to by activation of an BMFPS RBSP (in which bmfps\_sequence\_parameter\_set\_id is equal to the bmsps\_sequence\_parameter\_set\_id), it is activated for the particular basemesh. This BMSPS RBSP is called the active BMSPS RBSP for the particular basemesh until it is deactivated by the activation of another BMSPS RBSP for the particular basemesh. An BMSPS RBSP, with that particular value of bmsps\_mesh\_sequence\_parameter\_set\_id, shall be available to the decoding process prior to its activation, included in at least one access unit with BmTemporalID equal to 0 or provided through external means. An activated BMSPS RBSP shall remain active for the entire coded basemesh sequence (CBMS).

Any BMSPS NAL unit with bmesh\_nal\_layer\_id equal to 0 containing the value of bmsps\_sequence\_parameter\_set\_id for the active BMSPS RBSP for a CBMS shall have the same content as that of the active BMSPS RBSP for the CBMS, unless it follows the last access unit of the CBMS and precedes the first BMCL NAL unit of another CBMS.

All constraints that are expressed on the relationship between the values of the syntax elements and the values of variables derived from those syntax elements in BMSPSs and BMFPSs, and other syntax elements, are expressions of constraints that apply only to the active BMSPS RBSP and the active BMFPS RBSP. If any BMSPS RBSP and BMFPS RBSP is present that is never activated in the bitstream, its syntax elements shall have values that would conform to the specified constraints if it was activated by reference in an otherwise conforming bitstream.

During operation of the decoding process (see Clause H.9), the values of parameters of the active BMSPS RBSP and the active BMFPS RBSP are considered in effect. For interpretation of SEI messages, the values of the active BMSPS RBSP and the active BMFPS RBSP for the operation of the decoding process for the BMCL NAL units of the coded basemesh frame with bmesh\_nal\_layer\_id equal to 0 in the same access unit are considered in effect unless otherwise specified in the SEI message semantics.

Order of access units (AUs) and association to CBMSs

A bitstream conforming to this document consists of one or more CBMSs.

A CBMS consists of one or more access units. The order of NAL units and coded basemesh frames, and their association to access units are described in subclause H.8.4.2.3.3.

The first access unit of a CBMS is an IRAP coded mesh access unit with NoOutputBeforeRecoveryFlag equal to 1.

It is a requirement of bitstream conformance that, when present, the next access unit after an access unit that contains an end of sequence NAL unit or an end of a bitstream NAL unit shall be an IRAP coded mesh access unit, which may be an IDR access unit, a BLA access unit, or a CRA access unit.

Order of NAL units and coded basemesh frames, and their association to access units

This subclause specifies the order of NAL units and coded basemesh frames, and their association to access units for CBMSs that conform to one or more of the profiles specified in Clauses H.11 and that are decoded using the decoding process specified in Clauses  H.2 through H.9.

An access unit consists of one coded basemesh with bmesh\_nal\_layer\_id equal to 0, zero or more BMCL NAL units with bmesh\_nal\_layer\_id greater than 0, and zero or more non-BMCL NAL units. The association of BMCL NAL units to coded meshes is described in subclause  H.8.4.2.3.5.

The first access unit in the bitstream starts with the first NAL unit of the bitstream.

Let firstBlAFrmNalUnit be the first BMCL NAL unit of a coded basemesh frame with bmesh\_nal\_layer\_id equal to 0. The first of any of the following NAL units preceding firstBlAFrmNalUnit and succeeding the last BMCL NAL unit preceding firstBlAFrmNalUnit, if any, specifies the start of a new access unit:

NOTE  – The last BMCL NAL unit preceding firstBlAFrmNalUnit in decoding order can have bmesh\_nal\_layer\_id greater than 0.

* access unit delimiter NAL unit with bmesh\_nal\_layer\_id equal to 0 (when present),
* BMSPS NAL unit with bmesh\_nal\_layer\_id equal to 0 (when present),
* BMFPS NAL unit with bmesh\_nal\_layer\_id equal to 0 (when present),
* Prefix SEI NAL unit with bmesh\_nal\_layer\_id equal to 0 (when present),
* NAL units with bmesh\_nal\_unit\_type in the range of BNAL\_RSV\_NBMCL\_40..BNAL\_RSV\_NBMCL\_44 with bmesh\_nal\_layer\_id equal to 0 (when present),
* NAL units with bmesh\_nal\_unit\_type in the range of BNAL\_UNSPEC\_45..BNAL\_UNSPEC\_63 with bmesh\_nal\_layer\_id equal to 0 (when present).

NOTE  – The first NAL unit preceding firstBlAFrmNalUnit and succeeding the last BMCL NAL unit preceding firstBlAFrmNalUnit, if any, can only be one of the above-listed NAL units.

When there is none of the above NAL units preceding firstBlAFrmNalUnit and succeeding the last BMCL NAL preceding firstBlAFrmNalUnit, if any, firstBlAFrmNalUnit starts a new access unit.

The order of the coded basemesh frames and non-BMCL NAL units within an access unit shall obey the following constraints:

* When an access unit delimiter NAL unit with bmesh\_nal\_layer\_id equal to 0 is present, it shall be the first NAL unit. There shall be at most one access unit delimiter NAL unit with bmesh\_nal\_layer\_id equal to 0 in any access unit.
* When any BMSPS NAL units, BMFPS NAL units, prefix SEI NAL units, NAL units with bmesh\_nal\_unit\_type in the range of BNAL\_RSV\_NBMCL\_40.. BNAL\_RSV\_NBMCL\_44, or NAL units with bmesh\_nal\_unit\_type in the range of BNAL\_UNSPEC\_45..BNAL\_UNSPEC\_63 are present, they shall not follow the last BMCL NAL unit of the access unit.
* NAL units having bmesh\_nal\_unit\_type equal to BNAL\_FD, BNAL\_SUFFIX\_NSEI, BNAL\_SUFFIX\_ESEI or in the range of BNAL\_RSV\_NBMCL\_40..BNAL\_RSV\_NBMCL\_44 or BNAL\_UNSPEC\_45..BNAL\_UNSPEC\_63 shall not precede the first BMCL NAL unit of the access unit.
* When an end of sequence NAL unit with bmesh\_nal\_layer\_id equal to 0 is present, it shall be the last NAL unit among all NAL units with bmesh\_nal\_layer\_id equal to 0 in the access unit other than an end of a bitstream NAL unit (when present).
* When an end of a bitstream NAL unit is present, it shall be the last NAL unit in the access unit.

Order of BMCL NAL units and association to coded mesh frames

This subclause specifies the order of BMCL NAL units and association to coded basemesh frames.

Each BMCL NAL unit is part of a coded basemesh frame.

The order of the BMCL NAL units within a coded basemesh frame is constrained as follows:

* The first BMCL NAL unit of the coded basemesh frame shall have ath\_id equal to FirstSubmeshID.

The submeshs of a basemesh frame shall be in increasing order of their ath\_id values.

* + 1. Raw byte sequence payloads, trailing bits, and byte alignment semantics
       1. Basemesh sequence parameter set RBSP semantics
          1. General basemesh sequence parameter set RBSP semantics

**bmsps\_sequence\_parameter\_set\_id** provides an identifier for the basemesh sequence parameter set for reference by other syntax elements.

**bmsps\_intra\_mesh\_codec\_id** indicates a mapping index of a codec identifier of the static mesh decoder used to decode the static basemesh subbitstream. bmsps\_intra\_mesh\_codec\_id shall be in the range of 0 to 255, inclusive. This decoder may be identified through the profiles defined in Annex H.11, or by a component codec mapping SEI message, or through means outside this document.

**bmsps\_inter\_mesh\_codec\_id** indicates a mapping index of a codec identifier of the decoder used to decode the motion data. bmsps\_intr\_mesh\_codec\_id shall be in the range of 0 to 255, inclusive. This decoder may be identified through the profiles defined in Annex H.11, or by a component codec mapping SEI message, or through means outside this document.

**bmsps\_inter\_mesh\_max\_num\_mvp\_cand\_minus1** plus 1 indicates the maximum number of motion vector predictor candidate. bmsps\_inter\_mesh\_max\_num\_mvp\_cand\_minus1 shall be in the range of 0 to 2, inclusive.

**bmsps\_geometry\_3d\_bit\_depth\_minus1** plus 1 indicates the bit depth of the geometry coordinates of the reconstructed basemesh. bmsps\_geometry\_3d\_bit\_depth\_minus1 shall be in the range of 0 to 31, inclusive.

**bmsps\_mesh\_attribute\_count** indicates the number of attributes associated with the basemesh. bmsps\_mesh\_attribute\_count shall be in the range of 0 to 127, inclusive.

**bmsps\_mesh\_attribute\_index**[ i ]indicates the index of the mesh attribute in the static mesh sub-bitstream associated to the i-th basemesh attribute. bmsps\_mesh\_attribute\_index[ i ] shall be in the range of 0 to 127, inclusive.

**bmsps\_mesh\_attribute\_type\_id**[ i ] indicates the attribute type of the attribute with index i for the basemesh. Table H-2 describes the list of supported attributetypes and their relationship with bmsps\_mesh\_attribute\_type\_id[ i ].

Table H-2 – Relation between bmsps\_mesh\_attribute\_type\_id and attribute type

|  |  |  |
| --- | --- | --- |
| **bmsps\_mesh\_attribute\_type\_id**[ i ] | **Identifier** | **Attribute type** |
| 0 | ATTR\_TEXTURE | Texture |
| 1 | ATTR\_MATERIAL\_ID | Material ID |
| 2 | ATTR\_TRANSPARENCY | Transparency |
| 3 | ATTR\_REFLECTANCE | Reflectance |
| 4 | ATTR\_NORMAL | Normals |
| 5 | ATTR\_FACEGROUP\_ID | Facegroup ID |
| 6..14 | ATTR\_RESERVED | Reserved |
| 15 | ATTR\_UNSPECIFIED | Unspecified |

**bmsps\_mesh\_attribute\_dimension\_minus1** plus 1 specifies the dimensions for i-th basemesh attribute. bmsps\_mesh\_attribute\_dimension\_minus1[i] shall be in the range of 0 to 63, inclusive.

The value for bmsps\_mesh\_attribute\_dimension\_minus1 plus 1 for bmsps\_mesh\_attribute\_type\_id equal to 0 i.e. ATTR\_TEXTURE shall be 3.

The value for bmsps\_mesh\_attribute\_dimension\_minus1 plus 1 for bmsps\_mesh\_attribute\_type\_id equal to 1 i.e. ATTR\_MATERIAL\_ID shall be 1.

The value for bmsps\_mesh\_attribute\_dimension\_minus1 plus 1 for bmsps\_mesh\_attribute\_type\_id equal to 1 i.e. ATTR\_TRANSPARENCY shall be 1.

The value for bmsps\_mesh\_attribute\_dimension\_minus1 plus 1 for bmsps\_mesh\_attribute\_type\_id equal to 1 i.e. ATTR\_NORMAL shall be either 2 or 3.

The value for bmsps\_mesh\_attribute\_dimension\_minus1 plus 1 for bmsps\_mesh\_attribute\_type\_id equal to 1 i.e. ATTR\_FACEGROUP\_ID shall be 1.

**bmsps\_attribute\_bit\_depth\_minus1**[ i ] plus 1 indicates the bit depth of the attribute with index i for the mesh. bmsps\_attribute\_bit\_depth\_minus1[ i ] shall be in the range of 0 to 31, inclusive.

**bmsps\_attribute\_msb\_align\_flag**[ i ] indicates how the decoded attribute samples in the basemesh for the attribute with index i are converted to attribute samples at the nominal attribute bit depth.

**bmsps\_intra\_mesh\_post\_reindex\_method** indicates a method to use to reorder indices coming from the static mesh coder before motion estimation. Table H-3 describes the list of supported methods.

Table H-3 – Basemesh post reindex methods

|  |  |  |
| --- | --- | --- |
| **bmsps\_intra\_mesh\_post\_reindex\_method** | **Identifier** | **Method** |
| 0 | BM\_REINDEX\_NONE | Disabled |
| 1 | BM\_REINDEX\_VERTEX\_DEGREE | Vertex degree traversal |
| >1 | BM\_REINDEX\_RESERVED | Reserved |

**bmsps\_log2\_max\_mesh\_frame\_order\_cnt\_lsb\_minus4** plus 4 specifies the values of the variables Log2MaxMeshFrmOrderCntLsb and MaxMeshFrmOrderCntLsb that are used in the decoding process for the mesh frame order count as follows:

Log2MaxMeshFrmOrderCntLsb =   
 bmsps\_log2\_max\_mesh\_frame\_order\_cnt\_lsb\_minus4 + 4

MaxMeshFrmOrderCntLsb = 2Log2MaxMeshFrmOrderCntLsb

The value of bmsps\_log2\_max\_mesh\_frame\_order\_cnt\_lsb\_minus4 shall be in the range of 0 to 12, inclusive.

**bmsps\_max\_dec\_mesh\_frame\_buffering\_minus1** plus 1 specifies the maximum required size of the decoded basemesh frame buffer for the CBMS in units of basemesh frame storage buffers. The value of bmps\_max\_dec\_mesh\_frame\_buffering\_minus1 shall be in the range of 0 to 15, inclusive.

**bmsps\_max\_num\_reorder\_frames** specifies the maximum allowed number of frames that can precede any frame in in decoding order and follow that frame in output order for the CBMS. The value of bmsps\_max\_num\_reorder\_frames shall be in the range of 0 to bmsps\_max\_dec\_mesh\_frame\_buffering\_minus1, inclusive.

**bmsps\_max\_latency\_increase\_plus1** not equal to 0 is used to compute the value of MaxLatencyBaseMeshFrames, which specifies the maximum number of frames that can precede any frame in output order and follow that picture in decoding order for the CBMS.

When bmsps\_max\_latency\_increase\_plus1 is not equal to 0, the value of MaxLatencyBaseMeshFrames is specified as follows:

MaxLatencyBaseMeshFrames =  
 bmsps\_max\_num\_reorder\_frames + bmsps\_max\_latency\_increase\_plus1 − 1

When bmsps\_max\_latency\_increase\_plus1 is equal to 0, no corresponding limit is expressed. The value of bmsps\_max\_latency\_increase\_plus1 shall be in the range of 0 to 2^32 − 2, inclusive.

**bmsps\_long\_term\_ref\_mesh\_frames\_flag** equal to 0 specifies that no long-term reference basemesh frame is used for inter prediction of any coded basemesh frame in the CBMS. bmsps\_long\_term\_ref\_mesh\_frames\_flag equal to 1 specifies that long term reference mesh frames may be used for inter prediction of one or more coded mesh frames in the CBMS.

**bmsps\_num\_ref\_mesh\_frame\_lists\_in\_bmsps** specifies the number of the bmesh\_ref\_list\_struct( rlsIdx ) syntax structures included in the basemesh sequence parameter set. The value of bmsps\_num\_ref\_mesh\_frame\_lists\_in\_bmsps shall be in the range of 0 to 64, inclusive.

NOTE – A decoder allocates memory for a total number of bmesh\_ref\_list\_struct( rlsIdx ) syntax structures equal to (bmsps\_num\_ref\_mesh\_frame\_lists\_in\_bmsps + 1) since there can be one bmesh\_ref\_list\_struct( rlsIdx ) syntax structure directly signalled in the base submesh headers of the current base submesh.

**bmsps\_inter\_mesh\_motion\_group\_size\_minus1** plus 1 specifies the size of vertices grouping in motion vector coding. bmsps\_inter\_mesh\_motion\_group\_size\_minus1 shall be in the range of 0 to 255, inclusive.

**bmsps\_inter\_mesh\_max\_num\_neighbours\_minus1** plus 1 specifies the maximum number of vertex neighbours to use in the calculation of motion vector predictor. bmsps\_inter\_mesh\_max\_num\_neighbours\_minus1 shall be in the range of 0 to 7, inclusive. bmsps\_extension\_present\_flag equal to 1 specifies that bmsps\_extension\_count\_minus1 is present in bmesh\_sequence\_parameter\_set\_rbsp( ) syntax structure.

**bmsps\_extension\_count** specifies the number of BMSPS extensions present in the bmesh\_sequence\_parameter\_set( ) syntax structure. When not present bmsps\_extension\_count is inferred to be equal to 0. When bmsps\_extension\_count is equal to 0, then BmspsExtensionsLength, which specifies the cumulative length in bytes of all extensions that would have otherwise followed this syntax element, is equal to 0.

**bmsps\_codec\_specific\_parameters\_present\_flag** equal to 1 specifies that additional mesh codec prefix data is present in the base mesh sequence parameter set. bmsps\_codec\_specific\_parameters\_present\_flag equal to 0 specifies that additional mesh codec prefix data is not present in the basemesh sequence parameter set.

**bmsps\_mesh\_codec\_prefix\_length\_minus1** plus 1, when present, specifies the length in bytes, MeshCodecPrefixLength, that follow this syntax element. MeshCodecPrefixLength is computed as follows:

MeshCodecPrefixLength = bmsps\_mesh\_codec\_prefix\_length\_minus1 + 1

**bmsps\_mesh\_codec\_prefix\_data\_byte**[ i ] contains the i-th byte of MeshCodecPrefixData. The first byte, i equal 0, contains most significant, left-most byte of MeshCodecPrefixData. MeshCodecPrefixData contains MeshCodecPrefixLength bytes.

The basemesh decoder shall concatenate data contained in MeshCodecPrefixData and data contained in associated bm\_intra\_submesh\_unit () syntax element and pass the concatenated data to the mesh decoder to decode the sub-mesh of type I\_SUBMESH.

**bmsps\_extensions\_length\_minus1**, when present, specifies the cumulative length in bytes, BmspsExtensionsLength, for all extensions that follow this syntax element. BmspsExtensionsLength is computed as follows:

if( bmsps\_extension\_count == 0 )  
 BmspsExtensionsLength = 0  
 else  
 BmspsExtensionsLength = bmsps\_extensions\_length\_minus1 + 1

It is a requirement, when bmsps\_extension\_count is not equal to 0, that BmspsExtensionsLength is equal to 3 \* bmsps\_extension\_count plus the sum of all bmsps\_extension\_length[ i ].

**bmsps\_extension\_type**[ i ] indicates the BMSPS extension type for the extension with index i as specified in [CrossReferenceXXX]. Values indicated as reserved are reserved for future use by ISO/IEC and shall not be present in bitstreams conforming to this version of this document. Decoders conforming to this version of this document should ignore such reserved extensions. It is a requirement that a particular bmsps\_extension\_type[ i ] value shall only be present once in an entire BMSPS, while the order of extensions does not matter.

**bmsps\_extension\_length**[ i ] specifies the number of bytes used to represent the payload size of the syntax structure of the associated extension with index i. If bmsps\_extension\_length[ i ] is equal to 0, no extension payload is present for the extension with index i. Otherwise, the extension with index i shall have a payload size in bits in the range of 8 \* ( bmsps\_extension\_length[ i ] – 1 ) + 1 to 8 \* bmsps\_extension\_length[ i ], inclusive.

* + - * 1. Basemesh SPS extension semantics

**bmsps\_extension\_data\_byte** may have any value.

* + - * 1. Basemesh Profile, tier, and level semantics

**bmptl\_tier\_flag** specifies the tier context for the interpretation of bmptl\_level\_idc as specified in Annex H.11.

**bmptl\_profile\_codec\_group\_idc** specifies the codec group profile component to which the CBMS conforms as specified in Annex H.11. Bitstreams shall not contain values of bmptl\_profile\_codec\_group\_idc other than those specified in Annex H.11. Other values of bmptl\_profile\_codec\_group\_idc are reserved for future use by ISO/IEC.

**bmptl\_profile\_toolset\_idc** specifies the toolset combination profile component to which the CBMS conforms as specified in Annex H.11. Bitstreams shall not contain values of bmptl\_profile\_toolset\_idc other than those specified in Annex H.11. Other values of bmptl\_profile\_toolset\_idc are reserved for future use by ISO/IEC.

**bmptl\_reserved\_zero\_32bits**, when present, shall be equal to 0 in bitstreams conforming to this version of this document. Other values for bmptl\_reserved\_zero\_32bits are reserved for future use by ISO/IEC. Decoders shall ignore the value of bmptl\_reserved\_zero\_32bits.

**bmptl\_level\_idc** indicates a level to which the CBMS conforms as specified in Annex H.11. Bitstreams shall not contain values of bmptl\_level\_idc other than those specified in Annex H.11. Other values of bmptl\_level\_idc are reserved for future use by ISO/IEC.

**bmptl\_num\_sub\_profiles** specifies the number of the bmptl\_sub\_profile\_idc[ i ] syntax elements.

**bmptl\_extended\_sub\_profile\_flag** equal to 1 specifies that the bmptl\_sub\_profile\_idc[ i ] syntax elements, if present, should be represented using 64 bits. bmptl\_extended\_sub\_profile\_flag equal to 0 specifies that the bmptl\_sub\_profile\_idc[ i ] syntax elements, if present, should be represented using 32 bits.

**bmptl\_sub\_profile\_idc**[ i ] specifies the i-th interoperability metadata registered as specified by Rec. ITU-T T.35, the content of which is not specified in this document. The number of bits used to represent bmptl\_sub\_profile\_idc[ i ] is equal to (bmptl\_extended\_sub\_profile\_flag == 0 ? 32 : 64).

**bmptl\_toolset\_constraints\_present\_flag** equal to 1 specifies that an additional syntax structure, bmesh\_profile\_toolset\_constraints\_information( ), is present in the bitstream. bmptl\_toolset\_constraints\_present\_flag equal to 0 specifies that the syntax structure bmesh\_profile\_toolset\_constraints\_information( ) is not present.

* + - * 1. Basemesh Profile toolset constraints information semantics

**bmptc\_one\_mesh\_frame\_only\_flag,** when present, has semantics specified in Annex H.11. where the profile indicated by bmptl\_profile\_toolset\_idc is a profile specified in Annex H.11. When not present, ptc\_one\_mesh\_frame\_only\_flag is inferred to be equal to 0.

**bmptc\_intra\_frames\_only\_flag** equal to 1 specifies that the bitstream contains only sdu\_intra\_submesh\_unit( ). When not present, bmptc\_intra\_frames\_only\_flag is inferred to be equal to 0.

**bmptc\_motion\_vector\_derivation\_disable\_flag** equal to 1 specifies the motion vector derivation is disabled. When bmptc\_motion\_vector\_derivation\_disable\_flag equal to 1, bmidu\_mv\_signalled\_flag\_count[ i ] is always 0 for any possible submesh with submeshID i. When bmptc\_motion\_vector\_derivation\_disable\_flag is not present, bmptc\_motion\_vector\_derivation\_disable\_flag is inferred to be equal to 0.

**bmptc\_reserved\_zero\_5bits** shall be equal to 0 in bitstreams conforming to this version of this document. Other values of bmptc\_reserved\_zero\_5bits are reserved for future use by ISO/IEC and shall not be present in bitstreams conforming to this version of this document. Decoders conforming to this version of this document shall ignore values of bmptc\_reserved\_zero\_5bitsother than 0.

**bmptc\_num\_reserved\_constraint\_bytes** specifies the number of the reserved constraint bytes. The value of bmptc\_num\_reserved\_constraint\_bytes shall be 0 in bitstreams conforming to this version of this document. Other values of bmptc\_num\_reserved\_constraint\_bytes are reserved for future use by ISO/IEC and shall not be present in bitstreams conforming to this version of this document. Decoders conforming to this version of this document shall ignore values of bmptc\_num\_reserved\_constraint\_bytes other than 0.

**bmptc\_reserved\_constraint\_byte**[ i ] may have any value. Its presence and value do not affect decoder conformance to profiles specified in this version of this document. Decoders conforming to this version of this document shall ignore the values of all the bmptc\_reserved\_constraint\_byte[ i ] syntax elements.

* + - 1. Basemesh frame parameter set RBSP semantics
         1. General basemesh frame parameter set RBSP semantics

**bmfps\_sequence\_parameter\_set\_id** specifies the value of bmsps\_sequence\_parameter\_set\_id for the active basemesh sequence parameter set.

**bmfps\_parameter\_set\_id** identifies the basemesh frame parameter set for reference by other syntax elements.

**bmfps\_output\_flag\_present\_flag** equal to 1 indicates that the bmsh\_output\_flag syntax element is present in the associated submesh headers. bfps\_output\_flag\_present\_flag equal to 0 indicates that the bmsh\_output\_flag syntax element is not present in the associated submesh headers.

**bmfps\_num\_ref\_idx\_default\_active\_minus1** plus 1 specifies the inferred value of the variable BmNumRefIdxActive for the submesh with bmsh\_num\_ref\_idx\_active\_override\_flag equal to 0. The value of bfps\_num\_ref\_idx\_default\_active\_minus1 shall be in the range of 0 to 14, inclusive.

**bmfps\_additional\_lt\_mfoc\_lsb\_len** specifies the value of the variable MaxLtMeshFrmOrderCntLsb that is used in the decoding process for reference mesh frame lists as follows:

MaxLtMeshFrmOrderCntLsb =  
  1 << ( Log2MaxMeshFrmOrderCntLsb + bfps\_additional\_lt\_mfoc\_lsb\_len)

The value of bfps\_additional\_lt\_mfoc\_lsb\_len shall be in the range of 0 to 32 – Log2MaxMeshFrmOrderCntLsb, inclusive.

When bmsps\_long\_term\_ref\_mesh\_frames\_flag is equal to 0, the value of bmfps\_additional\_lt\_mfoc\_lsb\_len shall be equal to 0.

**bmfps\_extension\_present\_flag** equal to 1 specifies that the syntax element bmfps\_extension\_8bits is present in bmesh\_frame\_parameter\_set\_rbsp( ) syntax structure. bmfps\_extension\_present\_flag equal to 0 specifies that the syntax element bmfps\_extension\_8bits is not present in bmesh\_frame\_parameter\_set\_rbsp( ) syntax structure. The value of bmfps\_extension\_present\_flag shall be 0 in this version of this document

**bmfps\_extension\_8bits** equal to 0 specifies that no bfps\_extension\_data\_flag syntax elements are present in bmesh\_frame\_parameter\_set\_rbsp( ) syntax structure. When present, bfps\_extension\_8bits shall be equal to 0 in bitstreams conforming to this version of this document. Values of bfps\_extension\_8bits not equal to 0 are reserved for future use by ISO/IEC. Decoders shall allow the value of bfps\_extension\_8bits to be not equal to 0 and shall ignore all bfps\_extension\_data\_flag syntax elements in bmesh\_frame\_parameter\_set\_rbsp( ) syntax structure. When not present, the value of bfps\_extension\_8bits is inferred to be equal to 0.

**bmfps\_extension\_data\_flag** may have any value. Its presence and value do not affect decoder conformance to profiles specified in this version of this document. Decoders conforming to this version of this document shall ignore all bfps\_extension\_data\_flag syntax elements.

* + - * 1. Basemesh submesh information

**bmsi\_use\_single\_mesh\_flag** equal to 1 specifies that there is only one submesh in each mesh frame referring to the BFPS. bmsi\_use\_single\_mesh\_falg equal to 0 specifies that there may be more than one submeshes in each mesh frame referring to the BFPS.

**bmsi\_num\_submeshes\_minus2** plus 2 specifies the number of submeshes in each mesh frame referring to the BFPS. The value of bmsi\_num\_submeshes\_minus1shall be in the range of 0 to 62, inclusive.

The variable NumBmeshSubMeshes is computed as follows:

NumBmeshSubMeshes = bmsi\_num\_submeshes\_minus2 + 2

When bmsi\_num\_submeshes\_minus2 is not present and bmsi\_use\_single\_mesh\_flag is equal to 1, NumBmeshSubMeshes value is inferred to be equal to 1.

**bmsi\_signalled\_submesh\_id\_flag** equal to 1 specifies that the submesh ID for each mesh frame is signalled. bmsi\_signalled\_submesh\_id\_flag equal to 0 specifies that submesh IDs are not signalled.

**delta\_bmsi\_signalled\_submesh\_id\_delta\_length** plus 1 specifies the number of bits used to represent the syntax element bmsi\_submesh\_id[ i ] when present, and the syntax element submesh\_id in a submesh header. The value of delta\_bmsi\_signalled\_submesh\_id\_delta\_length shall be in the range of 0 to 15, inclusive. When not present, its value is inferred to be equal to Ceil( Log2( NumBmeshSubMeshes ) ) – 1.

**bmsi\_submesh\_id**[ i ] specifies the submesh ID of the i-th submesh. The length of the bmsi\_submesh\_id[ i ] syntax element is delta\_length + Ceil( Log2( bmsi\_num\_submeshesbmsi\_signalled\_submesh\_id\_delta\_length + Ceil( Log2( bmsi\_num\_submeshes\_minus1 + 1 ) ) bits. When not present, the value of bmsi\_submesh\_id[ i ] is inferred to be equal to i, for each i in the range of 0 to NumBmeshSubMeshes - 1, inclusive. It is a requirement of bitstream conformance that bmsi\_submesh\_id[ i ] shall not be equal to bmsi\_submesh\_id[ j ] for all i != j

The variable FirstSubmeshID is computed as follows:

FirstSubmeshID= bmsi\_submesh\_id[ 0 ]  
 for ( i = 1; i < NumBmeshSubMeshes; i++ )  
 FirstSubmeshID = Min(FirstSubmeshID, bmsi\_submesh\_id[ i ])

* + - 1. Basemesh submesh layer RBSP sementics
      2. Basemesh submesh header semantics

When present, the value of the submesh header syntax elements bmsh\_basemesh\_frame\_parameter\_set\_id, bmsh\_mesh\_output\_flag, bmsh\_no\_output\_of\_prior\_mesh\_frames\_flag, and bmsh\_mesh\_frm\_order\_cnt\_lsb, shall be the same in all submesh headers of a coded basemesh frame.

**bmsh\_no\_output\_of\_prior\_mesh\_frames\_flag** affects the output of previously-decoded mesh frames in the DBMB after the decoding of an mesh frame in a CS AU that is not the first AU in the bitstream as specified in Annex [CrossReferenceXXX]. When bmsh\_no\_output\_of\_prior\_mesh\_frames\_flag is not present, its value is inferred to be equal to 0.

It is a requirement of bitstream conformance that the value of bmsh\_no\_output\_of\_prior\_mesh\_frames\_flag shall be the same for all submeshes of the basemesh frame in an AU.

The value of bmsh\_no\_output\_of\_prior\_mesh\_frames\_flag in the submesh headers is also referred to as the output\_of\_prior\_mesh\_frames\_flag value of the AU.

**bmsh\_basemesh\_frame\_parameter\_set\_id** specifies the value of bfps\_basemesh\_frame\_parameter\_set\_id for the active basemesh frame parameter set for the current submesh.

**bmsh\_id** specifies the submesh ID associated with the current submesh. When not present, the value of bmsh\_idis inferred to be equal to 0.

The following applies:

* The length of bmsh\_id is delta\_ + Ceil(Log2( bmsi\_num\_submeshesbmsi\_signalled\_submesh\_id\_delta\_length + Ceil(Log2( bmsi\_num\_submeshes\_minus1 + 1 ) ) bits.
* The value of bmsh\_id shall be in the range of values specified by the array BaseMeshSubmeshIndexToID [ i ], for i in the range from 0 to NumBmeshSubMeshes - 1, inclusive.

It is a requirement of bitstream conformance that the following constraints apply:

* The value of bmsh\_id shall not be equal to the value of bmsh\_id of any other basemesh unit of the same basemesh frame.
* The submesh order of a basemesh frame shall be in increasing order of their bmsh\_id values.

**bmsh\_type** specifies the coding type of the current submesh according to Table H-4. The value of bmsh\_type shall be equal to 0, 1, or 2 in bitstreams conforming to this version of this document. Other values of bmsh\_type are reserved for future use by ISO/IEC. Decoders conforming to this version of this document shall ignore reserved values of bmsh\_type.

Table H-4 – Name association to bmsh\_type

|  |  |
| --- | --- |
| bmsh\_type | Name of bmsh\_type |
| 0 | P\_SUBMESH |
| 1 | I\_SUBMESH |
| 2 | SKIP\_SUBMESH |
| 3 - .. | RESERVED |

**bmsh\_mesh\_output\_flag** affects the decoded mesh output and removal processes as specified in [CrossReferenceXXX]. When bmsh\_mesh\_output\_flag is not present, it is inferred to be equal to 1.

**bmsh\_mesh\_frm\_order\_cnt\_lsb** specifies the mesh frame order count modulo MaxMeshFrmOrderCntLsb for the current submesh. The length of the bmsh\_mesh\_frm\_order\_cnt\_lsb syntax element is equal to Log2MaxMeshFrmOrderCntLsb bits. The value of the bmsh\_mesh\_frm\_order\_cnt\_lsb shall be in the range of 0 to MaxMeshFrmOrderCntLsb - 1, inclusive.

**bmsh\_ref\_mesh\_frame\_list\_bmsps\_flag** equal to 1 specifies that the reference bmesh frame list of the current submesh is derived based on one of the bmesh\_ref\_list\_struct( rlsIdx ) syntax structures in the active BMSPS. bmsh\_ref\_mesh\_frame\_list\_bmsps\_flag equal to 0 specifies that the reference bmesh frame list of the current submesh is derived based on the bmesh\_ref\_list\_struct( rlsIdx ) syntax structure that is directly included in the submesh header of the current submesh. When bmsps\_num\_ref\_mesh\_frame\_lists\_in\_bmsps is equal to 0, the value of bmsh\_ref\_mesh\_frame\_list\_bmsps\_flag is inferred to be equal to 0.

**bmsh\_ref\_mesh\_frame\_list\_idx** specifies the index, into the list of the bmesh\_ref\_list\_struct( rlsIdx ) syntax structures included in the active BMSPS, of the bmesh\_ref\_list\_struct( rlsIdx ) syntax structure that is used for derivation of the reference basemesh frame list for the current submesh. The syntax element bmsh\_ref\_mesh\_frame\_list\_idx is represented by Ceil( Log2( bmsps\_num\_ref\_mesh\_frame\_lists\_in\_bmsps ) ) bits. When not present, the value of bmsh\_ref\_mesh\_frame\_list\_idx is inferred to be equal to 0. The value of bmsh\_ref\_mesh\_frame\_list\_idx shall be in the range of 0 to bmsps\_num\_ref\_mesh\_frame\_lists\_in\_bmsps - 1, inclusive. When bmsh\_ref\_mesh\_frame\_list\_bmsps\_flag is equal to 1 and bmsps\_num\_ref\_mesh\_frame\_lists\_in\_bmsps is equal to 1, the value of bmsh\_ref\_mesh\_frame\_list\_idx is inferred to be equal to 0.

The variable BmRlsIdx for the current submesh is derived as follows:

BmRlsIdx = bmsh\_ref\_mesh\_frame\_list\_bmsps\_flag ?  
 bmsh\_ref\_mesh\_frame\_list\_idx : bmsps\_num\_ref\_mesh\_frame\_lists\_in\_bmsps

**bmsh\_additional\_mfoc\_lsb\_present\_flag**[ j ] equal to 1 specifies that bmsh\_additional\_mfoc\_lsb\_val[ j ] is present for the current submesh. bmsh\_additional\_mfoc\_lsb\_present\_flag[ j ] equal to 0 specifies that bmsh\_additional\_mfoc\_lsb\_val[ j ] is not present.

**bmsh\_additional\_mfoc\_lsb\_val**[ j ] specifies the value of FullMeshFrmOrderCntLsbLt[ BmRlsIdx ][ j ] for the current submesh as follows:

FullMeshFrmOrderCntLsbLt[ BmRlsIdx ][ j ] = bmsh\_additional\_mfoc\_lsb\_val[ j ] \*   
 MaxMeshFrmOrderCntLsb +mfoc\_lsb\_lt[ BmRlsIdx ][ j ]

The syntax element bmsh\_additional\_mfoc\_lsb\_val[ j ] is represented by bfps\_additional\_lt\_mfoc\_lsb\_len bits. When not present, the value of bmsh\_additional\_mfoc\_lsb\_val[ j ] is inferred to be equal to 0.

**bmsh\_num\_ref\_idx\_active\_override\_flag** equal to 1 specifies that the syntax element bmsh\_num\_ref\_idx\_active\_minus1 is present for the current submesh. bmsh\_num\_ref\_idx\_active\_override\_flag equal to 0 specifies that the syntax element bmsh\_num\_ref\_idx\_active\_minus1 is not present. If bmsh\_num\_ref\_idx\_active\_override\_flag is not present, its value shall be inferred to be equal to 0.

**bmsh\_num\_ref\_idx\_active\_minus1** is used for the derivation of the variable BmNumRefIdxActive as specified by Equation [CrossReferenceXXX] for the current submesh. The value of bmsh\_num\_ref\_idx\_active\_minus1 shall be in the range of 0 to 14, inclusive.

When the current submesh is a P\_SUBMESH submesh, bmsh\_num\_ref\_idx\_active\_override\_flag is equal to 1, and bmsh\_num\_ref\_idx\_active\_minus1 is not present, bmsh\_num\_ref\_idx\_active\_minus1 is inferred to be equal to 0.

The variable BmNumRefIdxActive is derived as follows:

if( bmsh\_type == P\_SUBMESH || bmsh\_type == SKIP\_SUBMESH ) {  
 if( bmsh\_num\_ref\_idx\_active\_override\_flag == 1 )  
 BmNumRefIdxActive = bmsh\_num\_ref\_idx\_active\_minus1 + 1  
 else {  
 if( num\_ref\_entries[ BmRlsIdx ] >= bfps\_num\_ref\_idx\_default\_active\_minus1 + 1 )  
 BmNumRefIdxActive = bfps\_num\_ref\_idx\_default\_active\_minus1 + 1  
 else  
 BmNumRefIdxActive = num\_ref\_entries[ BmRlsIdx ]  
 }  
 }  
 else   
 BmNumRefIdxActive = 0

BmNumRefIdxActive minus 1 specifies the maximum value of the mesh reference frame index that may be used to decode the current submesh.

* + - 1. Reference list structure semantics

**num\_ref\_entries**[ rlsIdx ] specifies the number of entries in the bmesh\_ref\_list\_struct( rlsIdx ) syntax structure, where rlsIdx is the index of a basemesh frame reference list. For P\_SUBMESH and SKIP\_SUBMESH, the value of num\_ref\_entries[ rlsIdx ] shall be in the range of 1 to bmsps\_max\_dec\_mesh\_frame\_buffering\_minus1 + 1. Otherwise, the value of num\_ref\_entries[ rlsIdx ] shall be in the range of 0 to bmsps\_max\_dec\_mesh\_frame\_buffering\_minus1 + 1.

**st\_ref\_mesh\_frame\_flag**[ rlsIdx ][ i ] equal to 1 specifies that the i-th entry in the bmesh\_ref\_list\_struct( rlsIdx ) syntax structure is a short term reference basemesh frame entry. st\_ref\_mesh\_frame\_flag[ rlsIdx ][ i ] equal to 0 specifies that the i-th entry in the ref\_list\_struct( rlsIdx ) syntax structure is a long term reference basemesh frame entry. When not present, the value of st\_ref\_mesh\_frame\_flag[ rlsIdx ][ i ] is inferred to be equal to 1.

The variable NumLtrMeshFrmEntries[ rlsIdx ] is derived as follows:

NumLtrMeshFrmEntries[ rlsIdx ] = 0  
 for( i = 0; i < num\_ref\_entries[ rlsIdx ]; i++ )  
 if( !st\_ref\_mesh\_frame\_flag[ rlsIdx ][ i ] )   
 NumLtrMeshFrmEntries[ rlsIdx ]++

**abs\_delta\_mfoc\_st**[ rlsIdx ][ i ], when the i-th entry is the first short term reference basemesh frame entry in bmesh\_ref\_list\_struct( rlsIdx ) syntax structure, specifies the absolute difference between the basemesh frame order count values of the current submesh and the basemesh frame referred to by the i-th entry, or, when the i-th entry is a short term reference basemesh frame entry but not the first short term reference basemesh frame entry in the bmesh\_ref\_list\_struct( rlsIdx ) syntax structure, specifies the absolute difference between the basemesh frame order count values of the basemesh frames referred to by the i-th entry and by the previous short term reference basemesh frame entry in the bmesh\_ref\_list\_struct( rlsIdx ) syntax structure. The value of abs\_delta\_mfoc\_st[ rlsIdx ][ i ] shall be in the range of 0 to 215 - 1, inclusive.

**straf\_entry\_sign\_flag**[ rlsIdx ][ i ] equal to 1 specifies that the i-th entry in the syntax structure bmesh\_ref\_list\_struct( rlsIdx ) has a value greater than or equal to 0. straf\_entry\_sign\_flag[ rlsIdx ][ i ] equal to 0 specifies that the i-th entry in the syntax structure bmesh\_ref\_list\_struct( rlsIdx ) has a value less than 0. When not present, the value of straf\_entry\_sign\_flag[ rlsIdx ][ i ] is inferred to be equal to 1.

The list DeltaMfocSt[ rlsIdx ][ i ] is derived as follows:

for( i = 0; i < num\_ref\_entries[ rlsIdx ]; i++ )  
 if( st\_ref\_mesh\_frame\_flag[ rlsIdx ][ i ] )   
 DeltaMfocSt[ rlsIdx ][ i ] =  
 ( 2 \* straf\_entry\_sign\_flag[ rlsIdx ][ i ] – 1 ) \* abs\_delta\_mfoc\_st[ rlsIdx ][ i ]   
 else  
 DeltaMfocSt[ rlsIdx ][ i ] = 0

**mfoc\_lsb\_lt**[ rlsIdx ][ i ] specifies the value of the mesh frame order count modulo MaxMeshFrmOrderCntLsb of the mesh frame referred to by the i-th entry in the bmesh\_ref\_list\_struct( rlsIdx ) syntax structure. The length of the mfoc\_lsb\_lt[ rlsIdx ][ i ] syntax element is Log2MaxMeshFrmOrderCntLsb bits.

* + - 1. Basemesh submesh data unit semantics

None

* + - 1. Basemesh intra submesh data unit semantics

None

* + - 1. Basemesh inter submesh data unit semantics
         1. Basemesh inter submesh data unit default semantics

**bm\_vertex\_count**[ submeshID ] indicates the number of vertices of submesh with submesh Id, submeshID.

**bmidu\_mv\_signalled\_flag\_count**[ submeshID ] specifies the number of bmidu\_mv\_signalled\_flag in the current submesh, with submesh ID equal to submeshID.

It is a requirement of bitstream conformance that if bmidu\_mv\_signalled flag\_count[ submeshID ] is larger than 0 for a submesh with submesh ID equal to submeshID, mesh\_deduplicate\_method, if present in the corresponding intra submesh data unit, shall be equal to MESH\_POSITION\_DEDUP\_NONE.

**bmidu\_mv\_signalled\_flag**[ submeshID ][ d ] indicates a motion vector for the vertex with index d, whose output of findIndexInArray() is not -1, is present in the bitstream. When bmidu\_mv\_signalled\_flag[ submeshID ][ d ] is not present in the bitstream, bmidu\_mv\_signalled\_flag[ submeshID ][ d ] is inferred as 1. bmidu\_mv\_signalled\_flag[ submeshID ][ d ] is used to derive BmiduMvFlag[ submeshID][ v ] variable, which indicates that a motion vector for the vertex with index v is present in the bitstream.

The BmiduMvFlag[ submeshID][ v ] variable is derived as follows:

d = 0  
 for( v = 0; v < vertexCount; v++ ) {  
 vRef = findIndexInArray( referenceSubmeshVertexPositions[ v ],  
 referenceSubmeshVertexPositions, v – 1 )  
 if( vRef != -1 ) {  
 BmiduMvFlag[ subMeshID ][ v ] = bmidu\_mv\_signalled\_flag[ subMeshID ][ d ]  
 d++  
 }else {  
 BmiduMvFlag[ subMeshID ][ v ] = 1  
 }  
 }

where findIndexInArray( ) function is defined in subclause H.5

**bmidu\_skip\_group\_flag**[ submeshID ][ g ] equal to 1 specifies a motion vector associated with vertices in the group with index g of the current submesh, with submesh ID equal to submeshID, is skipped. bmidu\_skip\_group\_flag [ submeshID ][ g ] equal to 0 specifies a motion vector associated with vertices in the group with index g of the current submesh, with submesh ID equal to submeshID, is not skipped. When not present, bmidu\_skip\_group\_flag[ submeshID ][ g ] is inferred to be equal to 0.

**bmidu\_skip\_group\_comp\_flag**[ subMeshID ][ g ][ k ] equal to 1 specifies the k-th motion vector component, associated with vertices in the group with index g of the current submesh, with submesh ID equal to subMeshID, is inferred to be equal to 0. When bmidu\_skip\_group\_comp\_flag[ subMeshID ][g ][ k ] is not present, bmidu\_skip\_group\_comp\_flag[ subMeshID ][ g ][ k ] is inferred to be equal to 0..

When bmidu\_skip\_group\_flag[ submeshID ][ g ] is equal to 0 and when the value of k is equal to 2 and bmidu\_skip\_group\_comp\_flag[ subMeshID ][ g ][ 0 ] and bmidu\_skip\_group\_comp\_flag[ subMeshID ][ g ][ 1 ] are both equal to 1, bmidu\_skip\_group\_comp\_flag[ subMeshID ][ g ][ 2 ] shall be equal to 0.

**bmidu\_mv\_pred\_mode\_group**[ subMeshID ][ g ][ k ] specifies the method used to predict the motion vector of the k-th motion vector component, associated with vertices in the group with index g of the current submesh, with submesh ID equal to subMeshID. The value of bmidu\_mv\_pred\_mode\_group[ subMeshID ][ g ][ k ] shall be in the range of 0 to 2

Table H-5 – Name association to bmidu\_mv\_pred\_mode\_group

|  |  |
| --- | --- |
| bmsh\_type | Name of bmidu\_mv\_pred\_mode\_group |
| 0 | MV\_PRED\_NONE |
| 1 | MV\_PRED\_NEIGHBOUR |
| 2 | MV\_DERIVED |

**bmidu\_mv\_residual\_abs\_gt0**[ submeshID ][ v ][ k ] equal to 1 specifies the k-th component of the motion vector prediction residual associated with the vertex with index v of the current submesh, with submesh ID equal to submeshID has a value higher than 0.bmidu\_mv\_residual\_abs\_gt0[ subMeshID ][ v ][ k ] equal to 0  indicates the k-th component of the motion vector prediction residual associated with the vertex with index v of the current submesh, with submesh ID equal to subMeshID, has a value equal to 0.

**bmidu\_mv\_residual\_sign**[ submeshID ][ v ][ k ] equal to 1 specifies whether the k-th component of the motion vector prediction residual associated with the vertex with index v of the current submesh, with submesh ID equal to submeshID has a value greater or equal to 0. bmidu\_mv\_residual\_sign[ submeshID ][ v ][ k ] equal to 0 specifies the k-th component of the motion vector prediction residual associated with the vertex with index v of the current submesh, with submesh ID equal to submeshID has a value less than 0, When bmidu\_mv\_residual\_sign[ v ][ k ] is not present it shall be inferred to be equal to 1.

**bmidu\_mv\_residual\_abs\_gt1**[ submeshID ][ v ][ k ] specifies whether the k-th component of the motion vector prediction residual associated with the vertex with index v of the current submesh, with submesh ID equal to submeshID has an absolute value higher than one (when 1), or not (when 0). If bmidu\_mv\_residual\_abs\_gt1[ v ][ k ] is not present it shall be inferred to be equal to 0.

**bmidu\_mv\_residual\_abs\_rem**[ submeshID ][ v ][ k ] specifies the absolute value of the k-th component of the motion vector prediction residual associated with the vertex with index v of the current submesh, with submesh ID equal to submeshID. When bmidu\_mv\_residual\_abs\_rem[ v ][ k ] is not present it shall be inferred to be equal to 0.

The k-th component of the motion vector prediction residual VertexMotionVectorResiduals[ v ][ k ] associated with the vertex with index v of the current submesh, with submesh ID equal to submeshID is computed as follows:

VertexMotionVectorResiduals[ v ][ k ] = bmidu\_mv\_residual\_sign[ v ][ k ] ? 1 : -1) \*  
 (bmidu\_mv\_residual\_abs\_gt0[ v ][ k ] + bmidu\_mv\_residual\_abs\_gt1[ v ][ k ] +   
 bmidu\_mv\_residual\_abs\_rem[ v ][ k ])

* + - 1. Embedded external data unit semantics

**embedded\_external\_data\_bit** stores a value which corresponds to the coded-mesh data. The syntax and semantics of the sequence of bits is specified by the basemesh codec profile group signaled through bmptl\_profile\_codec\_group\_idc or respective bmsps\_intra\_mesh\_codec\_id/bmsps\_inter\_mesh\_codec\_id signaled in the basemesh sequence parameter set.

* 1. Decoding process

Input to this decoding process is the basemesh sub-bitstream corresponding to the basemesh NAL units of the V3C basemesh component which is determined through either examining if vuh\_unit\_type is equal to V3C\_BMD or through external means if the V3C unit header is unavailable, is as specified in subclause H.9.1.

* + 1. General decoding process

Input to this process is a Basemesh bitstream.

Outputs of this process are:

* NumDecMeshFrames, indicating the number of decoded mesh frames,
* the 1D arrays of size NumDecMeshFrames , DecSubmeshCountPerFrame, indicating the number of submeshes,
* for each decoded mesh frame, the corresponding upper-case variables from subclauses H.8, and
* For each decoded submesh frame, an array of elements called DecSubmeshesFrame of size DecSubmeshCountPerFrame is defined. Each element of the DecSubmeshesFrame is of structure data type, denoted as structs, contains the following members:
  + - submeshId, indicating the id of the current submesh
    - verCoordCount, indicating the number of vertex coordinates,
    - faceCount, indicating the number of faces,
    - a 2D array verCoords, indicating vertex coordinate values, of size verCoordCount × 3,
    - a 2D array verCoordFaces, indicating vertex coordinates connectivity, of size faceCount × 3
    - an array of structs of size bmsps\_mesh\_attribute\_count, strAttributes, indicating the decoded attributes. For attrIdx-th element of the strAttributes has the following members:
    - attrTypeId equal to bmsps\_mesh\_attribute\_type\_id[ attrIdx ], indicating the type of the attribute
    - attrBitDepth equal to bmsps\_attribute\_bit\_depth\_minus1[ attrIdx ] + 1
    - attrValueCount, indicating the number of values of the attribute,
    - attrFaceCount, indicating the number of faces of the attribute,
    - attrValueDimension, indicating the dimension of values of the attribute,
    - a 2D array attrValues, indicating values of the attribute, where the first dimension range is 0..attrValueCount - 1, and the second dimension range is 0.. attrValueDimension,
    - attrValuePerVertex, indicating the attributes is per vertex
    - a 2D array attrFacesArray, indicating the connectivity of the attribute, of size attrFaceCount × 3.

If the bitstream is encapsulated using the sample stream format, as defined in Annex D then the composition time can be derived using the mesh frame order count, MFOC, and/or mesh frame timing information, if available. If the bitstream is encapsulated within the ISO media file format, the mesh composition time, DecMeshCompTime, for the bitstream is indicated according to ISO/IEC DIS 23090-10. Otherwise, the mesh composition time is provided through external means and it is outside the scope of this document.

Subclause H.9.2 is repeatedly invoked for each coded mesh frame in the current mesh bitstream to be decoded, in decoding order.

* + 1. Decoding process for a coded mesh frame

The decoding processes specified in this subclause apply to each coded mesh frame with bmesh\_nal\_layer\_id equal to 0, referred to as the current mesh frame.

The decoding process for the current mesh frame takes as inputs the syntax elements and upper-case variables from Clause H.8. When interpreting the semantics of each syntax element in each NAL unit, the term "the bitstream" (or part thereof, e.g., a CBMS of the bitstream) refers to the current mesh component bitstream (or part thereof).

When the current mesh frame is a BLA mesh frame that has bmesh\_nal\_unit\_type equal to BNAL\_BLA\_W\_LP or is a CRA mesh frame, the following applies:

* If some external means not specified in this document is available to set the variable UseAltCabParamsFlag to a value, UseAltCabParamsFlag is set equal to the value provided by the external means.
* Otherwise, the value of UseAltCabParamsFlag is set equal to 0.

When the current mesh frame is an IRAP coded mesh frame, the following applies:

* If the current mesh frame is an IDR mesh frame, a BLA mesh frame, the first mesh frame in the bitstream in decoding order, or the first mesh frame that follows an end of sequence NAL unit in decoding order, the variable NoOutputBeforeRecoveryFlag is set equal to 1.
* Otherwise, if some external means not specified in this document is available to set the variable HandleCraAsBlaFlag to a value for the current mesh frame, the variable HandleCraAsBlaFlag is set equal to the value provided by the external means and the variable NoOutputBeforeRecoveryFlag is set equal to HandleCraAsBlaFlag.
* Otherwise, the variable HandleCraAsBlaFlag is set equal to 0 and the variable NoOutputBeforeRecoveryFlag is set equal to 0.

The decoding process operates as follows for the current mesh frame:

* The decoding of mesh NAL units is specified in subclause H.9.3.
* The processes in subclause H.9.4 specify the following decoding processes using syntax elements in the submesh header layer and above:
  + Variables and functions relating to the mesh frame order count are derived as specified in subclause H.9.4.1.
  + At the beginning of the decoding process for each submesh, the reference mesh frame list construction process specified in subclause H.9.4.3 is invoked for the derivation of the reference mesh frame list, RefMeshFrmList.
  + The reference mesh frame marking process in subclause H.9.4.4 is invoked, wherein reference mesh frames might be marked as "unused for reference" or "used for long-term reference".
  + When the current mesh frame is a BLA mesh frame or is a CRA mesh frame with NoOutputBeforeRecoveryFlag equal to 1, the decoding process for generating unavailable reference mesh frames specified in subclause H.9.4.2 is invoked, which needs to be invoked only for the first submesh of a mesh frame.
  + The variable MeshFrameOutputFlag is set as follows:
    - If the current mesh frame is a RASL mesh frame and NoOutputBeforeRecoveryFlag of the associated IRAP coded mesh frame is equal to 1, MeshFrameOutputFlag is set equal to 0.
    - Otherwise, MeshFrameOutputFlag is set equal to bmsh\_mesh\_output\_flag.
* The processes in subclause H.9.4.5 specify the submesh decoding processes according to the submesh type as follows:
  + Decoding of intra submesh is specified in subclause H.9.4.5.2
  + Decoding of inter submesh is specified in subclause H.9.4.5.3
  + Decoding of skip submesh is specified in subclause H.9.4.5.4
* After all submeshes associated with the current mesh have been decoded, the current mesh is marked as "used for short-term reference".
  + 1. Mesh NAL unit decoding process

Inputs to this process are mesh NAL units of the current mesh frame and their associated non-BMCL NAL units.

Outputs of this process are the parsed RBSP syntax structures encapsulated within the mesh NAL units.

The decoding process for each mesh NAL unit extracts the RBSP syntax structure from the mesh NAL unit and then parses the RBSP syntax structure.

* + 1. Submesh header decoding process
       1. Mesh frame order count derivation process

Output of this process is MeshFrmOrderCntVal, the mesh frame order count of the current submesh.

Mesh frame order counts are used to identify the output order of mesh frames, as well as for decoder conformance checking.

Each coded mesh frame is associated with a mesh frame order count variable, denoted as MeshFrmOrderCntVal.

When the current mesh frame is not an IRAP coded mesh with NoOutputBeforeRecoveryFlag equal to 1, the variables prevMeshFrmOrderCntLsb and prevMeshFrmOrderCntMsb are derived as follows:

* Let prevMeshFrm be the previous mesh frame in decoding order that has BmTemporalID equal to 0 and that is not a RASL, RADL, or SLNR coded mesh.
* The variable prevMeshFrmOrderCntLsb is set equal to the mesh frame order count LSB value, bmsh\_mesh\_frm\_order\_cnt\_lsb, of prevMeshFrm.
* The variable prevMeshFrmOrderCntMsb is set equal to MeshFrmOrderCntMsb of prevMeshFrm.

The variable MeshFrmOrderCntMsb of the current mesh frame is derived as follows:

* If the current mesh is an IRAP coded mesh with NoOutputBeforeRecoveryFlag equal to 1, MeshFrmOrderCntMsb is set equal to 0.
* Otherwise, MeshFrmOrderCntMsb is derived as follows:

if( ( bmsh\_mesh\_frm\_order\_cnt\_lsb< prevMeshFrmOrderCntLsb ) &&  
 ( ( prevMeshFrmOrderCntLsb − bmsh\_mesh\_frm\_order\_cnt\_lsb) >=   
 ( MaxMeshFrmOrderCntLsb / 2 ) ) )  
 MeshFrmOrderCntMsb = prevMeshFrmOrderCntMsb + MaxMeshFrmOrderCntLsb (50)  
 else if( ( bmsh\_mesh\_frm\_order\_cnt\_lsb> prevMeshFrmOrderCntLsb ) &&  
 ( ( bmsh\_mesh\_frm\_order\_cnt\_lsb– prevMeshFrmOrderCntLsb ) >  
 ( MaxMeshFrmOrderCntLsb / 2 ) ) )  
 MeshFrmOrderCntMsb = prevMeshFrmOrderCntMsb – MaxMeshFrmOrderCntLsb (51)  
 else  
 MeshFrmOrderCntMsb = prevMeshFrmOrderCntMsb (52)

MeshFrmOrderCntVal is derived as follows:

MeshFrmOrderCntVal = MeshFrmOrderCntMsb + bmsh\_mesh\_frm\_order\_cnt\_lsb (53)

The value of MeshFrmOrderCntVal shall be in the range of −231 to 231 − 1, inclusive. In one CBMS, the MeshFrmOrderCntVal values for any two coded mesh frames with the same value of bmesh\_nal\_layer\_id shall not be the same.

The function MeshFrmOrderCnt( aFrmX ) is specified as follows:

MeshFrmOrderCnt( aFrmX ) = MeshFrmOrderCntVal of the mesh frame aFrmX (54)

The function DiffMeshFrmOrderCnt( aFrmA, aFrmB ) is specified as follows:

DiffMeshFrmOrderCnt( aFrmA, aFrmB ) =  
 MeshFrmOrderCnt( aFrmA ) – MeshFrmOrderCnt( aFrmB ) (55)

The bitstream shall not contain data that result in values of DiffMeshFrmOrderCnt( aFrmA, aFrmB) used in the decoding process that are not in the range of −215 to 215 − 1, inclusive.

NOTE  – Let X be the current mesh frame and Y and Z be two other mesh frames in the same CBMS, Y and Z are considered to be in the same output order direction from X when both DiffMeshFrmOrderCnt( X, Y ) and DiffMeshFrmOrderCnt( X, Z ) are positive or both are negative.

* + - 1. Decoding process for generating unavailable reference mesh frames
         1. General

This process is invoked once per coded mesh frame when the current mesh frame is a BLA mesh frame or a CRA mesh frame with NoOutputBeforeRecoveryFlag equal to 1.

NOTE – This process is primarily specified only for the specification of syntax constraints for RASL mesh frames. The entire specification of the decoding process for RASL mesh frames associated with an IRAP coded mesh frame that has NoOutputBeforeRecoveryFlag equal to 1 is included herein only for purposes of specifying constraints on the allowed syntax content of such RASL mesh frames. During the decoding process, any RASL mesh frames associated with an IRAP coded mesh frame that has NoOutputBeforeRecoveryFlag equal to 1 can be ignored, as these mesh frames are not specified for output and have no effect on the decoding process of any other mesh frames that are specified for output. However, in HRD operations as specified in Clause H.12, RASL access units could be taken into consideration in derivation of coded basemesh buffer (CBMB) arrival and removal times.

When this process is invoked, the following applies:

* For each RefMeshFrmList[ j ], with j in the range of 0 to num\_ref\_entries[ BmRlsIdx ] − 1, inclusive, that is equal to "no reference mesh frame", a mesh frame is generated as specified in subclause 9.2.4.2.2 and the following applies:
  + The value of bmesh\_nal\_layer\_id for the generated mesh frame is set equal to the bmesh\_nal\_layer\_id value of the current mesh frame.
  + If st\_ref\_mesh\_frame\_flag[ BmRlsIdx ][ j ] is equal to 1 the value of MeshFrmOrderCntVal for the generated mesh frame is set equal to RefMeshFrmList[ j ] and the generated mesh frame is marked as "used for short-term reference".
  + Otherwise, when st\_ref\_mesh\_frame\_flag[ BmRlsIdx ][ j ] is equal to 0 the value of MeshFrmOrderCntVal for the generated mesh frame is set equal to RefMeshFrmList[ j ], the value of bmsh\_mesh\_frm\_order\_cnt\_lsbfor all submeshs in the generated mesh frame is inferred to be equal to ( RefMeshFrmList[ j ] & ( MaxMeshFrmOrderCntLsb – 1 ) ), and the generated mesh frame is marked as "used for long-term reference".
  + The value of MeshFrameOutputFlag for the generated reference mesh frame is set equal to 0.
  + RefMeshFrmList[ j ] is set to be the generated reference mesh frame.
  + The value of BmTemporalID for the generated mesh frame is set equal to the BmTemporalID value of the current mesh frame.
  + The value of bm\_mesh\_frame\_parameter\_set\_id for the generated mesh frame is set equal to bm\_mesh\_frame\_parameter\_set\_id of the current mesh frame.

The value of bmesh\_nal\_layer\_id for the generated mesh frame is set equal to bmesh\_nal\_layer\_id of the current mesh frame.

* + - * 1. Generation of one unavailable mesh frame

When this process is invoked, an unavailable mesh frame is generated as follows:

* For all submeshes that are associated with this mesh frame the following applies:

for( t = 0; t< NumBmeshSubMeshes; t++ ) {  
 submeshID = BaseMeshSubmeshIndexToID( t )  
 bm\_vert\_count[ submeshID ] = 0  
 …  
 }

* + - 1. Reference mesh frame list construction process

This process is invoked at the beginning of the decoding process for each submesh of a mesh frame.

Reference mesh frames are addressed through reference indices. A reference index is an index into a reference mesh frame list (RAFL). When decoding an I\_SUBMESH submesh, no RAFL is used in decoding of the submesh data. When decoding a SKIP\_SUBMESH or a P\_SUBMESH submesh, a single reference mesh frame list, RefMeshFrmList, is used in decoding of the submesh data.

At the beginning of the decoding process for each submesh, the RAFL RefMeshFrmList is derived. The RAFL is used in marking of reference mesh frames as specified in subclause 9.2.4.4 or in decoding of the submesh data.

NOTE  – For an I\_SUBMESH submesh of a mesh frame, RefMeshFrmList could be derived for bitstream conformance checking purposes, but its derivation is not necessary for the decoding of the current mesh frame or for mesh frames that follow the current mesh frame in decoding order.

The reference mesh frame list RefMeshFrmList is constructed as follows:

for( j = 0, mfocBase = MeshFrmOrderCntVal; j < num\_ref\_entries[ BmRlsIdx ]; j++) { (56)  
 if( st\_ref\_mesh\_frame\_flag[ BmRlsIdx ][ j ] ) {  
 RefMeshFrmMfocList[ j ] = mfocBase − DeltaMfocSt[ BmRlsIdx ][ j ]  
 if( there is a reference mesh frame afA in the DBMB with   
 MeshFrmOrderCntVal equal to RefMeshFrmMfocList[ j ] )  
 RefMeshFrmList[ j ] = afA  
 else  
 RefMeshFrmList[ j ] = "no reference mesh frame"  
 mfocBase = RefMeshFrmMfocList[ j ]  
 } else {  
 if( there is a reference mesh frame afA in the DBMB with MeshFrmOrderCntVal & ( MaxLtMeshFrmOrderCntLsb − 1 )  
 equal to FullMeshFrmOrderCntLsbLt[ BmRlsIdx ][ j ] )  
 RefMeshFrmList[ j ] = afA  
 else  
 RefMeshFrmList[ j ] = "no reference mesh frame"  
 }  
 }

The first BmNumRefIdxActive entries in RefMeshFrmList are referred to as the active entries in RefMeshFrmList and the other entries in RefMeshFrmList are referred to as the inactive entries in RefMeshFrmList.

If the current submesh is a SKIP\_SUBMESH, then the array RefTotalNumSubmeshes is set equal to 1 that corresponds to the first entry in the RefMeshFrmList, RefMeshFrmList[ 0 ].

It is a requirement of bitstream conformance that the following constraints apply:

* num\_ref\_entries[ BmRlsIdx ] shall not be less than BmNumRefIdxActive.
* The mesh frame referred to by each active entry in RefMeshFrmList shall be present in the DBMB and shall have BmTemporalID less than or equal to that of the current mesh frame.
* The mesh frame referred to by each entry in RefMeshFrmList shall not be the current mesh frame.
* A short term reference mesh frame entry and a long term reference mesh frame entry in RefMeshFrmList of an submesh shall not refer to the same mesh frame.
* There shall be no long term reference mesh frame entry in RefMeshFrmList for which the difference between the MeshFrmOrderCntVal of the current submesh and the MeshFrmOrderCntVal of the mesh frame referred to by the entry is greater than or equal to 224.
* Let setOfRefMeshFrms be the set of unique mesh frames referred to by all entries in RefMeshFrmList that have the same bmesh\_nal\_layer\_id as the current mesh frame. The number of mesh frames in setOfRefMeshFrms shall be less than or equal to asps\_max\_dec\_mesh\_frame\_buffering\_minus1, and setOfRefMeshFrms shall be the same for all submeshs of a mesh frame.
* The mesh frames referred to by each active entry in RefMeshFrmList shall have exactly the same number of submeshs as the current mesh frame.
* The RefMeshFrmList of all submeshs in the current mesh frame shall contain the same unique mesh frames, without, however, any restrictions in ordering of the reference mesh frames.
* When the current mesh frame, with bmesh\_nal\_layer\_id equal to a particular value layerID, is an IRAP coded mesh, there shall be no mesh referred to by an entry in RefMeshFrmList that precedes, in output order or decoding order, any preceding IRAP coded mesh with bmesh\_nal\_layer\_id equal to layerID in decoding order (when present).
* When the current mesh frame is not a RASL coded mesh associated with a CRA coded mesh with NoOutputBeforeRecoveryFlag equal to 1, there shall be no mesh referred to by an active entry in RefMeshFrmList that was generated by the decoding process for generating unavailable reference mesh frames for the CRA coded mesh associated with the current mesh.
* When the current mesh frame follows an IRAP coded mesh having the same value of bmesh\_nal\_layer\_id in both decoding order and output order, there shall be no mesh frame referred to by an active entry in RefMeshFrmList that precedes that IRAP coded mesh in output order or decoding order.
* When the current mesh frame follows an IRAP coded mesh having the same value of bmesh\_nal\_layer\_id and all the leading mesh frames, if any, associated with that IRAP coded mesh in both decoding order and output order, there shall be no mesh referred to by an entry in RefMeshFrmList that precedes that IRAP coded mesh in output order or decoding order.

When the current mesh frame is a RADL coded mesh, there shall be no active entry in RefMeshFrmList that is a mesh frame that precedes the associated IRAP coded mesh of the RADL coded mesh in decoding order.

* + - 1. Reference mesh frame marking process

This process is invoked once per mesh frame, after decoding of a submesh header and the decoding process for RAFL construction for the mesh frame as specified in subclause H.9.4.4, but prior to the decoding of the submesh data. This process might result in one or more reference mesh frames in the DBMB being marked as "unused for reference" or "used for long-term reference".

A decoded mesh frame in the DBMB can be marked as "unused for reference", "used for short-term reference", or "used for long-term reference", but only one among these three at any given moment during the operation of the decoding process. Assigning one of these markings to a mesh frame implicitly removes another of these markings when applicable. When a mesh frame is referred to as being marked as "used for reference", this collectively refers to the mesh frame being marked as "used for short-term reference" or "used for long-term reference" (but not both).

Short term reference mesh frames are identified by their bmesh\_nal\_layer\_id and MeshFrmOrderCntVal values. Long term reference mesh frames are identified by their bmesh\_nal\_layer\_id values and by the Log2( MaxLtMeshFrmOrderCntLsb ) least significant bits of their MeshFrmOrderCntVal values.

For all cases, the following applies:

* For each long term reference mesh frame entry in RefMeshFrmList, when the mesh frame is marked as "used for short-term reference" and has the same bmesh\_nal\_layer\_id as the current mesh frame, the mesh frame is marked as "used for long-term reference".
* Each reference mesh frame in the same bmesh\_nal\_layer\_id as the current mesh frame in the DBMB that is not referred to by any entry in RefMeshFrmList is marked as "unused for reference".
  + - 1. Decoding process for submesh data units
         1. General decoding process for submesh data units

The decoding process of a submesh data unit with submesh ID submeshID is performed as follows.

* The data for the submesh data unit is extracted from bmesh\_submesh\_unit() syntax structure by invoking the extraction process in H.9.4.5.5.
* If bmsh\_type == I\_SUBMESH, the decoding process in H.9.4.5.2 is invoked with the extracted data.
* If bmsh\_type == P\_SUBMESH, the decoding process in H.9.4.5.3 is invoked with the extracted data.
* If bmsh\_type == SKIP\_SUBMESH, the decoding process in H.9.4.5.4 is invoked with the extracted data.

The outputs of this process are:

* A struct, DecSubmesh, indicating a decoded submesh. The members of DecSubmeshFrame are the same as the member of DecSubmeshesFrame[ submeshID ]

The conversion from the output of Annex I to Annex H. may be provided either in the bitstream or by external means not specified in this document. The composition time and the output order index may be utilized to determine timing relationships of, and to perform operations as defined in [CrossReferenceXXX], if required.

* + - * 1. Decoding process for intra submesh data units

The decoding process of an intra submesh data units, with submesh ID submeshID, is performed as follows.

* If bmptl\_profile\_codec\_group\_idc is 0 and bmsps\_codec\_specific\_parameters\_present\_flag is 0, the intra submesh decoding process defined in Annex I is invoked with the extracted submesh data as the input.
* If bmptl\_profile\_codec\_group\_idc is 0 and bmsps\_codec\_specific\_parameters\_present\_flag is 1, the intra submesh decoding process defined in Annex I is invoked with data that is concatenation of data contain in MeshCodecPrefixData and the extracted submesh data as the input.
* If bmptl\_profile\_codec\_group\_idc is equal to 127 and the component codec mapping SEI message is present, the codec can be determined as specified in subclause with bmsps\_intra\_mesh\_codec\_id, as the codec mapping index.
* Otherwise, the codec can be determined to be the MeshCodecGroup entry corresponding to the value of ptl\_profile\_codec\_group\_idc in[CrossReferenceXXX]. Then, the intra submesh decoding process, according to the corresponding coding specification, is invoked for the extracted submesh data as the input.

Output of this process is a struct, DecIntraSubmesh, indicating the decoded submesh. The members of DecIntraSubmesh are as follows:

* submeshId, indicating the submeshId of the current submesh
* verCoordCount, indicating the number of vertex coordinates,
* a 1D array vertexNeighboursCounts, of size verCoordCount × 1, indicating the number of neighbours for each vertex, value of each entry of the array shall be equal to 0.
* a 2D array vertexNeighbours, of size of size verCoordCount × ( bmsps\_inter\_mesh\_max\_num\_neighbours\_minus1 + 1 ), indicating the indices of its neighbours of each vertex, value of each each entry of the array shall be equal to -1.
* a 1D array vDuplicated of size verCoordCount × 1, each element of the array shall be set to –2,
* faceCount, indicating the number of faces,
* a 2D array verCoords, indicating vertex coordinate values, of size verCoordCount × 3,
* decVerCoordBitDepth, indicating the bit depth of the vertex coordinates,
* a 2D array verCoordFaces, indicating vertex coordinates connectivity, of size faceCount × 3, where i indicates the number of vertices of the i-th face
* attributeCount, indicating the number of attributes,
* an array of structs of attributeCount, strAttributes, indicating the decoded attributes. For attrIdx-th element of the strAttributes has the following members:
  + attrType, indicating the type of the attribute,
  + attrValueCount, indicating the number of values of the attribute,
  + attrFaceCount, indicating the number of faces of the attribute,
  + attrValueDimension, indicating the dimension of values of the attribute,
  + attrValueBitDepth, indicating the bit depth of values of the attribute,
  + attrValuePerVertex, indicating the attributes is per vertex
  + a 2D array attrValues, indicating values of the attribute, where the first dimension range is 0..attrValueCount - 1, and the second dimension range is 0.. attrValueDimension,
  + a 2D array attrFacesArray, indicating the connectivity of the attribute, where the first dimension range is 0..attrFaceCount - 1, and the second dimension range is 0.. 2,

The decoded submesh may require the application of additional transformations, as described in [CrossReferenceXXX], per each frame. For example, the different attributes count, bitdepth, attribute type etc need to be converted to a nominal format. The outputs of [CrossReferenceXXX] is a struct, DecIntraSubmeshNF.

* + - * 1. Decoding process for inter submesh data units

General Decoding process for inter submesh data units

The decoding process of an inter submesh data units, with submesh ID submeshID, is performed as follows.

submeshMotionVectors and refMeshFrame, the reference frame are decoded as :

* If bmptl\_profile\_codec\_group\_idc is 0, the inter submesh decoding process defined in H.9.4.5.3.2 is invoked with the extracted submesh data as the input.
* If bmptl\_profile\_codec\_group\_idc is equal to 127 and the component codec mapping SEI message is present, the codec can be determined as specified in subclause [CrossReferenceXXX] with bmsps\_intra\_mesh\_codec\_id, as the codec mapping index.
* Otherwise, the codec can be determined to be the MeshCodecGroup entry corresponding to the value of ptl\_profile\_codec\_group\_idc in [CrossReferenceXXX]. Then, the inter submesh decoding process, according to the corresponding coding specification, is invoked using the extracted submesh data as the input.

Default Decoding process for inter submesh data units

Inputs to this process is current submesh ID, submeshID.

An output of this process is a struct, DecSubmesh, indicating the decoded submesh. The members of DecSubmesh are as follows:

* submeshId, indicating the submeshId of the current submesh
* verCoordCount, indicating the number of vertex coordinates,
* a 1D array vertexNeighboursCounts, of size verCoordCount × 1, indicating the number of neighbours for each vertex
* a 2D array vertexNeighbours, of size of size verCoordCount × ( bmsps\_inter\_mesh\_max\_num\_neighbours\_minus1 + 1 ) indicating the indices of its neighbours of each vertex
* a 1D array vDuplicated of size verCoordCount × 1, each element indicating the index of vertex with identical coordinates as current vertex or -1 or -2,faceCount, indicating the number of faces,
* a 2D array verCoords, indicating vertex coordinate values, of size verCoordCount × 3,
* decVerCoordBitDepth, indicating the bit depth of the vertex coordinates,
* a 2D array verCoordFaces, indicating vertex coordinates connectivity, of size faceCount × 3,
* attributeCount, indicating the number of attributes,
* an array of structs of attributeCount, strAttributes, indicating the decoded attributes. For attrIdx-th element of the strAttributes has the following members:
  + attrType, indicating the type of the attribute,
  + attrValueCount, indicating the number of values of the attribute,
  + attrFaceCount, indicating the number of faces of the attribute,
  + attrValueDimension, indicating the dimension of values of the attribute,
  + attrValueBitDepth, indicating the bit depth of values of the attribute,
  + attrValuePerVertex, indicating the attributes is per vertex
  + a 2D array attrValues, indicating values of the attribute, where the first dimension range is 0..attrValueCount - 1, and the second dimension range is 0.. attrValueDimension,
  + a 2D array attrFacesArray, indicating the connectivity of the attribute, of size attrFaceCount × 3,

It is a requirement that any element of 2D array, verCoordFaces shall have values less than verCoordCount and that any elements of 2D array, attrValues[ attIdx ] shall have values less than attrValueCount[ attIdx ].

It is a requirement that number of the reconstructed vertices verCoordCount shall be the same as the vertex count of the prediction frame.

[Ed. Note: The requirement above needs to be refined. Also, it has to be considered if a requirement saying ‘all the values from 0 to verCoordCount/0 to attrValueCount[ attIdx ] need to be used’ is needed]

First, refIdx is set to 0. the reference mesh, refSubmesh is derived as described in subclause H.9.4.5.3.4 is invoked using the variables refIdx.

Then, DecSubmesh is initialized as follows:

DecSubmesh.verCoordCount = refSubmesh.verCoordCount  
 DecSubmesh.vertexNeighboursCounts = refSubmesh.vertexNeighboursCounts  
 DecSubmesh.vertexNeighbours = refSubmesh.vertexNeighbours  
 DecSubmesh.vDuplicated = refSubmesh.vDuplicated  
 DecSubmesh.faceCount = refSubmesh.faceCount   
 DecSubmesh.verCoordFaces = refSubmesh.verCoordFaces  
 DecSubmesh.verCoords = refSubmesh.verCoords  
 DecSubmesh.verCoordCount = refSubmesh.verCoordCount  
 for( attrIdx=0; attrIdx < bmsps\_mesh\_attribute\_count; attrIdx++ ){  
 attrValueCount[ attrIdx ] = refSubmesh.attrValueCount[ attrIdxNF ]  
 attrFaceCount[ attrIdx ] = refSubmesh.attrFaceCount[ attrIdxNF ]   
 attrFacesArray[ attrIdx ] = refSubmesh.attrFacesArray[ attrIdxNF ]   
 attrValueDimension[ attrIdx ] = refSubmesh.attrValueDimension  
 attrValues[ attrIdx ] = refSubmesh.attrValues[ attrIdxNF ]  
 }  
 DecSubmesh.submeshId = submeshId  
 DecSubmesh.vDuplicated = refSubmesh.vDuplicated  
 DecSubmesh.verCoordCount = refSubmesh.verCoordCount DecSubmesh.faceCount = refSubmesh.faceCount   
 DecSubmesh.verCoords =refSubmesh.verCoords  
 DecSubmesh.decVerCoordBitDepth = refSubmesh.decVerCoordBitDepth  DecSubmesh.verCoordFaces =refSubmesh.verCoordFaces  
 DecSubmesh.attributeCount = refSubmesh.attributeCount  
 for( attrIdx=0; attrIdx < refSubmesh.attributeCount; attrIdx++ ){  
 DecSubmesh.strAttributes[attrIdx].attrValueCount =  
 refSubmesh.strAttributes[attrIdx].attrValueCount  
 DecSubmesh.strAttributes[attrIdx].attrFaceCount =  
 refSubmesh.strAttributes[attrIdx].attrFaceCount  
 DecSubmesh.strAttributes[attrIdx].attrValueDimension =  
 refSubmesh.strAttributes[attrIdx].attrValueDimension  
 DecSubmesh.strAttributes[attrIdx].attrValueBitDepth =  
 refSubmesh.strAttributes[attrIdx].attrValueBitDepth  
 DecSubmesh.strAttributes[attrIdx].attrValuePerVertex =  
 refSubmesh.strAttributes[attrIdx].attrValuePerVertex  
 DecSubmesh.strAttributes[attrIdx].attrValues =  
 refSubmesh.strAttributes[attrIdx].attrValues  
 DecSubmesh.strAttributes[attrIdx].attrValues =  
 refSubmesh.strAttributes[attrIdx].attrValues  
 DecSubmesh.strAttributes[attrIdx].attrFacesArray =   
 refSubmesh.strAttributes[attrIdx].attrFaceCount  
 }

The vertex neighbour table calculation in H.9.4.5.3.3 with the variable refSubmesh.faceCount and the 2D array refSubmesh.verCoordFaces as inputs. A 1D array vertexNeighboursCounts of size refSubmesh.verCoordCount and a 2D array vertexNeighbours are outputs of the process. The size of the first dimension of the i-th element vertexNeighbours is refSubmesh.verCoordCount and the size of the second dimension of the i-th element vertexNeighbours is vertexNeighboursCounts[ i ].

vN = 0

For v = 0..bm\_vertex\_count[ submeshID ] − 1, the following applies to update the v-th vertex coordinate

* The group index g is derived as follows:

vB = v - vN  
 g = vB / ( bmsps\_inter\_mesh\_motion\_group\_size\_minus1 + 1 )

* If bmidu\_skip\_group\_flag[ submeshID ][ g ] is equal to 1, then

currentSubmeshMotionVectors[ v ][ k ] = 0, where k = 0..2

* Otherwise, the prediction mode, mvPredMode[ submeshID ][ v ][ k ] where k = 0..2, is derived as follows:

mvPredMode[ submeshID ][ v ][ k ] = BmiduMvFlag[ submeshID ][ v ] ? MV\_DERIVED  
 : bmidu\_mv\_pred\_mode\_group[ submeshID][ g ][ k ]

* + if the prediction mode, mvPredMode[ submeshID ][ v ][ k ] is equal to MV\_DERIVED, then

vN++

* + - vRef is derived as follow:

if ( DecSubmesh.vDuplicated[ v ] == -2 ){  
 vRef = findIndexInArray (referenceSubmeshVertexPositions[ v ],  
 referenceSubmeshVertexPositions, v-1)  
 DecSubmesh.vDuplicated[ v ] = vRef  
 else {  
 vRef = DecSubmesh.vDuplicated[ v ]  
 }

* + - if vRef = -1,then

currentSubmeshMotionVectors[ v ][ k ] is set as 0

* + - else

currentSubmeshMotionVectors[ v ][ k ] =  
 currentSubmeshMotionVectors[ vRef ][ k ]

* + else if the prediction mode, mvPredMode[ submeshID ][ v ][ k ] is equal to MV\_PRED\_NONE, then

currentSubmeshMotionVectors[ v ][ k ] =  
 VertexMotionVectorResiduals[ submeshID ][ vB ][ k ]

* + else (when bmidu\_mv\_pred\_mode\_group[submeshID][ v ] is equal to MV\_PRED\_NEIGHBOUR),

currentSubmeshMotionVectors[ v ][ k ] =   
 VertexMotionVectorResiduals[ submeshID ][ vB ][ k ] +   
 currentSubmeshPredictedMotionVectors[ v ][ k ]

* The predicted motion vector, currentSubmeshPredictedMotionVectors[ v ][ k ] where k = 0..2 is derived by applying the following process:

for( k = 0; k < 3; k++ ) {  
 mv[ k ] = 0  
 count = 0  
 for( n = 0; n < vertexNeighboursCounts[ v ]; n++ ) {  
 w = vertexNeighbours[ v ][ n ]  
 if ( w < v ) {  
 mv[ k ] += currentSubmeshMotionVectors[ v ][ w ]  
 count += 1  
 }  
 }  
 if ( count > 1) {  
 offset = bmidu\_mv\_pred\_mode\_group[ submeshID ][ v ] == 2 ? count >> 1 : 0  
 if ( mv[ k ] > 0 ) {  
 mv [ k ]= (mv[ k ] + offset ) / count  
 } else if ( mv[ k ] < 0 ) {  
 mv [ k ]= -(-mv[ k ] + offset ) / count  
 }  
 }  
 currentSubmeshPredictedMotionVectors[ v ] = mv  
 }

* + If the prediction mode, MvPredMode[ submeshID ][ v ] is equal to MV\_PRED\_NONE, then

currentSubmeshMotionVectors[ v ][ k ] =  
 VertexMotionVectorResiduals[ submeshID ][ v ][ k ]

* + Otherwise (when bmidu\_mv\_pred\_mode\_group[ submeshID ][ v ] is equal to MV\_PRED\_NEIGHBOUR),

currentSubmeshMotionVectors[ v ][ k ] =  
 VertexMotionVectorResiduals[ submeshID ][ vB ][ k ] +  
 currentSubmeshPredictedMotionVectors[ v ][ k ]

* The v-th vertex coordinate of the output struct, DecSubmesh is updated as follows:
  + If v is less than or equal to refSubmesh.verCoordCount

DecSubmesh.verCoords[ v ][ k ] =  
 currentSubmeshMotionVectors[ v ][ k ] +  
 refSubmesh.verCoords[ v ][ k ], where k = 0..2

Vertex neighbour table calculation

Inputs to this process are:

* vertexCount, which is a variable indicating the number of vertices
* faceCount, which is a variable indicating the number of faces,
* faceIndices, which is a 2D array of size faceCount by 3 indicating the connectivity indices.
* refSubmesh, which is the reference submesh.

The outputs of this process are:

* vertexNeighboursCounts, which is a 1D array of size vertexCount indicating the number of neighbours for each vertex .
* vertexNeighbours, which is a 2D array indicating the indices of its neighbours of each vertex

The maximum number of neighbours maxVertexNeighbourCount is set equal to bmsps\_inter\_mesh\_max\_num\_neighbours\_minus1 + 1.

When refSubmesh.vertexNeighboursCounts only contains zero values, vertexNeighboursCounts and vertexNeighbours are updated as follows:

for( v = 0; v < vertexCount; v++ ) {  
 vertexNeighboursCounts[ v ] = 0  
 }

for( f = 0; f < faceCount; f++ ) {  
 v0 = faceIndices[ f ][ 0 ]  
 v1 = faceIndices[ f ][ 1 ]  
 v2 = faceIndices[ f ][ 2 ]  
 AddNeighbour( v0, v1 )  
 AddNeighbour( v1, v0 )  
 AddNeighbour( v0, v2 )  
 AddNeighbour( v2, v0 )  
 AddNeighbour( v1, v2 )  
 AddNeighbour( v2, v1 )  
 }

AddNeighbour( indexA,indexB ) {  
 availableOld = 0  
 nCount = min(vertexNeighboursCounts[ indexA ], maxNumNeighboursMotion)  
 if (indexA > indexB ) {  
 available = findIndexInArray( indexB, vertexNeighbours[ indexA ], nCount -1 )  
 if (available < 0) {  
 if (nCount == maxNumNeighboursMotion) {  
 vertexNeighbours[ indexA ][maxNumNeighboursMotion - 1] = indexB   
 } else {  
 vertexNeighbours[ indexA ][nCount] = indexB   
 vertexNeighboursCounts[ indexA ] = nCount + 1;  
 }  
 }  
 }  
 }

Derivation of inter reference submesh

Inputs to this process are the mesh frame reference index, refIdx and the submesh ID, submeshID.

Outputs to this process are a struct refSubmesh.

First, the submesh, refSubmesh, is determined that has an index equal to submeshID in the submeshes of the ( refIdx + 1 )-th entry of the reference mesh frame list RefMeshFrmList, RefMeshFrmList[ refIdx ].

Then the members of the output struct, refSubmesh are derived as follows:

refSubmesh.submeshId = submeshId  
 refSubmesh.verCoordCount = RefMeshFrmList[ refIdx ].verCoordCount  
 refSubmesh.faceCount = RefMeshFrmList[ refIdx ].faceCount   
 refSubmesh.verCoords = RefMeshFrmList[ refIdx ].verCoords  
 refSubmesh.decVerCoordBitDepth = RefMeshFrmList[ refIdx ].decVerCoordBitDepth  refSubmesh.verCoordFaces = RefMeshFrmList[ refIdx ].verCoordFaces  
 refSubmesh.attributeCount = RefMeshFrmList[ refIdx ].attributeCount  
  
 for( attrIdx=0; attrIdx < refSubmesh.attributeCount; attrIdx++ ){  
 refSubmesh.strAttributes[attrIdx].attrValueCount =  
 RefMeshFrmList[ refIdx ].strAttributes[attrIdx].attrValueCount  
 refSubmesh.strAttributes[attrIdx].attrFaceCount =  
 RefMeshFrmList[ refIdx ].strAttributes[attrIdx].attrFaceCount  
 refSubmesh.strAttributes[attrIdx].attrValueDimension =  
 RefMeshFrmList[ refIdx ].strAttributes[attrIdx].attrValueDimension  
 refSubmesh.strAttributes[attrIdx].attrValueBitDepth =  
 RefMeshFrmList[ refIdx ].strAttributes[attrIdx].attrValueBitDepth  
 refSubmesh.strAttributes[attrIdx].attrValuePerVertex =  
 RefMeshFrmList[ refIdx ].strAttributes[attrIdx].attrValuePerVertex  
 refSubmesh.strAttributes[attrIdx].attrValues =  
 RefMeshFrmList[ refIdx ].strAttributes[attrIdx].attrValues  
 refSubmesh.strAttributes[attrIdx].attrValues =  
 RefMeshFrmList[ refIdx ].strAttributes[attrIdx].attrValues  
 refSubmesh.strAttributes[attrIdx].attrFacesArray =  
 RefMeshFrmList[ refIdx ].strAttributes[attrIdx].attrFaceCount  
 }

* + - * 1. Decoding process for skip submesh data units

Inputs to this process is current submesh ID, submeshID.

An output of this process is a struct, DecSubmesh, indicating the decoded submesh. The members of DecSubmesh are as follows:

* submeshId, indicating the submeshId of the current submesh
* verCoordCount, indicating the number of vertex coordinates,
* a 1D array vertexNeighboursCounts, of size verCoordCount × 1, indicating the number of neighbours for each vertex
* a 2D array vertexNeighbours, of size of size verCoordCount × ( bmsps\_inter\_mesh\_max\_num\_neighbours\_minus1 + 1 ) indicating the indices of its neighbours of each vertex
* a 1D array vDuplicated of size verCoordCount × 1, each element indicating the index of vertex with identical coordinates as current vertex or -1 or -2,
* faceCount, indicating the number of faces,
* a 2D array verCoords, indicating vertex coordinate values, of size verCoordCount × 3,
* decVerCoordBitDepth, indicating the bit depth of the vertex coordinates,
* a 2D array verCoordFaces, indicating vertex coordinates connectivity, of size faceCount × 3,
* attributeCount, indicating the number of attributes,
* an array of structs of attributeCount, strAttributes, indicating the decoded attributes. For attrIdx-th element of the strAttributes has the following members:
  + attrType, indicating the type of the attribute,
  + attrValueCount, indicating the number of values of the attribute,
  + attrFaceCount, indicating the number of faces of the attribute,
  + attrValueDimension, indicating the dimension of values of the attribute,
  + attrValueBitDepth, indicating the bit depth of values of the attribute,
  + attrValuePerVertex, indicating the attributes is per vertex
  + a 2D array attrValues, indicating values of the attribute, where the first dimension range is 0..attrValueCount - 1, and the second dimension range is 0.. attrValueDimension,
  + a 2D array attrFacesArray, indicating the connectivity of the attribute, of size attrFaceCount × 3.

First, the reference mesh frame refMeshFrm is selected as follows:

* refIdx is set to 0.
* The process described in subclause H.9.4.5.3.4 is invoked using the variables refIdx, and submeshID as inputs, and the output is refSubmesh, the reference submesh.

Then, DecSubmesh is initialized as follows:

DecSubmesh.verCoordCount = refSubmesh.verCoordCount  
 DecSubmesh.vertexNeighboursCounts = refSubmesh.vertexNeighboursCounts  
 DecSubmesh.vertexNeighbours = refSubmesh.vertexNeighbours  
 DecSubmesh.faceCount = refSubmesh.faceCount   
 DecSubmesh.verCoordFaces = refSubmesh.verCoordFaces  
 DecSubmesh.verCoords = refSubmesh.verCoords  
 DecSubmesh.verCoordCount = refSubmesh.verCoordCount  
 for( attrIdx=0; attrIdx < bmsps\_mesh\_attribute\_count; attrIdx++ ){  
 attrValueCount[ attrIdx ] = refSubmesh.attrValueCount[ attrIdxNF ]  
 attrFaceCount[ attrIdx ] = refSubmesh.attrFaceCount[ attrIdxNF ]   
 attrFacesArray[ attrIdx ] = refSubmesh.attrFacesArray[ attrIdxNF ]   
 attrValues[ attrIdx ] = refSubmesh.attrValues[ attrIdxNF ]  
 }  
 DecSubmesh.submeshId = submeshId  
 DecSubmesh.verCoordCount = refSubmesh.verCoordCount DecSubmesh.faceCount = refSubmesh.faceCount   
 DecSubmesh.verCoords = refSubmesh.verCoords  
 DecSubmesh.decVerCoordBitDepth = refSubmesh.decVerCoordBitDepth   
 DecSubmesh.verCoordFaces = refSubmesh.verCoordFaces  
 DecSubmesh.attributeCount = refSubmesh.attributeCount  
 for( attrIdx=0; attrIdx < refSubmesh.attributeCount; attrIdx++ ){  
 DecSubmesh.strAttributes[attrIdx].attrValueCount =  
 refSubmesh.strAttributes[attrIdx].attrValueCount  
 DecSubmesh.strAttributes[attrIdx].attrFaceCount =  
 refSubmesh.strAttributes[attrIdx].attrFaceCount  
 DecSubmesh.strAttributes[attrIdx].attrValueDimension =  
 refSubmesh.strAttributes[attrIdx].attrValueDimension  
 DecSubmesh.strAttributes[attrIdx].attrValueBitDepth =  
 refSubmesh.strAttributes[attrIdx].attrValueBitDepth  
 DecSubmesh.strAttributes[attrIdx].attrValuePerVertex =  
 refSubmesh.strAttributes[attrIdx].attrValuePerVertex  
 DecSubmesh.strAttributes[attrIdx].attrValues =  
 refSubmesh.strAttributes[attrIdx].attrValues  
 DecSubmesh.strAttributes[attrIdx].attrValues =  
 refSubmesh.strAttributes[attrIdx].attrValues  
 DecSubmesh.strAttributes[attrIdx].attrFacesArray =  
 refSubmesh.strAttributes[attrIdx].attrFaceCount  
 }

* + - * 1. Extraction process for submesh data unit

The extraction process of a submesh data unit with submesh ID submeshID is performed as follows.

* The submesh data is extracted by concatenating embedded\_external\_data\_bit in sub-clause H.8.3.3.11 until more\_rbsp\_data() return FALSE.
  1. Parsing process

Refer to Annex K.

* 1. Profiles and levels

[Ed. Note (LK)] Input required.

* 1. Basemesh hypothetical reference decoder

[Ed. Note (LK)] Input required.

* 1. Supplemental enhancement information

[Ed. Note (LK)] Input required.

1. (Normative)  
     
   MPEG EdgeBreaker Static Mesh Coding
   1. Scope

This annex specifies syntax, semantics, and decoding process for MPEG EdgeBreaker (MEB) static mesh coding. In this annex, a bitstream refers to an MEB bitstream unless otherwise indicated.

* 1. Normative references

The list of normative references in Clause 2 applies.

* 1. Terms and definitions

For the purpose of this annex, the following definitions apply in addition to the definitions in Clause 3. These definitions are either not present in Clause 3 or replace definitions in Clause 3.

I.3.1

mesh

We restrict the definition of a mesh, to be a 3D mesh of triangles. It comprises a set of triangles that are connected by their common edges and vertices. A mesh is an undirected graph where one or several attributes can be attached to each vertex. A mesh systematically has at least one 3D position attribute per vertex. The mesh is thus a 3D mesh.

I.3.2

connected component

In an [undirected graph](https://en.wikipedia.org/wiki/Undirected_graph) G, two [vertices](https://en.wikipedia.org/wiki/Vertex_(graph_theory)) u and v are called connected if G contains a [path](https://en.wikipedia.org/wiki/Path_(graph_theory)) from u to v. Otherwise, they are called disconnected. If the two vertices are additionally connected by a path of length 1 (that is, they are the endpoints of a single edge), the vertices are called adjacent.

A [graph](https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)) is said to be connected if every pair of vertices in the graph is connected.

A connected component of an [undirected graph](https://en.wikipedia.org/wiki/Undirected_graph) is a [connected](https://en.wikipedia.org/wiki/Connected_graph) [subgraph](https://en.wikipedia.org/wiki/Glossary_of_graph_theory#subgraph) that is not part of any larger connected subgraph.

I.3.3

primary attribute

The first 3D position attribute attached to a vertex of a mesh is named primary attribute. The primary attributes are always defined per-vertex.

I.3.4

auxiliary attribute

Any other attribute attached either to the vertices of the mesh (e.g. a second position attribute, a texture coordinate, a color, a normal vector, etc.) or to its faces (e.g. a color, a material Id, a normal vector, etc.) is named auxiliary attribute.

I.3.5

attribute arrays  
attribute tables

The attributes of a mesh, be it per vertex or per face, are commonly stored into arrays and dereferenced by their indices.

For instance, the primary attribute can be stored as a 2D array pos of size nbPositions x 3, where nbPositions defines the number of 3D position values, and pos[n][0], pos[n][1], and pos[n][2] respectively contains the x, y and z components of the 3D position. Such an array is commonly referred as a position array and each position can be dereferenced by its index n, with 0 <= n < nbPositions.

As a second example, an auxiliary attribute depicting per vertex texture coordinates can be stored as a 2D array uvcoord of size nbUVCoords x 2, where nbUVCoords defines the number of texture coordinates values, and where uvcoord[n][0] and uvcoord [n][2] respectively contains the u and v components of the texture coordinate. Such an array is commonly referred as a UV coordinates array and each UV coordinate can be dereferenced by its index n, with 0 <= n < nbUVCoords.

The same applies with any other type of attributes (normal, IDs, etc.)

I.3.6

connectivity

The graph made of edges and vertices, defining the triangles of the mesh, is named the **connectivity**.

Each vertex of the graph can be assigned a unique index in 0, number of vertices -1 included. This vertex index can then be used to dereference attributes values from associated attribute tables.

I.3.7

corner to vertex table, or indexed triangle set, or index array

The connectivity of a mesh is commonly described and stored into arrays of indices. For each triangle, for each **corner** of the triangle, we store the index of the vertex used to dereference the associated attribute arrays.

For instance, suppose the tables pos and uvcoord presented in clause 0 be defined and of same size (i.e. nbPositions = nbUVCoords). Suppose also that those tables are aligned, in the sense that for a vertex index n we can find its associated position at pos[n] and we can find its associated texture coordinate uvcoord[n].

Then a connectivity can be defined by using a 2D array triangles, of size nbTriangles x 3, where nbTriangles defines the number of triangles for the mesh, and where triangles[t][0], triangles[t][1], triangles[t][2] respectively contains, for each corner of the triangle of index t the index of the attribute in the attribute arrays.

For convenience and easy arithmetic for dereferencing corners, we can also store the connectivity in a 1D table triangles of size nbTriangles \* 3. Where triangles[ t \* 3 + 0 ], triangles triangles[ t \* 3 + 1 ], triangles triangles[ t \* 3 + 2 ] respectively contains, for each corner of the triangle of index t the index of the attribute in the attribute arrays. In this annex, such a 1D table is named a corner to vertex table, where corners are numbered from 0 to 3 \* nbTriangles- 1.

Both memory representations are also commonly called indexed triangle set (or indexed face set) in the literature.

I.3.8

primary connectivity

The primary connectivity is the connectivity which supports the primary attribute. Some auxiliary attributes can be attached to the primary connectivity, either per vertex or per face.

The primary connectivity is defined using a primary corner to vertex table that dereferences one or several attribute tables (at least, the primary positions and optionally some auxiliary attributes).

I.3.9

auxiliary connectivity

In some cases, it is needed to affect a different per vertex attribute to the corners of its associated triangles. This is common with texture coordinates, where texture maps are made of several patches. In this case, some triangles connected by the same vertex in the primary connectivity might need different texture coordinates to dereference different texture coordinates.

To solve this problem there are two possibilities. Either duplicate the vertices of the primary connectivity, but this implies to also duplicate all the attributes using primary connectivity to preserve the alignments, which implies more memory consumption. Or use an auxiliary connectivity aligned with the primary connectivity (i.e. same number of triangles depicted in same order) but where per vertex attribute indices dereference the auxiliary attribute table. This auxiliary connectivity is defined using an auxiliary corner to vertex table.

I.3.10

boundary edge

Edges connected to only one face are called Boundary edges.

I.3.11

attribute seam

A seam is a set of boundary edges on the auxiliary connectivity that are not boundary edges on the primary connectivity.

For instance, two triangles that are adjacent in the primary connectivity might not be adjacent in an auxiliary connectivity attached to texture coordinates that refers to two different patches of a texture map. Each set of connected edges of the auxiliary connectivity presenting this characteristic is called an attribute seam.

I.3.12

isolated vertices

An isolated vertex is a vertex that is not connected to any other vertex. The indexed triangle set representation is not able to represent such isolated vertices since it only depicts triangles. However, the attributes (primary or auxiliary) of an isolated vertex might be stored in the attributes arrays even if not referenced in the associated connectivity. This leads to corner to vertex arrays that may not reference the entirety of the indices covered by the attribute tables.

I.3.13

degenerate triangle

A degenerate triangle is a triangle which dereferences the same attribute index for at least two of its corners.

This is not to be confused with a “collapsed” triangle which is topologically valid (i.e. three different vertex indices) but for which at least two vertex positions (or UV coordinates) may be equal, reducing the triangle to a “line”, or three vertex positions (or UV coordinates) may be equal reducing the position to a “point”, in 3D space (in 2D space for UV coordinates).

I.3.14

Non-manifold mesh

A non-manifold mesh is a mesh that has at least one of the following topology issues:

The mesh present some “T-Shape” issues where an edge is connected to more than two triangles.

The mesh present some “Bowtie” issues where, for instance, two cubes have a single point in common.

The mesh has non-coherent triangle normal vectors, with normal vector obtained by cross product of first and second edge of each triangle. That is, the normal on two adjacent triangles points in opposite directions. In other terms, the order of face corners enumeration is not the same for all the faces. In previous example with two triangles, one might have its vertex indices enumerated clockwise and the other counter-clockwise.

I.3.15

Opposite corner table and Corner table

The indexed triangle set data structure is convenient for rendering or other simple processing. However, it requires complex operations to perform triangle traversals.

For the case where the mesh is not non-manifold it is possible to generate and maintain an additional table to the vertex one, of same size, that is named the opposite corner table. This table is aligned with the corner to vertex table and stores for each corner, the index of its opposite corner.

Let t1 and t2 be two triangles connected by an edge e, and c1 and c2 be respectively the corner of t1 and t2 that is not member of the common edge e. Then c1 and c2 are opposite corners.

Knowing this additional information, it is then very easy to navigate through the connectivity of the mesh with very low complexity, hence, to perform efficient traversals of the triangles.

In addition, it is possible to handle boundaries of a mesh by setting the value of opposite corners to negative values.

The combination of the corner to vertex table and the opposite corners table is known as a corner table.

One corner table is used to describe the primary connectivity and each auxiliary connectivity is described using one additional corner table.

I.3.16

CLERS

Set of symbols used to encode mesh connectivity.

* 1. Abbreviated terms

For the purposes of this Annex, the following abbreviations apply.

|  |  |
| --- | --- |
| MEB | MPEG EdgeBreaker |

* 1. Conventions

The specifications in Clause 5 and its subclauses apply with the following additions.

A “break” statement, which is used within loops, is defined as follows:

The “break” statement, when encountered inside a loop, exits the loop. This results in skipping the execution of subsequent statements inside the body of the loop for the current and subsequent iterations. For example:

for( j =0; j < N; j++ ) {  
 statement 0  
 if( condition 1 )  
 break  
 statement 2  
 statement 3  
}

is equivalent to the following:

broken = 0  
for( j =0; j < N; j++ ) {  
 if( broken )  
 continue  
 statement 0  
 if( condition 1 ) {  
 broken=1  
 continue  
 }  
 statement 2  
 statement 3  
}

In addition, a “return” statement, which is used within processes, is defined as follows:

The “return” statement, when encountered inside a process, exits the process.

* 1. Overall MEB characteristics, decoding operations, and post-decoding processes
     1. MEB characteristics

This annex enables the encoding and decoding processes of static triangular meshes.

* + 1. MEB bitstream characteristics, decoding operations, and post decoding processes

This subclause provides high-level description of the characteristics and the operations needed for the decoding of MEB bitstreams.

MEB specific syntax elements and their semantics are specified in Clause I.8.

Clause I.9 invokes the decoding process of an MEB bitstream carrying static mesh content, with outputs composed of ...

* 1. Bitstream format
  2. Syntax and semantics
     1. Method of specifying syntax in tabular form

The specifications in subclause 8.1 apply.

* + 1. Specification of syntax functions and descriptors

The specifications in subclause 8.2 apply with the following additions.

The following descriptors specify additional syntax element parsing processes:

* ae(v): context-adaptive arithmetic entropy-coded syntax element. The parsing process for this descriptor is specified in clause I.10.3.
* vu(v): unsigned integer using a variable number of bytes as groups of 8 bits. The parsing process for this descriptor is specified by the return values of the functions read\_bits( 1 ) followed by read\_bits( 7 ). The second value is appended to the resulting bit sequence and the process iterated while the first value is equal to 1. In particular, the parsing process for this descriptor is specified as follows:

vu(v) {  
 value = 0  
 do {  
 continue = read\_bits( 1 )  
 partial\_value = read\_bits( 7 )  
 value = ( value << 7 ) | partial\_value  
 }  
 while( continue )  
 return value  
}

* vi(v): signed integer using a variable number of bytes as groups of 8 bits. The parsing process for this descriptor is specified by the return values of the functions read\_bits( 1 ) followed by read\_bits( 7 ). The second value is appended to the resulting bit sequence and the process iterated while the first value is equal to 1. The resulting bit sequence interpreted as an unsigned integer is assigned to a signed integer value considering that syntax elements have been mapped to unsigned integers by ordering those by their absolute value in increasing order and representing the positive value for a given absolute value with the highest coded value. Table I-1 provides the assignment rule. In particular, the parsing process for this descriptor is specified as follows:

vi(v) {  
 value = 0  
 do {  
 continue = read\_bits( 1 )  
 partial\_value = read\_bits( 7 )  
 value = ( value << 7 ) | partial\_value  
 }  
 while( continue )  
 is\_positive = !( value & 1 )  
 value = ( value >> 1 )  
 if(is\_positive)  
 return value  
 else  
 return –( value + 1 )  
}

Table I-1 – Assignment of syntax element to codeNum

|  |  |
| --- | --- |
| **codeNum** | **syntax element value** |
| 0 | 0 |
| 1 | −1 |
| 2 | 1 |
| 3 | −2 |
| 4 | 2 |
| 5 | -3 |
| 6 | 3 |
| k | ( ( k + 1 ) / 2 ) \* ( −1 )k |

* + 1. Syntax in tabular form
       1. General Mesh coding syntax

|  |  |
| --- | --- |
| mesh\_coding( ) { | **Descriptor** |
| mesh\_coding\_header( ) |  |
| mesh\_position\_coding\_payload( ) |  |
| mesh\_attribute\_coding\_payload( ) |  |
| } |  |

* + - 1. Mesh coding header syntax

|  |  |
| --- | --- |
| mesh\_coding\_header( ) { | **Descriptor** |
| **mesh\_codec\_type** | u(2) |
| **mesh\_vertex\_traversal\_method** | u(2) |
| mesh\_position\_encoding\_parameters( ) |  |
| **mesh\_position\_dequantize\_flag** | u(1) |
| if( mesh\_position\_dequantize\_flag ) |  |
| mesh\_position\_dequantize\_parameters( ) |  |
| **mesh\_attribute\_count** | u(5) |
| for( i = 0; i < mesh\_attribute\_count; i++ ) { |  |
| **mesh\_attribute\_type**[ i ] | u(3) |
| if( mesh\_attribute\_type[ i ] == MESH\_ATTR\_TEXCOORD ) |  |
| NumComponents[ i ] = 2 |  |
| else if( mesh\_attribute\_type[ i ] == MESH\_ATTR\_NORMAL ) { |  |
| **mesh\_normal\_octahedral\_flag**[ i ] | u(1) |
| if( mesh\_normal\_octahedral\_flag[ i ] ) |  |
| NumComponents[ i ] = 2 |  |
| else |  |
| NumComponents[ i ] = 3 |  |
| } |  |
| } |  |
| else if( mesh\_attribute\_type[ i ] == MESH\_ATTR\_COLOR ) |  |
| NumComponents[ i ] = 3 |  |
| else if( mesh\_attribute\_type[ i ] == MESH\_ATTR\_MATERIAL\_ID ) |  |
| NumComponents[ i ] = 1 |  |
| else if( mesh\_attribute\_type[ i ] == MESH\_ATTR\_GENERIC ) { |  |
| **mesh\_attribute\_num\_components\_minus1**[ i ] | u(2) |
| NumComponents[ i ] = mesh\_attribute\_num\_components\_minus1[ i ] + 1 |  |
| } |  |
| mesh\_attribute\_encoding\_parameters ( i ) |  |
| **mesh\_attribute\_dequantize\_flag**[ i ] | u(1) |
| if( mesh\_attribute\_dequantize\_flag[ i ] ) |  |
| mesh\_attribute\_dequantize\_parameters( i ) |  |
| } |  |
| **mesh\_deduplicate\_method** | ue(v) |
| padding\_to\_byte\_alignment( ) |  |
| } |  |

* + - 1. Mesh position encoding parameters syntax

|  |  |
| --- | --- |
| mesh\_position\_encoding\_parameters( ) { | **Descriptor** |
| **mesh\_position\_bit\_depth\_minus1** | u(5) |
| **mesh\_position\_prediction\_method** | ue(v) |
| **mesh\_position\_reverse\_unification\_flag** | u(1) |
| } |  |

* + - 1. Mesh position dequantize parameters syntax

|  |  |
| --- | --- |
| mesh\_position\_dequantize\_parameters( ) { | **Descriptor** |
| for( i = 0; i < 3; i++ ) { |  |
| **mesh\_position\_min**[ i ] | fl(32) |
| **mesh\_position\_max**[ i ] | fl(32) |
| } |  |
| } |  |

* + - 1. Mesh attributes encoding parameters syntax

|  |  |
| --- | --- |
| mesh\_attribute\_encoding\_parameters( index ) { | **Descriptor** |
| **mesh\_attribute\_bit\_depth\_minus1**[ index ] | u(5) |
| **mesh\_attribute\_per\_face\_flag**[ index ] | u(1) |
| if( !mesh\_attribute\_per\_face\_flag[ index ] ) { |  |
| **mesh\_attribute\_separate\_index\_flag**[ index ] | u(1) |
| if( !mesh\_attribute\_separate\_index\_flag[ index ] ) |  |
| **mesh\_attribute\_reference\_index\_plus1**[ index ] | ue(v) |
| } |  |
| **mesh\_attribute\_prediction\_method**[ index ] | ue(v) |
| } |  |

* + - 1. Mesh attributes dequantize parameters syntax

|  |  |
| --- | --- |
| mesh\_attribute\_dequantize\_parameters( index ) { | **Descriptor** |
| for( j = 0; j < NumComponents[ index ]; j++ ) { |  |
| **mesh\_attribute\_min**[ index ][ j ] | fl(32) |
| **mesh\_attribute\_max**[ index ][ j ] | fl(32) |
| } |  |
| } |  |

* + - 1. Mesh position coding payload syntax

|  |  |
| --- | --- |
| mesh\_position\_coding\_payload( ) { | **Descriptor** |
| **mesh\_triangle\_count** | vu(v) |
| **mesh\_position\_start\_count** | vu(v) |
| **mesh\_position\_fine\_residuals\_count** | vu(v) |
| **mesh\_position\_coarse\_residuals\_count** | vu(v) |
| **mesh\_clers\_count** | vu(v) |
| **mesh\_cc\_with\_boundary\_count** | vu(v) |
| **mesh\_handles\_count** | vu(v) |
| MinHandles = 10 |  |
| if( mesh\_handles\_count < MinHandles ) { |  |
| for( i=0; i < mesh\_handles\_count; i++ ){ |  |
| **mesh\_handle\_first\_delta**[ i ] | vi(v) |
| **mesh\_handle\_second\_delta**[ i ] | vi(v) |
| } |  |
| } else { |  |
| **mesh\_coded\_handle\_size** | vu(v) |
| for( i=0; i< mesh\_handles\_count; i++ ){ |  |
| **mesh\_handle\_first\_sign**[ i ] | ae(v) |
| **mesh\_handle\_second\_shift**[ i ] | ae(v) |
| **mesh\_handle\_first\_variable\_delta\_length4\_minus1**[ i ] | ae(v) |
| **mesh\_handle\_first\_variable\_delta**[ i ] | ae(v) |
| **mesh\_handle\_second\_variable\_delta\_length4\_minus1**[ i ] | ae(v) |
| **mesh\_handle\_second\_variable\_delta**[ i ] | ae(v) |
| } |  |
| } |  |
| padding\_to\_byte\_alignment( ) |  |
| **mesh\_coded\_clers\_symbols\_size** | vu(v) |
| for( i=0; i <mesh\_clers\_count; i++ ) { |  |
| **mesh\_clers\_symbol**[ i ] | ae(v) |
| } |  |
| padding\_to\_byte\_alignment( ) |  |
| NumPositionStart = mesh\_position\_start\_count |  |
| for( i=0; i < NumPositionStart; i++ ) { |  |
| for( j = 0; j < 3; j++ ) { |  |
| **mesh\_position\_start**[ i ][ j ] | u(v) |
| } |  |
| } |  |
| padding\_to\_byte\_alignment( ) |  |
| NumPredictedFinePositions = mesh\_position\_fine\_residuals\_count |  |
| if( mesh\_position\_fine\_residuals\_count > 0 ) { |  |
| **mesh\_coded\_position\_fine\_residuals\_size** | vu(v) |
| for( j = 0; j < 3; j++ ){ |  |
| for( i = 0; i < NumPredictedFinePositions; i++ ) { |  |
| **mesh\_position\_fine\_residual**[ i ][ j ] | ae(v) |
| } |  |
| } |  |
| padding\_to\_byte\_alignment( ) |  |
| } |  |
| NumPredictedCoarsePositions = mesh\_position\_coarse\_residuals\_count |  |
| if( mesh\_position\_coarse\_residuals\_count > 0 ) { |  |
| **mesh\_coded\_position\_coarse\_residuals\_size** | vu(v) |
| for( j = 0; j < 3; j++ ){ |  |
| for( i = 0; i < NumPredictedCoarsePositions; i++ ) { |  |
| **mesh\_position\_coarse\_residual**[ i ][ j ] | ae(v) |
| } |  |
| } |  |
| padding\_to\_byte\_alignment( ) |  |
| } |  |
| mesh\_position\_deduplicate\_information( ) |  |
| if( mesh\_position\_reverse\_unification\_flag ) { |  |
| mesh\_difference\_information( ) |  |
| } |  |
| } |  |

* + - 1. Mesh position deduplicate information syntax

|  |  |
| --- | --- |
| mesh\_position\_deduplicate\_information( ) { | **Descriptor** |
| if( mesh\_deduplicate\_method == MESH\_\_DEDUP\_DEFAULT ) { |  |
| **mesh\_position\_deduplicate\_count** | vu(v) |
| if( mesh\_position\_deduplicate\_count > 0 ){ |  |
| NumSplitVertex = 0 |  |
| for( i=0; i < mesh\_position\_deduplicate\_count; i++ ) { |  |
| **mesh\_position\_deduplicate\_idx**[ i ] | vu(v) |
| NumSplitVertex = Max( NumSplitVertex,  mesh\_position\_deduplicate\_idx[ i ] + 1 ) |  |
| } |  |
| NumAddedDuplicatedVertex = mesh\_position\_deduplicate\_count  – NumSplitVertex |  |
| NumPositionIsDuplicateFlags = NumPositionStart  + NumPredictedFinePositions  + NumPredictedCoarsePositions  + NumAddedDuplicatedVertex |  |
| **mesh\_position\_coded\_is\_duplicate\_size** | vu(v) |
| for( i = 0; i< NumPositionIsDuplicateFlags; i++ ) { |  |
| **mesh\_position\_is\_duplicate\_flag**[ i ] | ae(v) |
| } |  |
| padding\_to\_byte\_alignment( ) |  |
| } |  |
| } |  |
| } |  |

* + - 1. Mesh difference information syntax

|  |  |
| --- | --- |
| mesh\_difference\_information( ) { | **Descriptor** |
| **mesh\_position\_added\_vertices\_count** | vu(v) |
| if( mesh\_position\_added\_vertices\_count > 0 ) { |  |
| for( i = 0; i < mesh\_position\_added\_vertices\_count; i++ ) { |  |
| **mesh\_position\_added\_vertices\_idx\_delta**[ i ] | vi(v) |
| } |  |
| } |  |
| padding\_to\_byte\_alignment( ) |  |
| **mesh\_position\_deleted\_vertices\_count** | vu(v) |
| if( mesh\_position\_deleted\_vertices\_count >0 ) { |  |
| for( i = 0; i < mesh\_position\_deleted\_vertices\_count; i++ ) { |  |
| **mesh\_position\_deleted\_vertices\_idx\_delta**[ i ] | vi(v) |
| } |  |
| } |  |
| padding\_to\_byte\_alignment( ) |  |
| **mesh\_modified\_count** | vu(v) |
| if( mesh\_modified\_count >0 ) { |  |
| for( i = 0; i < mesh\_modified\_count; i++ ) { |  |
| **mesh\_modified\_triangles\_idx\_delta**[ i ] | vi(v) |
| } |  |
| for( i = 0; i < mesh\_modified\_count; i++ ) { |  |
| **mesh\_modified\_vertices\_relative\_idx**[ i ] | vu(v) |
| } |  |
| for( i = 0; i < mesh\_modified\_count; i++ ) { |  |
| **mesh\_target\_modified\_vertices\_idx\_delta**[ i ] | vi(v) |
| } |  |
| padding\_to\_byte\_alignment( ) |  |
| } |  |
| } |  |

* + - 1. Mesh attribute coding payload syntax

|  |  |
| --- | --- |
| mesh\_attribute\_coding\_payload( ) { | **Descriptor** |
| for( i = 0; i < mesh\_attribute\_count; i++ ) { |  |
| **mesh\_attribute\_start\_count**[ i ] | vu(v) |
| **mesh\_attribute\_fine\_residuals\_count**[ i ] | vu(v) |
| **mesh\_attribute\_coarse\_residuals\_count**[ i ] | vu(v) |
| if( mesh\_attribute\_separate\_index\_flag[ i ]) { |  |
| **mesh\_attribute\_seams\_count**[ i ] | vu(v) |
| if( mesh\_attribute\_seams\_count > 0 ) { |  |
| **mesh\_coded\_attribute\_seams\_size**[ i ] | vu(v) |
| for( j = 0; j < mesh\_attribute\_seams\_count[ i ]; j++ ) { |  |
| **mesh\_attribute\_seam**[ i ][ j ] | ae(v) |
| } |  |
| } |  |
| padding\_to\_byte\_alignment( ) |  |
| } |  |
| NumAttributeStart[ i ] = mesh\_attribute\_start\_count[ i ] |  |
| if( mesh\_attribute\_type[ i ] == MESH\_ATTR\_NORMAL ) { |  |
| NumAttributeStartComponents[ i ] = 3 |  |
| } else { |  |
| NumAttributeStartComponents[ i ] = NumComponents[ i ] |  |
| } |  |
| for( j = 0; j < mesh\_attribute\_start\_count[ i ]; j++ ) { |  |
| for( k = 0; k< NumAttributeStartComponents[ i ]; k++ ) { |  |
| **mesh\_attribute\_start**[ i ][ j ][ k ] | u(v) |
| } |  |
| } |  |
| padding\_to\_byte\_alignment( ) |  |
| if( mesh\_attribute\_fine\_residuals\_count[ i ] ){ |  |
| **mesh\_coded\_attribute\_fine\_residuals\_size**[ i ] | vu(v) |
| for( j = 0; j < mesh\_attribute\_fine\_residuals\_count[ i ]; j++ ) { |  |
| for( k = 0; k < NumComponents[ i ]; k++ ) { |  |
| **mesh\_attribute\_fine\_residual**[ i ][ j ][ k ] | ae(v) |
| } |  |
| } |  |
| padding\_to\_byte\_alignment( ) |  |
| } |  |
| if( mesh attribute\_coarse\_residuals\_count[ i ] > 0 ){ |  |
| **mesh\_coded\_attribute\_coarse\_residuals\_size**[ i ] | vu(v) |
| for( j = 0; j < mesh\_attribute\_coarse\_residuals\_count[ i ]; j++ ) { |  |
| for( k = 0; k < NumComponents[ i ]; k++ ) { |  |
| **mesh\_attribute\_coarse\_residual**[ i ][ j ][ k ] | ae(v) |
| } |  |
| } |  |
| padding\_to\_byte\_alignment( ) |  |
| } |  |
| if(mesh\_attribute\_separate\_index\_flag[ i ]) |  |
| mesh\_attribute\_deduplicate\_info( i ) |  |
| /\* extra data dependent on the selected prediction scheme \*/ |  |
| AttributeType = mesh\_attribute\_type[ i ] |  |
| AttributePredictionMethod = mesh\_attribute\_prediction\_method[ i ] |  |
| mesh\_attribute\_extra\_data( i, AttributeType, AttributePredictionMethod ) |  |
| } |  |
| padding\_to\_byte\_alignment( ) |  |
| } |  |

* + - 1. Mesh extra attribute data syntax

|  |  |
| --- | --- |
| mesh\_attribute\_extra\_data( index, type, method ) { | **Descriptor** |
| if( type == MESH\_ATTR\_TEXCOORD ) { |  |
| if( method == MESH\_TEXCOORD\_MSTRETCH ) |  |
| mesh\_texcoord\_stretch\_extra\_data( index ) |  |
| } |  |
| else if( type == MESH\_ATTR\_NORMAL ) |  |
| if( mesh\_normal\_octahedral\_flag[ index ] ) { |  |
| mesh\_normal\_octahedral\_extra\_data( index ) |  |
| } |  |
| else if( type == MESH\_ATTR\_COLOR ) |  |
| /\* No extra data defined for specified prediction methods applied on colors \*/ |  |
| else if( type == MESH\_ATTR\_MATERIAL\_ID ) { |  |
| if( method == MESH\_MATERIALID\_DEFAULT ) |  |
| mesh\_materialid\_default\_extra\_data( index ) |  |
| } |  |
| else if( type == MESH\_ATTR\_GENERIC ) |  |
| /\* No extra data defined for specified prediction methods applied on generic\*/ |  |
| } |  |

* + - 1. Mesh texcoord stretch extra data syntax

|  |  |
| --- | --- |
| mesh\_texcoord\_stretch\_extra\_data( index ) { | **Descriptor** |
| **mesh\_texcoord\_stretch\_orientations\_count**[ index ] | vu(v) |
| if(mesh\_texcoord\_stretch\_orientations\_count[ index ] > 0 ) { |  |
| **mesh\_coded\_texcoord\_stretch\_orientations\_size**[ index ] | vu(v) |
| for( j = 0; j < mesh\_texcoord\_stretch\_orientations\_count[ index ]; j++ ) { |  |
| **mesh\_texcoord\_stretch\_orientation**[ index ][ j ] | ae(v) |
| } |  |
| } |  |
| padding\_to\_byte\_alignment( ) |  |
| } |  |

* + - 1. Mesh materialid default extra data syntax

|  |  |
| --- | --- |
| mesh\_materialid\_default\_extra\_data( index ) { | **Descriptor** |
| **mesh\_materialid\_default\_not\_equal\_count**[ index ] | vu(v) |
| **mesh\_materialid\_default\_left\_count**[ index ] | vu(v) |
| **mesh\_materialid\_default\_right\_count**[ index ] | vu(v) |
| **mesh\_materialid\_default\_facing\_count**[ index ] | vu(v) |
| if( mesh\_materialid\_default\_not\_equal\_count[ index ] > 0 ){ |  |
| **mesh\_coded\_materialid\_default\_not\_equal\_size**[ index ] | vu(v) |
| for( j = 0; j < mesh\_materialid\_default\_not\_equal\_count[ index ]; j++ ){ |  |
| **mesh\_materialid\_default\_not\_equal\_flag**[ index ][ j ] | ae(v) |
| } |  |
| } |  |
| if( mesh\_materialid\_default\_left\_count[ index ] > 0 ){ |  |
| **mesh\_coded\_materialid\_default\_left\_size**[ index ] | vu(v) |
| for( j = 0; j < mesh\_materialid\_default\_left\_count[ index ]; j++ ) { |  |
| **mesh\_materialid\_default\_left\_flag**[ index ][ j ] | ae(v) |
| } |  |
| } |  |
| if( mesh\_materialid\_default\_right\_count[ index ] > 0 ) { |  |
| **mesh\_coded\_materialid\_default\_right\_size**[ index ] | vu(v) |
| for( j = 0; j < mesh\_materialid\_default\_right\_count[ index ]; j++ ) { |  |
| **mesh\_materialid\_default\_right\_flag**[ index ][ j ] | ae(v) |
| } |  |
| } |  |
| if( mesh\_materialid\_default\_facing\_count[ index ] > 0 ){ |  |
| **mesh\_coded\_materialid\_default\_facing\_size**[ index ] | vu(v) |
| for( j = 0; j < mesh\_materialid\_default\_facing\_count[ index ]; j++ ) { |  |
| **mesh\_materialid\_default\_facing\_flag**[ index ][ j ] | ae(v) |
| } |  |
| } |  |
| padding\_to\_byte\_alignment( ) |  |
| } |  |

* + - 1. Mesh normal octahedral extra data syntax

|  |  |
| --- | --- |
| mesh\_normal\_octahedral\_extra\_data( index ) { | **Descriptor** |
| **mesh\_normal\_octahedral\_bit\_depth\_minus1**[ index ] | u(5) |
| **mesh\_normal\_octahedral\_second\_residual\_flag**[ index ] | u(1) |
| padding\_to\_byte\_alignment( ) |  |
| if( mesh\_normal\_octahedral\_second\_residuals\_flag[ index ] ){ |  |
| **mesh\_normal\_octahedral\_second\_residuals\_count**[ index ] | vu(v) |
| if ( mesh\_normal\_octrahedral\_second\_residuals\_count[ index ] ) { |  |
| **mesh\_normal\_octahedral\_second\_residuals\_size**[ index ] | vu(v) |
| for( j = 0; j < mesh\_normal\_octrahedral\_second\_residuals\_count[ index ]; j++ ) { |  |
| for( k = 0; k < 3; k++ ) { |  |
| **mesh\_normal\_octahedral\_second\_residual**[ index ][ j ][ k ] | ae(v) |
| } |  |
| } |  |
| } |  |
| } |  |
| padding\_to\_byte\_alignment( ) |  |
| } |  |

* + - 1. Mesh attribute deduplicate information syntax

|  |  |
| --- | --- |
| mesh\_attribute\_deduplicate\_info( index ) { | **Descriptor** |
| if( mesh\_deduplicate\_method == MESH\_\_DEDUP\_DEFAULT ) { |  |
| **mesh\_attribute\_deduplicate\_count**[ index ] | vu(v) |
| if( mesh\_position\_deduplicate\_count[ index ] > 0 ){ |  |
| NumSplitAttribute[ index ] = 0 |  |
| for( i = 0; i < mesh\_attribute\_deduplicate\_count[ index ]; i++ ) { |  |
| **mesh\_attribute\_deduplicate\_idx**[ index ][ i ] | vu(v) |
| NumSplitAttribute[ index ] = Max(NumSplitAttribute[ index ], mesh\_attribute\_deduplicate\_idx[ index ][ i ] + 1) |  |
| } |  |
| NumAddedDuplicatedAttribute[ index ] =  mesh\_attribute\_deduplicate\_count[ index ] – NumSplitAttribute[ index ] |  |
| NumAttributeIsDuplicateFlags[ index ] = NumAttributeStart[ index ]  +mesh\_attribute\_fine\_residuals\_count[ index ]  +mesh\_attribute\_coarse\_residuals\_count[ index ] +NumAddedDuplicatedAttribute[ index ] |  |
| **mesh\_attribute\_coded\_is\_duplicate\_size**[ index ] | vu(v) |
| for( i = 0; i < NumAttributeIsDuplicateFlags[ index ]; i++ ) { |  |
| **mesh\_attribute\_is\_duplicate\_flag**[ index ][ i ] | ae(v) |
| } |  |
| padding\_to\_byte\_alignment( ) |  |
| } |  |
| } |  |

* + - 1. Padding to byte alignment syntax

|  |  |
| --- | --- |
| padding\_to\_byte\_alignment( ) { | **Descriptor** |
| while( !byte\_aligned( ) ) |  |
| padding\_to\_byte\_alignment\_bit\_equal\_to\_zero /\* equal to 0 \*/ | f(1) |
| } |  |

* + 1. Semantics
       1. General Mesh Coding semantics

None.

* + - 1. Mesh coding header semantics

**mesh\_codec\_type** indicates the identifier of the selected codec method. Table I-2 describes the list of supported methods.

Table I-2 – Mesh codec types

|  |  |  |
| --- | --- | --- |
| **mesh\_codec\_type** | **Identifier** | **Codec Type** |
| 0 | CODEC\_TYPE\_REVERSE | Reverse |
| 1..3 | CODEC\_TYPE\_RESERVED | Reserved |

**mesh\_vertex\_traversal\_method** specifies the method used to traverse the vertices to perform the prediction of all the vertex positions and vertex attributes. Table I-3 describes the list of supported methods.

Table I-3 – Mesh vertex traversal methods

|  |  |  |
| --- | --- | --- |
| **mesh\_vertex\_traversal\_method** | **Identifier** | **Traversal Method** |
| 0 | MESH\_TRAVERSAL\_EB | Use Edge Breaker connectivity traversal as vertex order |
| 1 | MESH\_TRAVERSAL\_DEGREE | Use prediction degree traversal as vertex order |
| >1 | MESH\_TRAVERSAL\_RESERVED | Reserved |

**mesh\_position\_dequantize\_flag** equal to 1 specifies that the decoded 3D position shall be dequantized using bounding box information. mesh\_position\_dequantize\_flag equal to 0 specifies that decoded 3D position coordinates are (quantized) unsigned integers.

**mesh\_attribute\_count** indicates the number of encoded attributes.

**mesh\_attribute\_type**[ i ]indicates the attribute type of the i-th attribute where i is in the range 0 to 7. Table I-4 describes the list of supported attribute types.

Table I-4 – Mesh attributes types

|  |  |  |
| --- | --- | --- |
| **mesh\_attribute\_type[ i ]** | **Identifier** | **Attribute type** |
| 0 | MESH\_ATTR\_TEXCOORD | Texture coordinate |
| 1 | MESH\_ATTR\_NORMAL | Normal |
| 2 | MESH\_ATTR\_COLOR | Color |
| 3 | MESH\_ATTR\_MATERIAL\_ID | Material ID |
| 4 | MESH\_ATTR\_GENERIC | Generic |
| 5..6 | MESH\_ATTR\_RESERVED | Reserved |
| 7 | MESH\_ATTR\_UNSPECIFIED | Unspecified |

NOTE – Generic attributes will have their number of components specified by the value of the corresponding mesh\_attribute\_generic\_num\_components\_minus1[ index ]. Other attributes have a fixed number of components.

**mesh\_normal\_octahedral\_flag**[ i ] equals to 1 indicates that the normal attributes shall be decoded using octahedral representation. mesh\_normal\_octahedral\_flag[ i ] equals to 0 indicate that the normal attributes shall be decoded in 3D cartesian coordinates.

**mesh\_attribute\_num\_components\_minus1**[ i ] plus 1 specifies the number of components of the i-th attribute when mesh\_attribute\_type[ i ] is equal to MESH\_ATTR\_GENERIC.

**mesh\_attribute\_dequantize\_flag**[ i ] equal to 1 specifies that the decoded attributes with index i shall be dequantized using bounding box information. mesh\_attribute\_dequantize\_flag[ i ] equal to 0 specifies that decoded components of attributes with index I are (quantized) unsigned integers.

**mesh\_deduplicate\_method** specifies the method used to deduplicate positions and attributes. Table I-5 describes the list of supported methods.

Table I-5 – Mesh deduplication methods

|  |  |  |
| --- | --- | --- |
| **mesh\_deduplicate\_method** | **Identifier** | **Prediction Method** |
| 0 | MESH\_DEDUP\_NONE | None |
| 1 | MESH\_DEDUP\_DEFAULT | Default |
| > 0 | MESH\_DEDUP\_RESERVED | Reserved |

* + - 1. Mesh position encoding parameters semantics

**mesh\_position\_bit\_depth\_minus1** plus 1 specifies the number of bits used to represent 3D position coordinates.

**mesh\_position\_prediction\_method** specifies the method used to predict vertex positions and compute residuals mesh\_position\_fine\_residual[ i ][ j ], and mesh\_position\_coarse\_residual[ i ][ j ]. Table I-6 describes the list of supported methods.

Table I-6 – Mesh position prediction methods

|  |  |  |
| --- | --- | --- |
| **mesh\_position\_prediction\_method** | **Identifier** | **Prediction Method** |
| 0 | MESH\_POSITION\_MPARA | Multiple Parallelograms |
| >0 | MESH\_POSITION\_RESERVED | Reserved |

**mesh\_position\_reverse\_unification\_flag** equals to 1 indicates that the duplicated position shall be recovered. mesh\_position\_reverse\_unification\_flag equal to 0 indicates that the duplicated positions shall not be recovered.

* + - 1. Mesh position dequantize parameters semantics

**mesh\_position\_min**[ i ] specifies the minimum value of the dequantized i-th component of 3D positions. A quantized position i-th component value of 0 will be mapped to mesh\_position\_min[ i ] after dequantization.

**mesh\_position\_max**[ i ] specifies the maximum value of the dequantized i-th components of 3D positions. A quantized position i-th component value of 2mesh\_position\_bit\_depth\_minus1 +1 – 1 will be mapped to mesh\_position\_max[ i ] after dequantization.

* + - 1. Mesh attributes encoding parameters semantics

**mesh\_attribute\_bit\_depth\_minus1**[ i ]plus 1 specifies the number of bits used to represent the components of the i-th attribute.

**mesh\_attribute\_per\_face\_flag**[ i ]equal to 1 indicates that the i-th attribute has values defined per face. mesh\_attribute\_per\_face\_flag[ i ] equal to 0 indicates that the i-th attribute has values defined per vertex.

NOTE 1 – When attributes are defined per face there is no index ambiguity. When attributes are defined per vertex they can each have their own set of indices.

**mesh\_attribute\_separate\_index\_flag**[ i ] equal to 1 specifies if the i-th attribute as a specific index sequence attached. mesh\_attribute\_separate\_index\_flag[ i ] equal to 0 specifies that the i-th attribute indices are replicated from either the position indices or the indices of another attribute. When mesh\_attribute\_per\_face\_flag[ i ] is equal to 1 mesh\_attribute\_separate\_index\_flag[ i ] is set to its default value 0

**mesh\_attribute\_reference\_index\_plus1**[ i ] minus 1 specifies the reference index for the i-th attribute when mesh\_attribute\_separate\_index\_flag[ i ] is equal to 0. When equal to -1 the reference index is the position index, otherwise it specifies the attribute from which the index is used as reference.

It is a requirement of bitstream conformance that mesh\_attribute\_reference\_index\_plus1[ i ] minus 1 shall be less than i.

**mesh\_attribute\_prediction\_method**[ i ] specifies the method used to predict the i-th attribute values and compute residuals mesh\_attribute\_fine\_residual[ i ][ j ][ k ] and mesh\_attribute\_fine\_residual[ i ][ j ][ k ]. Table I-7 describes the list of supported methods for attributes with mesh\_attribute\_type equal to MESH\_ATTR\_TEXCOORD, Table I-8 for attributes with mesh\_attribute\_type equal to MESH\_ATTR\_NORMAL, Table I-9 for attributes with mesh\_attribute\_type equal to MESH\_ATTR\_COLOR, Table I-10 for attributes with mesh\_attribute\_type equal to MESH\_ATTR\_MATERIAL\_ID, Table I-11 for attributes with mesh\_attribute\_type equal to MESH\_ATTR\_GENERIC.

Table I-7 – Mesh attribute prediction methods for MESH\_ATTR\_TEXCOORD type attributes

|  |  |  |
| --- | --- | --- |
| **mesh\_attribute\_prediction\_method[ i ]** | **Identifier** | **Prediction Method** |
| 0 | MESH\_TEXCOORD\_MSTRETCH | Stretch |
| > 0 | MESH\_TEXCOORD\_RESERVED | Reserved |

Table I-8 – Mesh attribute prediction methods for MESH\_ATTR\_NORMAL type attributes

|  |  |  |
| --- | --- | --- |
| **mesh\_attribute\_prediction\_method[ i ]** | **Identifier** | **Prediction Method** |
| 0 | MESH\_NORMAL\_DELTA | Delta Coding |
| 1 | MESH\_NORMAL\_MPARA | Multiple parallelograms |
| 2 | MESH\_NORMAL\_CROSS | Cross product |
| > 2 | MESH\_NORMAL\_RESERVED | Reserved |

Table I-9 – Mesh attribute prediction methods for MESH\_ATTR\_COLOR type attributes

|  |  |  |
| --- | --- | --- |
| **mesh\_attribute\_prediction\_method[ i ]** | **Identifier** | **Prediction Method** |
| 0 | MESH\_COLOR\_DEFAULT | Default (TODO) |
| >0 | MESH\_COLOR\_RESERVED | Reserved |

Table I-10 – Mesh attribute prediction methods for MESH\_ATTR\_MATERIAL\_ID type attributes

|  |  |  |
| --- | --- | --- |
| **mesh\_attribute\_prediction\_method[ i ]** | **Identifier** | **Prediction Method** |
| 0 | MESH\_MATERIALID\_DEFAULT | Default |
| >0 | MESH\_MATERIALID\_RESERVED | Reserved |

Table I-11 – Mesh attribute prediction methods for MESH\_ATTR\_GENERIC type attributes

|  |  |  |
| --- | --- | --- |
| **mesh\_attribute\_prediction\_method[ i ]** | **Identifier** | **Prediction Method** |
| 0 | MESH\_GENERIC\_DEFAUT | Default (TODO) |
| >0 | MESH\_GENERIC\_RESERVED | Reserved |

* + - 1. Mesh attributes dequantize parameters semantics

**mesh\_attribute\_min**[ index ][ j ] specifies the minimum value of the dequantized j-th components of the i-th attribute. A quantized index-th attribute j-th component value of 0 will be mapped to mesh\_attribute\_min[ index ][ j ] after dequantization.

**mesh\_attribute\_max**[ index ][ j ] specifies the maximum value of the dequantized j-th components of the i-th attribute. A quantized index-th attribute j-th component value of 2mesh\_attribute\_bit\_depth\_minus1[ index ] + 1 – 1 will be mapped to mesh\_attribute\_max[ index ][ j ] after dequantization.

* + - 1. Mesh position coding payload semantics

**mesh\_triangle\_count** specifies the number of triangles in the decoded mesh.

**mesh\_position\_start\_count** specifies the number of start positions.

**mesh\_position\_fine\_residuals\_count** specifies the size of the mesh\_position\_fine\_residual array containing fine residual values for the 3 components of the predicted positions.

**mesh\_position\_coarse\_residuals\_count** specifies the size of the mesh\_position\_coarse\_residual array containing coarse residual values for the 3 components of the predicted positions.

**mesh\_clers\_count** specifies the number of encoded CLERS symbols.

**mesh\_cc\_with\_boundary\_count** specifies the number of connected components with boundaries comprised in the coded mesh. The number of components without boundaries is deduced from the decoded CLERS symbols sequence (see clause I.9.3).

**mesh\_handles\_count**[ i ] specifies the number of handles comprised in the i-th connected component with non zero handle count.

**mesh\_handle\_first\_delta**[ i ] specifies the difference between the i-th handle first corner and the (i – 1)-th handle first corner when i is greater than 0. When i is equal to 0 mesh\_handle\_index\_first\_delta[ 0 ] specifies the first handle first corner.

**mesh\_handle\_second\_delta**[ i ] specifies the difference between the i-th handle second corner and the (i – 1)-th handle second corner when i is greater than 0. When i is equal to 0 mesh\_handle\_index\_second\_delta[ 0 ] specifies the first handle second corner.

**mesh\_coded\_handle\_size** specifies the size in bytes of the arithmetically coded sequence of mesh\_handle\_first\_sign, mesh\_handle\_second\_shift, mesh\_handle\_first\_variable\_delta\_length4\_minus1, mesh\_handle\_first\_variable\_delta, mesh\_handle\_second\_variable\_delta\_length4\_minus1, and mesh\_handle\_second\_variable\_delta values, including the final byte alignment.

**mesh\_handle\_first\_sign**[ i ] specifies if the handle is associated with a boundary or not. When mesh\_handle\_first\_sign[ i ] is equal to 0, the corner index associated with the i-th handle first corner will be smaller than zero, indicating that the handle is associated with a boundary. When mesh\_handle\_first\_sign[ i ] is equal to 1, the corner index associated with the i-th handle first corner will be greater than zero, indicating that the handle is not associated with a boundary

**mesh\_handle\_second\_shift**[ i ] specifies the shift to apply when computing the corner index associated with the i-th handle second corner.

NOTE 1 – handle indices are relative to a triangle/face index as related corner indices can be deduced implicitly. The corner index of the fist handle is conforming to either ( 3 \* T + 2 ) or ( -3 \* T - 2 ). The corner index of the second handle f index is conforming to either ( 3 \* T + 1 ) or ( 3 \* T + 2 ). mesh\_handle\_first\_sign[ i ] and mesh\_handle\_index\_second\_shift[ i ] are used to disciminate those cases.

**mesh\_handle\_first\_variable\_delta\_length4\_minus1**[ i ] specifies the number of groups of four bits used to represent mesh\_handle\_first\_variable\_delta[ i ].

**mesh\_handle\_first\_variable\_delta**[ i ] specifies an intermediate value used to evaluate the corner index associated with the i-th handle first corner. The number of bits used to represent **mesh\_handle\_first\_variable\_delta**[ i ] is equal to ( 4 \* ( mesh\_handle\_index\_first\_variable\_delta\_length4\_minus1 + 1 ) )

**mesh\_handle\_index\_second\_variable\_delta\_length4\_minus1**[ i ] specifies the number of groups of four bits used to represent mesh\_handle\_second\_variable\_delta[ i ].

**mesh\_handle\_second\_variable\_delta**[ i ] specifies an intermediate value used to evaluate the corner index associated with the i-th handle second corner. The number of bits used to represent **mesh\_handle\_second\_variable\_delta**[ i ] is equal to ( 4 \* ( mesh\_handle\_index\_second\_variable\_delta\_length4\_minus1 + 1 ) )

**mesh\_coded\_clers\_symbols\_size** specifies the size in bytes of the arithmetically coded sequence of CLERS symbols, including the final byte alignment.

NOTE 2 – specifying mesh\_coded\_clers\_symbols\_size explicitely enables parallel parsing of CLERS symbols.

**mesh\_clers\_symbol**[ i ] specifies the value of the i-th CLERS symbol. The values are positive integers interpreted as specified by Table I-12.

Table I-12 – Interpretation of mesh\_clers\_symbol[ i ]

|  |  |
| --- | --- |
| **mesh\_clers\_symbol[ i ]** | **Identifier** |
| 0 | CLERS\_C |
| 1 | CLERS\_R |
| 3 | CLERS\_S |
| 7 | CLERS\_L |
| 15 | CLERS\_E |
| other values | CLERS\_RESERVED |

**mesh\_position\_start**[ i ][ j ] specifies the value of the j-th component of the first position of the i-th connected component as an unsigned integer coded using mesh\_position\_bit\_depth\_minus1 + 1 bits.

**mesh\_coded\_position\_fine\_residuals\_size** specifies the size in bytes of the arithmetically coded sequence of fine positions residuals, including the final byte alignment.

**mesh\_position\_fine\_residual**[ i ][ j ] specifies the value of the i-th fine position prediction residual associated with the j-th component.

**mesh\_coded\_position\_coarse\_residuals\_size** specifies the size in bytes of the arithmetically coded sequence of coarse positions residuals, including the final byte alignment.

**mesh\_position\_coarse\_residual**[ i ][ j ] specifies the value of the i-th coarse prediction residual associated with the j-th component.

* + - 1. Mesh position deduplicate information semantics

**mesh\_position\_deduplicate\_count** specifies the number of duplicated vertices.

NOTE 1 – The number of duplicated vertices is the sum of the number of original vertices that have been splitted, and the number of vertices added during the process of duplication to fix the manifoldness of the mesh.

**mesh\_position\_deduplicate\_idx**[ i ]specifies a value that is common to duplicated vertices sharing a common parent. mesh\_position\_deduplicate\_idx[ i ] values range from 0 to the number of original vertices that have been splitted minus one.

**mesh\_position\_coded\_is\_duplicate\_size** specifies the size in bytes of the arithmetically coded sequence of binary duplicate indications mesh\_position\_is\_duplicate\_flag[ i ], including the final byte alignment.

**mesh\_position\_is\_duplicate\_flag**[ i ]equal to 1 specifies that the i-th decoded vertex is a duplicated vertex. mesh\_position\_is\_duplicate\_flag[ i ] equal to 0 specifies that the i-th decoded vertex is not a duplicated vertex.

* + - 1. Mesh difference information syntax

**mesh\_position\_added\_vertices\_count** specifies the number of added vertices.

NOTE 1 – The number of added vertices is the number of vertices that shall be added to the reconstructed mesh at first.

**mesh\_position\_added\_vertices\_idx\_delta**[ i ] specifies the difference between the values of the i-th added point index and the (i – 1)-th added point index when i is greater than 0. When i equals 0, mesh\_position\_added\_vertices\_idx\_delta[ 0 ] specifies the value of the index of the first added point.

**mesh\_position\_deleted\_vertices\_count** specifies the number of deleted vertices.

NOTE 2 – The number of deleted vertices is the number of vertices of the reconstructed mesh at first that shall be deleted.

**mesh\_position\_deleted\_vertices\_idx\_delta**[ i ] specifies the difference between the values of the i-th deleted point index and the (i – 1)-th deleted point index when i is greater than 0. When i equals 0, mesh\_position\_deleted\_vertices\_idx\_delta[ 0 ] specifies the value of the index of the first deleted point.

**mesh\_modified\_count** specifies the number of modified information.

NOTE 3 – The number of modified information is the number of times the triangles that shall be modified in the reconstructed mesh at first.

**mesh\_modified\_triangles\_idx\_delta**[ i ] specifies the difference between the values of the i-th modified triangle index and the (i – 1)-th modified triangle index when i is greater than 0. When i equals 0, mesh\_modified\_triangles\_idx\_delta[ 0 ] specifies the value of the index of the first modified triangle.

**mesh\_modified\_vertices\_relative\_idx**[ i ] specifies the value of the i-th modified point relative index in the i-th modified triangles. Its range is [0, 1, 2].

**mesh\_target\_modified\_vertices\_idx\_delta**[ i ] specifies the difference between the values of the i-th target point index and the (i – 1)-th target point index when i is greater than 0. When i equals 0, mesh\_target\_modified\_vertices\_idx\_delta[ 0 ] specifies the value of the index of the first target point index.

NOTE 4 – The i-th target point index is the index of the i-th target point in the reconstructed mesh at first that the i-th modified point shall be modified to.

* + - 1. Mesh attribute coding payload semantics

**mesh\_attribute\_seams\_count**[ i ] specifies the size of the i-th attribute mesh\_attribute\_seam array.

**mesh\_coded\_attribute\_seams\_size**[ i ] specifies the size in bytes of the arithmetically coded sequence of binary seam indications mesh\_attribute\_seam[ i ][ j ], including the final byte alignment.

**mesh\_attribute\_seam**[ i ][ j ] specifies the value of the j-th binary seam indication of the i-th attribute.

**mesh\_attribute\_start\_count**[ i ] specifies the size of the mesh\_attribute\_start[ i ] array containing attribute values which are not predicted.

**mesh\_attribute\_start**[ i ][ j ][ k ] specifies the value of the k-th component of j-th value not predicted for the i-th attribute and the value of the k-th component as an unsigned integer coded using mesh\_attribute\_bit\_depth\_minus1[ i ] + 1 bits.

**mesh\_attribute\_fine\_residuals\_count**[ i ] specifies the size of the mesh\_attribute\_fine\_residual[ i ] array containing fine residual values for the NumComponents[ i ] components of the i-th attribute.

**mesh\_coded\_attribute\_fine\_residuals\_size**[ i ] specifies the size in bytes of the arithmetically coded sequence of the i-th attribute residuals mesh\_attribute\_fine\_residual[ i ][ j ][ k ], including the final byte alignment.

**mesh\_attribute\_fine\_residual**[ i ][ j ][ k ] specifies the value of the k-th component of the j-th fine prediction residual associated with the i-th attribute.

**mesh\_attribute\_coarse\_residuals\_count**[ i ] specifies the size of the mesh\_attribute\_coarse\_residual[ i ] array containing coarse residual values for the NumComponents[ i ] components of the i-th attribute.

**mesh\_coded\_attribute\_coarse\_residuals\_size**[ i ] specifies the size in bytes of the arithmetically coded sequence of the i-th attribute coarse prediction residuals mesh\_attribute\_coarse\_residual[ i ][ j ][ k ], including the final byte alignment.

**mesh\_attribute\_coarse\_residual**[ i ][ j ][ k ] specifies the value of the k-th component of the j-th coarse prediction residual associated with the i-th attribute.

* + - 1. Mesh attribute extra data semantics
      2. Mesh texcoord stretch extra data semantics

**mesh\_texcoord\_stretch\_orientations\_count**[ i ] specifies the size of the array containing orientation information mesh\_texcoord\_stretch\_orientation used to predict texture coordinates when the mesh\_attribute\_type of the i-th attribute is equal to MESH\_ATTR\_TEXCOORD and when the related mesh\_attribute\_prediction\_method is equal to MESH\_TEXCOORD\_MSTRETCH.

**mesh\_coded\_texcoord\_stretch\_orientations\_size**[ i ]specifies the size in bytes of the arithmetically coded sequence of orientations contained in the array mesh\_texcoord\_stretch\_orientation[ i ], including the final byte alignment.

**mesh\_texcoord\_stretch\_orientation**[ i ][ j ] specifies the value of the j-th orientation bit in the sequence used to generate the i-th attribute values from the arrays mesh\_attribute\_residual[ i ], and mesh\_attribute\_start[ i ].

* + - 1. Mesh materialid default extra data semantics

**mesh\_materialid\_default\_not\_equal\_count**[ i ] specifies the size of the array containing boolean information mesh\_materialid\_default\_not\_equal\_flag used to predict material ids when the mesh\_attribute\_type of the i-th attribute is equal to MESH\_ATTR\_MATERIAL\_ID and when the related mesh\_attribute\_prediction\_method is equal to MESH\_MATERIALID\_DEFAULT.

**mesh\_coded\_materialid\_default\_not\_equal\_size**[ i ]specifies the size in bytes of the arithmetically coded sequence of booleans contained in the array mesh\_materialid\_default\_not\_equal\_flag[ i ], including the final byte alignment.

**mesh\_materialid\_default\_not\_equal\_flag**[ i ][ j ] specifies the value of the j-th bit in the sequence used to generate the i-th attribute value from the arrays mesh\_attribute\_residual[ i ], and mesh\_attribute\_start[ i ].

**mesh\_materialid\_default\_left\_count**[ i ] specifies the size of the array containing boolean information mesh\_materialid\_default\_left\_flag used to predict material ids when the mesh\_attribute\_type of the i-th attribute is equal to MESH\_ATTR\_MATERIAL\_ID and when the related mesh\_attribute\_prediction\_method is equal to MESH\_MATERIALID\_DEFAULT.

**mesh\_coded\_materialid\_default\_left\_size**[ i ]specifies the size in bytes of the arithmetically coded sequence of booleans contained in the array mesh\_materialid\_default\_left\_flag[ i ], including the final byte alignment.

**mesh\_materialid\_default\_left\_flag**[ i ][ j ] specifies the value of the j-th bit in the sequence used to generate the i-th attribute value from the arrays mesh\_attribute\_residual[ i ], and mesh\_attribute\_start[ i ].

**mesh\_materialid\_default\_right\_count**[ i ] specifies the size of the array containing boolean information mesh\_materialid\_default\_right\_flag used to predict material ids when the mesh\_attribute\_type of the i-th attribute is equal to MESH\_ATTR\_MATERIAL\_ID and when the related mesh\_attribute\_prediction\_method is equal to MESH\_MATERIALID\_DEFAULT.

**mesh\_coded\_materialid\_default\_right\_size**[ i ]specifies the size in bytes of the arithmetically coded sequence of booleans contained in the array mesh\_materialid\_default\_right\_flag[ i ], including the final byte alignment.

**mesh\_materialid\_default\_right\_flag**[ i ][ j ] specifies the value of the j-th bit in the sequence used to generate the i-th attribute value from the arrays mesh\_attribute\_residual[ i ], and mesh\_attribute\_start[ i ].

**mesh\_materialid\_default\_facing\_count**[ i ] specifies the size of the array containing boolean information mesh\_materialid\_default\_facing\_flag used to predict material ids when the mesh\_attribute\_type of the i-th attribute equal to MESH\_ATTR\_MATERIAL\_ID and when the related mesh\_attribute\_prediction\_method equal to MESH\_MATERIALID\_DEFAULT.

**mesh\_coded\_materialid\_default\_facing\_size**[ i ]specifies the size in bytes of the arithmetically coded sequence of booleans contained in the array mesh\_materialid\_default\_facing\_flag[ i ], including the final byte alignment.

**mesh\_materialid\_default\_facing\_flag**[ i ][ j ] specifies the value of the j-th bit in the sequence used to generate the i-th attribute value from the arrays mesh\_attribute\_residual[ i ], and mesh\_attribute\_start[ i ].

* + - 1. Mesh normal octahedral extra data semantics

**mesh\_normal\_octahedral\_bit\_depth\_minus1**[ i ] plus 1 specifies the number of bits used to represent normal attribute when the mesh\_attribute\_type of the i-th attribute is equal to MESH\_ATTR\_NORMAL and when the related mesh\_normal\_octahedral\_flag is equal to 1.

**mesh\_normal\_octahedral\_second\_residual\_flag**[ i ] equals to 1 indicates that the second residual for normals shall be decoded. mesh\_normal\_octahedral\_second\_residual\_flag[ i ] equals to 0 indicates that the second residual for normals shall not be decoded. This shall happen when the mesh\_attribute\_type of the i-th attribute is equal to MESH\_ATTR\_NORMAL and when the related mesh\_normal\_octahedral\_flag is equal to 1.

**mesh\_normal\_octahedral\_second\_residuals\_count**[ i ] specifies the size of the mesh\_normal\_octahedral\_second\_residual[ i ] array containing residual values for three components when the mesh\_attribute\_type of the i-th attribute is equal to MESH\_ATTR\_NORMAL and when the related mesh\_normal\_octahedral\_flag is equal to 1.

**mesh\_normal\_octahedral\_second\_residuals\_size**[ i ] specifies the size in bytes of the arithmetically coded sequence of the residuals mesh\_normal\_octahedral\_second\_residual[ i ][ j ][ k ], including the final byte alignment when the mesh\_attribute\_type of the i-th attribute is equal to MESH\_ATTR\_NORMAL and when the related mesh\_normal\_octahedral\_flag is equal to 1.

**mesh\_normal\_octahedral\_second\_residual**[ i ][ j ][ k ] specifies the value of the residual associated with the k-th component of the j-th value of the i-th attribute when the mesh\_attribute\_type of the i-th attribute is equal to MESH\_ATTR\_NORMAL and when the related mesh\_normal\_octahedral\_flag is equal to 1.

* + - 1. Mesh attribute deduplicate information semantics

**mesh\_attribute\_deduplicate\_count**[ i ] specifies the number of duplicated attribute vertices for the i-th attribute.

**mesh\_attribute\_deduplicate\_idx**[ i ][ j ]specifies a value that is common to duplicated attribute vertices for the i-th attribute, sharing a common parent. mesh\_attribute\_deduplicate\_idx[ i ][ j ] values range from 0 to the number of original attribute vertices that have been splitted minus one.

**mesh\_attribute\_coded\_is\_duplicate\_size**[ i ] specifies the size in bytes of the arithmetically coded sequence of binary duplicate indications mesh\_attribute\_is\_duplicate\_flag[ i ][ j ], including the final byte alignment.

**mesh\_attribute\_is\_duplicate\_flag**[ i ][ j ]equal to 1 specifies that the j-th decoded attribute vertex of the i-th attribute is a duplicated attribute vertex.

* + - 1. Padding to byte alignment syntax

**padding\_to\_byte\_alignment\_bit\_equal\_to\_zero** shall be equal to 0.

* 1. Decoding Process
     1. General

The decoding process is specified such that all decoders will produce numerically identical outputs when invoking the decoding process for a conformant bitstream. Any decoding process that produces identical outputs to those produced by the process described herein conforms to the decoding process requirements of this document.

The decoding process takes as inputs the syntax elements and upper-case variables from Clause I.8.

The decoding process applies when mesh\_codec\_type is equal to CODEC\_TYPE\_REVERSE. Any other value for mesh\_codec\_type is not supported in this version of this document.

Outputs of this process are:

* variables and arrays specifying general mesh information as follows :
  + a variable VertCoordCount, indicating the number of vertex coordinates,
  + a variable DecVertCoordBitDepth, indicating the vertex coordinate bit depth,
  + a variable DecVertCoordDequantizeFlag, indicating if dequantization parameters are defined for vertex coordinates,
  + the following 1D arrays:
    - DecVertCoordDequantizeMin, indicating the minimal value of vertex coordinates after dequantization, where the dimension correspond to the vertex coordinates dimension,
    - DecVertCoordDequantizeMax, indicating the maximal value of vertex coordinates after dequantization, where the dimension correspond to the vertex coordinates dimension,
  + a variable AttributeCount, indicating the number of attributes,
  + the following 1D arrays:
    - AttrType, indicating the type of attribute, according to Table I-4,
    - AttrValueDimension, indicating the dimension of attribute values,
    - AttrValueCount, indicating the number of attribute values,
    - AttrValueIsPerVertex, indicating if the attribute values are defined per face or per vertex,
    - DecAttrValueBitDepth, indicating the attribute value bit depth, where the dimension corresponds to the attribute index,
    - DecAttrDequantizeFlag, indicating if dequantization parameters are defined for the attribute, where the dimension corresponds to the attribute index,
  + The following 2D arrays:
    - DecAttrDequantizeMin, indicating the minimal value of attribute components after dequantization, where the dimensions correspond to the attribute index and the attribute dimension respectively,
    - DecAttrDequantizeMax, indicating the maximal value of attribute components after dequantization, where the dimensions correspond to the attribute index and the attribute dimension respectively,
* arrays containing decoded mesh data as follows :
  + a 1D array VertCoordIndices, indicating vertex connectivity, where the dimension corresponds to the vertex coordinate value index,
  + a 2D array VertCoordValues, indicating vertex coordinate values, where the dimensions correspond to the vertex coordinate value index and the vertex coordinate value dimension index, respectively,
  + a 2D array AttrIndices, indicating attribute connectivity, where the dimensions correspond to the attribute index and the attribute value index, respectively,
  + a 3D array AttrValues, indicating attribute values, where the dimensions correspond to the attribute index, the attribute value index, and the attribute value dimension index, respectively,

The decoding process operates as follows:

* First, the global initialization process described in subclause I.9.2 is invoked.
* Next, the connectivity pre-pass process described in subclause I.9.3 is invoked.
* Next, the connectivity decoding process described in subclause I.9.4 is invoked.
* Next, the auxiliary connectivity decoding process described in subclause I.9.5 is invoked.
* Next, the primary per vertex attributes decoding process described in subclause I.9.6, is invoked.
* Next, the auxiliary per vertex attributes decoding process described in subclause I.9.7, is invoked for each attribute which index attributeIndex is in range 0 to mesh\_attribute\_count - 1 inclusive, and where mesh\_attribute\_per\_face\_flag[ attributeIndex ] is equal to 0.
* Next, the auxiliary per face attributes decoding process described in subclause I.9.8, is invoked for each attribute which index attributeIndex is in range 0 to mesh\_attribute\_count - 1 inclusive, and where mesh\_attribute\_per\_face\_flag[ attributeIndex ] is equal to 1.
* Next, if mesh\_deduplicate\_method is equal to MESH\_DEDUP\_DEFAULT, then the following applies:
  + First, the per fan deduplication process described in subclause I.9.9, is invoked for primary per vertex attributes.
  + Then the per fan deduplication process described in subclause I.9.9, is invoked for each for auxiliary per vertex attribute which index attributeIndex is in range 0 to mesh\_attribute\_count - 1 inclusive, and where mesh\_attribute\_separate\_index\_flag[ attrIndex ] is equal to 1.
* Next, if bmsps\_intra\_mesh\_post\_reindex\_method is equal to BM\_REINDEX\_VERTEX\_DEGREE, the post reindexing process described in subclause I.9.10, is invoked.
* Next, the conversion from corner table to indexed face set process described in subclause I.9.11, is invoked.
* Finally, if mesh\_position\_reverse\_unification\_flag is equal to 1, the following applies:
  + First, the duplication of per vertex position process described in subclause   
    I.9.12, is invoked.
  + Then the adjustment of the decoded mesh process described in subclause I.9.13, is invoked.
    1. Global initialization

This process defines and initializes a set of global variables and arrays that are shared among the different processes composing the decoding process.

Let the 1D arrays AuxiliaryStartIndexArray, AuxiliaryDeltaIndexArray, AuxiliaryDeltaCoarseIndexArray, and AuxiliaryUvOrientationIndexArray, of size mesh\_attribute\_count, specifying the consumption index for the fine residual values, the coarse residual values, the start values of each attribute and the UV orientations respectively, be initialized as follows.

for( i = 0; i < mesh\_attribute\_count; i++ ) {  
 AuxiliaryStartIndexArray[ i ] = 0  
 AuxiliaryDeltaIndexArray[ i ] = 0  
 AuxiliaryDeltaCoarseIndexArray[ i ] = 0  
 AuxiliaryUvOrientationIndexArray[ i ] = 0  
 }

Let the 1D array NormalSecondResidualIndexArray specify the consumption index for the normal second residual values when the mesh\_attribute\_type of the i-th attribute is equal to MESH\_ATTR\_NORMAL and when the related mesh\_normal\_octahedral\_flag is equal to 1.

NormalSecondResidualIndexArray = 0

Let the variable ConnectedComponentCnt, specifying the number of connected components be initialized as follows:

ConnectedComponentCnt = 0

Let the variable CornerCnt, specifying the number of corners, be initialized as follows:

CornerCnt = 3 \* mesh\_triangle\_count

Let the variable DecVertCoordBitDepth be initialized as follows:

DecVertCoordBitDepth = mesh\_position\_bit\_depth\_minus1 + 1

Let the variable PrimaryVerticesCnt, specifying the number of vertices for the decoded primary connectivity before deduplication is optionally applied, be initialized as follows:

PrimaryVerticesCnt = NumPositionStart + NumPredictedFinePositions  
 + NumPredictedCoarsePositions+ NumAddedDuplicatedVertex

Then, the array VertCoordValues is initialized as follows:

for(i = 0; i < PrimaryVerticesCnt; i++ ) {  
 VertCoordValues[ i ][ 0 ] = 0  
 VertCoordValues[ i ][ 1 ] = 0  
 VertCoordValues[ i ][ 2 ] = 0  
 }

The variable DecAttrDequantizeFlag and the arrays DecVertCoordDequantizeMin and DecVertCoordDequantizeMax are initialized as follows:

DecAttrDequantizeFlag = mesh\_position\_dequantize\_flag  
 for(i = 0; i < 3 ; i++ ) {  
 if( DecAttrDequantizeFlag[ attrIndex ] ) {  
 DecVertCoordDequantizeMin[ attrIndex ][ c ] =  
 mesh\_position\_min[ attrIndex ][ c ]  
 DecVertCoordDequantizeMax[ attrIndex ][ c ] =  
 mesh\_position\_max[ attrIndex ][ c ]  
 } else {  
 DecVertCoordDequantizeMin[ attrIndex ][ c ] = 0  
 DecVertCoordDequantizeMax[ attrIndex ][ c ] = 0  
 }  
 }

Let OppositeCornersArray be a 1D array, of size CornerCnt, specifying for the primary attribute, the opposite corner index for each corner index in the range of 0 to CornerCnt - 1, inclusive.

Let CornerToVertexArray be a 1D array, of size CornerCnt, specifying for the primary attribute, the vertex index for each corner index in the range of 0 to CornerCnt - 1, inclusive.

Then, the 1D arrays OppositeCornersArray and CornerToVertexArray are initialized as follows:

for( i = 0; i < CornersCnt; i++ ) {  
 OppositeCornersArray[ i ] = -1  
 CornerToVertexArray[ i ] = -1  
 }

Let AuxiliaryOppositeCornersArray be a 2D array, of size mesh\_attribute\_count × CornerCnt, where the dimensions specify the index of the auxiliary attribute, and the opposite corner index for each corner index in the range of 0 to CornerCnt - 1 inclusive, respectively.

Let AuxiliaryCornerToVertexArray be a 2D array, of size mesh\_attribute\_count × CornerCnt, where the dimensions specify the index of the auxiliary attribute, and the vertex index for each corner index in the range of 0 to CornerCnt – 1 inclusive, respectively.

The 2D arrays AuxiliaryOppositeCornersArray and AuxiliaryCornerToVertexArray are initialized as follows:

for( i = 0; i < mesh\_attribute\_count; i++)  
 for( j = 0; j < CornersCnt; j++) {  
 AuxiliaryOppositeCornersArray[ i ][ j ] = -3  
 AuxiliaryCornerToVertexArray[ i ][ j ] = -1  
 }  
 }

Then, the variable AttributeCount and the 1D arrays AttrType, AttrValueDimension, AttrValueCount, AttrValueIsPerVertex DecAttrValueBitDepth and DecAttrDequantizeFlag of size AttributeCount are initialized as follows:

AttributeCount = mesh\_attribute\_count  
 for( attrIndex = 0; attrIndex < AttributeCount; attrIndex++ ) {  
 AttrType[ attrIndex ] = mesh\_attribute\_type[ attrIndex ]  
 AttrValueDimension[ attrIndex ] = NumComponents[ attrIndex ]  
 AttrValueIsPerVertex[ attrIndex ] = !mesh\_attribute\_per\_face\_flag[ attrIndex ]  
 DecAttrValueBitDepth[ attrIndex ] =  
 mesh\_attribute\_bit\_depth\_minus1[ attrIndex ] + 1  
 DecAttrDequantizeFlag[ attrIndex ] = mesh\_attribute\_dequantize\_flag[ attrIndex ]  
 if( mesh\_attribute\_per\_face\_flag[ attrIndex ] )  
 AttrValueCount[ attrIndex ] = mesh\_triangle\_count  
 else if( !mesh\_attribute\_separate\_index\_flag[ attrIndex ] ) {  
 if( mesh\_attribute\_reference\_index\_plus1 == 0 )  
 AttrValueCount[ attrIndex ] = PrimaryVerticesCnt  
 else   
 AttrValueCount[ attrIndex ] =  
 AttrValueCount[ mesh\_attribute\_reference\_index\_plus1 – 1 ]  
 }  
 else  
 AttrValueCount = mesh\_attribute\_start\_count[ attrIndex ]  
 + mesh\_attribute\_residuals\_fine\_count[ attrIndex ]  
 + mesh\_attribute\_residuals\_coarse\_count[ attrIndex ]  
 + NumAddedDuplicatedAttribute[ attrIndex ]  
 }

Then, the 2D array DecAttrDequantizeMin and DecAttrDequantizeMin are initialized as follows:

for( attrIndex = 0; attrIndex < AttributeCount; attrIndex++ ) {  
 for( c = 0; c < AttrValueDimension[ attrIndex ]; c++ ) {  
 if( DecAttrDequantizeFlag[ attrIndex ] ) {  
 DecAttrDequantizeMin[ attrIndex ][ c ] =  
 mesh\_attribute\_min[ attrIndex ][ c ]  
 DecAttrDequantizeMax[ attrIndex ][ c ] =  
 mesh\_attribute\_max[ attrIndex ][ c ]  
 } else {  
 DecAttrDequantizeMin[ attrIndex ][ c ] = 0  
 DecAttrDequantizeMax[ attrIndex ][ c ] = 0  
 }  
 }  
 }

Then, the 3D array AttrValues is initialized as follows:

for( attrIndex = 0; attrIndex < AttributeCount; attrIndex++ ) {  
 for( i = 0; i < AttrValueCount[ attrIndex ]; i++ ) {  
 for( c = 0; c < AttrValueDimension[ attrIndex ]; c++ ) {  
 AttrValues[ attrIndex ][ i ][ c ] = 0  
 }  
 }  
 }

Let FirstTriangleArray be a 1D array of size mesh\_cc\_with\_boundary\_count, specifying for each connected component of the primary attribute with a boundary, a triangle index used to detect the end of the connected component in the CLERS table, initialized as follows:

for( i = 0; i < mesh\_cc\_with\_boundary\_count; i++ ) {  
 FirstTriangleArray[ i ] = 0  
 }

Let VertexMarkingArray be a 1D array of size PrimaryVerticesCnt, where all values are initialized to 0 that specifies if a vertex is marked (element value = 1) or not marked (element value = 0). This table may be resized by processes to be used for marking vertices from auxiliary topologies.

Let the variable ProcessedCornersCnt, specifying the size of the ProcessedCornersArray array, be set to 0.

Let ProcessedCornersArray, be a 1D array, of size ProcessedCornersCnt, specifying the corners in their order of visit by the connectivity decoding, initially empty.

Let the variable ReadCornersCnt, specifying the size of the ReadCornersArray array, be initialized to 0.

Let ReadCornersArray be a 1D array, of size ReadCornersCnt, specifying the corners where a vertex must be predicted in their order of visit, initially empty.

The variable TraversalCornersCnt, specifying the size of the TraversalCornersArray array, is initialized to 0.

Let TraversalCornersArray be a 1D array, of size TraversalCornersCnt, specifying a sequence of corners defining a vertex traversal order, initially empty.

The variable DuplicatesCnt, initialized to 0, that specifies the number of duplicate vertices.

Let DuplicatesArray be a 2D array, of size DuplicatesCnt × 2, initially empty, specifying for each corner associated to a duplicated vertex the vertex index of the first traversed corner associated to a duplicated vertex sharing the same duplicate group.

Let SkippedCornersArray be a 1D array, of size CornerCnt, where all values are initialized to -1, specifying if corner attributes shall be predicted (value equal to -1) or if prediction shall be skipped and use the existing predicted value from the vertex index stored in the element of the table.

Let a 2D array HandlesArray, of size mesh\_handles\_count × 2, specifying for each handle two associated corner indices, be derived as follows:

* Let the variables handleFirst, handleSecond, firstSign and secondSign be initialized to 0.
* If mesh\_handles\_count is less than MinHandles, the following applies:

for( i = 0; i< mesh\_handles\_count; i++ ) {  
 handleFirst += mesh\_handle\_first\_delta[ i ]  
 handleSecond += mesh\_handle\_second\_delta[ i ]  
 HandlesArray[ i ][ 0 ] = handleFirst  
 HandlesArray[ i ][ 1 ] = handleSecond  
 }

* Else, if mesh\_handles\_count is greater than or equal to MinHandles, the following applies:

for( i = 0; i< mesh\_handles\_count; i++ ){  
 firstSign = 1 - 2 \* ( mesh\_handle\_first\_variable\_delta[ i ] & 1)  
 handleFirst += firstSign \* mesh\_handle\_first\_variable\_delta[ i ] + 1 ) / 2  
 secondSign = 1 - 2 \* ( mesh\_handle\_second\_variable\_delta[ i ] & 1)  
 handleSecond += secondSign \* mesh\_handle\_second\_variable\_delta[ i ] + 1 ) / 2  
 HandlesArray[ i ][ 0 ] =  
 ( 3 \* handleFirst + 2 ) \* (2 \* mesh\_handle\_first\_sign[ i ] - 1 )  
 HandlesArray[ i ][ 1 ] =  
 ( 3 \* handleSecond + 1 ) + mesh\_handle\_second\_shift[ i ]  
 }

Let the variable CLERS\_P, specifying the end of a connected component with no boundaries, be initialized to -1. The letter P stands for parenthesis since the connectivity pre-pass described in clause I.9.3 does open and closes virtual parenthesis to detect such components.

Let the variable ExtendedClersSymbolsCnt, specifying the size of the ExtendedClersSymbolsArray array, be initialized to 0.

Let the 1D array ExtendedClersSymbolsArray, of size ExtendedClersSymbolsCnt, specifying the sequence of extended CLERS symbols, be initially empty. The values of array elements shall be either shall be either CLERS\_P, or values from the list specified in Table I-12.

Let the variable ConnectedComponentCnt, specifying the number of connected components of the primary connectivity, be initialized to 0.

Let the variable PrimaryStartCornersCnt, specifying the number of start corners of the primary connectivity be initialized to 0.

Let the 1D array PrimaryStartCornersArray, of size PrimaryStartCornersCnt, specifying the initial corner index of each connected component of the primary connectivity, be initially empty.

Let the variable AuxiliaryStartCornersCount, specifying the number of start corners of the auxiliary connectivity be initialized to 0.

Let the 1D array AuxiliaryStartCornersArray, of size AuxiliaryStartCornersCount, specifying the initial corner index of each connected component of the auxiliary connectivity, be initially empty.

If the mesh\_position\_reverse\_unification\_flag is equal to 1, the following applies:

* Let 1D array addPointList, of size mesh\_position\_added\_vertices\_count, and the variable addedIdx, both specifying the index of added per vertex position, be initialized to all 0s. If mesh\_position\_added\_vertices\_count is greater than 0, the following applies

for( i = 0; i< mesh\_position\_added\_vertices\_count; i++ ) {  
 addedIdex += mesh\_position\_added\_vertices\_idx\_delta[ i ]  
 addPointList[ i ] = addedIdex  
 }

* Let 1D array deletedPointList, of size mesh\_position\_deleted\_vertices\_count, and the variable deletedIdx, both specifying the index of deleted per vertex position, be initialized to all 0s. If mesh\_position\_deleted\_vertices\_count is greater than 0, the following applies

for( i = 0; i< mesh\_position\_deleted\_vertices\_count; i++ ) {  
 deletedIdex += mesh\_position\_deleted\_vertices\_idx\_delta[ i ]  
 deletedPointList[ i ] = deletedIdex  
 }

* Let 1D array modifiedTriangles, of size mesh\_modified\_count, and the variable modifiedTri, both specifying the modified triangle index, be initialized to all 0s.
* Let 1D array modifiedVertices, of size mesh\_modified\_count, specifying the modified vertex relative index in the modified triangles, be initialized to all 0s.
* Let 1D array targetVertices, of size mesh\_modified\_count, and the variable targetVert, both specifying the target vertex index of the modified vertex, be initialized to all 0s.
* If mesh\_modified\_count is greater than 0, the following applies

for( i = 0; i< mesh\_modified\_count; i++ ) {  
 modifiedTri += mesh\_modified\_triangles\_idx\_delta[ i ]  
 modifiedTriangles[ i ] = modifiedTri  
 modifiedVertices[ i ] = mesh\_modified\_vertices\_relative\_idx  
 targetVert += mesh\_target\_modified\_vertices\_idx\_delta[ i ]  
 targetVertices[ i ] = targetVert  
 }

* + 1. Connectivity pre pass

This process updates the array of opposite corners of the primary connectivity OppositeCornersArray using the handles information, it determines the number of connected components ConnectedComponentCnt, and generates an extended CLERS table ExtendedClersSymbolsArray to be used by the connectivity decoding process described in clause I.9.4.

This process modifies:

* the variables ConnectedComponentCnt, PrimaryStartCornersCnt and ExtendedClersSymbolsCnt, defined in clause I.9.2.
* the arrays OppositeCornersArray, ExtendedClersSymbolsArray, PrimaryStartCornersArray, and FirstTriangleArray, defined in clause I.9.2.

The following applies:

* a variable clersIndex, indicating the current CLERS symbol to process, is initialized to 0,
* a variable triangleIndex, indicating the current processed triangle, is initialized to 0,
* a variable componentIndex, indicating the current connected component processed, is initialized to 0,
* The OppositeCornersArray array is first updated as follows:

for( i = 0; i < mesh\_handles\_count; ++i) {  
 h0 = HandlesArray[ i ][ 0 ]  
 h1 = HandlesArray[ i ][ 1 ]  
 if(h0 > 0) {  
 OppositeCornersArray[ h0 ] = h1  
 OppositeCornersArray[ h1 ] = h0  
 } else {  
 OppositeCornersArray[ -h0 ] = -h1  
 OppositeCornersArray[ h1 ] = h0  
 }

* The ExtendedClersSymbolsArray, PrimaryStartCornersArray, and FirstTriangleArray arrays, and the variables PrimaryStartCornersCnt and ExtendedClersSymbolsCnt are derived as follows:

parenthesisCnt = 0  
 if( mesh\_cc\_with\_boundary\_count == 0 ) {  
 PrimaryStartCornersArray[ PrimaryStartCornersCnt++ ] = PreviousCorner( 0 )  
 ExtendedClersSymbolsArray[ ExtendedClersSymbolsCnt++ ] = CLERS\_P  
 triangleIndex++  
 }  
 else  
 PrimaryStartCornersArray[ PrimaryStartCornersCnt++ ] = 0  
 while( triangleIndex < mesh\_triangle\_count ) {   
 currentClersSymbol = mesh\_clers\_symbol[ clersIndex++ ]  
 currentCorner = 3 \* triangleIndex  
 ExtendedClersSymbolsArray[ ExtendedClersSymbolsCnt++ ] = currentClersSymbol  
 triangleIndex++  
 if( currentClersSymbol == CLERS\_S   
 && ( LeftCorner( OppositeCornersArray, currentCorner ) ) == -1 ) )  
 parenthesisCnt++  
 if( currentClersSymbol == CLERS\_E ) {  
 if( parenthesisCnt > 0)  
 parenthesisCnt --  
 else {  
 componentIndex++  
 if( triangleIndex < mesh\_triangle\_count ) {  
 currentStartCorner = triangleIndex \* 3  
  if( componentIndex >= mesh\_cc\_with\_boundary\_count )   
 currentStartCorner = PreviousCorner( currentStartCorner )  
 PrimaryStartCornersArray[ PrimaryStartCornersCnt++ ] =  
 currentStartCorner  
 }  
 if( componentIndex < mesh\_cc\_with\_boundary\_count ) {  
 FirstTriangleArray[ componentIndex ] = triangleIndex  
 }  
 if( ( componentIndex >= mesh\_cc\_with\_boundary\_count )  
 && triangleIndex < mesh\_triangle\_count ) {  
 ExtendedClersSymbolsArray[ ExtendedClersSymbolsCnt++ ] = CLERS\_P  
 triangleIndex++  
 }  
 }  
 }  
 }  
 ConnectedComponentCnt = componentIndex

* + 1. Connectivity decoding
       1. General

This process updates the array of opposite corners of the primary connectivity OppositeCornersArray using handle descriptors. It updates the vertex indices associated to the corners of the primary connectivity. During the process, it generates the traversal arrays ReadCornersArray and ProcessedCornersArray.

This process modifies:

* the variables ReadCornersCnt, and ProcessedCornersCnt defined in clause I.9.2.
* the arrays OppositeCornersArray, ReadCornersArray, and ProcessedCornersArray defined in clause I.9.2.

This process uses the functions DecodeBoundary, CloseStar, PreviousCorner, NextCorner, RightCorner, and SetMatch that are defined in subclauses I.9.4.2, I.9.4.3, I.9.15.1, I.9.15.2, I.9.15.4, and I.9.4.4 respectively.

The following applies:

* Let a variable cornerStackCnt be initialized to 0, and a 1D array cornerStackArray be initially empty,
* Let the variables triangleIndex, vertexIndex, and componentIndex be set as follows:

triangleIndex = ExtendedClersSymbolsCnt - 1  
 vertexIndex = PrimaryVerticesCnt – 1  
 componentIndex = ConnectedComponentCount

* While componentIndex > 0, the following applies repeatedly:
  + Let the variables cornerIndex and processNextClersSymbol be set as follows:

cornerIndex = -1  
 processNextClersSymbol = ( triangleIndex >= 0 &&  
 ( ( componentIndex > mesh\_cc\_with\_boundary\_count ) ||  
 ( ( componentIndex > 0 ) &&  
 ( triangleIndex >= FirstTriangleArray[componentIndex - 1] ) ) )

* + While processNextClersSymbol is not equal to 0 the following applies repeatedly:

currentClersSymbol = ExtendedClersSymbolsArray[ triangleIndex]  
 if( currentClersSymbol == CLERS\_C ) {  
 SetMatch( cornerIndex , 3 \* triangleIndex + 1 )  
 CloseStar( 3 \* triangleIndex + 2, vertexIndex + 0 )  
 ReadCornersArray[ ReadCornersCnt++ ] = NextCorner( triangleIndex + 2 )  
 vertexIndex--  
 } else if( currentClersSymbol == CLERS\_L ) {  
 SetMatch( cornerIndex, 3 \* triangleIndex + 1 )  
 } else if( currentClersSymbol == CLERS\_R ) {  
 SetMatch( cornerIndex, 3 \* triangleIndex + 2 )  
 } else if( currentClersSymbol == CLERS\_S ) {  
 SetMatch( cornerIndex, 3 \* triangleIndex + 1 )  
 cornerIndex = 3 \* triangleIndex + 2  
 if( OppositeCornersArray[ cornerIndex ] == -1 ) {  
 SetMatch( cornerIndex , cornerStackArray[ cornerStackCnt - 1 ] )  
 cornerStackCnt--  
 } else if( OppositeCornersArray[ cornerIndex ] < 0 ) {  
 SetMatch( cornerIndex, -OppositeCornersArray[ cornerIndex ] )  
 cornerIndex = PreviousCorner( cornerIndex )  
 while( RightCorner( OppositeCornersArray, cornerIndex) >= 0 ) {  
 cornerIndex = RightCorner( OppositeCornersArray, cornerIndex )  
 }  
 DecodeBoundary( PreviousCorner( cornerIndex ), vertexIndex )  
 }  
 } else if( currentClersSymbol == CLERS\_E ) {  
 if( cornerIndex > 0 ) {  
 cornerStackArray[ cornerStackCnt++ ] = cornerIndex  
 }  
 } else if( currentClersSymbol == CLERS\_P ) {  
 SetMatch( cornerIndex, 3 \* triangleIndex )  
 CloseStar( 3 \* triangleIndex + 1, vertexIndex - 2 )  
 CloseStar( 3 \* triangleIndex + 2, vertexIndex - 1 )  
 CloseStar( 3 \* triangleIndex + 0, vertexIndex - 0 )  
 ReadCornersArray[ ReadCornersCnt++ ] = 3 \* triangleIndex + 1  
 ReadCornersArray[ ReadCornersCnt++ ] = 3 \* triangleIndex + 0  
 ReadCornersArray[ ReadCornersCnt++ ] = 3 \* triangleIndex + 2  
 vertexIndex -= 3  
 componentIndex--  
 cornerIndex = -1  
 }  
 cornerIndex = 3 \* triangleIndex  
 if( currentClersSymbol == CLERS\_P )  
 ProcessedCornersArray[ ProcessedCornersCnt++ ] =   
 PreviousCorner( cornerIndex )  
 else  
 ProcessedCornersArray[ ProcessedCornersCnt++ ] = cornerIndex)  
 triangleIndex--  
 processNextClersSymbol = ( triangleIndex >= 0 &&  
 ( ( componentIndex > boundaryComponentCount ) ||  
 ( ( componentIndex >= 0 ) &&  
 ( triangleIndex >=  
 FirstTriangleArray[componentIndex - 1] ) ) )

* + Then the DecodeBoundary function is invoked when the mesh has connected components with boundaries as follows:

if( mesh\_cc\_with\_boundary\_count != 0 ){  
 componentIndex--  
 vertexIndex = DecodeBoundary( 3 \* FirstTriangleArray[ componentIndex ] + 1,  
 vertexIndex ) }

* + - 1. Decoding of boundaries

Let the function DecodeBoundary(cornerIndex, vertexIndex) be defined as follows:

DecodeBoundary( cornerIndex, vertexIndex ) {  
 while( LeftCorner( OppositeCornersArray, cornerIndex) >= 0 )   
 cornerIndex = LeftCorner( OppositeCornersArray, cornerIndex )  
 }  
 do {  
 ReadCornersArray[ ReadCornersCnt++ ] = cornerIndex  
 VertexMarkingArray[ vertexIndex ] = 1  
 SetVertexIndex( CornerToVertexArray, cornerIndex, vertexIndex )  
 cornerIndex = PreviousCorner( cornerIndex )  
 SetOpposite( OppositeCornersArray, cornerIndex, -3 )  
 while( LeftCorner( OppositeCornersArray, cornerIndex ) >= 0) {  
 cornerIndex = LeftCorner( OppositeCornersArray, cornerIndex)  
 SetVertexIndex( CornerToVertexArray,  
 NextCorner( cornerIndex ), vertexIndex )  
 }  
 vertexindex--  
 } while( GetVertexIndex( CornerToVertexArray, cornerIndex ) < 0 )  
 return vertexIndex  
 }

Where the functions PreviousCorner, NextCorner, LeftCorner, SetVertexIndex, and GetVertexIndex are defined in subclauses I.9.15.1, I.9.15.2, I.9.15.3, I.9.15.6, and I.9.15.7respectively.

* + - 1. Closing vertex star

Let the function CloseStar be defined as follows, where the variable borderIndex specifies the index of a corner on a border:

CloseStar( cornerIndex, vertexIndex ) {  
 MarkedVertexArray[ vertexIndex ] = 1  
 borderIndex = cornerIndex  
 while( LeftCorner( OppositeCornersArray, borderIndex ) >= 0  
 && LeftCorner( OppositeCornersArray, borderIndex ) != cornerIndex ) {  
 SetVertex( NextCorner( borderIndex ), vertexIndex )  
 borderIndex = LeftCorner( OppositeCornersArray, borderIndex )  
 }  
 SetVertexIndex( CornerToVertexArray, NextCorner( borderIndex ), vertexIndex )  
 SetMatch( PreviousCorner( borderIndex ) , cornerIndex )  
 }

Where the functions PreviousCorner , NextCorner, LeftCorner, SetVertexIndex, and SetMatch are defined in subclauses I.9.15.1, I.9.15.2, I.9.15.3, I.9.15.6, and I.9.4.4, respectively.

* + - 1. Setting matching corners

Let the function SetMatch be defined as follows:

SetMatch( c0, c1 ) {  
 if( ( c0 > 0 ) && ( c0 < CornerCnt ) ) OppositeCornersArray[ c0 ] = c1  
 if( ( c1 > 0 ) && ( c1 < CornerCnt ) ) OppositeCornersArray[ c1 ] = c0  
}

* + 1. Auxiliary connectivity decoding

This process decodes connectivity information for all attributes having separate indices as specified by the values in the mesh\_attribute\_separate\_index\_flag array.

This process modifies:

* The arrays AuxiliaryOppositeCornersArray and AuxiliaryCornerToVertexArray defined in clause I.9.2
* The variable AuxiliaryStartCornersCount and the array AuxiliaryStartCornersArray defined in clause I.9.2

This process invokes the functions PreviousCorner NextCorner, and CornerToTriangle, that are defined in subclauses I.9.15.1, I.9.15.2, and I.9.15.5, respectively.

For attrIdx = 0 to mesh\_attribute\_count – 1 inclusive, if mesh\_attribute\_separate\_index\_flag[ attrIdx ] is equal to 1, the following applies:

* a 1D array markedCornersArray, of size CornerCnt is initialized to 0,
* If mesh\_vertex\_traversal\_method is not equal to MESH\_TRAVERSAL\_EB, the following applies:

for( i = 0; i < CornerCnt; i++ ) {  
 AuxiliaryOppositeCornersArray[ attrIdx ][ i ] = OppositeCornersArray[ i ]  
 }  
 AuxiliaryStartCornersCount[ attrIdx ] = PrimaryStartCornersCnt  
 for( i = 0; i < AuxiliaryStartCornersCount[ attrIdx ]; i++ ) {  
 AuxiliaryStartCornersArray[ attrIdx ][ i ] = PrimaryStartCornersArray[ i ]  
 }

* The AuxiliaryOppositeCornersArray[ attrIdx ] array, and optionally the variable auxiliaryStartCornersCount[ attrIdx ] and the array AuxiliaryStartCornersArray[ attrIdx ], are updated as follows. Let the variable seamsIndicatorIndex, specifying the consumption index for the array of seam mesh\_attribute\_seam, be initialized to 0. Let the variables processedCornerIndex, and srcFaceIndex, respectively describing the index of a corner and the index of the face associated to this corner, be initialized to 0. Let the 1D array cornersArray, of size 3, specifying the three corners of the triangle of index srcFaceIndex. Let the variables curCorner, and oppCorner, respectively specifying a corner of the triangle of index srcFaceIndex, and its opposite corner, be initialized to 0. Let the variable oppFaceIndex, specifying the index of the face attached to the corner oppCorner, be initialized to 0. The following applies:

seamsIndicatorIndex = 0  
 for( i = 0; i < ProcessedCornersCnt; i++ ) {  
 processedCornerIndex = ProcessedCornersArray[ i ]  
 srcFaceIndex = CornerToTriangle( processedCornerIndex )  
 cornersArray = { processedCornerIndex, NextCorner( processedCornerIndex ),  
 PreviousCorner( processedCornerIndex ) }  
 for( c = 0; c < 3; ++c ) {  
 curCorner = cornersArray[ c ]  
 oppCorner = OppositeCornersArray[ curCorner ]  
 if( oppCorner < 0 || markedCornersArray[ oppCorner ] > 0 )  
 continue  
 markedCornersArray[ oppCorner] = 1  
 oppFaceIndex = CornerToTriangle( opp\_corner )  
 if( oppFaceIndex > srcFaceIndex )  
 continue  
 if( mesh\_attribute\_seam[ seamsIndicatorIndex++ ] ) {  
 AuxiliaryOppositeCornersArray[ attrIdx ][ curCorner ] = -2  
 AuxiliaryOppositeCornersArray[ attrIdx ][ oppCorner ] = -2  
 if( mesh\_vertex\_traversal\_method != MESH\_TRAVERSAL\_EB ) {  
 AuxiliaryStartCornersArray  
 [ attrIdx ][ AuxiliaryStartCornersCount[ attrIdx ]++ ]  
  = cur\_corner  
 AuxiliaryStartCornersArray  
 [ attrIdx ][ AuxiliaryStartCornersCount[ attrIdx ]++ ]  
 = opp\_corner  
 }  
 }  
 }  
 }

* Then attribute indices are generated, and the AuxiliaryCornerToVertexArray and AuxiliaryOppositeCornersArray arrays are updated as follow. Let the variable attributeIndicesCnt, specifying the number of attribute indices, be initialized to 0. Let the variables cornerIndex, newIndex, and movC, respectively specifying one corner of the triangle attached to the corner processedCornerIndex, the new attribute index, and movC a corner index to iterate around the corner processedCornerIndex, be initialized to 0. The following applies:

attributeIndicesCnt = 0  
 for( i = 0; i < ProcessedCornersCnt; i++ ) {  
 processedCornerIndex = ProcessedCornersArray[ i ]  
 cornersArray = { processedCornerIndex, NextCorner( processedCornerIndex ),  
 PreviousCorner( processedCornerIndex ) }  
 for( c = 0; c < 3; ++c ) {  
 cornerIndex = cornersArray[ c ]  
 if( AuxiliaryCornerToVertexArray[ attrIdx ][ cornerIndex ] >= 0)  
 continue  
 newIndex = attributeIndicesCnt++  
 AuxiliaryCornerToVertexArray[ attrIdx ][ cornerIndex ] = newIndex  
 movC = cornerIndex   
 while( OppositeCornersArray[ NextCorner( movC ) ] >= 0  
 && AuxiliaryOppositeCornersArray  
 [ attrIdx ][ NextCorner( movC ) ] >= -1 ) {  
 movC = NextCorner( OppositeCornersArray[ NextCorner( movC ) ] )  
 AuxiliaryCornerToVertexArray[ movC ] = newIndex  
 if( movC == cornerIndex ) break  
 }  
 movC = cornerIndex  
 while( OppositeCornersArray[ PreviousCorner( movC ) ] >= 0  
 && AuxiliaryOppositeCornersArray  
 [ attrIdx ][ PreviousCorner( movC ) ] >= -1 ) {  
 movC = PreviousCorner( OppositeCornersArray[PreviousCorner( movC)])  
 AuxiliaryCornerToVertexArray[ attrIdx ][ movC ] = newIndex  
 if( movC == cornerIndex ) break  
 }  
 }  
 }

* + 1. Decoding primary per vertex attributes
       1. General

This process has no direct output.

The following applies:

* If mesh\_vertex\_traversal\_method is equal to MESH\_TRAVERSAL\_EB, then the variable TraversalCornersCnt is set to ReadCornersCnt, the 1D array TraversalCornersArray size is TraversalCornersCnt, and the following applies:

for( i = 0; i< TraversalCornersCnt; i++ ){  
 TraversalCornersArray[ i ]= ReadCornersArray[ i ]  
 }

* Else, if mesh\_vertex\_traversal\_method is equal to MESH\_TRAVERSAL\_DEGREE, then the process described in subclause I.9.12 is invoked with the variables PrimaryVerticesCnt, mesh\_triangle\_count, PrimaryStartCornersCnt, and arrays PrimaryStartCornersArray, and OppositeCornersArray, and CornerToVertexArray as inputs, and the outputs are the variable TraversalCornersCnt and the 1D array TraversalCornersArray of size TraversalCornersCnt.
* Else, any other value of mesh\_vertex\_traversal\_method is unsupported.
* Finally, the attribute values are derived by invoking subclause I.9.6.2 with variable TraversalCornersCnt and array TraversalCornersArray as inputs.
  + - 1. Decoding and deduplication of per vertex position attributes

Inputs to this process are:

* a variable traversalOrderCnt, specifying the number of corners of the traversal order array,
* a 1D array traversalOrderArray, of size traversalOrderCnt, specifying the indices of corners to use for the traversal in their order of appearance in the array.

This process modifies the arrays SkippedCornersArray, VertexMarkingArray and DuplicatesArray.

Let the variable deduplicate be set to 1 if mesh\_deduplicate\_methodis not equal to MESH\_DEDUP\_NONE, or to 0 if mesh\_deduplicate\_method is equal to MESH\_DEDUP\_NONE.

Let the 1D array processedDupIdxArray of size NumSplitVertex, specifying the vertex index associated with a group of duplicated vertices identified by a split index from the array mesh\_position\_deduplicate\_idx be initialized as follows:

for( i = 0; i < NumSplitVertex; i++ ) {  
 processedDupIdxArray[ i ] = -1  
 }

Let positionDecodeWithPrediction(c) refer to the invocation of the process described in subclause I.9.6.3 with variables c as input. The function eventually modifies the two arrays VertexMarkingArray and VertexPositionsArray as well as the counters StartIndex and DeltaIndex that are eventually incremented.

The following applies:

for( i = 0; i < CornerCnt; i++ ) {  
 SkippedCornersArray[ i ] = -1  
 }  
 for( i = 0; i < PrimaryVerticesCnt; i++ ) {  
 VertexMarkingArray[ i ] = -1  
 }

Let the variable currentDuplicate, specify the index of the current duplicated vertex, and the variable c specify the index of a corner, be initialized as follow:

c = 0  
 currentDuplicate = 0

The following applies:

/\* goes through the corners, for the positions \*/  
 for( currentVertex = 0; currentVertex < traversalOrderCnt; currentVertex++ ) {  
 c = traversalOrder[ currentVertex ]  
 /\* perform deduplication if needed \*/  
 if( deduplicate && NumPositionIsDuplicateFlags != 0  
 && mesh\_position\_is\_duplicate\_flag[ currentVertex ] )  
 {  
 splitIdx = mesh\_position\_deduplicate\_idx[ currentDuplicate++ ]  
 dupVtx = processedDupIdxArray[ splitIdx ]  
 if( dupVtx > =0 )  
 {  
 DuplicatesArray[ ++DuplicatesCnt ][ 0 ] = c  
 DuplicatesArray[ ++DuplicatesCnt ][ 1 ] = dupVtx  
   
 vertexPositionsArray[ CornerToVertexArray[ c ] ][ 0 ] =  
 vertexPositionsArray[ dupVtx ][ 0 ]  
 vertexPositionsArray[ CornerToVertexArray[ c ] ][ 1 ] =  
 vertexPositionsArray[ dupVtx ][ 1 ]  
 vertexPositionsArray[ CornerToVertexArray[ c ] ][ 2 ] =  
 vertexPositionsArray[ dupVtx ][ 2 ]  
 /\* used by attributes using main index table \*/  
 SkippedCornersArray[ c ] = dupVtx  
 }  
 else  
 {  
 processedDupIdx[ splitIdx ] = CornerToVertexArray[ c ]  
 }  
 }  
 /\* perform prediction if needed \*/  
 if(SkippedCornersArray[ c ] == -1) {  
 positionDecodeWithPrediction( c )  
 }  
 }

* + - 1. Prediction of per vertex position attributes

Inputs to this process are:

* a variable c specifying the index of the corner for which vertex position will be predicted.

Output of this process is indirect:

* It modifies the array VertexMarkingArray, defined in clause I.9.2 and the array VertCoordValues defined in clause I.9.1.

Let the alias pO refer to the variable OppositeCornersArray.

Let the alias pV refer to the variable CornerToVertexArray.

Let the alias pG refer to the variable VertCoordValues.

Let the alias mV refer to the variable VertexMarkingArray.

Let the variable maxParallelograms, specifying the maximum number of parallelogram predictions, and the variable v, specifying the index of the vertex position associated with c, be initialized as follows:

maxParallelograms = 4  
 v = GetVertexIndex( CornerToVertexArray, c )

If mV[v] is strictly greater than 0, the vertex v is already predicted, then the process does nothing and returns. Otherwise, the following applies:

/\* we mark the vertex \*/  
 mV[ v ] = 1  
   
 /\* search for some parallelogram estimations around the vertex of the corner  
    the triangle fan might not be complete,  
    but we know that a vertex is allways manifold, so we have only one fan per vertex  
    also some opposite corners might not be deifned due to boundaries  
    so we use OV accessors and test results to filter negative values \*/

Let the 1D array predPos, of size 3, specifying the predicted position of the vertex associated with c, and the variable altC, specifying a corner index, and the variable nextC, specifying a corner index, be initialized as follows:

predPos[ 0 ] = 0  
 predPos[ 1 ] = 0  
 predPos[ 2 ] = 0  
 altC = c  
 nextC = NextCorner( pO[ NextCorner( altC ) ] )  
   
 /\* loop through corners attached to the current vertex \*/  
 /\* swing around the fan until we find a border \*/  
 while( nextC >= 0 && nextC != c)  
 {  
 altC = nextC  
 nextC = NextCorner( pO[ NextCorner( altC ) ] )  
 }

Let the variable isBoundary, specify if nextC is on a boundary, and the variable count, specify the number of valid parallelogram predictions found, and the variable startC, specify the index of the extreme corner of the fan, and be initialized as follows:

isBoundary = ( nextC != c )  
 startC = altC  
 count = 0

Let the variables prevV, oppoV and nextV, specifying the index of the vertex associated with previous, opposite, and next corners respectively, be set to 0.

The following applies:

/\* now we are position on the right most corner sharing v \*/  
 /\* we turn left an evaluate the possible predictions \*/  
 do  
 {  
 oppoV = GetVertexIndex( pV , pO[ altC ] )  
 prevV = GetVertexIndex( pV , PreviousCorner( altC ) )  
 nextV = GetVertexIndex( pV , NextCorner( altC ) )  
   
 if( ( oppoV > -1 && prevV > -1 && nextV > -1 ) &&  
 ( ( mV[ oppoV ] > 0 ) && ( mV[ prevV ] > 0 ) && ( mV[ nextV ] > 0 ) ) )  
 {  
 /\* parallelogram prediction estG = prevG + nextG – oppoGd \*/  
 /\* accumulate parallelogram predictions \*/  
 predPos[ 0 ] = predPos[ 0 ]  
 + pG[prevV][ 0 ] + pG[ nextV ][ 0 ] - pG[ oppoV ][ 0 ]  
 predPos[ 1 ] = predPos[ 1 ]  
 + pG[prevV][ 1 ] + pG[ nextV ][ 1 ] - pG[ oppoV ][ 1 ]  
 predPos[ 2 ] = predPos[ 2 ]  
 + pG[prevV][ 2 ] + pG[ nextV ][ 2 ] - pG[ oppoV ][ 2 ]  
 count++  
 }  
 /\* swing around the triangle fan \*/  
 altC = PreviousCorner( pO[ PreviousCorner( altC ) ] )  
 /\* stop on incomplete fan or full rotation or max predictions reached \*/  
 } while( altC >= 0 && altC != startC && count < maxParallelograms )  
   
 /\* 1. use parallelogram prediction when possible \*/  
 if( count > 0 ) {  
 pG[ 0 ] = mesh\_position\_fine\_residual[ deltaIndex ][ 0 ]  
 + Round( predPos[ 0 ] ¸ count )  
 pG[ 1 ] = mesh\_position\_fine\_residual[ deltaIndex ][ 1 ]  
 + Round( predPos[ 1 ] ¸ count )  
 pG[ 2 ] = mesh\_position\_fine\_residual[ deltaIndex ][ 2 ]  
 + Round( predPos[ 2 ] ¸ count )  
 deltaIndex = deltaIndex + 1  
 return  
 }  
   
 /\* 2. or fallback to delta with available values if any \*/  
 prevV = GetVertexIndex( v , PreviousCorner( c ) )  
 nextV = GetVertexIndex( v , NextCorner( c ) )  
   
 if( prevV > -1 && mV[ prevV ] > -1 ) {  
 pG[ v ][ 0 ] = mesh\_position\_coarse\_residual[ deltaCoarseIndex ][ 0 ]  
 + pG[ prevV ][ 0 ]  
 pG[ v ][ 1 ] = mesh\_position\_coarse\_residual[ deltaCoarseIndex ][ 1 ]  
 + pG[ prevV ][ 1 ]  
 pG[ v ][ 2 ] = mesh\_position\_coarse\_residual[ deltaCoarseIndex ][ 2 ]  
 + pG[ prevV ][ 2 ]  
 deltaCoarseIndex = deltaCoarseIndex + 1  
 return  
 }  
   
 if( nextV > -1 && mV[ nextV ] > -1 ) {  
 pG[ v ][ 0 ] = mesh\_position\_coarse\_residual[ deltaCoarseIndex][ 0 ]  
 + pG[nextV][ 0 ]  
 pG[ v ][ 1 ] = mesh\_position\_coarse\_residual[ deltaCoarseIndex][ 1 ]  
 + pG[nextV][ 1 ]  
 pG[ v ][ 2 ] = mesh\_position\_coarse\_residual[ deltaCoarseIndex][ 2 ]  
 + pG[nextV][ 2 ]  
 deltaCoarseIndex = deltaCoarseIndex + 1  
 return  
 }

Let the variable b, specifying the index of the previous corner on the boundary.   
Let the variable bv, specifying the index of the vertex associated with b.

/\* 3. or maybe we are on a boundary \*/  
 /\* then we may use delta from previous vertex on the boundary \*/  
 if( isBoundary ) {  
 b = PreviousCorner( startC )   
 bv = GetVertexIndex( pV, b )  
 if( mV[ bv ] > -1 ) {  
 pG[ v ][ 0 ] = mesh\_position\_coarse\_residual[ deltaCoarseIndex ][ 0 ]  
 + pG[ bv ][ 0 ]  
 pG[ v ][ 1 ] = mesh\_position\_coarse\_residual[ deltaCoarseIndex ][ 1 ]  
 + pG[ bv ][ 1 ]  
 pG[ v ][ 2 ] = mesh\_position\_coarse\_residual[ deltaCoarseIndex ][ 2 ]  
 + pG[ bv ][ 2 ]  
 deltaCoarseIndex = deltaCoarseIndex + 1  
 return  
 }  
 }  
   
 /\* 4. no more choices, we use an absolute value (i.e. a start) \*/  
 pG[ v ][ 0 ] = mesh\_position\_start[ startIndex ][ 0 ]  
 pG[ v ][ 1 ] = mesh\_position\_start[ startIndex ][ 1 ]  
 pG[ v ][ 2 ] = mesh\_position\_start[ startIndex ][ 2 ]  
 startIndex = startIndex + 1

* + 1. Decoding auxiliary per vertex attributes
       1. General

This process performs specific vertex traversals, depending on the values of mesh\_attribute\_separate\_index\_flag[ attrIndex ] and mesh\_vertex\_traversal\_method, to invoke the attribute prediction for each vertex of an auxiliary attribute specified by attrIndex.

Inputs to this process is:

* a variable attrIndex, specifying the index of the attribute on which to perform predictions.

This process has no direct outputs. The sub-processes described in sub-clauses I.9.7.2, I.9.7.3 and I.9.7.4 modify global variables and arrays.

Let the alias posV refer to the variable CornerToVertexArray.

Let the alias aO refer to the variable OppositeCornersArray.

Let the alias auxV refer to the variable AuxiliaryCornerToVertexArray[ attrIndex ].

Let the alias aAOC refer to the variable AuxiliaryOppositeCornersArray[ attrIndex ].

The following applies:

/\* reset the vertex marking table \*/  
 for( i = 0; i < AttrValueCount[ attrIndex ]; i++ ) {  
 VertexMarkingArray[ i ] = -1  
 }

Let the variable hasOwnIndices, specifying if the auxiliary attribute uses an auxiliary index table, be set to the value of mesh\_attribute\_separate\_index\_flag[ attrIndex ].

Let the variable c, specify the index of the corner for which vertex position will be predicted, and the variable skipIdx, specify the index by which current vertex associated with c shall be substituted or -1 otherwise.

The following applies:

* If hasOwnIndices is equal to 0 and mesh\_vertex\_traversal\_method is equal to MESH\_TRAVERSAL\_EB, the following applies:

for( i = 0; i < ReadCornersCnt; i++ ) {  
 c = ReadCorners[ i ]  
 skipIdx = ( SkippedCornersCnt != 0 ? SkippedCorners[ c ] : -1 )  
 if( skipIdx >= 0 ) {  
 AttrValues[ attrIndex ][ posV[ c ] ] = AttrValues[ attrIndex ][ skipIdx ]  
 }  
 else {

The prediction of per vertex attributes process described in subclause I.9.7.3 is invoked with attrIndex, c and posV as input parameters.

}  
 }

* If hasOwnIndices is equal to 0 and mesh\_vertex\_traversal\_method is equal to MESH\_TRAVERSAL\_DEGREE, the following applies:

for( i = 0; i < TraversalCornerCnt; i++ ) {  
 c = TraversalCornerArray[ i ]  
 skipIdx = ( SkippedCornersCnt != 0 ? SkippedCornersArray[ c ] : -1 )  
 if( skipIdx >= 0) {  
 AttrValues[ attrIndex ][ posV[ c ] ] = AttrValues[ attrIndex ][ skipIdx ]  
 }  
 else {

The prediction of per vertex attributes process described in subclause I.9.7.3 is invoked with attrIndex, c and posV as input parameters.

}  
 }

* If hasOwnIndices is equal to 1, let the variable currentDuplicate, specify the index of the current duplicated vertex, and the variable currentVertex, specify the index of the current traversed vertex, be initialized as follow:

currentDuplicate = 0  
 currentVertex = 0

* Let the alias iDuplicateSplitVertexIdx refer to mesh\_attribute\_deduplicate\_idx[ attrIndex ].
* Let the 1D array processedDupIdxArray, of size NumSplitAttribute[ attrIndex ], specifying the vertex index associated with a group of duplicated vertices identified by a split index from the array iDuplicateSplitVertexIdx, be initialized as follows:

for( i = 0; i < NumSplitAttribute[ attrIndex ]; i++ ) {  
 processedDupIdxArray[ i ] = -1  
 }

* If hasOwnIndices is equal to 1 and mesh\_vertex\_traversal\_method is equal to MESH\_TRAVERSAL\_EB, let the 1D array corners, of size 3, specify a triplet of corner indices, the following applies:
  + Attribute values are derived as follows:

for( i = 0; i < ProcessedCornersCnt; i++ ) {  
 corners[ 0 ] = ProcessedCorners[ i ]  
 corners[ 1 ] = NextCorner(ProcessedCorners[ i ] )  
 corners[ 2 ] = PreviousCorner(ProcessedCorners[ i ] )  
 for(int j = 0; j < 3; ++j ) {  
 c = corners[ j ]  
 if( VertexMarkingArray[ auxV[ c ] ] > 0 )  
 continue

Then, the attribute values are derived by invoking subclause I.9.7.2 with attrIndex, c and auxV as input parameters.

}  
 }

* + Finally, the variable aAOC is updated by invoking subclause I.9.7.4 with aO, attrIndex, ProcessedCornersCnt and ProcessedCorners as input parameters.
* If hasOwnIndices is equal to 1 and mesh\_vertex\_traversal\_method is equal to MESH\_TRAVERSAL\_DEGREE, the following applies:
  + Let the variable auxTraversalCornersCnt specify the size of the table auxTraversalCornersArray and the 1D array auxTraversalCornersArray of size auxTraversalCornersCnt, specify an ordered set of corners for the traversal of the auxiliary UV connectivity.
  + Then, the process described in subclause I.9.12 is invoked with the variables AttrValueCount[ attrIndex ], mesh\_triangle\_count, and AuxiliaryStartCornersCntArray[ attrIndex ] and arrays AuxiliaryStartCornersArray[ attrIndex ], and AuxiliaryOppositeCornersArray[ attrIndex ], and AuxiliaryCornerToVertexArray[ attrIndex ], as inputs and the variable TraversalCornersCnt and the 1D array auxTraversalCornersArray of size auxTraversalCornersCnt as outputs.
  + Then, the following applies:
    - Attribute values are derived as follows:

/\* perform the predictions using the order \*/  
 for( i = 0; i < auxTraversalCornersCnt; i++ ) {  
 c = auxTraversalCornerArray[ i ]

Then, the attributes values are derived by invoking subclause I.9.7.2 with attrIndex, c and auxV as input parameters.

}

* + - Finally, the variable aAOC is updated by invoking subclause I.9.7.4 with aO, attrIndex, auxTraversalCornersCnt and auxTraversalCornersArray as input parameters.
      1. Decoding and deduplication of per vertex attributes

Inputs to this process are:

* a variable attrIndex, specifying the index of the attribute on which to perform decoding and deduplication.
* a variable c specifying the index of the corner for which vertex position will be predicted.
* a 1D array auxV, of size CornerCnt, specifying the connectivity to be used to dereference attribute values with index attrIndex.

This process has no direct output.

Let the variable deduplicate be set to 1 if mesh\_deduplicate\_method[ attrIndex ] is not equal to MESH\_DEDUP\_NONE, or to 0 if mesh\_deduplicate\_method[ attrIndex ] is equal to MESH \_DEDUP\_NONE.

Let the variable predictAttribute, specify if corner attributes shall be predicted (value equal to 1), or if prediction shall be skipped and use the existing predicted value from its duplicate vertex index stored in AttrValues[ attrIndex ], be initialized as follow:

predictAttribute = 1

The following applies:

/\* perform deduplication if needed \*/  
 if ( deduplicate && NumAttributeIsDuplicateFlags[ attrIndex ] != 0   
 && mesh\_attribute\_is\_duplicate\_flag[ attrIndex ][ currentVertex ] )  
 {  
 splitIdx = iDuplicateSplitVertexIdx[ currentDuplicate++ ]  
 dupVtx = processedDupIdxArray[ splitIdx ]  
 if( dupVtx > =0 )  
 {  
 DuplicatesArray[ duplicatesCnt ][ 0 ] = c  
 DuplicatesArray[ duplicatesCnt ][ 1 ] = dupVtx  
   
 AttrValues[ attrIndex ][ auxV [ c ] ] =  
  AttrValues[ attrIndex ][ dupVtx ]  
 predictAttribute = 0   
 } else {  
 processedDupIdx[ splitIdx ] = auxV[ c ]  
 }  
 }  
 /\* perform prediction if needed \*/  
 if (predictAttribute == 1) {

The prediction of per vertex attributes process described in subclause I.9.7.3 is invoked with attrIndex, c and auxV as input parameters.

}

* + - 1. Prediction of per vertex attributes

The process described by this subclause switches on the type of attribute to invoke the proper predictor (UV coordinates, Normal vectors, Colors, etc.).

Inputs to this process are:

* a variable attrIndex, specifying the index of the attribute on which to perform predictions.
* a variable c specifying the index of the corner for which vertex position will be predicted.
* a 1D array auxV, of size CornerCnt, specifying the connectivity to be used to dereference attribute values for the attribute with index attrIndex.

This process has no direct output.

The following applies:

* If mesh\_attribute\_type[ attrIndex ] is equal to MESH\_ATTR\_TEXCOORD, the process described in subclause I.9.7.5 is invoked with the variables attrIndex, c and the array auxV as inputs.
* Else if mesh\_attribute\_type[ attrIndex ] is equal to MESH\_ATTR\_NORMAL, the process described in subclause I.9.7.6 is invoked with the variables attrIndex, c and the array auxV as inputs.
* Else if mesh\_attribute\_type[ attrIndex ] is equal to MESH\_ATTR\_COLOR, the process described in subclause I.9.7.6.5 is invoked with the variables attrIndex, c and the array auxV as inputs.
* Else if mesh\_attribute\_type[ attrIndex ] is equal to MESH\_ATTR\_GENERIC, the process described in subclause I.9.7.8 is invoked with the variables attrIndex, c and the array auxV as inputs.
* Any other value for mesh\_attribute\_type[ attrIndex ] is not supported.
  + - 1. Updating opposite corners of per vertex attributes

This process updates the array of opposite corners of per vertex attributes (aAOC), used to handle per fan deduplication in subclause I.9.9.

Inputs to this process are:

* a 1D array aO refer to the variable OppositeCornersArray.
* a variable attrIndex, specifying the index of the attribute on which to perform updating opposite corners.
* a variable traCorCnt, specifying the size of the table traCornersArray. Its value is equal to ProcessedCornersCnt if mesh\_vertex\_traversal\_method is equal to MESH\_TRAVERSAL\_EB, or to auxTraversalCornersCnt if mesh\_vertex\_traversal\_method is equal to MESH\_TRAVERSAL\_DEGREE.
* a 1D array traCornersArray, of size traCorCnt, specifying the corners in their order of visit for the traversal of the auxiliary attributes connectivity. Its values are equal to ProcessedCorners if mesh\_vertex\_traversal\_method is equal to MESH\_TRAVERSAL\_EB, or to auxTraversalCornersArray if mesh\_vertex\_traversal\_method is equal to MESH\_TRAVERSAL\_DEGREE.

This process has no direct output.

If deduplicate is equal to 1, and NumAttributeIsDuplicateFlags [ attrIndex ] is not equal to 0. The following applies:

* a 1D array markedCornersArray, of size CornerCnt is initialized to 0,
* Let the 1D array corners, of size 3, specify a triplet of corner indices, the following applies:

for( i = 0; i < traCorCnt; i++) {  
 corners[ 0 ] = traCornersArray[ i ]  
 corners[ 1 ] = NextCorner( traCornersArray[ i ] )  
 corners[ 2 ] = PreviousCorner( traCornersArray[ i ] )  
 for( j = 0; j < 3; j++ ) {  
 c = corners[ j ]  
 if( markedCornersArray[ c ] )  
 continue  
 markedCornersArray[c] = 1  
 oppc = aO[ c ]  
 if( oppc >= 0 ) {  
 if( aAOC[ c ] != -2 ) {  
 aAOC[ c ] = oppc  
 aAOC[ oppc ] = cur\_corner  
 }  
 markedCornersArray[ oppc ] = 1  
 }  
 }  
 }

* + - 1. Prediction of per vertex UV coordinates attributes
         1. General

Inputs to this process are:

* a variable attrIndex, specifying the index of the attribute on which to perform predictions.
* a variable c specifying the index of the corner for which vertex position will be predicted.
* a 1D array auxV, of size CornerCnt, specifying the connectivity to be used to dereference UV coordinates.

Output of this process is indirect:

* It modifies the arrays VertexMarkingArray, AuxiliaryStartIndex[ attrIndex  AuxiliaryDeltaIndex[ attrIndex ] and AuxiliaryDeltaCoarseIndexArray[ attrIndex ] defined in clause I.9.2 and the array AttrValues[ attrIndex ] defined in clause I.9.1.

Let the variable hasOwnIndices, specifying if the auxiliary attribute uses an auxiliary index table, be set to the value of mesh\_attribute\_separate\_index\_flag[ attrIndex ]

Let the alias mV refer to the variable VertexMarkingArray.

Let the alias pO refer to the variable OppositeCornersArray.

Let the alias pV refer to the variable CornerToVertexArray.

Let the alias auxUV refer to the variable AttrValues[attrIndex].

Let the alias auxO refer to the variable AuxiliaryOppositeCornersArray[ attrIndex ].

Let the alias auxStartIndex refer to the variable AuxiliaryStartIndex[ attrIndex ].

Let the alias auxDeltaIndex refer to the variable AuxiliaryDeltaIndexArray[ attrIndex ].

Let the alias auxDeltaCoarseIndex refer to the variable AuxiliaryDeltaCoarseIndexArray[ attrIndex ].

Let uvPrediction( attrIndex, altC, TC, predUV ) denote the invocation of the process defined in subclause I.9.7.5.2 when mesh\_attribute\_prediction\_method[ attrIndex ] is equal to MESH\_TEXCOORD\_MSTRETCH, with the parameters attrIndex, altC, TC and predUV as inputs, and the parameter predUV as outputs.

Let the variable maxUVPreds, specifying the maximum number of UV coordinates predictions be initialized as follows:

maxUVPreds = 8

Let the variable v, specifying the index of the texture coordinate of the vertex associated with c, be initialized as follows:

v = auxV[ c ]

If mV[v] is strictly greater than 0, the vertex v is already predicted, then the process does nothing and returns. Otherwise, the following applies:

/\* we mark the vertex \*/  
 mV[ v ] = 1  
 /\* search for valid min stretch estimations around the vertex of the corner \*/  
 /\* the triangle fan might not be complete, \*/  
 /\* but we know that each vertex is manifold, so we have only one - potentally \*/  
 /\* partial - fan per vertex. Some opposite corners might not be defined due to \*/  
 /\* boundaries so we use corner table accessors and test to filter negative values \*/

Let the 1D array predUV, of size 2, specifying the cumulated UV coordinate prediction of the vertex associated with c, and the variable altC, and nextC, specifying corner indices, and the variable onSeam, specifying if altC is on a seam, be initialized as follows:

predUV[ 0 ] = 0  
 predUV[ 1 ] = 0  
 altC = c  
 onSeam = ( hasOwnIndices ? ( auxO[NextCorner( altC ) ] == -2 ) : 0 )   
 nextC  = NextCorner( pO[ NextCorner( altC ) ] )

The following applies:

/\* loop through corners attached to the current vertex \*/  
 /\* by swinging around the fan until we find a border or a seam \*/  
 while( nextC >= 0 && nextC != c && !onSeam )  
 {  
 altC = nextC  
 onSeam = ( hasOwnIndices ? ( auxO[ NextCorner( altC ) ] == -2 ) : 0 )  
 nextC = NextCorner( pO[ NextCorner( altC ) ] )  
 }

Let the variable isBoundary, specifying if nextC is on a boundary but not on an attribute seam, and the variable count, specifying the number of valid stretch predictions found, and the variable startC, specifying the index of the extreme corner of the fan, be initialized as follows:

isBoundary = ( !onSeam && nextC != c )  
 count = 0  
 startC = altC

Let the variables prevV, oppoV and nextV, specifying the index of the vertex associated with previous, opposite, and next corners respectively, be set to 0.

The following applies:

/\* now we are position on the right most corner sharing v \*/  
 /\* we turn left and evaluate the possible predictions \*/  
 do {  
 altV  = auxV[ altC ]  
 prevV = auxV[ PreviousCorner( altC ) ]  
 nextV = auxV[ NextCorner( altC ) ]  
 if( ( altV > -1 && prevV > -1 && nextV > -1 ) &&  
 ( ( mV[ altV ] > 0 ) && ( mV[ prevV ] > 0 ) && ( mV[ nextV ] > 0) ) )  
 {  
 uvPrediction( attrIndex, altC, auxV, predUV )  
 ++count  
 }  
 onSeam = ( hasOwnIndices ? (auxO[ PreviousCorner( altC ) ] == -2 ) : 0 )  
 // swing around the triangle fan  
 altC = PreviousCorner( pO[ PreviousCorner(altC) ] )  
   
 // stop on incomplete fan or full rotation or max predictions reached  
 } while( altC >= 0 && altC != startC && !onSeam && count < maxUVPreds )  
 /\* 1. use min stretch prediction when possible \*/  
 if( count > 0 ) {  
 auxUV[ v ][ 0 ] = mesh\_attribute\_fine\_residual[ attrIndex ][ auxDeltaIndex ][ 0 ]  
 + predUV[ 0 ] ¸ count  
 auxUV[ v ][ 0 ] = mesh\_attribute\_fine\_residual[ attrIndex ][ auxDeltaIndex ][ 1 ]  
 + predUV[ 1 ] ¸ count  
 auxDeltaIndex = auxDeltaIndex + 1  
 return  
 }  
   
 /\* 2. or fallback to delta with available values if any \*/  
 prevV = GetVertexIndex( auxV, PreviousCorner( c ) )  
 nextV = GetVertexIndex( auxV, NextCorner( c ) )  
   
 if( prevV > -1 && mV[ prevV ] > -1 ) {  
 auxUV[ v ][ 0 ] =  
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 0 ]  
 + auxUV[ prevV ][ 0 ]  
 auxUV[ v ][ 1 ] =  
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 1 ]  
 + auxUV[ prevV ][ 1 ]  
 auxDeltaCoarseIndex = auxDeltaCoarseIndex + 1  
 return  
 }  
   
 if( nextV > -1 && mV[ nextV ] > -1 ) {  
 auxUV[ v ][ 0 ] =  
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex][ 0 ]  
 + auxUV[ nextV ][ 0 ]  
 auxUV[ v ][ 1 ] =  
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex][ 1 ]  
 + auxUV[ nextV ][ 1 ]  
 auxDeltaCoarseIndex = auxDeltaCoarseIndex + 1  
 return  
 }

Let the variable b, specifying the index of the previous corner on the boundary.

Let the variable bv, specifying the index of the vertex associated with b.

/\* 3. or maybe we are on a boundary \*/  
 /\* then we may use delta from previous vertex on the boundary \*/  
 if( isBoundary ) {  
 b = PreviousCorner( startC )  
 bV = GetVertexIndex( auxV, b )  
 if( mV[ bv ] > -1 ) {  
 auxUV[ v ][ 0 ] =  
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex][ 0 ]  
 + auxUV[ bv ][ 0 ]  
 auxUV[ v ][ 1 ] =  
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex][ 1 ]  
 + auxUV[ bv ][ 1 ]  
 auxDeltaCoarseIndex = auxDeltaCoarseIndex + 1  
 return  
 }  
 }  
   
 /\* 4. no more choices, we use an absolute value (i.e. a start) \*/  
 auxUV[ v ][ 0 ] = mesh\_attribute\_start[ auxStartIndex ][ 0 ]  
 auxUV[ v ][ 1 ] = mesh\_attribute\_start[ auxStartIndex ][ 1 ]  
 auxStartIndex = auxStartIndex + 1

* + - * 1. Minimal stretch prediction of UV coordinates

Inputs to this process are:

* a variable attrIndex, specifying the index of the attribute on which to perform predictions.
* a variable c specifying the index of the corner for which vertex position will be predicted.
* a 1D array auxV, of size CornerCnt, specifying the connectivity to be used to dereference UV coordinates.

Output of this process is:

* a 1D array predUV, of size 2, specifying the predicted UV coordinate.

Let the variable hasOwnIndices, specifying if the auxiliary attribute uses an auxiliary index table, be set to the value of mesh\_attribute\_separate\_index\_flag[ attrIndex ]

Let the alias mV refer to the variable VertexMarkingArray.

Let the alias pO refer to the variable OppositeCornersArray.

Let the alias pV refer to the variable CornerToVertexArray.

Let the alias pG refer to the variable VertCoordValues.

Let the alias auxUV refer to the variable AttrValues[attrIndex].

Let the alias auxO refer to the variable AuxiliaryOppositeCornersArray[ attrIndex ]

Let the variable v, specifying the index of the texture coordinate of the vertex associated with c, be initialized as follows:

v = auxV[ c ]

Let the 1D arrays uvPrev and uvNext, of size 2, specifying the UV coordinate of previous and next corner, respectively, be initialized as follows:

uvPrev[ 0 ] = auxUV[ auxV[ PreviousCorner( c ) ] ][ 0 ]  
 uvPrev[ 1 ] = auxUV[ auxV[ PreviousCorner( c ) ] ][ 1 ]  
 uvNext[ 0 ] = auxUV[ auxV[ NextCorner(c) ] ][ 0 ]  
 uvNext[ 1 ] = auxUV[ auxV[ NextCorner(c) ] ][ 1 ]

Let the 1D arrays gPrev and gNext and gCurr, of size 3, specifying the 3D position of previous, next, and current corner, respectively, be initialized as follows:

gPrev[ 0 ] = pG[ pV[ PreviousCorner( c ) ] ][ 0 ]  
 gPrev[ 1 ] = pG[ pV[ PreviousCorner( c ) ] ][ 1 ]  
 gPrev[ 2 ] = pG[ pV[ PreviousCorner( c ) ] ][ 2 ]  
   
 gNext[ 0 ] = pG[ pV[ NextCorner( c ) ] ][ 0 ]  
 gNext[ 1 ] = pG[ pV[ NextCorner( c ) ] ][ 1 ]  
 gNext[ 2 ] = pG[ pV[ NextCorner( c ) ] ][ 2 ]  
   
 gCurr[ 0 ] = pG[ pV[ c ] ][ 0 ]  
 gCurr[ 1 ] = pG[ pV[ c ] ][ 1 ]  
 gCurr[ 2 ] = pG[ pV[ c ] ][ 2 ]

Let the 1D arrays gNgP, gNgC of size 3 and uvNuvP of size 2, specifying some difference vectors as follows:

gNgP[ 0 ] = gPrev[ 0 ] – gNext[ 0 ]  
 gNgP[ 1 ] = gPrev[ 1 ] – gNext[ 1 ]  
 gNgP[ 2 ] = gPrev[ 2 ] – gNext[ 2 ]  
   
 gNgC[ 0 ] = gCurr[ 0 ] – gNext[ 0 ]  
 gNgC[ 1 ] = gCurr[ 1 ] – gNext[ 1 ]  
 gNgC[ 2 ] = gCurr[ 2 ] – gNext[ 2 ]  
   
 uvNuvP[ 0 ] = uvPrev[ 0 ] - uvNext[ 0 ]  
 uvNuvP[ 1 ] = uvPrev[ 1 ] - uvNext[ 1 ]  
 uvNuvP[ 2 ] = uvPrev[ 2 ] - uvNext[ 2 ]

Let the variable d2gNgP, specifying the dot product result, be defined as follows:

d2gNgP = gNgP[ 0 ] \* gNgP[ 0 ] + gNgP[ 1 ] \* gNgP[ 1 ] + gNgP[ 2 ] \* gNgP[ 2 ]

If d2\_gNgP is less than or equal to 0, the following applies:

predUV[ 0 ] = Round( (auxUV[ auxV[ NextCorner( c ) ] ][ 0 ]  
 + auxUV[ auxV[ PreviousCorner( c ) ] ][ 0 ] ) ¸ 2)  
 predUV[ 1 ] = Round( (auxUV[ auxV[ NextCorner( c ) ] ][ 1 ]  
 + auxUV[ auxV[ PreviousCorner( c ) ] ][ 1 ] ) ¸ 2)

Else the following applies:

Let the variable gNgPdotgNgC, specifying the dot product results, be defined as follows:

gNgPdotgNgC = gNgP[ 0 ] \* gNgC[ 0 ]   
 + gNgP[ 1 ] \* gNgC[ 1 ]  
 + gNgP[ 2 ] \* gNgC[ 2 ]

Let the 1D arrays uvProj and gProj, of size 3, specifying the projections as follows:

uvProj[ 0 ] = uvNext[ 0 ] + uvNuvP[ 0 ] \* ( gNgPdotgNgC ¸ d2\_gNgP )  
 uvProj[ 1 ] = uvNext[ 1 ] + uvNuvP[ 1 ] \* ( gNgPdotgNgC ¸ d2\_gNgP )  
   
 gProj[ 0 ] = gNext[ 0 ] + gNgP[ 0 ] \* ( gNgPdotgNgC ¸ d2\_gNgP )  
 gProj[ 1 ] = gNext[ 1 ] + gNgP[ 1 ] \* ( gNgPdotgNgC ¸ d2\_gNgP )  
 gProj[ 2 ] = gNext[ 2 ] + gNgP[ 2 ] \* ( gNgPdotgNgC ¸ d2\_gNgP )

Let the 1D arrays gDiff, of size 3, specifying geometric vector between gCurr and gProj be defined as follows:

gDiff[ 0 ] = gCur[ 0 ] – gProj[ 0 ]  
 gDiff[ 1 ] = gCur[ 1 ] – gProj[ 1 ]  
 gDiff[ 2 ] = gCur[ 2 ] – gProj[ 2 ]

Let the variable d2gProjgCurr, specifying the dot product of dDiff with itself, be defined as follows:

d2gProjgCurr = gDiff[ 0 ] \* gDiff[ 0 ]  
 + gDiff[ 1 ] \* gDiff[ 1 ]  
 + gDiff[ 2 ] \* gDiff[ 2 ]

Let the 1D array uvProjuvCurr, of size 2, specifying the projection of the current UV coordinates, be defined as follows:

uvProjUvCurr[ 0 ] = uvNuvP.y \* Sqrt( d2gProjgCurr ¸ d2gNgP )  
 uvProjUvCurr[ 1 ] = -uvNuvP.x \* Sqrt( d2gProjgCurr ¸ d2gNgP )

Let the 1D arrays predUV0 and predUV1, of size 2, specifying potential UV predictions, be defined as follows:

predUV0[ 0 ] = uvProj[ 0 ] + uvProjUvCurr[ 0 ]  
 predUV0[ 1 ] = uvProj[ 1 ] + uvProjUvCurr[ 1 ]  
   
 predUV1[ 0 ] = uvProj[ 0 ] - uvProjUvCurr[ 0 ]  
 predUV1[ 1 ] = uvProj[ 1 ] - uvProjUvCurr[ 1 ]

Let the variables, useOpp, specifying whether opposite corner of c can be used for the prediction or not, and onSeam specifying whether the corner c is located on a seam or not, and checkOpposite specifying whether opposite corner of c shall be checked for validity to be used for the prediction or not, be initialized as follows:

/\* we cannot use the opposite if beyond a seam \*/  
 onSeam = ( hasOwnIndices ? ( auxO[ c ] == -2) : 0)  
 /\* we cannot use the opposite if beyond a seam \*/  
 checkOpposite = ( !onSeam && pO[ c ] >= 0 && auxV[ pO[ c ] ] >= 0  
 && mV[ auxV[ pO[ c ] ] ] > 0)  
 useOpp = 0

If checkOpposite is TRUE, the following applies:

Let the variables, uvOpp, specifying the UV coordinates of the opposite vertex, nP specifying the UV coordinates vector from the next corner vertex to the previous corner vertex, and nO specifying the UV coordinates vector from the next corner vertex to the opposite corner vertex, be initialized as follows:

uvOpp[ 0 ] = auxUV[ auxV[ pO[ c ] ] ][ 0 ]  
 uvOpp[ 1 ] = auxUV[ auxV[ pO[ c ] ] ][ 1 ]  
 /\* this test should be using 64b integers - this is ( vecNP ^ vec NO ) \*/  
 nP[ 0 ] = uvPrev[ 0 ] – uvNext[ 0 ]  
 nP[ 1 ] = uvPrev[ 1 ] – uvNext[ 1 ]  
 nO[ 0 ] = uvOpp[ 0 ]  – uvNext[ 0 ]  
 nO[ 1 ] = uvOpp[ 1 ]  – uvNext[ 1 ]

Let the variable nPnO, defined in R, specifying a cross product, be initialized as follows:

/\* check that O is not aligned with N and P \*/  
 /\* evaluate cross product \*/  
 nPnO = nP[ 0 ] \* nO[ 1 ] – nP[ 1 ] \* nO[ 0 ]  
 /\* in current implementation nPnO is integer \*/  
 /\* and check can be to strict zero \*/  
 useOpp = ( nPnO != 0 )

Then, if useOpp is TRUE, the following applies:

Let the 1D arrays of size 2, uvOpp, specifying the UV coordinates of the opposite corner of c, and oUv0 specifying a residual between uvOpp and predUV0, and oUv1 specifying a residual between uvOpp and predUV1, be initialized as follows:

uvOpp[ 0 ] = auxUV[ auxV[ pO[ c ] ] ][ 0 ]  
 uvOpp[ 1 ] = auxUV[ auxV[ pO[ c ] ] ][ 1 ]  
 oUv0[ 0 ] = uvOpp[ 0 ] - predUV0[ 0 ]  
 oUv0[ 1 ] = uvOpp[ 1 ] - predUV0[ 1 ]  
 oUv1[ 0 ] = uvOpp[ 0 ] – predUV1[ 0 ]  
 oUv1[ 1 ] = uvOpp[ 1 ] – predUV1[ 1 ]

Let the variables dotUv0, specifying the dot product of oUV0 with itself, and dotUv1, specifying the dot product of oUV1 with itself, be initialized as follows:

dotUv0 = oUv0[ 0 ] \* oUv0[ 0 ] + oUv0[ 1 ] \* oUv0[ 1 ]  
 dotUv1 = oUv1[ 0 ] \* oUv1[ 0 ] + oUv1[ 1 ] \* oUv1[ 1 ]

If dotUv0 < dotUv1, the following applies:

predUV[ 0 ] = predUV[ 0 ] + predUV1[ 0 ]  
 predUV[ 1 ] = predUV[ 1 ] + predUV1[ 1 ]

Otherwise, the following applies:

predUV[ 0 ] = predUV[ 0 ] + predUV0[ 0 ]  
 predUV[ 1 ] = predUV[ 1 ] + predUV0[ 1 ]

Otherwise, useOpp is FALSE, and the following applies:

Let the variable ori, be initialized as follows:

ori = mesh\_texcoord\_stretch\_orientation[ attrIndex ]  
 [ AuxiliaryUvOrientationIndexArray[ attrIndex ]++ ]

Let the 1D array predUVd, of dimension 2, with real values, be initialized as follows:

predUVd[ 0 ] = ori ? predUV0[ 0 ] : predUV1[ 0 ]  
 predUVd[ 1 ] = ori ? predUV0[ 1 ] : predUV1[ 1 ]

The following applies:

predUV[ 0 ] = predUV[ 0 ] + Round( predUVd[ 0 ] )  
 predUV[ 1 ] = predUV[ 1 ] + Round( predUVd[ 1 ] )

* + - 1. Prediction of per vertex normal vector attributes
         1. General

When mesh\_attribute\_type equals to MESH\_ATTR\_NORMAL, the parameter mesh\_attribute\_prediction\_method[ index ] specifies which normal prediction scheme to use which is defined in Table I-8. When it is MESH\_NORMAL\_DELTA, the delta prediction scheme is employed. When it is MESH\_NORMAL\_MPARA, the multiple parallelogram prediction scheme for normals is employed. When it is MESH\_NORMAL\_CROSS, the cross-product prediction scheme is employed.

Let normalDecodeWithPredictionDelta( c ), normalDecodeWithPredictionMPARA( c ), and normalDecodeWithPredictionCross( c ) refer to the invocation of the process described in subclause I.9.7.6.2, I.9.7.6.3, I.9.7.6.4 with variables c as input. These functions eventually modifies the arrays VertexMarkingArray, AuxiliaryStartIndex[ attrIndex ], AuxiliaryDeltaIndexArray[ attrIndex ], and AuxiliaryDeltaCoarseIndexArray[ attrIndex ] defined in clause I.9.2 and the array AttrValues[ attrIndex ] defined in clause I.9.1.

Let decodeOctahedral( attrIndex, prediction, residual, reconstructed ) denote the invocation of the process defined in subclause I.9.7.6.5.

* + - * 1. Delta prediction of normals

Inputs to this process are:

* a variable attrIndex, specifying the index of the attribute on which to perform predictions.
* a variable c specifying the index of the corner for which vertex position will be predicted.

Output of this process is indirect:

* It modifies the array VertexMarkingArray, AuxiliaryStartIndex[ attrIndex ] and AuxiliaryDeltaIndex[ attrIndex ] defined in clause I.9.2 and the array AttrValues[ attrIndex ] defined in clause I.9.1.

Let the alias mV refer to the variable VertexMarkingArray.

Let the alias pO refer to the variable OppositeCornersArray.

Let the alias pV refer to the variable CornerToVertexArray.

Let the alias auxNorm refer to the variable AttrValues[attrIndex].

Let the alias auxStartIndex refer to the variable AuxiliaryStartIndex[ attrIndex ]

Let the alias auxDeltaIndex refer to the variable AuxiliaryDeltaIndexArray[ attrIndex ]

Let the alias auxDeltaCoarseIndex refer to the variable AuxiliaryDeltaCoarseIndexArray[ attrIndex ]

Let the variable v, specifying the index of the vertex associated with c, be initialized as follows:

v = GetVertexIndex( CornerToVertexArray, c )

If mV[v] is strictly greater than 0, the vertex v is already predicted, then the process does nothing and returns. Otherwise, the following applies:

// we mark the vertex  
 mV[ v ] = 1

Let the variable altC, specifying a corner index, and the variable nextC, specifying a corner index, be initialized as follows:

altC = c  
 nextC = NextCorner( pO[ NextCorner( altC ) ] )  
   
 // loop through corners attached to the current vertex  
 // swing around the fan until we find a border  
 while ( nextC >= 0 && nextC != c)  
 {  
 altC = nextC  
 nextC = NextCorner( pO[ NextCorner( altC ) ] )  
 }

Let the variable isBoundary, specify if nextC is on a boundary be initialized as follows:

isBoundary = ( nextC != c )

Let the variables prevV, oppoV and nextV, specifying the index of the vertex associated with previous, opposite, and next corners respectively, be set to 0.

The following applies:

// 1. Use delta with available values if any  
 prevV = GetVertexIndex( v , PreviousCorner( c ) )  
 nextV = GetVertexIndex( v , NextCorner( c ) )  
   
 if( prevV > -1 && mV[ prevV ] > -1 ) {  
 if( mesh\_normal\_octahedral\_flag[ attrIndex ] ) {  
 prediction = auxNorm[ prevV ]  
 residual = mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ]  
 decodeOctahedral( attrIndex, prediction, residual, auxNorm[ v ] )  
 } else {  
 auxNorm[ v ][ 0 ] =  
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 0 ]  
 + auxNorm[ prevV ][ 0 ]  
 auxNorm[ v ][ 1 ] =  
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 1 ]  
 + auxNorm[ prevV ][ 1 ]  
 auxNorm[ v ][ 2 ] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 2 ]  
 + auxNorm[ prevV ][ 2 ]  
 }  
 auxDeltaCoarseIndex = auxDeltaCoarseIndex + 1  
 return  
 }  
  
 if( nextV > -1 && MV[ nextV ] > -1 ) {  
 if ( mesh\_normal\_octahedral\_flag[ attrIndex ] ) {  
 prediction = auxNorm[ nextV ]  
 residual = mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ]  
 decodeOctahedral( attrIndex, prediction, residual, auxNorm[ v ] )  
 } else {  
 auxNorm[ v ][ 0 ] =  
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 0 ]  
 + auxNorm[ nextV ][ 0 ]  
 auxNorm[ v ][ 1 ] =  
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 1 ]  
 + auxNorm[ nextV ][ 1 ]  
 auxNorm[ v ][ 2 ] =  
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 2 ]  
 + auxNorm[ nextV ][ 2 ]  
 }  
 auxDeltaCoarseIndex = auxDeltaCoarseIndex + 1  
 return  
 }

Let the variable b, specifying the index of the previous corner on the boundary.   
Let the variable bV, specifying the index of the vertex associated with b.

// 2. If we are on a boundary  
 // then we may use delta from previous vertex on the boundary  
 if( isBoundary ) {  
 b = PreviousCorner( startC )  
 bV = GetVertexIndex( pV, b )  
 if ( mV[ bV ] > -1 ) {  
 if ( mesh\_normal\_octahedral\_flag[ attrIndex ] ) {  
 prediction = auxNorm[ bV ]  
 residual =  
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ]  
 decodeOctahedral( attrIndex, prediction, residual, auxNorm[ v ] )  
 } else {  
 auxNorm[ v ][ 0 ] =  
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 0 ]  
 + auxNorm[ bV ][ 0 ]  
 auxNorm[ v ][ 1 ] =  
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 0 ]  
 + auxNorm[ bV ][ 1 ]  
 auxNorm[v][2] =  
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 2 ]  
 + auxNorm[ bV ][ 2 ]  
 }  
 auxDeltaCoarseIndex = auxDeltaCoarseIndex + 1  
 return  
 }  
 }  
   
 // 3. No more choices, we use an absolute value (i.e. a start)  
 auxNorm[ v ][ 0 ] = mesh\_attribute\_start[ attrIndex ][ auxStartIndex ][ 0 ]  
 auxNorm[ v ][ 1 ] = mesh\_attribute\_start[ attrIndex ][ auxStartIndex ][ 1 ]  
 auxNorm[ v ][ 2 ] = mesh\_attribute\_start[ attrIndex ][ auxStartIndex ][ 2 ]  
 auxStartIndex = auxStartIndex + 1

* + - * 1. Multiple parallelograms prediction of normals

Inputs to this process are:

* a variable attrIndex, specifying the index of the attribute on which to perform predictions.
* a variable c specifying the index of the corner for which vertex position will be predicted.

Output of this process is indirect:

* It modifies the array VertexMarkingArray, AuxiliaryStartIndex[ attrIndex ] and AuxiliaryDeltaIndex[ attrIndex ] defined in clause I.9.2 and the array AttrValues[ attrIndex ] defined in clause I.9.1.

Let the alias mV refer to the variable VertexMarkingArray.

Let the alias pO refer to the variable OppositeCornersArray.

Let the alias pV refer to the variable CornerToVertexArray.

Let the alias auxNorm refer to the variable AttrValues[attrIndex].

Let the alias auxStartIndex refer to the variable AuxiliaryStartIndex[ attrIndex ]

Let the alias auxDeltaIndex refer to the variable AuxiliaryDeltaIndexArray[ attrIndex ]

Let the alias auxDeltaCoarseIndex refer to the variable AuxiliaryDeltaCoarseIndexArray[ attrIndex ]

Let the variable maxParallelograms, specifying the maximum number of parallelogram predictions, and the variable v, specifying the index of the vertex position associated with c, be initialized as follows:

maxParallelograms = 4  
 v = GetVertexIndex( CornerToVertexArray, c )

If mV[v] is strictly greater than 0, the vertex v is already predicted, then the process does nothing and returns. Otherwise, the following applies:

// we mark the vertex  
 mV[ v ] = 1

Let the 1D array predNorm, of size 3, specifying the predicted normal of the vertex associated with c, and the variable altC, specifying a corner index, and the variable nextC, specifying a corner index, be initialized as follows:

predNorm[ 0 ] = 0  
 predNorm[ 1 ] = 0  
 predNorm[ 2 ] = 0  
 altC = c  
 nextC = NextCorner( pO[ NextCorner( altC ) ] )  
   
 // loop through corners attached to the current vertex  
 // swing around the fan until we find a border  
 while ( nextC >= 0 && nextC != c)  
 {  
 altC = nextC  
 nextC = NextCorner( pO[ NextCorner( altC ) ] )  
 }

Let the variable isBoundary, specify if nextC is on a boundary, and the variable count, specify the number of valid parallelogram predictions found, and the variable startC, specify the index of the extreme corner of the fan, and be initialized as follows:

isBoundary = ( nextC != c )  
 startC = altC  
 count = 0

Let the variables prevV, oppoV and nextV, specifying the index of the vertex associated with previous, opposite, and next corners respectively, be set to 0.

Let the function *normalize* be the normalization of a vector into a unit vector. It employs the L2 norm for normalization, resulting in an output within the range of -1.0 to 1.0.

Let the function *normal\_quantize(vector, qn)* convert a float unit *vector* centered at 0 to an unsigned integer with bitdepth=*qn*.

The following applies:

// now we are position on the right most corner sharing v  
 // we turn left an evaluate the possible predictions  
 do  
 {  
 oppoV = GetVertexIndex( pV , pO[ altC ] )  
 prevV = GetVertexIndex( pV , PreviousCorner( altC ) )  
 nextV = GetVertexIndex( pV , NextCorner( altC ) )  
   
 if ( ( oppoV > -1 && prevV > -1 && nextV > -1 ) &&  
 ( ( mV[ oppoV ] > 0 ) && ( mV[ prevV ] > 0 ) && ( mV[ nextV ] > 0 ) ) )  
 {  
 // parallelogram prediction estNrm = prevNrm + nextNrm – oppoNrm  
 // accumulate parallelogram predictions  
 predNorm[ 0 ] = predNorm[ 0 ]+  
 auxNorm[ prevV ][ 0 ]+auxNorm[ nextV ][ 0 ] - auxNorm[ oppoV ][ 0 ]  
 predNorm[ 1 ]=predNorm[ 1 ]+  
 auxNorm[ prevV ][ 1 ]+auxNorm[ nextV ][ 1 ] - auxNorm[ oppoV ][ 1 ]  
 predNorm[ 2 ]=predNorm[ 2 ]+  
 auxNorm[ prevV ][ 2 ]+auxNorm[ nextV ][ 2 ] - auxNorm[ oppoV ][ 2 ]  
 count++  
 }  
 // swing around the triangle fan  
 altC = PreviousCorner( pO[ PreviousCorner( altC ) ] )  
 // stop on incomplete fan or full rotation or max predictions reached  
 } while ( altC >= 0 && altC != startC && count < maxParallelograms )  
   
 // 1. use parallelogram prediction when possible  
 if( count > 0 ) {   
 // Normalize prediction to a unit vector  
 predNorm = normalize( predNorm )  
 // Convert float unit vector to an unsigned integer with quantized bitdepth  
 predNorm = normal\_quantize( predNorm,  
 mesh\_attribute\_bit\_depth\_minus1[ attrIndex ] + 1 )  
 if ( mesh\_normal\_octahedral\_flag[ attrIndex ] ) {  
 residual = mesh\_attribute\_residual[ attrIndex ][ auxDeltaCoarseIndex ]  
 decodeOctahedral( attrIndex, predNorm, residual, auxNorm[ v ] )  
 } else {  
 auxNorm[ 0 ] = mesh\_attribute\_residual[ attrIndex ][ auxDeltaIndex ][ 0 ]  
 + predNorm[ 0 ]  
 auxNorm[ 1 ] = mesh\_attribute\_residual[ attrIndex ][ auxDeltaIndex ][ 1 ]  
 + predNorm[ 1 ]  
 auxNorm[ 2 ] = mesh\_attribute\_residual[ attrIndex ][ auxDeltaIndex ][ 2 ]  
 + predNorm[ 2 ]  
 }  
 auxDeltaIndex = auxDeltaIndex + 1  
 return  
  
 }  
   
 // 2. or fallback to delta with available values if any  
 prevV = GetVertexIndex( pV , PreviousCorner( c ) )  
 nextV = GetVertexIndex( pV , NextCorner( c ) )  
   
 if( prevV > -1 && mV[ prevV ] > -1 ) {  
 if ( mesh\_normal\_octahedral\_flag[ attrIndex ] ) {  
 prediction = auxNorm[ prevV ]  
 residual = mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ]   
 decodeOctahedral( attrIndex, prediction, residual, auxNorm[ v ] )  
 } else {  
 auxNorm[ v ][ 0 ] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 0 ]  
 + auxNorm[ prevV ][ 0 ]  
 auxNorm[ v ][ 1 ] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 1 ]  
 + auxNorm[ prevV ][ 1 ]  
 auxNorm[ v ][ 2 ] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 2 ]  
 + auxNorm[ prevV ][ 2 ]  
 }  
 auxDeltaCoarseIndex = auxDeltaCoarseIndex + 1  
 return  
 }  
   
 if ( nextV > -1 && MV[ nextV ] > -1 )  {  
 if ( mesh\_normal\_octahedral\_flag[ attrIndex ] ) {  
 prediction = auxNorm[ nextV ]  
 residual = mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ]   
 decodeOctahedral( attrIndex, prediction, residual, auxNorm[ v ] )  
 } else {  
 auxNorm[ v ][ 0 ] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 0 ]  
 + auxNorm[ nextV ][ 0 ]  
 auxNorm[ v ][ 1 ] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 1 ]  
 + auxNorm[ nextV ][ 1 ]  
 auxNorm[ v ][ 2 ] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 2 ]  
 + auxNorm[ nextV ][ 2 ]  
 }  
 auxDeltaCoarseIndex = auxDeltaCoarseIndex + 1  
 return  
 }

Let the variable b, specifying the index of the previous corner on the boundary.

Let the variable bV, specifying the index of the vertex associated with b.

// 3. If we are on a boundary  
 // then we may use delta from previous vertex on the boundary  
 if( isBoundary ) {  
 b = PreviousCorner( startC )  
 bV = GetVertexIndex( pV, b )  
 if ( mV[ bV ] > -1 ) {  
 if ( mesh\_normal\_octahedral\_flag[ attrIndex ] ) {  
 prediction = auxNorm[ bV ]  
 residual = mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ]   
 decodeOctahedral( attrIndex, prediction, residual, auxNorm[ v ] )  
 } else {  
 auxNorm[ v ][ 0 ] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 0 ]   
 + auxNorm[ bV ][ 0 ]  
 auxNorm[ v ][ 1 ] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 0 ]  
 + auxNorm[ bV ][ 1 ]  
 auxNorm[v][2] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 2 ]  
 + auxNorm[ bV ][ 2 ]  
 }  
 auxDeltaCoarseIndex = auxDeltaCoarseIndex + 1  
 return  
 }  
  
   
 // 4. no more choices, we use an absolute value (i.e. a start)  
 auxNorm[ v ][ 0 ] = mesh\_attribute\_start[ attrIndex ][ auxStartIndex ][ 0 ]  
 auxNorm[ v ][ 1 ] = mesh\_attribute\_start[ attrIndex ][ auxStartIndex ][ 1 ]  
 auxNorm[ v ][ 2 ] = mesh\_attribute\_start[ attrIndex ][ auxStartIndex ][ 2 ]  
 auxStartIndex = auxStartIndex + 1

* + - * 1. Cross-product prediction of normals

Inputs to this process are:

* a variable attrIndex, specifying the index of the attribute on which to perform predictions.
* a variable c specifying the index of the corner for which vertex position will be predicted.

Output of this process is indirect:

* It modifies the array VertexMarkingArray, AuxiliaryStartIndex[ attrIndex ] and AuxiliaryDeltaIndex[ attrIndex ] defined in clause I.9.2 and the array AttrValues[ attrIndex ] defined in clause I.9.1.

Let the alias mV refer to the variable VertexMarkingArray.

Let the alias pO refer to the variable OppositeCornersArray.

Let the alias pV refer to the variable CornerToVertexArray.

Let the alias pG refer to the variable VertCoordValues.

Let the alias auxNorm refer to the variable AttrValues[attrIndex].

Let the alias auxStartIndex refer to the variable AuxiliaryStartIndex[ attrIndex ]

Let the alias auxDeltaIndex refer to the variable AuxiliaryDeltaIndexArray[ attrIndex ]

Let the alias auxDeltaCoarseIndex refer to the variable AuxiliaryDeltaCoarseIndexArray[ attrIndex ]

Let the variable v, specifying the index of the vertex position associated with c, be initialized as follows:

v = GetVertexIndex( CornerToVertexArray, c )

If mV[v] is strictly greater than 0, the vertex v is already predicted, then the process does nothing and returns. Otherwise, the following applies:

// we mark the vertex  
 mV[ v ] = 1

Let the 1D array predNorm, of size 3, specifying the predicted normal of the vertex associated with c, and the variable altC, specifying a corner index, and the variable nextC, specifying a corner index, be initialized as follows:

predNorm[ 0 ] = 0  
 predNorm[ 1 ] = 0  
 predNorm[ 2 ] = 0  
 altC = c  
 nextC = NextCorner( pO[ NextCorner( altC ) ] )  
   
 // loop through corners attached to the current vertex  
 // swing around the fan until we find a border  
 while ( nextC >= 0 && nextC != c)  
 {  
 altC = nextC  
 nextC = NextCorner( pO[ NextCorner( altC ) ] )  
 }

Let the variable isBoundary, specify if nextC is on a boundary, and the variable count, specify the number of valid parallelogram predictions found, and the variable startC, specify the index of the extreme corner of the fan, and be initialized as follows:

isBoundary = ( nextC != c )  
 startC = altC  
 count = 0

Let the variables prevV, oppoV and nextV, specifying the index of the vertex associated with previous, opposite, and next corners respectively, be set to 0.

Let the variable vCP of size 3 be the vector pointing from the current vertex position to the previous vertex position. Let the variable vCN of size 3 be the vector pointing from the current vertex position to the next vertex position.

Let the function *cross\_product* be the cross product of two vectors. Let the function *normalize* be the normalization of a vector into a unit float vector. Let the function *normal\_quantize(vector, qn)* convert a float unit *vector* centered at 0 to an unsigned integer with bitdepth=*qn*.

The following applies:

// now we are position on the right most corner sharing v  
 // we turn left an evaluate the possible predictions  
 do  
 {  
 prevV = GetVertexIndex( pV , PreviousCorner( altC ) )  
 nextV = GetVertexIndex( pV , NextCorner( altC ) )  
   
 if( prevV > -1 && nextV > -1 )  
 {  
 vCP[0] = pG[ prevV ][ 0 ] – pG[ v ][ 0 ]  
 vCP[1] = pG[ prevV ][ 1 ] – pG[ v ][ 1 ]  
 vCP[2] = pG[ prevV ][ 2 ] – pG[ v ][ 2 ]  
 vCN[0] = pG[ nextV ][ 0 ] – pG[ v ][ 0 ]  
 vCN[1] = pG[ nextV ][ 1 ] – pG[ v ][ 1 ]  
 vCN[2] = pG[ nextV ][ 2 ] – pG[ v ][ 2 ]  
 predNorm = cross\_product( vCP , vCN )   
 count++  
 }  
 // swing around the triangle fan  
 altC = PreviousCorner( O[ PreviousCorner( altC ) ] )  
 // stop on incomplete fan or full rotation or max predictions reached  
 } while ( altC >= 0 && altC != startC && count < maxParallelograms )  
   
 // 1. use cross product prediction when possible  
 if( count > 0 ) {   
 // Normalize prediction to a unit vector  
 predNorm = normalize( predNorm )  
 // Convert float unit vector to an unsigned integer with quantized bitdepth  
 predNorm = normal\_quantize( predNorm,  
 mesh\_attribute\_bit\_depth\_minus1[ attrIndex ] + 1 )  
 if ( mesh\_normal\_octahedral\_flag[ attrIndex ] ) {   
 residual = mesh\_attribute\_residual[ attrIndex ][ auxDeltaCoarseIndex ]  
 decodeOctahedral( attrIndex, predNorm, residual, auxNorm[ v ] )  
 } else {  
 auxNorm[ 0 ] = mesh\_attribute\_residual[ attrIndex ][ auxDeltaIndex ][ 0 ]   
 + predNorm[ 0 ]  
 auxNorm[ 1 ] = mesh\_attribute\_residual[ attrIndex ][ auxDeltaIndex ][ 1 ]   
 + predNorm[ 1 ]  
 auxNorm[ 2 ] = mesh\_attribute\_residual[ attrIndex ][ auxDeltaIndex ][ 2 ]   
 + predNorm[ 2 ]  
 }  
 auxDeltaIndex = auxDeltaIndex + 1  
 return  
 }  
   
 // 2. or fallback to delta with available values if any  
  
 prevV = GetVertexIndex( pV , PreviousCorner( c ) )  
 nextV = GetVertexIndex( pV , NextCorner( c ) )  
   
 if( prevV > -1 && mV[ prevV ] > -1 ) {  
 if ( mesh\_normal\_octahedral\_flag[ attrIndex ] ) {  
 prediction = auxNorm[ prevV ]  
 residual = mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ]  
 decodeOctahedral( attrIndex, prediction, residual, auxNorm[ v ] )  
 } else {  
 auxNorm[ v ][ 0 ] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 0 ]  
 + auxNorm[ prevV ][ 0 ]  
 auxNorm[ v ][ 1 ] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 1 ]  
 + auxNorm[ prevV ][ 1 ]  
 auxNorm[ v ][ 2 ] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 2 ]  
 + auxNorm[ prevV ][ 2 ]  
 }  
 auxDeltaCoarseIndex = auxDeltaCoarseIndex + 1  
 return  
 }  
 if ( nextV > -1 && MV[ nextV ] > -1 )  {  
 if ( mesh\_normal\_octahedral\_flag[ attrIndex ] ) {  
 prediction = auxNorm[ nextV ]  
 residual = mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ]   
 decodeOctahedral( attrIndex, prediction, residual, auxNorm[ v ] )  
 } else {  
 auxNorm[ v ][ 0 ] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 0 ]  
 + auxNorm[ nextV ][ 0 ]  
 auxNorm[ v ][ 1 ] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 1 ]  
 + auxNorm[ nextV ][ 1 ]  
 auxNorm[ v ][ 2 ] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 2 ]  
 + auxNorm[ nextV ][ 2 ]  
 }  
 auxDeltaCoarseIndex = auxDeltaCoarseIndex + 1  
 return  
 }

Let the variable b, specifying the index of the previous corner on the boundary.   
Let the variable bV, specifying the index of the vertex associated with b.

// 3. If we are on a boundary  
 // then we may use delta from previous vertex on the boundary  
 if( isBoundary ) {  
 b = PreviousCorner( startC )  
 bV = GetVertexIndex( pV, b )  
 if ( mV[ bV ] > -1 ) {  
 if ( mesh\_normal\_octahedral\_flag[ attrIndex ] ) {  
 prediction = auxNorm[ bV ]  
 residual = mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ]   
 decodeOctahedral( attrIndex, prediction, residual, auxNorm[ v ] )  
 } else {  
 auxNorm[ v ][ 0 ] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 0 ]   
 + auxNorm[ bV ][ 0 ]  
 auxNorm[ v ][ 1 ] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 0 ]  
 + auxNorm[ bV ][ 1 ]  
 auxNorm[v][2] =   
 mesh\_attribute\_coarse\_residual[ attrIndex ][ auxDeltaCoarseIndex ][ 2 ]  
 + auxNorm[ bV ][ 2 ]  
 }  
 auxDeltaCoarseIndex = auxDeltaCoarseIndex + 1  
 return  
 }

// 4. no more choices, we use an absolute value (i.e. a start)  
 auxNorm[ v ][ 0 ] = mesh\_attribute\_start[ attrIndex ][ auxStartIndex ][ 0 ]  
 auxNorm[ v ][ 1 ] = mesh\_attribute\_start[ attrIndex ][ auxStartIndex ][ 1 ]  
 auxNorm[ v ][ 2 ] = mesh\_attribute\_start[ attrIndex ][ auxStartIndex ][ 2 ]  
 auxStartIndex = auxStartIndex + 1

* + - * 1. Decode octahedral normals

Inputs to this process are:

* a variable *attrIndex*, specifying the index of the attribute on which to perform predictions.
* a variable *prediction*, specifying the three dimensional normal vector that was predicted.
* a variable *residual*, specifying the 2D octahedral representation of the first residual of normal.

Output of this Inputs to this process are:

* a variable *reconstructed*, specifying the three dimensional normal vector that was reconstructed after addition of first and possibly second residual.

Let the alias *secondRes* refer to the variable mesh\_normal\_octahedral\_second\_residual[ attrIndex ]

Let the alias *normSecondResidualIndex* refer to the variable NormalSecondResidualIndexArray defined in subclause I.9.2.

Let convert3Dto2Doctahedral( attrIndex, 3Dvector ) denote the invocation of the process defined in subclause I.9.7.6.5.1.

Let convert2DoctahedralTo3D( attrIndex, 2Dvector ) denote the invocation of the process defined in subclause I.9.7.6.5.2.

pred2D = convert3Dto2Doctahedral( attrIndex, prediction )  
 rec\_2D = pred2D + residual  
 rec\_3D = convert2DoctahedralTo3D( attrIndex, rec\_2D )  
 if( mesh\_normal\_octahedral\_second\_residual\_flag[ attrIndex ] ) {  
 reconstructed[ 0 ] = rec\_3D[ 0 ]  
 + secondRes[ attrIndex ][ normSecondResidualIndex ][ 0 ]  
 reconstructed[ 1 ] = rec\_3D[ 1 ]  
 + secondRes[ attrIndex ][ normSecondResidualIndex ][ 1 ]  
 reconstructed[ 2 ] = rec\_3D[ 2 ]  
 + secondRes[ attrIndex ][ normSecondResidualIndex ][ 2 ]  
 normSecondResidualIndex = normSecondResidualIndex + 1  
  } else {  
  reconstructed = rec\_3D\_without\_second\_residual  
  }

Convert 3D to 2D octahedral.

Inputs to this process are:

* a variable attrIndex, specifying the index of the attribute on which to perform predictions.
* a variable 3Dvector, specifying the three dimensional vector in unsigned integer format.

Output of this Inputs to this process are:

* a variable 2Dvector, specifying the two dimensional octahedral representation in unsigned integer format.

Let alias *qn* refer to the normal bit depth defined by *mesh\_attribute\_bit\_depth\_minus1[ attrIndex ]+1*

Let the function *floatToint(float , qn)* converts a float centered at 0 which between -1.0 and +1.0 range, to an unsigned integer with bitdepth=*qn*.

Let *center* be a three dimensional point defining the middle point (center point) of the normal 3D representation defined as:

center[ 0 ] = 2 ^ ( qn – 1 )  
 center[ 1 ] = 2 ^ ( qn – 1 )  
 center[ 2 ] = 2 ^ ( qn – 1 )

The input 3Dvector is first centered to zero and then normalized.

Let the function *absolute* give the modulus (absolute value) of a real number .

Let the funciton *copysign(input)* that extracts the sign from the *input* and applies it to a float of value 1.0 and returns the result.

3Dvector[ 0 ] = 3Dvector[ 0 ] - center[ 0 ]   
 3Dvector[ 1 ] = 3Dvector[ 1 ] - center[ 1 ]   
 3Dvector[ 2 ] = 3Dvector[ 2 ] - center[ 2 ]   
   
 // Then normalized to become a float in the range -1.0 to +1.0  
 sum = absolute( 3Dvector[ 0 ] )  
 + absolute( 3Dvector[ 1 ] ) + absolute( 3Dvector[ 2 ] )  
 3Dvector[ 0 ] = 3Dvector[ 0 ] / sum  
 3Dvector[ 1 ] = 3Dvector[ 1 ] / sum  
 3Dvector[ 2 ] = 3Dvector[ 2 ] / sum

Then convert the float 3D vector to a 2D octahedral representation:

if ( 3Dvector [ 2 ] >= 0){  
 2Dvector[ 0 ] = floatToint( 3Dvector[ 0 ] , qn )  
 2Dvector[ 1 ] = floatToint( 3Dvector[ 1 ] , qn )  
 } else {  
 2Dvector[0] = floatToint(  
 ( 1 - absolute( 3Dvector[ 1 ] ) \* copysign( 3Dvector[ 0 ] ) ), qn )  
 2Dvector[1] = floatToint(  
 ( 1 - absolute( 3Dvector[ 0 ] ) \* copysign( 3Dvector[ 1 ] ) ), qn )  
 }  
 return 2Dvector

Convert 2D octahedral to 3D.

Inputs to this process are:

* a variable attrIndex, specifying the index of the attribute on which to perform predictions.
* a variable 2Dvector, specifying the two dimensional octahedral representation in unsigned integer format.

Output of this Inputs to this process are:

* a variable 3Dvector, specifying the three dimensional vector in unsigned integer format.

Let alias *qn* refer to the normal bit depth defined by *mesh\_attribute\_bit\_depth\_minus1[ attrIndex ]**+1*

Let the function *intTofloat(int , qn)* convert an unsigned integer with bitdepth=*qn* to a float centered at 0 which is between -1.0 and +1.0 range.

Let the function *absolute* give the modulus (absolute value) of a real number .

Let the funciton *copysign* be able to copy the sign of the variable and assign it to a float of value 1.0

First step involves converting 2D octahedral representation to 3D vector.

3Dvector[ 0 ] = intTofloat( 2Dvector[ 0 ] , qn )   
 3Dvector[ 1 ] = intTofloat( 2Dvector[ 1 ] , qn )   
 3Dvector[ 2 ] = 1 – ( absolute( 2Dvector[ 0 ] ) + absolute( 2Dvector[ 1 ] ))  
   
 if ( 3Dvector[ 2 ] < 0 ) {  
 temporary\_x = 3Dvector[ 0 ]  
 3Dvector[ 0 ] = ( 1 – absolute( 3Dvector[ 1 ] ) ) \* copysign( temporary\_x )   
 3Dvector[ 1 ] = ( 1 – absolute( temporary\_x )) \* copysign( 3Dvector[ 1 ] )   
 }

Let the function *normalize* transform a given vector into a unit vector. It employs the L2 norm for normalization, resulting in an output within the range of -1.0 to 1.0.

Let the function *normal\_quantize(vector, qn)* convert a float unit *vector* centered at 0 to an unsigned integer with bitdepth=*qn*.

Second step involves normalizing the 3D vector and quantizing it to an unsigned integer.

// Normalize the 3D vector.  
 3Dvector = normalize( 3Dvector )  
 // Convert float unit vector to an unsigned integer with quantized bitdepth  
 3Dvector = normal\_quantize( 3Dvector, qn)  
 return 3Dvector

* + - 1. Prediction of per vertex color attributes

[Ed. Note Input required]

* + - 1. Prediction of per generic attributes

[Ed. Note Input required]

* + 1. Decoding auxiliary per face attributes
       1. General

Inputs to this process are:

* a variable attrIndex, specifying the index of the attribute on which to perform predictions.

This process has no direct output.

The following applies:

* If mesh\_attribute\_type[ attrIndex ] is equal to MESH\_ATTR\_MATERIAL\_ID, the process described in subclause I.9.8.2 is invoked with the variable attrIndex.
* Any other value for mesh\_attribute\_type[ attrIndex ] is not supported in this version of this document.
  + - 1. Decoding per face material ID attributes

Inputs to this process are:

* a variable attrIndex, specifying the index of the attribute on which to perform predictions.

This process has no direct output.

This process updates the array AttrValues[ attrIndex ], defined in clause I.9.1.

Let the variable prevFid specify the latest decoded face attribute value, and prevTri specify the triangle index of the latest decoded face, and the variable fidNotPredictedIdx specify a running index in the mesh\_attribute\_fine\_residual[ attrIndex ] 1D array, and the variable fidIsRightIdx specify a running index in the mesh\_materialid\_default\_right\_flag[ attrIndex ] 1D array, and the variable fidIsLeftIdx specify a running index in the mesh\_materialid\_default\_left\_flag[ attrIndex ] 1D array, and the variable fidIsFacingIdx specify a running index in the mesh\_materialid\_default\_facing\_flag[ attrIndex ] 1D array, and fidIsDifferentIdx specify a running index in the mesh\_materialid\_default\_not\_equal\_flag[ attrIndex ] 1D array, and the 1D array triangleMarkedArray, of size mesh\_triangle\_count, specify for each triangle if it Is marked or not, be initialized as follows:

prevFid = -1  
 prevTri = -1  
 fidNotPredictedIdx = 0  
 fidIsRightIdx = 0  
 fidIsLeftIdx = 0  
 fidIsFacingIdx = 0  
 fidIsDifferentIdx = 0  
 for( i = 0; i < mesh\_triangle\_count; ++i ) {  
 ptriangleMarkedArray[ i ] = 0  
 }

Let the 1D array corners, of size 3, specify a triplet of corner indices, and the variable tri specify the index of the current triangle for which a face ID must be decoded, the following applies:

for( i = 0; i < ProcessedCornersCnt; i++ ) {  
 corners[ 0 ] = ProcessedCorners[ i ]  
 corners[ 1 ] = NextCorner( ProcessedCorners[ i ] )   
 corners[ 2 ] = PreviousCorner( ProcessedCorners[ i ] )   
 tri = CornerToTriangle( ProcessedCorners[ i ] )  
 if(!mesh\_coded\_materialid\_default\_not\_equal\_flag[ attrIndex ]  
 [ fidIsDifferentIdx++ ] ) {  
 AttrValues[ attrIndex ][ tri ] = prevFid  
 triangleMarkedArray[ tri ] = 1  
 prevTri = tri  
 continue  
 }

Let the variable tFIdx specify the index of the facing triangle, and the variable tRIdx specify the index of the right triangle, and the varible tLIdx specify the index of the left triangle, and the variable decodedFacing specify if the font triangle decoded attribute value is available, and the variable decodedRight specify if the right triangle decoded attribute value is available, and the variable decodedLeft specify if the left triangle decoded attribute value is available, and the variable faceIdFacing specify the attribute value associated with the facing triangle if already decoded, and the variable faceIdRight specify the attribute value associated with the right triangle if already decoded, and the variable faceIdLeft specify the attribute value associated with the left triangle if already decoded, be initialized as follows:

tFIdx = CornerToTriangle( OppositeCorner( corners[ 0 ] ) )  
 tRIdx = CornerToTriangle( OppositeCorner( corners[ 1 ] ) )  
 tLIdx = CornerToTriangle( OppositeCorner( corners[ 2 ] ) )  
 decodedFacing = (tFIdx < 0 ? 0 : triangleMarkedArray[ tFIdx ] )  
 && !( tFIdx == prevTri )  
 decodedRight  = (tRIdx < 0 ? 0 : triangleMarkedArray[ tRIdx ] )  
 && !( tRIdx == prevTri )  
 decodedLeft   = (tLIdx < 0 ? 0 : triangleMarkedArray[ tLIdx ] )  
 && !( tLIdx == prevTri )  
 faceIdFacing = decodedFacing ? AttrValues[ attrIndex ]  
 [ CornerToTriangle( OppositeCorner( corners[ 0 ] ) ) ] : -1  
 faceIdRight  = decodedRight ? AttrValues[ attrIndex ]  
 [ CornerToTriangle( OppositeCorner( corners[ 1 ] ) ) ] : -1  
 faceIdLeft   = decodedLeft ? AttrValues[ attrIndex ]  
 [ CornerToTriangle( OppositeCorner( corners[ 2 ] ) ) ] : -1

Let the variable dec, specify if the adjacent triangle of current triangle tri is decoded or not, and rc specify the index of a rotating corner, the following applies:

for(k = 0; k < 3; k++)  
 {  
 dec = ( k == 0 ) ? decodedRight :  
 ( ( k == 1 ) ? decodedLeft : decodedFacing )  
 if( !dec ) {  
 /\* initialise a starting“"rotating corne”" as a corner of the non \*/  
 /\* encoded adjacent triangle (R for k==0, L if k==1, F if k==2) \*/  
 rc = OppositeCorner( corners[ ( k + 1 ) % 3 ] )  
 /\* then start looping through adjacent triangles, swinging around a \*/  
 /\* first corner shared with the triangle we are encoding \*/  
 rc = OppositeCorner( NextCorner( rc ) )  
 /\* stopping if an encoded triangle is found, or if the analyzed face \*/  
 /\* is one of the 3 original R,L,F triangles \*/  
 while(  
 rc >= 0   
 && rc != PreviousCorner( OppositeCorner( corners[ k % 3 ] ) )  
 && !triangleMarkedArray[ CornerToTriangle( rc ) ]  
 ) {  
 rc = OppositeCorner( NextCorner( rc ) )  
 }  
 if(rc < 0 || !triangleMarkedArray[ CornerToTriangle( rc ) ] ) {  
 /\* if no encoded face found, swing around the other shared corners \*/  
 rc = OppositeCorner( corners[ (k + 1) % 3 ] )  
 rc = OppositeCorner( PreviousCorner( rc ) )  
 while(  
 rc >= 0   
 && rc != NextCorner( OppositeCorner( corners[ ( k + 2 ) % 3 ] ) )  
 && !triangleMarkedArray[ CornertoTriangle( rc ) ]   
 ) {  
 rc = OppositeCorner( PreviousCroner( rc ) )  
 }  
 }

Let the variable newDecoded specifying a decoded attribute value be initialized as follows:

newDecoded = -1 /\* negative if no new decoded was found \*/

The following applies:

if( rc >= 0 && triangleMarkedArray[ CornertoTriangle( rc ) ]  
 && !( CornerToTriangle( rc ) == prevTri ) ) {  
 newDecoded = AttrValue[ CornertoTriangle( rc ) ]  
 }  
 /\* if an already encoded face is found during the procedure, then update \*/  
 /\* the corresponding“"decode”" (mean can be decoded) status and Id value \*/  
 if( newDecoded >= 0) {  
 if(k == 0) {  
 decodedRight = 1  
 faceIdRight = newDecoded  
 } else if( k == 1) {  
  decodedLeft = 1  
  faceIdLeft = newDecoded  
 } else {  
  decodedFacing = 1  
  faceIdFacing = newDecoded  
 }  
 }  
 }  
 } /\* end of inner for loop \*/  
 triangleMarkedArray[ tri ] = 1

Let the variable fid specifying the final face attribute value be initialized as follows:

fid = -1

Then, the following applies:

if( decodedRight ) {  
 if( mesh\_materialid\_default\_right\_flag[ attrIndex ][ fidIsRightIdx++ ] )  
 fid = faceIdRight  
 }  
 if( ( fid < 0 ) && decodedLeft && ( faceIdLeft != faceIdRight ) ) {  
 if( mesh\_materialid\_default\_left\_flag[ attrIndex ][ fidIsLeftIdx++ ] )  
 fid = faceIdLeft  
 }  
 if( ( fid < 0 ) && decodedFacing && ( faceIdFacing != faceIdRight )   
 && ( faceIdFacing != faceIdLeft ) ) {  
 if( mesh\_materialid\_default\_right\_facing[ attrIndex ][ fidIsFacingIdx++ ] )  
 fid = faceIdFacing  
 }  
 if( fid < 0 ) {  
 fid = mesh\_attribute\_fine\_residual[ attrIndex ][ fidNotPredictedIdx++ ]  
 }  
 prevFid = fid  
 prevTri = tri  
 AttrValues[ attrIndex ][ tri ] = fid  
 }

* + 1. Per fan deduplications

This process has no direct input.

This process has no direct output.

This process updates the array CornerToVertexArray, and AuxiliaryCornerToVertexArray[ attrIndex ] if the mesh\_attribute\_separate\_index\_flag[ attrIndex ] is equal to 1, defined in clause I.9.2, to finalize the deduplication of the vertices.

Let the alias aV refer to the current array to be updated, it is equal to CornerToVertexArray for the primary per vertex attributes, or it is equal to AuxiliaryCornerToVertexArray[ attrIndex ] for auxiliary per vertex attributes, invoked in clause I.9.1.

Let the alias aO refer to the current array to be used, it is equal to OppositeCornersArray for the primary per vertex attributes, or it is equal to AuxiliaryOppositeCornersArray[ attrIndex ] for auxiliary per vertex attributes, invoked in clause I.9.1.

Let the variables currCorner specify the index of a corner, and nextCorner specify the index of a corner, and splitVtxToVertex specify the index of a vertex. The following applies:

for( i = 0; i < DuplicatesCnt; i++ ) {  
 currCorner = DuplicatesArray[ i ][ 0 ]  
 splitVtxToVertex = DuplicatesArray[ i ][ 1 ]  
   
 aV[ currCorner ] = splitVtxToVertex  
 nextCorner = currCorner  
 while(  
 nextCorner = NextCorner( aO[ NextCorner( nextCorner ) ] )  
 , nextCorner != currCorner && nextCorner >= 0 ) {  
 aV[ nextCorner ] = splitVtxToVertex  
 }  
 if( nextCorner < 0 ) {  
 nextCorner = currCorner  
 while(  
 nextCorner = NextCorner( aO[ NextCorner( nextCorner ) ] )  
 , nextCorner != currCorner && nextCorner >= 0 )   
 {  
 aV[ nextCorner ] = splitVtxToVertex  
 }  
 }  
 }

* + 1. Post reindexing per vertex attributes
       1. General

This process reindexes the positions and attributes, attached to main connectivity, to provide an implicit traversal coherent with the one provided to motion prediction at encoding.

This process has no direct input.

This process has no direct output.

This process updates:

* the array VertCoordValues, defined in clause I.9.1.
* the array CornerToVertexArray, defined in clause I.9.2.

The sub-process described in sub-clause I.9.10.2 modifies global variables and arrays.

The following applies:

* If mesh\_vertex\_traversal\_method is not equal to MESH\_TRAVERSAL\_DEGREE, then the process described in subclause I.9.12 is invoked with the variables PrimaryVerticesCnt, mesh\_triangle\_count, and PrimaryStartCornersCnt and arrays PrimaryStartCornersArray, and OppositeCornersArray, and CornerToVertexArray as inputs, and the outputs are the variable TraversalCornersCnt and the 1D array TraversalCornersArray of size TraversalCornersCnt.
* Otherwise, there is no need to update the TraversalCornersArray, since it was initialized during the prediction of the attributes, see clause I.9.7.1.

Let the 1D array newIdx, of size PrimaryVerticesCnt, specify a set of indices and the 2D array newPos, of size PrimaryVerticesCnt x 3 specify a set of positions. The following applies:

/\* reindex the position indices and attributes \*/  
 for( i = 0; i < TraversalCornersCnt; ++i ) {  
 newIdx[ TraversalCornersArray[ i ] ] = i  
 newPos[ i ][ 0 ] = VertCoordValues[  
 CornerToVertexArray[ TraversalCornersArray[ i ] ] ][ 0 ]  
 newPos[ i ][ 1 ] = VertCoordValues[   
 CornerToVertexArray[ TraversalCornersArray[ i ] ] ][ 1 ]  
 newPos[ i ][ 2 ] = VertCoordValues[   
 CornerToVertexArray[ TraversalCornersArray[ i ] ] ][ 2 ]  
 }  
 /\* finalize vertex positions update \*/  
 for( i = 0; i < PrimaryVerticesCnt; ++i ) {  
 VertCoordValues[ i ][ 0 ] = newPos[ i ][ 0 ]  
 VertCoordValues[ i ][ 1 ] = newPos[ i ][ 1 ]  
 VertCoordValues[ i ][ 2 ] = newPos[ i ][ 2 ]  
 }

Next, the process described in subclause I.9.10.2 is invoked for each attribute which index attributeIndex is in range 0 to mesh\_attribute\_count - 1, inclusive and where mesh\_attribute\_per\_face\_flag[ attributeIndex ], and where mesh\_attribute\_separate\_index\_flag[ attrIndex ] is equal to 0 , and where AttrValuesCnt[ attributeIndex ] is greater than 0, with attrIndex as input.

Next, the following applies:

/\* finalize corner table update \*/  
 for( c = 0; c < CornerCnt; ++c ) {  
 CornerToVertexArray[ c ] = newIdx[ CornerToVertexArray[ c ] ]  
 }

* + - 1. Post reindexing auxiliary attributes

Inputs to this process are:

* a variable attrIndex, specifying the index of the attribute on which to perform predictions.

This process has no direct output.

This process updates:

* the array AttrValue[ attrIndex ], defined in clause I.9.1.

Let the 2D array newValues, of size PrimaryVerticesCnt × AttrValueDimension[ attrIndex ] specify a set of attribute values. The following applies:

for( i = 0; i < TraversalCornersCnt; ++i ) {  
 for( j = 0; j < AttrValueDimension[ attrIndex ]; j++ ) {  
 newValues[ i ][ j ] = AttrValues[attrIndex]  
 [ CornerToVertexArray[ TraversalCornersArray[ i ] ] ][ j ]  
 }  
 }  
 for( i = 0; i < AttrValueCount[ attrIndex ]; ++i ) {  
 for( j = 0; j < AttrValueDimension[ attrIndex ]; j++ ) {  
 AttrValues[ attrIndex ][ i ][ j ] = newValues[ i ][ j ]  
 }  
 }

* + 1. Conversion from corner table to indexed face set

At the beginning of this process, the values of the positions and attributes are already set. The process then performs the copy of the connectivity from the corner tables to the decoded mesh and perform the deduplication of non-manifold vertices if needed.

The following applies:

* First, the vertex coordinate indices are initialized as follows

for(i = 0; i < CornerCnt; ++i ) {  
 VertCoordIndices[ i ] = CornerToVertexArray[ i ]  
 }

* Next, for each attribute which index attributeIndex is in range 0 to mesh\_attribute\_count-1, inclusive and where mesh\_attribute\_per\_face\_flag[ attributeIndex ] is equal to 0, and where mesh\_attribute\_separate\_index\_flag[ attributeIndex ] is equal to 1, the following applies:

for( i = 0; i < CornerCnt; ++i ) {  
 AttrIndices[ attributeIndex ][ i ] =  
 AuxiliaryCornerToVertexArray[ attributeIndex ][ i ]  
 }

* Next, if mesh\_deduplicate\_methodis not equal to MESH\_DEDUP\_NONE, the following applies to deduplicate the non-manifold vertices:
  + Let the variable idx, specify a vertex index, and the 1D array mapping, of size VertCoordCount, be initialized as follows:

idx = 0  
 for( i = 0; i < VertCoordCount; ++i ) {  
 mapping[ i ] = -1  
 }

* + Next, the vertex coordinate indices are reconstructed as follows:

for( i = 0; i < CornerCnt; ++i ) {  
 if( mapping[ VertCoordIndices[ i ] ] < 0 )  
 {  
 mapping[ VertCoordIndices[ i ] ] = idx  
 idx = idx + 1  
 }  
 }  
 for( i = 0; i < CornerCnt; ++i ) {  
 VertCoordIndices[ i ] = mapping[ VertCoordIndices[ i ] ]  
 }

* + Let the 2D array newPositions, of size idx × 3, specify temporary placeholder for reorganized position values, be initialized as follows:

for( i = 0; i < VertCoordCount; ++i ) {  
 if(mapping[ i ] >= 0)  
 {  
 for( j = 0; j < 3; j++ ) {  
 newPositions[ mapping[ i ]][ j ] = VertCoordValues[ i ][ j ]  
 }  
 }  
 }

* + Next, the size of the vertex coordinates of the decoded mesh is reset to the size of newPositions, and its values are set to the values of newPositions as follows:

VertCoordCount = idx  
 for( i = 0; i < VertCoordCount; ++i ) {  
 for( j = 0; j < 3; j++ ){  
 VertCoordValues[ i ][ j ] = newPositions[ i ][ j ]  
 }  
 }

* Next, for each attribute which index attributeIndex is in range 0 to mesh\_attribute\_count-1, inclusive and where mesh\_attribute\_per\_face\_flag[ attributeIndex ] is equal to 0, and where mesh\_attribute\_separate\_index\_flag[ attributeIndex ] is equal to 0, the following applies:
  + Let the 2D array newAttrValues, of size idx × AttrValueDimension[ attributeIndex ], specify a temporary placeholder for reorganized attribute values. The following applies:

for( i = 0; i < VertCoordCount; ++i ) {  
 if( mapping[ i ] >= 0 ) {  
 for( j = 0; j < AttrValueDimension[ attributeIndex ]; j++ )  
 {  
 newAttrValues[mapping[ i ]][ j ] = AttrValues[ i ][ j ]  
 }  
 }  
 }

* + Next, the attribute values of the mesh are set to the values of newAttrValues as follows:

for( i = 0; i < VertCoordCount; ++i ) {  
 for( j = 0; j < AttrValueDimension[ attributeIndex ]; j++ )  
 {  
 AttrValues[ i ][ j ] = newAttrValues[ i ][ j ]  
 }  
 }

* Next, for each attribute which index attributeIndex is in range 0 to mesh\_attribute\_count-1, inclusive and where mesh\_attribute\_per\_face\_flag[ attributeIndex ] is equal to 0, and where mesh\_attribute\_separate\_index\_flag[ attributeIndex ] is equal to 1, the following applies:
  + Let the variable idx, specify a vertex index, and the 1D array attrMapping, of size AttrValueCount[ attributeIndex ], be initialized as follows:

idx = 0  
 for( i = 0; i < AttrValueCount[ attributeIndex ]; ++i ) {  
 attrMapping[ i ] = -1  
 }

* + Next, the attribute value indices with attributeIndex are reconstructed as follows:

for( i = 0; i < CornerCnt; ++i ) {  
 if( attrMapping[ AttrIndices[ attributeIndex ][ i ] ] < 0 )  
 {  
 attrMapping[ AttrIndices[ attributeIndex ][ i ] ] = idx  
 idx = idx + 1  
 }  
 }  
 for( i = 0; i < CornerCnt; ++i ) {  
 AttrIndices[ attributeIndex ][ i ] =  
 attrMapping[ AttrIndices[ i ] ]  
 }

* + Let the 2D array newAttrValues, of size idx × AttrValueDimension[ attributeIndex ], specify a temporary placeholder for reorganized attribute values. The following applies:

for( i = 0; i < AttrValueCount[ attributeIndex ]; ++i ) {  
 if( attrMapping [ i ] >= 0 ) {  
 for( j = 0; j < AttrValueDimension[ attributeIndex ]; j++ )  
 {  
 newAttrValues[attrMapping [ i ]][ j ] =   
 AttrValues[ i ][ j ]  
 }  
 }  
 }

* + Next, the size of the attribute values of the decoded mesh is reset to the size of newAttrValues, and its values are set to the values of newAttrValues as follows:

AttrValueCount[ attributeIndex ] = idx  
 for( i = 0; i < AttrValueCount[ attributeIndex ]; ++i ) {  
 for( j = 0; j < AttrValueDimension[ attributeIndex ]; j++  )  
 {  
 AttrValues[ attributeIndex ][ i ][ j ] =   
 newAttrValues[ i ][ j ]  
 }  
 }

* + 1. Duplication of per vertex position

This process has no direct input.

This process has no direct output.

This process duplicates per vertex position if mesh\_position\_reverse\_unification\_flag is equal to 1, invoked in clause I.9.1.

Let the alias UVCoordCount refer to AttrValueCount of UV coordinates.

Let the alias UVCoordIndices refer to AttrIndices of UV coordinates.

Let the 1D array posToTexture, of size VertCoordCount, specifying the mapping between VertCoordIndices and UVCoordIndices in the triangle, be initialized as follows:

for( i = 0; i < VertCoordCount; i++ ) {  
 posToTexture[ i ] = -1  
 }

Let the 1D array texToPosition, of size UVCoordCount, specifying the mapping between the UVCoordIndices and VertCoordIndices in the triangle, be initialized as follows:

for ( i = 0; i < UVCoordCount; i++ ) {  
 texToPosition[ i ] = -1  
 }

The following applies:

* If the size of UVCoordIndices is equal to 0, then return.
* Next, VertCoordValues is updated as follows:

for(i = 0; i < CornerCnt; i++ ) {  
 if( posToTexture[ VertCoordIndices[ i ] ] == -1 ) {  
 posToTexture[ VertCoordIndices[ i ] ] = UVCoordIndices[ i ]  
 texToPosition[ UVCoordIndices[ i ] ] = VertCoordIndices[ i ]  
 } else if( posToTexture[ VertCoordIndices[ i ] ] !=   
 UVCoordIndices[ i ] ) {  
 if( texToPosition[ UVCoordIndices[ i ] ] == -1 ) {  
 /\* Build duplicated point \*/  
 pt = VertCoordIndices[ i ]  
 for ( j = 0; j < 3; j++ ) {  
 VertCoordValues[ VertCoordCount ][ j ] =   
 VertCoordValues [ pt ][ j ]  
 }  
 posToTexture[ VertCoordCount ] = UVCoordIndices[ i ]  
 texToPosition[ UVCoordIndices[ i ] ] = VertCoordCount  
 VertCoordIndices[ i ] = VertCoordCount  
 VertCoordCount = VertCoordCount + 1  
 } else {  
 if( VertCoordValues[ VertCoordIndices[ i ] ][ 0 ] ==   
 VertCoordValues[texToPosition[ UVCoordIndices[ i ] ][ 0 ]  
 && VertCoordValues[ VertCoordIndices[ i ] ][ 1 ] ==   
 VertCoordValues[texToPosition[ UVCoordIndices[ i ] ][ 1 ]  
 && VertCoordValues[ VertCoordIndices[ i ] ][ 2 ] ==   
 VertCoordValues[texToPosition[ UVCoordIndices[ i ] ][ 2 ] ) {  
 VertCoordIndices[ i ] =   
 texToPosition[ UVCoordIndices[ i ] ]  
 }  
 }  
 }  
 }

* + 1. Adjustment of the decoded mesh
       1. General

This process has no direct input.

This process has no direct output.

This process modifies the attributes and connectivity of the decoded mesh if mesh\_position\_reverse\_unification\_flag is equal to 1, invoked in clause I.9.1.

Let the variable singleCoordCount specify the number of vertices with different position.

Let the 1D array coordCount, of size singleCoordCount, and the variable tmpCoordCount, both specifying the number of vertices with same position, be initialized to all 0s.

Let the variable i specify the current vetex index.

Let the 2D map recPointMap, of size singleCoordCount × CoordCount[ VertCoordValues[ i ] ], specifying the mapping between VertCoordValues and VertCoordIndices, be derived as follows:

for( i = 0; i < VertCoordCount; i++ ) {  
 tmpCoordCount = coordCount[ VertCoordValues[ i ] ]  
 recPointMap[ VertCoordValues[ i ] ][ tmpCoordCount ] = i  
 coordCount[ VertCoordValues[ i ] ]++  
 }

Let the 1D array triCount, of size VertCoordCount, and the variable tmpTriCount, both specifying the number of triangles each vertex belongs to, be initialized to all 0s.

Let the 2D array recVertexToTriangles, of size VertCoordCount × triCount[ VertCoordIndices[ i ] ], specifying the mapping between VertCoordIndices and triangle index, be derived as follows:

for( i = 0; i < CornerCnt; i++ ) {  
 tmpTriCount = triCount[ VertCoordIndices[ i ] ]  
 recVertexToTriangles[ VertCoordIndices[ i ] ][ tmpTriCount ] = i / 3  
 triCount[ VertCoordIndices[ i ] ]++  
 }

Let the variable dupIdx specify the current added vertex index, and the 1D array tmpPoint, of size 3, specify the current added vertex position. If mesh\_position\_added\_vertices\_count is greater than 0, the following applies:

for( i = 0; i < mesh\_position\_added\_vertices\_count; i++ ) {  
 dupIdx = addPointList[ i ]  
 tmpPoint = VertCoordValues[ dupIdx ]  
 VertCoordValues[ VertCoordCount ] = tmpPoint  
 recPointMap[ tmpPoint ] = VertCoordCount  
 VertCoordCount = VertCoordCount + 1  
 }

Let the variable delIdx specify the current deleted vertex index, and the 1D array tmpPoint, of size 3, specify the current deleted vertex position.

Let the the variable newPointIndex specify the target vertex index with same vertex position as tmpPoint, to be merged to.

Let a 1D array triangleList, of size triCount[ delIdx ], specify the triangle index.

If mesh\_position\_deleted\_vertices\_count is greater than 0, the following applies:

for( i = 0; i < mesh\_position\_deleted\_vertices\_count; i++) {  
 delIdx = deletedPointList[ i ]  
 tmpPoint = VertCoordValues[ delIdx ]  
 newPointIndex = recPointMap[ tmpPoint ][ 0 ]  
 triangleList = recVertexToTriangles[ delIdx ]  
 for( j = 0; j < triCount[ delIdx ]; j++ ){  
 for( k =0; k < 3; k++){  
 if( VertCoordIndices[3 \* triangleList[ j ] + k] == delIdx ){  
 VertCoordIndices[3 \* triangleList[ j ] + k] = newPointIndex  
 }  
 }  
 }  
 }

Let the variables mt, mp, mi specify the current modified triangle index, the current modified vertex relative index in the modified triangle and the current target vertex index of the modified vertex, respectively.

If mesh\_modified\_count is greater than 0, the following applies:

for( i = 0; i < mesh\_modified\_count; i++) {  
 mt = modifiedTriangles[ i ]  
 mp = modifiedVertices [ i ]  
 mi = targetVertices[ i ]  
 VertCoordIndices[ 3 \* mt + mp ] = mi  
 }

Finally, if mesh\_position\_deleted\_vertices\_count is greater than 0, the deletion of per vertex positions process described in subclause I.9.13.2 is invoked with deletePointList as input parameter.

* + - 1. Deletion of per vertex positions

Inputs to this process are:

* a 1D array deletePointList, of size mesh\_position\_deleted\_vertices\_count, specifying the deleted vertex indices.

This process has no direct output.

This process modifies the per vertex position of the decoded mesh.

First, sort deletePointList in ascending order.

Let the variables dp, slow, specifying the deleted vertex index, and the vertex index after deletion, be initialized to all 0s.

Let a 1D array mapping, of size VertCoordCount, be initialized as follows:

for( i = 0; i < VertCoordCount; i++ ) {  
 mapping[ i ] = -1  
 }

Then, the following applies:

for( i = 0; i < VertCoordCount; i++ ) { if( dp >= mesh\_position\_deleted\_vertices\_count )  
 dp = 0  
 if( i != deletePointList[ dp ] ) {  
 mapping[i] = slow  
 for ( j = 0; j < 3; j++ ) {  
 VertCoordValues[ slow ][ j ] = VertCoordValues[ i ][ j ]  
 }  
 slow++  
 } else {  
 dp ++  
 }  
 }

Next, the values in VertCoordValues after the index slow is erased, and VertCoordCount is reset to slow.

Finally, the vertex connectivity is updated as follows:

for( i = 0; i < CornerCnt; i++ ) {  
 VertCoordIndices[ i ] = mapping[ VertCoordIndices[ i ] ]  
 }

* + 1. Vertex degree traversal

The vertex degree traversal process performs a vertex-degree guided traversal of an input mesh connectivity provided as inputs to produce an ordered set of visited corners as an output.

Inputs to this process are:

* a variable vertexCnt, specifying the number of vertices in the input mesh,
* a variable triangleCnt, specifying the number of triangles in the input mesh,
* a variable ccStartCornersCnt, specifying the number of start corners to be used to initiate the traversals, usually equivalent to the number of connected components of the input mesh,
* a 1D array ccStartCornersArray, of size ccStartCornersArray, specifying the indices of the mesh corners that will be used to start the traversal process,
* a 1D array oppositeCornersArray, of size CornerCnt, specifying for the input mesh, the opposite corner index for each corner index in the range of 0 to CornerCnt - 1, inclusive,
* a 1D array cornerToVertexArray, of size CornerCnt, specifying for the input mesh, the vertex index for each corner index in the range of 0 to CornerCnt - 1, inclusive.

The outputs are:

* a variable visitedCornersCnt, specifying the number of corners that were traversed, initialized to 0,
* a 1D array visitedCornersArray, of size visitedCornersCnt, containing the indices of visited corners covering the vertex chain filled by the traversal method.

The process makes use of the following functions:

* IsCornerFaceVisited, defined in sub-clause I.9.14.5,
* NextCorner, defined in sub-clause I.9.15.2,
* PreviousCorner, defined in sub-clause I.9.15.1,
* RightCorner, defined in sub-clause I.9.15.4,
* LeftCorner, defined in sub-clause I.9.15.3,
* SetVertexVisited, defined in sub-clause I.9.14.9
* IsVertexVisited, defined in sub-clause I.9.14.8,
* SetFaceVisited, defined in sub-clause I.9.14.6,
* IsFaceVisited, defined in sub-clause I.9.14.4,
* SetCornerFaceVisited, defined in sub-clause I.9.14.7
* PopNextCornerToTraverse, defined in sub-clause I.9.14.1,
* CornerToTriangle, defined in sub-clause I.9.15.5,
* GetVertexIndex, defined in sub-clause I.9.15.7,
* ComputePriority, defined in sub-clause I.9.14.3,
* AddCornerToTraversalStack, defined in sub-clause I.9.14.2.

Let the constant maxPriority, specifying the number of priority levels for corner traversal, be initialized to 3.

Let the 1D array isFaceVisitedArray, of size triangleCnt, specifying whether each face of the mesh at a given index in the array has already been visited (isFaceVisitedArray[ index ] = 1) or not (isFaceVisitedArray[ index ] = 0) by the traversal algorithm, be initialized as follows:

for( i = 0; i < triangleCnt; i++ ) {  
 isFaceVisitedArray[ i ] = 0  
 }

Let the 1D array isVertexVisitedArray, of size vertexCnt, specifying whether each vertex of the mesh at a given index in the array has already been visited (isVertexVisitedArray[ index ] = 1) or not (isVertexVisitedArray[ index ] = 0) by the traversal algorithm, be initialized as follows:

for( i = 0; i < vertexCnt; i++ ) {  
 isVertexVisitedArray[ i ] = 0  
 }

Let the 1D array traversalStacksCntArray, of size maxPriority, that specifies the number of elements of each stack of corners in traversalStacksArray, be defined as follows:

for( i = 0; i < maxPriority; i++ ) {  
 traversalStacksCntArray[ i ] = 0  
 }

Let the 2D array traversalStacksArray, of size maxPriority × traversalStacksCntArray[ index ], with index ranging from 0 to maxPriority – 1 inclusive, specifying the stacks of available corners for each priority level.

Let the 1D array predictionDegreeArray, of size vertexCnt, specifying the prediction degree (i.e. the number of neighbor triangles for which vertex predictions have already been solved by the decoder) available for each vertex, be initialized as follows:

for( i = 0; i < vertexCnt; i++ ) {  
 predictionDegreeArray[ i ] = 0  
 }

Let the variable bestPriority, specifying the current best available priority for corner processing, be initialized to 0.

Let the variable cornerId specify the index of the corner from ccStartCornersArray that is currently being processed.

Let the variable currentVertex specify the vertex index of the corner corresponding to the current cornerId and the current face that are being processed.

Let the variable nextVertex specify the vertex index of the next corner to process from ccStartCornersArray, relative to the current cornerId and the current face that are being processed.

Let the variable previousVertex specify the vertex index of the previous corner that was processed from ccStartCornersArray, relative to the current cornerId and the current face that are being processed.

Let the variable faceId specify the face index corresponding to the current corner (indexed by cornerId) that is being processed.

Let the variable vertId specify the vertex index of the vertex corresponding to the current cornerId.

Let the variable rightCornerId specify the index of the corner to the right of the corner indexed by the current cornerId.

Let the variable leftCornerId specify the index of the corner to the left of the corner indexed by the current cornerId.

Let the variable rightFaceId specify the index of the triangle face corresponding to rightCornerId.

Let the variable leftFaceId specify the index of the triangle face corresponding to leftCornerId.

Let the variable isRightFaceVisited specify whether the face indexed by rightFaceId has already been visited (isRightFaceVisited = 1) or not (isRightFaceVisited = 0) by the traversal process.

Let the variable isLeftFaceVisited specify whether the face indexed by leftFaceId has already been visited (isLeftFaceVisited = 1) or not (isLeftFaceVisited = 0) by the traversal process.

Let the variable leftPriority specify the traversal priority of the corner indexed by leftCornerId.

Let the variable rightPriority specify the traversal priority of the corner indexed by rightCornerId.

The following applies:

for( i = 0; i < ccStartCornersCnt; i++ ) {  
   
 cornerId = ccStartCornersArray[ i ]  
   
 /\* If the corner with the current cornerId has already been visited, \*/  
 /\* move onto the next corner inside ccStartCornersArray \*/  
 if( IsCornerFaceVisited( cornerId, isFaceVisitedArray ) ) {  
 continue  
 }  
   
 traversalStacksArray[ 0 ][ traversalStacksCnt[ 0 ]++ ] = cornerId  
   
 /\* bestPriority is a variable that keeps track of the best available \*/  
 /\* priority for corner processing \*/  
 bestPriority = 0  
   
 /\* For the current face related to cornerId, fetch the vertex indices \*/  
 /\* of the remaining corners of this face \*/  
 nextVertex = cornerToVertexArray( NextCorner( cornerId ) )  
 previousVertex = cornerToVertexArray( PreviousCorner( cornerId ) )  
 currentVertex = cornerToVertexArray( cornerId )  
   
 if( nextVertex < 0 || previousVertex < 0 ) {  
 /\* Move onto the next corner in ccStartCornersArray \*/  
 continue  
 }  
   
 if( !isVertexVisitedArray( currentVertex ) ) {  
 SetVertexVisited( currentVertex, isVertexVisitedArray )  
 visitedCornersArray[ visitedCornersCnt++ ] = cornerId  
 }  
 if( !isVertexVisitedArray( nextVertex ) ) {  
 SetVertexVisited( nextVertex, isVertexVisitedArray )  
 visitedCornersArray[ visitedCornersCnt++ ] = NextCorner( cornerId )  
 }  
 if( !isVertexVisitedArray( previousVertex ) ) {  
 SetVertexVisited( previousVertex, isVertexVisitedArray )  
 visitedCornersArray[ visitedCornersCnt++ ] = PreviousCorner( cornerId )  
 }  
   
 /\* Start the actual traversal, from the corner indexed by cornerId \*/  
 /\* This traversal is going to follow either the right or left neighbouring \*/  
 /\* faces to the current corner, based on their prediction degree \*/  
 while( ( cornerId = PopNextCornerToTraverse( bestPriority,  
 maxPriority,  
 traversalStacksArray,  
 traversalStacksCntarray,  
 nextCornerId) ) != -1 ) {  
 /\* Get the face index corresponding to the current corner \*/  
 faceId = CornerToTriangle( cornerId )  
 if( isFaceVisitedArray( faceId ) ) {  
 /\* Move onto the next cornerId \*/  
 continue  
 }  
   
 while( 1 ) {  
 faceId = CornerToTriangle( cornerId )  
 SetFaceVisited( faceId, isFaceVisitedArray )  
   
 /\* Fetch the vertex index of the newly reached vertex related \*/  
 /\* to the current cornerId \*/  
 vertId = GetVertexIndex( cornerToVertexArray, cornerId )  
   
 if( !isVertexVisitedArray( vertId ) ) {  
 SetVertexVisited( vertId, isVertexVisitedArray )  
 visitedCornersArray[ visitedCornersCnt++ ] = cornerId  
 }  
   
 rightCornerId = RightCorner( oppositeCornersArray, cornerId )  
 leftCornerId = LeftCorner( oppositeCornersArray, cornerId )  
   
 rightFaceId = CornerToTriangle( rightCornerId )  
 leftFaceId = CornerToTriangle( leftCornerId )  
   
 isRightFaceVisited = isFaceVisitedArray( rightFaceId )  
 isLeftFaceVisited = isFaceVisitedArray( leftFaceId )  
   
 if( !isLeftFaceVisited ) {  
 leftPriority = ComputePriority( cornerToVertexArray,  
 leftCornerId,  
 predictionDegreeArray,  
 maxPriority,  
 leftPriority )  
 if( isRightFaceVisited && leftPriority <= bestPriority ) {  
 /\* The left face will be visited next, so no need to put \*/  
 /\* it on the stack \*/  
 cornerId = leftCornerId  
 continue  
 }  
 else {  
 AddCornerToTraversalStack( leftCornerId,  
 leftPriority,  
 bestPriority,  
 traversalStacksArray,  
 traversalStacksCnt )  
 }  
 }  
   
 if( !isRightFaceVisited ) {  
 rightPriority = ComputePriority( cornerToVertexArray,  
 rightCornerId,  
 predictionDegreeArray,  
 maxPriority,  
 rightPriority )  
 if( rightPriority <= bestPriority ) {  
 /\* The right face will be visited next, so no need to put \*/  
 /\* it on the stack \*/  
 cornerId = rightCornerId  
 continue  
 }  
 else {  
 AddCornerToTraversalStack( rightCornerId,  
 rightPriority,  
 bestPriority,  
 traversalStacksArray,  
 traversalStacksCntArray )  
 }  
 }  
 /\* Couldn't proceed directly to the next corner \*/  
 break  
 } /\* Inner while loop \*/  
 } /\* Outer while loop \*/  
 } /\* for loop ccStartCornersCnt \*/

* + - 1. Getting the index of the next corner to traverse and removing it from the traversal stack

Let the function PopNextCornerToTraverse be defined as follows:

PopNextCornerToTraverse(  
 bestPriority, maxPriority, traversalStacksArray,  
 traversalStacksCnt, nextCornerId )  
 {  
 for( i = bestPriority; i < maxPriority; i++ ) {  
 if(traversalStacksCntArray[ i ] > 0) {  
 nextCornerId =  
 traversalStacksArray[ i ][ traversalStacksCntArray[ i ] - 1 ]  
 traversalStacksCntArray[ i ]--  
 bestPriority = i  
 return nextCornerId  
 }  
 }  
 return -1  
 }

* + - 1. Adding a corner to a traversal stack according to its traversal priority

Let the function AddCornerToTraversalStack be defined as follows:

AddCornerToTraversalStack(  
 cornerId, priority, bestPriority,  
 traversalStacksArray, traversalStacksCntArray )  
 {  
 traversalStacksArray[ priority ][ traversalStacksCntArray[ priority ]++ ] =  
 cornerId  
 if( priority < bestPriority ) {  
 bestPriority = priority  
 }  
 }

* + - 1. Getting the traversal priority of a corner

The function ComputePriority defined in this clause makes use of the following functions:

* GetVertexIndex, defined in sub-clause I.9.15.7,
* IsVertexVisited, defined in sub-clause I.9.14.8.

Let the function ComputePriority be defined as follows:

ComputePriority( cornerToVertexArray, cornerId, predictionDegreeArray,  
 maxPriority, cornerTraversalPriority )  
 {  
 currentVertex = GetVertexIndex( cornerToVertexArray, cornerId )  
 cornerTraversalPriority = 0  
   
 if( !IsVertexVisited( currentVertex ) ) {  
 predictionDegreeArray[ currentVertex ] += 1  
 cornerTraversalPriority =  
 ( predictionDegreeArray[ currentVertex ] > 1 ? 1 : 2 )  
 }  
   
 if( cornerTraversalPriority >= maxPriority ) {  
 cornerTraversalPriority = maxPriority - 1  
 }  
   
 return cornerTraversalPriority  
 }

* + - 1. Testing if a face is set as visited

Let the function IsFaceVisited be defined as follows:

IsFaceVisited( faceId, isFaceVisitedArray )  
 {  
 if( faceId < 0 ) {  
 return 1  
 }  
 return isFaceVisitedArray[ faceId ]  
 }

* + - 1. Testing if the face of a corner is set as visited

The function IsCornerFaceVisited defined in this clause makes use of the function CornerToTriangle defined in clause I.9.15.5.

Let the function IsCornerFaceVisited be defined as follows:

IsCornerFaceVisited( cornerId, isFaceVisitedArray )  
 {  
 if( cornerId < 0 ) {  
 return 1  
 }  
 return isFaceVisitedArray[ CornerToTriangle( cornerId ) ]  
 }

* + - 1. Setting a face as visited given its index

Let the function SetFaceVisited be defined as follows:

SetFaceVisited( faceId, isFaceVisitedArray )  
 {  
 isFaceVisitedArray[ faceId ] = 1  
 }

* + - 1. Setting a face as visited given the index of one of its three corners

The function SetCornerFaceVisited defined in this clause makes use of the function CornerToTriangle defined in clause I.9.15.5.

Let the function SetCornerFaceVisited be defined as follows:

SetCornerFaceVisited( cornerId, isFaceVisitedArray )  
 {  
 isFaceVisitedArray[ CornerToTriangle( cornerId ) ] = 1  
 }

* + - 1. Testing if a vertex is set as visited

Let the function IsVertexVisited be defined as follows:

IsVertexVisited( vertId, isVertexVisitedArray )  
 {  
 return isVertexVisitedArray[ vertId ]  
 }

* + - 1. Setting a vertex as visited

Let the function SetVertexVisited be defined as follows.

SetVertexVisited( vertId, isVertexVisitedArray )  
 {  
 isVertexVisitedArray[ vertId ] = 1  
 }

* + 1. Corner Table
       1. Getting the previous corner

Let the function PreviousCorner be defined as follows:

PreviousCorner( corner ) {  
 return( corner < 0 ) ? -1 : ( corner % 3 ? corner - 1 : corner + 2 )  
 }

* + - 1. Getting the next corner

Let the function NextCorner be defined as follows:

NextCorner( corner ) {  
 return( corner < 0 ) ? -1 : ( ++corner % 3 ? corner : corner – 3 )  
 }

* + - 1. Getting the left corner using opposites

Let the function LeftCorner be defined as follows:

LeftCorner( oppositeCornersArray, corner ) {  
 return ( corner < 0 ) ? -1 : oppositeCornersArray[ PreviousCorner( corner ) ]  
 }

* + - 1. Getting the right corner using opposites

Let the function RightCorner be defined as follows:

RightCorner( oppositeCornersArray, corner ) {  
 return ( corner < 0 ) ? -1 : oppositeCornersArray[ NextCorner( corner ) ]  
 }

* + - 1. Converting corner index to triangle index

Let the function CornerToTriangle be defined as follows:

CornerToTriangle( corner ) {  
 return( corner < 0 ) ? -1 : ( c / 3 )  
 }

* + - 1. Setting a vertex index corresponding to a corner index

Let the function SetVertexIndex be defined as follows:

SetVertexIndex( cornerToVertexArray, corner, vertex ) {  
 if( corner >= 0 && corner < CornerCnt )  
 cornerToVertexArray[ corner ] = vertex  
}

* + - 1. Getting a vertex index corresponding to a corner index

Let the function GetVertexIndex be defined as follows:

GetVertexIndex( cornerToVertexArray, corner ) {  
 return ( corner < 0 ) ? -1 : cornerToVertexArray[ corner ]  
 }

* + - 1. Setting an opposite corner

Let the function SetOpposite be defined as follows:

SetOpposite( oppositeCornersArray, corner, opposite ) {  
 if( corner >= 0 && corner < CornersCnt )  
 oppositeCornersArray[ corner ] = opposite  
}

* + - 1. Getting an opposite corner from primary corner table

Let the function OppositeCorner be defined as follows:

OppositeCorner( corner, opposite ) {  
 if( corner >= 0 && corner < CornersCnt ) {  
 return OppositeCornersArray[ corner ]  
 }  
 else return -1  
 }

* 1. Parsing Process
     1. General

Inputs to this process are bits from the MEB bitstream.

Outputs of this process are syntax element values.

This process is invoked when the descriptor of a syntax element in the syntax tables is equal to ue(v), or ae(v) (see clause I.10.3).

* + 1. Parsing process for 0-th order Exp-Golomb codes

The specifications in ISO/IEC 23090-5(2E):2023 clause 14.2 and its subclauses apply.

* + 1. CABAC parsing process
       1. General

Inputs to this process are a request for a value of a syntax element and values of prior parsed syntax elements.

Output of this process is the value of the syntax element.

The initialization process as specified in clause I.10.3.2 is invoked when starting the parsing of a MEB bitstream.

For each requested value of a syntax element a binarization is derived as specified in clause K.1 and Table K-2 (or … The specifications in subclause K.1 apply with the following additions, … and move the table here ? – this table includes reference to bin selection currently in I.10.3.4)

The binarization for the syntax element and the sequence of parsed bins determines the decoding process flow as described in clause K.2 (including reference to K.2.7 which is Annex I specific for mesh\_clers\_symbol … or move K.2.7 here ?).

The state update process as specified in clause I.10.3.3 is invoked after decoding any syntax element.

* + - 1. Initialization process

Output of this process are initialized CABAC internal variables, and the initialized state variables used when parsing mesh\_clers\_symbol elements.

First the initialization process for CABAC parsing is invoked as specified in clause K.3.1.

Then the initialization process for state variables used when parsing mesh\_clers\_symbol elements is invoked as specified in clause I.10.3.2.1.

* + - * 1. Initialization process for mesh\_clers\_symbol[] syntax elements

A variable Crun, specifying the length of successive parsed values equal to CLERS\_C, is initialized to 0.

Two variables ClersSymbol0 and ClersSymbol1 specify the values of the last two previously parsed values in the mesh\_clers\_symbol array, where ClersSymbol0 is the latest parsed symbol, are initialized as follows:

ClersSymbol1 = CLERS\_C  
 ClersSymbol0 = CLERS\_R

A variable CtxIdxClers , specifying the current contextual probability model, is initialized to XXX.

The value of the variable CtxIdxClers is updated before parsing each individual bin of syntax elements in the mesh\_clers\_symbol array as defined in clause I.10.3.4.1.

The value of the variables Crun, ClersSymbol0, ClersSymbol1 are updated after parsing each element in the mesh\_clers\_symbol array as defined in clause I.10.3.3.1.

* + - 1. State update process

Output of this process are updated state variables used when parsing mesh\_clers\_symbol elements.

* + - * 1. State update process for mesh\_clers\_symbol[ i ] syntax elements

The variables ClersSymbol0 ClersSymbol1 and Crun are updated after parsing each element in the mesh\_clers\_symbol array.

When the i-th element has been parsed, those variables are updated as follows:

ClersSymbol1 = ClersSymbol0  
 ClersSymbol0 = mesh\_clers\_symbol[ i ]  
 if( mesh\_clers\_symbol[ i ] == CLERS\_C )   
 Crun += 1  
 else  
 Crun = 0

* + - 1. Context Selection
         1. Determination of CtxIdxClers for a bin of mesh\_clers\_symbol[]

Contextualization depends upon BinIdxClers, ClersSymbol0, ClersSymbol1, Crun and mesh\_clers\_count.

CtxIdxClers is determined as follows:

if( ClersSymbol0 == CLERS\_C )  
 CtxIdxClers = ( mesh\_clers\_count <= 3000 ) ? 0   
 : Crun < 2 ? 0   
 : Crun < 7 ? 1 : 2  
 else if( ( ClersSymbol0== CLERS\_S )  
 CtxIdxClers = 3  
 else if( ( ClersSymbol0== CLERS\_L )  
 CtxIdxClers = 4  
 else if( ClersSymbol0 == CLERS\_R )  
 CtxIdxClers = 5 + ( ClersSymbol1 == CLERS\_C ? 0  
 : ( ClersSymbol1 == CLERS\_R ? 1 : 2 ) )  
 else /\* if ClersSymbol0 == CLERS\_E \*/  
 CtxIdxClers = 8  
 CtxIdxClers += 9 \* BinIdxClers  
 if (BinIdxClers > 1) CtxIdxClers -= 3 \* (BinIdxClers - 1)

1. (Normative)  
     
   Arithmetic coded displacement sub-bitstream   
   1. Scope

This annex specifies the displacement sub-bitstream. In this section, a bitstream refers to a displacement sub-bitstream unless otherwise indicated.

* 1. Normative references

The specifications in Clause 2 apply.

* 1. Terms and definitions

The specifications in Clause 3 apply.

* 1. Abbreviated terms

DDB Decoded Displacement Buffer

* 1. Conventions

The specifications in Clause 5 apply.

* 1. Bitstream format
     1. NAL bitstream formats

This subclause specifies the relationship between the network abstraction layer (NAL) unit stream and the NAL sample stream, either of which are referred to as the NAL bitstream.

The bitstream can be in one of two formats: the NAL unit stream format or the sample stream format. The NAL unit stream format is conceptually the more "basic" type. It consists of a sequence of syntax structures called NAL units. This sequence is ordered in decoding order, as described in subclause J.7.2.1.3. There are constraints imposed on the decoding order (and contents) of the NAL units in the NAL unit stream.

NOTE – The NAL unit stream format is commonly not intended to be used in any applications on its own since it requires additional information, i.e. sub-bitstream size information, for decoding its associated sub-bitstreams. One method of achieving this is through the use of the NAL sample stream format.

The NAL sample stream format can be constructed from the NAL unit stream format by ordering the NAL units in decoding order and prefixing each NAL unit with a heading that specifies the exact size, in bytes, of the NAL unit. A sample stream header is included at the beginning of the sample stream bitstream that specifies the precision, in bytes, of the signalled NAL unit size. The NAL unit stream format can be extracted from the sample stream format by traversing through the sample stream format, reading the size information and appropriately extracting each NAL unit. Methods of framing NAL units in a manner other than the use of the sample stream format are outside the scope of this document. The sample stream format is specified in Annex D.

* 1. Syntax and semantics
     1. Syntax in tabular form
        1. NAL unit syntax
           1. General NAL unit syntax

|  |  |
| --- | --- |
| displ\_nal\_unit( NumBytesInNalUnit ) { | **Descriptor** |
| displ\_nal\_unit\_header( ) |  |
| NumBytesInRbsp = 0 |  |
| for( i = 2; i < NumBytesInNalUnit; i++ ) |  |
| **rbsp\_byte**[ NumBytesInRbsp++ ] | b(8) |
| } |  |

* + - * 1. NAL unit header syntax

|  |  |
| --- | --- |
| displ\_nal\_unit\_header( ) { | **Descriptor** |
| **displ\_nal\_forbidden\_zero\_bit** | f(1) |
| **displ\_nal\_unit\_type** | u(6) |
| **displ\_nal\_layer\_id** | u(6) |
| **displ\_nal\_temporal\_id\_plus1** | u(3) |
| } |  |

* + - 1. Raw byte sequence payloads, trailing bits, and byte alignment syntax
         1. Displacement sequence parameter set RBSP syntax

General displacement sequence parameter set RBSP syntax

|  |  |
| --- | --- |
| displ\_sequence\_parameter\_set\_rbsp( ) { | **Descriptor** |
| **dsps\_sequence\_parameter\_set\_id** | u(4) |
| **dsps\_codec\_id** | u(8) |
| dsps\_profile\_tier\_level( ) |  |
| **dsps\_single\_dimension\_flag** | u(1) |
| **dsps\_msb\_align\_flag** | u(1) |
| **dsps\_log2\_max\_displ\_frame\_order\_cnt\_lsb\_minus4** | ue(v) |
| **dsps\_max\_dec\_displ\_frame\_buffering\_minus1** | ue(v) |
| **dsps\_max\_num\_reorder\_frames** | ue(v) |
| **dsps\_max\_latency\_increase\_plus1** | ue(v) |
| **dsps\_long\_term\_ref\_displ\_frames\_flag** | u(1) |
| **dsps\_num\_ref\_displ\_frame\_lists\_in\_dsps** | ue(v) |
| for( i = 0; i < dsps\_num\_ref\_displ\_frame\_lists\_in\_dsps; i++ ) |  |
| displ\_ref\_list\_struct( i ) |  |
| **dsps\_geometry\_3d\_bit\_depth\_minus1** | u(5) |
| **dsps\_subdivision\_method** | u(3) |
| if( dsps\_subdivision\_method != 0 ) { |  |
| **dsps\_subdivision\_iteration\_count** | u(3) |
| DspsSubdivisionCount = dsps\_subdivision\_iteration\_count |  |
| } else |  |
| DspsSubdivisionCount = 0 |  |
| **dsps\_displacement\_reference\_qp** | u(7) |
| displ\_quantization\_parameters( 0, DspsSubdivisionCount ) |  |
| **dsps\_extension\_present\_flag** | u(1) |
| if( dsps\_extension\_present\_flag ) { |  |
| **dsps\_extension\_count** | u(7) |
| } |  |
| if( bmsps\_extension\_count ){ |  |
| **dsps\_extensions\_length\_minus1** | ue(v) |
| for( i = 0; i < dsps\_extension\_count; i++ ) { |  |
| **dsps\_extension\_type**[ i ] | u(8) |
| **dsps\_extension\_length**[ i ] | u(16) |
| dsps\_extension( dsps\_extension\_type[ i ], dsps\_extension\_length[ i ] ) |  |
| } |  |
| } |  |
| rbsp\_trailing\_bits( ) |  |
| } |  |

Displacement SPS extension syntax

|  |  |
| --- | --- |
| dsps\_extension( extension\_type, extension\_length ) { | **Descriptor** |
| for( j = 0; j < extension\_length; j++ ) |  |
| **dsps\_extension\_data\_byte** | u(8) |
| length\_alignment( ) |  |
| } |  |

Quantization parameters syntax

|  |  |
| --- | --- |
| displ\_quantization\_parameters( qpIndex, subdivisionCount ){ | **Descriptor** |
| **dqp\_lod\_quantization\_flag**[ qpIndex ] | u(1) |
| **dpq\_bitdepth\_offset**[ qpIndex ] | se(v) |
| if( dqp\_lod\_quantization\_flag[ qpIndex ] == 0 ) { |  |
| for( k = 0; k < MaxDimension; k++) { |  |
| **dqp\_quantization\_parameters**[ qpIndex ][ k ] | u(7) |
| for( i=0 ; i < subdivisionCount + 1; i++ ) |  |
| QuantizationParameter[ qpIndex ][ i ][ k ] =  dqp\_quantization\_parameters[ qpIndex ][ k ] |  |
| **dqp\_log2\_lod\_inverse\_scale**[ qpIndex ][ k ] | u(2) |
| } |  |
| } else { |  |
| for( i=0 ; i < subdivisionCount + 1; i++ ) { |  |
| for( k = 0; k < MaxDimension; k++ ) { |  |
| **dqp\_lod\_delta\_quantization\_parameter\_value**[ qpIndex ][ i ][ k ] | ue(v) |
| if( dqp\_lod\_delta\_quantization\_parameter\_value[ qpIndex ][ i ][ k ] ) |  |
| **dqp\_lod\_delta\_quantization\_parameter\_sign**[ qpIndex ][ i ][ k ] | u(1) |
| if( qpIndex = 0 )  QuantizationParameter[ qpIndex ][ i ][ k ] =  dsps\_displacement\_reference\_qp + ( 1 – 2 \*  dqp\_lod\_delta\_quantization\_parameter\_sign[ qpIndex ][ i ][ k ] ) \*   dqp\_lod\_delta\_quantization\_parameter\_value[ qpIndex ][ i ][ k ]  else  QuantizationParameter[ qpIndex ][ i ][ k ] =  QuantizationParameter[ qpIndex – 1 ][ i ][ k ] + ( 1 – 2 \*  dqp\_lod\_delta\_quantization\_parameter\_sign[ qpIndex ][ i ][ k ] ) \*   dqp\_lod\_delta\_quantization\_parameter\_value[ qpIndex ][ i ][ k ] |  |
| } |  |
| } |  |
| } |  |
| } |  |

Displacement profile, tier, and level syntax

|  |  |
| --- | --- |
| dsps\_profile\_tier\_level( ) { | **Descriptor** |
| **dptl\_tier\_flag** | u(1) |
| **dptl\_profile\_codec\_group\_idc** | u(7) |
| **dptl\_profile\_toolset\_idc** | u(8) |
| **dptl\_reserved\_zero\_32bits** | u(32) |
| **dptl\_level\_idc** | u(8) |
| **dptl\_num\_sub\_profiles** | u(6) |
| **dptl\_extended\_sub\_profile\_flag** | u(1) |
| for( i = 0; i < dptl\_num\_sub\_profiles; i++ ) { |  |
| **dptl\_sub\_profile\_idc**[ i ] | u(v) |
| } |  |
| **dptl\_toolset\_constraints\_present\_flag** | u(1) |
| if( dptl\_toolset\_constraints\_present\_flag ) { |  |
| dptl\_profile\_toolset\_constraints\_information( ) |  |
| } |  |
| } |  |

Profile toolset constraints information syntax

|  |  |
| --- | --- |
| dptl\_profile\_toolset\_constraints\_information( ) { | **Descriptor** |
| **dptc\_one\_displacement\_frame\_only\_flag** | u(1) |
| **dptc\_reserved\_zero\_7bits** | u(6) |
| **dptc\_num\_reserved\_constraint\_bytes** | u(8) |
| for( i = 0; i < dptc\_num\_reserved\_constraint\_bytes; i++ ) |  |
| **dptc\_reserved\_constraint\_byte**[ i ] | u(8) |
| } |  |

* + - * 1. Displacement frame parameter set RBSP syntax

General displacement frame parameter set RBPS syntax

|  |  |
| --- | --- |
| displ\_frame\_parameter\_set\_rbsp( ) { | **Descriptor** |
| **dfps\_displ\_sequence\_parameter\_set\_id** | u(4) |
| **dfps\_displ\_frame\_parameter\_set\_id** | u(4) |
| displ\_information( ) |  |
| **dfps\_output\_flag\_present\_flag** | u(1) |
| **dfps\_num\_ref\_idx\_default\_active\_minus1** | ue(v) |
| **dfps\_additional\_lt\_dfoc\_lsb\_len** | ue(v) |
| **dfps\_overriden\_flag** | u(1) |
| if( dfps\_overriden\_flag ) { |  |
| **dfps\_subdivision\_enable\_flag** | u(1) |
| **dfps\_quantization\_parameters\_enable\_flag** | u(1) |
| } |  |
| if( dfps\_subdivision\_enable\_flag ) { |  |
| **dfps\_subdivision\_method** | u(3) |
| if( dfps\_subdivision\_method != 0 ) { |  |
| **dfps\_subdivision\_iteration\_count** | u(3) |
| DfpsSubdivisonCount = dfps\_subdivision\_iteration\_count |  |
| } else { |  |
| DfpsSubdivisonCount = 0 |  |
| } |  |
| } else |  |
| DfpsSubdivisonCount = DspsSubdivisionCount |  |
| if( dfps\_quantization\_parameters\_enable\_flag ) |  |
| displ\_quantization\_parameters( 1, DfpsSubdivisonCount ) |  |
| **dfps\_extension\_present\_flag** | u(1) |
| if( dfps\_extension\_present\_flag ) |  |
| **dfps\_extension\_8bits** | u(8) |
| if( dfps\_extension\_8bits ) |  |
| while( more\_rbsp\_data( ) ) |  |
| **dfps\_extension\_data\_flag** | u(1) |
| rbsp\_trailing\_bits( ) |  |
| } |  |

Displacement information

|  |  |
| --- | --- |
| displ\_information( ) { | **Descriptor** |
| **di\_use\_single\_displ\_flag** | u(1) |
| if(!di\_use\_single\_displ\_flag){ |  |
| **di\_num\_displs\_minus2** | ue(v) |
| NumDispls =di\_num\_displs\_minus2 + 2 |  |
| } |  |
| else |  |
| NumDispls = 1 |  |
| **di\_signalled\_displ\_id\_flag** | u(1) |
| if( di\_signalled\_displ\_id\_flag ) { |  |
| **di\_signalled\_displ\_id\_length\_minus1** | ue(v) |
| for( i = 0; < NumDispls; i++ ) |  |
| **di\_displ\_id**[ i ] | u(v) |
| DisplIDToIndex[ di\_displ\_id[ i ] ] = i |  |
| DisplIndexToID[ i ] = di\_displ\_id[ i ] |  |
| } |  |
| } |  |
| else |  |
| for( i = 0; i < NumDispls; i++ ) { |  |
| di\_displ\_id[ i ] = i |  |
| DisplIDToIndex[ i ] = i |  |
| DisplIndexToID[ i ] = i |  |
| } |  |
| } |  |

* + - * 1. Displacement reference list structure syntax

|  |  |
| --- | --- |
| displ\_ref\_list\_struct( rlsIdx ) { | **Descriptor** |
| **drl\_num\_ref\_entries**[ rlsIdx ] | ue(v) |
| for( i = 0; i < drl\_num\_ref\_entries[ rlsIdx ]; i++ ) { |  |
| if( dsps\_long\_term\_ref\_displ\_frames\_flag ) |  |
| **drl\_st\_ref\_displ\_frame\_flag**[ rlsIdx ][ i ] | u(1) |
| if( drl\_st\_ref\_displ\_frame\_flag[ rlsIdx ][ i ] ) { |  |
| **drl\_abs\_delta\_dfoc\_st**[ rlsIdx ][ i ] | ue(v) |
| if( drl\_abs\_delta\_dfoc\_st[ rlsIdx ][ i ] > 0 ) |  |
| **drl\_straf\_entry\_sign\_flag**[ rlsIdx ][ i ] | u(1) |
| } else |  |
| **drl\_dfoc\_lsb\_lt**[ rlsIdx ][ i ] | u(v) |
| } |  |
| } |  |

* + - * 1. Displacement layer RBSP syntax

|  |  |
| --- | --- |
| displ\_layer\_rbsp( ) { | **Descriptor** |
| displ\_header( ) |  |
| displ\_data\_unit( displ\_id ) |  |
| rbsp\_trailing\_bits( ) |  |
| } |  |

* + - * 1. Displacement header syntax

|  |  |
| --- | --- |
| displ\_header( ) { | **Descriptor** |
| if( nal\_unit\_type >= DNAL\_BLA\_W\_LP && nal\_unit\_type <= DNAL\_RSV\_IRAP\_DCL\_29 ) |  |
| **dh\_no\_output\_of\_prior\_displ\_frames\_flag** | u(1) |
| **dh\_frame\_parameter\_set\_id** | u(4) |
| **dh\_id** | u(v) |
| displID = dh\_id |  |
| **dh\_type** | ue(v) |
| if( dfps\_output\_flag\_present\_flag ) |  |
| **dh\_output\_flag** | u(1) |
| **dh\_frm\_order\_cnt\_lsb** | u(v) |
| if( dsps\_num\_ref\_displ\_frame\_lists\_in\_dsps > 0 ) |  |
| **dh\_ref\_displ\_frame\_list\_dsps\_flag** | u(1) |
| if( dh\_ref\_displ\_frame\_list\_dsps\_flag == 0 ) |  |
| displ\_ref\_list\_struct( dsps\_num\_ref\_displ\_frame\_lists\_in\_dsps ) |  |
| else if( dsps\_num\_ref\_displ\_frame\_lists\_in\_dsps > 1 ) |  |
| **ref\_displ\_frame\_list\_idx** | u(v) |
| for( j = 0; j < NumLtrDisplFrmEntries[ RlsIdx ]; j++ ) { |  |
| **dh\_additional\_dfoc\_lsb\_present\_flag**[ j ] | u(1) |
| if( additional\_dfoc\_lsb\_present\_flag[ j ] ) |  |
| **dh\_additional\_dfoc\_lsb\_val**[ j ] | u(v) |
| } |  |
| **dh\_parameters\_override\_flag** | u(1) |
| if( dh\_parameters\_override\_flag ){ |  |
| **dh\_subdivision\_override\_flag** | u(1) |
| **dh\_quantization\_override\_flag** | u(1) |
| } |  |
| if( dh\_subdivision\_override\_flag ){ |  |
| **dh\_subdivision\_method** | u(3) |
| if( dh\_subdivision\_method != 0 ){ |  |
| **dh\_subdivision\_iteration\_count** | u(3) |
| DhSubdivisionCount = dh\_subdivision\_iteration\_count |  |
| } else { |  |
| DhSubdivisionCount = 0 |  |
| } |  |
| } else { |  |
| DhSubdivisionCount = DfpsSubdivisonCount |  |
| } |  |
| if( dh\_quantization\_override\_flag ) |  |
| displ\_quantization\_parameters( 2, DhSubdivisionCount ) |  |
| if( dh\_type == P\_DISPLACEMENT && num\_ref\_entries[ RlsIdx ] > 1 ) { |  |
| **dh\_num\_ref\_idx\_active\_override\_flag** | u(1) |
| if( num\_ref\_idx\_active\_override\_flag ) |  |
| **dh\_num\_ref\_idx\_active\_minus1** | ue(v) |
| **dh\_layer\_inter\_flag** | u(1) |
| if( !dh\_layer\_inter\_flag ) |  |
| for( i=0 ; i < DhSubdivisionCount + 1; i++ ) |  |
| **dh\_lod\_inter\_flag**[ i ] | u(1) |
| } |  |
| byte\_alignment( ) |  |
| } |  |

* + - * 1. Displacement data unit syntax

|  |  |
| --- | --- |
| displ\_data\_unit( displID ) { | **Descriptor** |
| **ddu\_lod\_count**[ displID ] | ae(v) |
| for( i = 0; i < ddu\_lod\_count; i++ ){ |  |
| **ddu\_vertex\_count\_lod**[ displID ][ i ] | ae(v) |
| **ddu\_lod\_split\_flag**[ displID ][ i ] | ae(1) |
| **ddu\_num\_subblock\_lod**[ displID ][ i ] | u(8) |
| totalVertCount += ddu\_vertex\_count\_lod[ displID ][ i ] |  |
| } |  |
| for ( k = 0; k < MaxDimension; k++ ){ |  |
| for( level = 0; level < ddu\_lod\_count; level++ ){ |  |
| levelBlockSize[ displID ][ level ] = 1 <<  log2 ( vertCount[ displID ][ level ] / ( ddu\_lod\_split\_flag[ displID ][ level ] ?   ddu\_num\_subblock\_lod[ displID ][ level ] : 1 ) ) |  |
| for ( block = 0; block < ddu\_num\_subblock\_lod[ level ]; block ++ ){ |  |
| **ddu\_nz\_subBlock**[ displID ][ k ][ block ] | ae(v) |
| if ( dpdu\_nz\_subBlock[ displID ][ k ][ block ]   && ddu\_lod\_split\_flag[ displID ][ level ]){ |  |
| vStart[ level ] =  ( level == 0? 0 : vStart[ level – 1 ] + vertCount[ level – 1 ]) |  |
| vBlockStart[ level ][ block ] =  vStart[ level ] + block \* levelBlockSize[ level ] |  |
| vBlockEnd[ level ][ block ] =  min( vBlockStart[ level ][ block ] +  levelBlockSize[ level ], totalVertCount ) |  |
| for( v= vBlockStart[ level ][ block ];v < vBlockEnd[ level ][ block ]; v++ ){ |  |
| **ddu\_coeff\_abs\_level\_gt0**[ displID ][ v ][ k ] | ae(v) |
| if( ddu\_coeff\_abs\_level\_gt0[ displID ][ v ][ k ] ) { |  |
| **ddu\_coeff\_abs\_level\_gt1**[ displID ][ v ][ k ] | ae(v) |
| **ddu\_coeff\_sign**[ displID ][ v ][ k ] | u(1) |
| if( ddu\_coeff\_abs\_level\_gt1[ displID ][ v ][ k ] ) { |  |
| **ddu\_coeff\_abs\_level\_gt2**[ displID ][ v ][ k ] | ae(v) |
| if( ddu\_coeff\_abs\_level\_gt2[ displID ][ v ][ k ] ) { |  |
| **ddu\_coeff\_abs\_level\_gt0** [ displID ][ v ][ k ] | ae(v) |
| if( ddu\_coeff\_abs\_level\_gt3[ displID ][ v ][ k ]) { |  |
| **ddu\_coeff\_abs\_level\_rem**[ displID ][ v ][ k ] | ae(v) |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |
| } |  |

* + 1. Semantics
       1. NAL unit semantics
          1. General NAL unit semantics

**NumBytesInNalUnit** specifies the size of the NAL unit in bytes. This value is required for decoding of the NAL unit. Some form of demarcation of NAL unit boundaries is necessary to enable inference of NumBytesInNalUnit. One such demarcation method is specified in Annex D for the sample stream format. Other methods of demarcation can be specified outside this document.

NOTE 1 – The displacement coding layer (DCL) is specified to efficiently represent the content of the displacement data. The NAL is specified to format that data and provide header information in a manner appropriate for conveyance on a variety of communication channels or storage media. All data are contained in NAL units, each of which contains an integer number of bytes. A NAL unit specifies a generic format for use in both packet-oriented and bitstream systems. The format of NAL units for both packet-oriented transport and sample streams is identical except that in the sample stream format specified in Annex TBD each NAL unit can be preceded by an additional element that specifies the size of the NAL unit.

**rbsp\_byte**[ i ] is the i-th byte of an RBSP. An RBSP is specified as an ordered sequence of bytes as follows:

The RBSP contains a string of data bits**(**SODB) as follows:

* If the SODB is empty (i.e., zero bits in length), the RBSP is also empty.
* Otherwise, the RBSP contains the SODB as follows:
  + The first byte of the RBSP contains the first (most significant, left-most) eight bits of the SODB; the next byte of the RBSP contains the next eight bits of the SODB, etc., until fewer than eight bits of the SODB remain.
  + The rbsp\_trailing\_bits( ) syntax structure is present after the SODB as follows:
    - The first (most significant, left-most) bits of the final RBSP byte contain the remaining bits of the SODB (if any).
    - The next bit consists of a single bit equal to 1 (i.e., rbsp\_stop\_one\_bit).
    - When the rbsp\_stop\_one\_bit is not the last bit of a byte-aligned byte, one or more bits equal to 0 (i.e. instances of rbsp\_alignment\_zero\_bit) are present to result in byte alignment.

Syntax structures having these RBSP properties are denoted in the syntax tables using an "\_rbsp" suffix. These structures are carried within NAL units as the content of the rbsp\_byte[ i ] data bytes. The association of the RBSP syntax structures to the NAL units is as specified in Table J-1 .

NOTE 2 – When the boundaries of the RBSP are known, the decoder can extract the SODB from the RBSP by concatenating the bits of the bytes of the RBSP and discarding the rbsp\_stop\_one\_bit, which is the last (least significant, right-most) bit equal to 1, and discarding any following (less significant, farther to the right) bits that follow it, which are equal to 0. The data necessary for the decoding process is contained in the SODB part of the RBSP.

* + - * 1. NAL unit header semantics

**displ\_nal\_forbidden\_zero\_bit** shall be equal to 0

**displ\_nal\_unit\_type** specifies the type of the RBSP data structure contained in the NAL unit as specified in Table J-1.

Table J-1 – NAL unit type codes and NAL unit type classes

|  |  |  |  |
| --- | --- | --- | --- |
| displ\_nal\_unit\_type | Name of displ\_nal\_unit\_type | Content of displacement NAL unit and RBSP syntax structure | NAL unitype class |
| 0 1 | DNAL\_TRAIL\_N DNAL\_TRAIL\_R | Coded displacement of a non-TSA, non STSA trailing displacement frame displ\_layer\_rbsp( ) | DCL |
| 2 3 | DNAL\_TSA\_N DNAL\_TSA\_R | Coded displacement of a TSA displacement frame displ\_layer\_rbsp( ) | DCL |
| 4 5 | DNAL\_STSA\_N DNAL\_STSA\_R | Coded displacement of a STSA displacement frame displ\_layer\_rbsp( ) | DCL |
| 6 7 | DNAL\_RADL\_N DNAL\_RADL\_R | Coded displacement of a RADL displacement frame displ\_layer\_rbsp( ) | DCL |
| 8 9 | DNAL\_RASL\_N DNAL\_RASL\_R | Coded displacement of a RASL displacement frame displ\_layer\_rbsp( ) | DCL |
| 10 11 | DNAL\_SKIP\_N DNAL\_SKIP\_R | Coded displacement of a skipped displacement frame displ\_layer\_rbsp( ) | DCL |
| 12 14 | DNAL\_RSV\_DCL\_N12 DNAL\_RSV\_DCL\_N14 | Reserved non-IRAP sub-layer non-reference DCL displacement NAL unit types | DCL |
| 13 15 | DNAL\_RSV\_DCL\_R13 DNAL\_RSV\_DCL\_R15 | Reserved non-IRAP sub-layer reference DCL displacement NAL unit types | DCL |
| 16 17 18 | DNAL\_BLA\_W\_LP DNAL\_BLA\_W\_RADL DNAL\_BLA\_N\_LP | Coded displacement of a BLA displacement frame  displ\_layer\_rbsp( ) | DCL |
| 19 20 | DNAL\_IDR\_W\_RADL DNAL\_IDR\_N\_LP | Coded displacement of an IDR displacement frame  displ\_layer\_rbsp( ) | DCL |
| 21 | DNAL\_CRA | Coded displacement of a CRA displacement frame  displ\_layer\_rbsp( ) | DCL |
| 22 23 | DNAL\_RSV\_IRAP\_DCL\_22 DNAL\_RSV\_IRAP\_DCL\_23 | Reserved IRAP DCL NAL unit types | DCL |
| 24..29 | DNAL\_RSV\_DCL\_24.. DNAL\_RSV\_DCL\_29 | Reserved non-IRAP DCL NAL unit types | DCL |
| 30 | DNAL\_DSPS | Displacement sequence parameter set displ\_sequence\_parameter\_set\_rbsp( ) | non-DCL |
| 31 | DNAL\_DFPS | Displacement frame parameter set displ\_frame\_parameter\_set\_rbsp( ) | non-DCL |
| 32 | DNAL\_DAUD | Access unit delimiter access\_unit\_delimiter\_rbsp( ) | non-DCL |
| 33 | DNAL\_DEOS | End of sequence end\_of\_sequence\_rbsp( ) | non-DCL |
| 34 | DNAL\_DEOB | End of bitstream end\_of\_displ\_sub\_bitstream\_rbsp( ) | non-DCL |
| 35 | DNAL\_FD | Filler filler\_data\_rbsp( ) | non-DCL |
| 36 37 | DNAL\_PREFIX\_NSEI  DNAL\_SUFFIX\_NSEI | Non-essential supplemental enhancement information sei\_rbsp( ) | non-DCL |
| 38 39 | DNAL\_PREFIX\_ESEI DNAL\_SUFFIX\_ESEI | Essential supplemental enhancement information sei\_rbsp( ) | non-DCL |
| 40..44 | DNAL\_RSV\_NDCL\_40 DNAL\_RSV\_NDCL\_44 | Reserved non-DCL NAL unit types | non-DCL |
| 45..63 | DNAL\_UNSPEC\_45 DNAL\_UNSPEC\_63 | Unspecified non-DCL NAL unit types | non-DCL |

**displ\_nal\_layer\_id** specifies the identifier of the layer to which an DCL NAL unit belongs or the identifier of a layer to which a non-DCL NAL unit applies. The value of displ\_nal\_layer\_id shall be in the range of 0 to 62, inclusive. The value of 63 may be specified in the future by ISO/IEC. For purposes other than determining the amount of data in the decoding units of the bitstream, decoders shall ignore all data that follow the value 63 for disp\_nal\_layer\_id in a NAL unit, and decoders conforming to a profile specified in Clause J.10 shall ignore (i.e., remove from the bitstream and discard) all NAL units with values of displ\_nal\_layer\_id not equal to 0.

**displ\_nal\_temporal\_id\_plus1** minus 1 specifies a temporal identifier for the NAL unit. The value of nal\_temporal\_id\_plus1 shall not be equal to 0.

* + - * 1. Order of NAL units and displacement frames, and association to coded displacement frames, access units, and coded displacement sequences

[Ed. Note : Input required]

* + - 1. Raw byte sequence payloads, trailing bits, and byte alignment semantics
         1. Displacement sequence parameter set RBSP semantics

General displacement sequence parameter set RBSP semantics

**dsps\_sequence\_parameter\_set\_**id provides an identifier for the displacement sequence parameter set for reference by other syntax elements.

**dsps\_codec\_id** indicates the identifier of the codec used to compress the displacement. dsps\_codec\_id shall be in the range of 0 to 255, inclusive. This codec may be identified through the profiles defined in Annex J.10, a component codec mapping SEI message, or through means outside this document. It may be associated with a specific displacement codec through the profiles specified in the corresponding specification, or could be explicitly indicated with an SEI message as is done in the V3C specification for the video sub-bitstreams.

**dsps\_single\_dimension\_flag** indicates the number of dimensions for the displacements associated with the displacements. dsps\_single\_dimension\_flag equal to 0 indicates three components for the displacements are used. dsps\_single\_dimension\_flag equal to 1 indicates only normal component for the displacements is used. The variable MaxDimension is derived as follows:

MaxDimension = dsps\_single\_dimension\_flag ? 1 : 3

**dsps\_msb\_align\_flag** indicates how the decoded displacement samples are converted to samples at the displacement range bit depth as specified in B.2.8.

**dsps\_log2\_max\_displ\_frame\_order\_cnt\_lsb\_minus4** plus 4 specifies the values of the variables Log2MaxDisplFrmOrderCntLsb and MaxDisplFrmOrderCntLsb that are used in the decoding process for the displacement frame order count as follows:

Log2MaxDisplFrmOrderCntLsb =   
 dsps\_log2\_max\_displ\_frame\_order\_cnt\_lsb\_minus4 + 4

MaxDisplFrmOrderCntLsb = 2Log2MaxDisplFrmOrderCntLsb

The value of dsps\_log2\_max\_displ\_frame\_order\_cnt\_lsb\_minus4 shall be in the range of 0 to 12, inclusive.

**dsps\_max\_dec\_displ\_frame\_buffering\_minus1** plus 1 specifies the maximum required size of the decoded displacement frame buffer for the CDS in units of displacement frame storage buffers. The value of dsps\_max\_dec\_displ\_frame\_buffering\_minus1 shall be in the range of 0 to 15, inclusive.

**dsps\_max\_num\_reorder\_frames** specifies the maximum allowed number of frames that can precede any frame in in decoding order and follow that frame in output order for the CDS. The value of dsps\_max\_num\_reorder\_frames shall be in the range of 0 to bmsps\_max\_dec\_mesh\_frame\_buffering\_minus1, inclusive.

**dsps\_max\_latency\_increase\_plus1** not equal to 0 is used to compute the value of MaxLatencyDisplacementFrames, which specifies the maximum number of frames that can precede any frame in output order and follow that picture in decoding order for the CDS.

When dsps\_max\_latency\_increase\_plus1 is not equal to 0, the value of MaxLatencyDisplacementFrames is specified as follows:

MaxLatencyDisplacementFrames =  
 dsps\_max\_num\_reorder\_frames + dsps\_max\_latency\_increase\_plus1 − 1

When dsps\_max\_latency\_increase\_plus1 is equal to 0, no corresponding limit is expressed. The value of dsps\_max\_latency\_increase\_plus1 shall be in the range of 0 to 2^32 − 2, inclusive.

**dsps\_long\_term\_ref\_displ\_frames\_flag** equal to 0 specifies that no long-term reference displacement is used for inter prediction of any coded displacement frame in the CDS. dsps\_long\_term\_ref\_displ\_frames\_flag equal to 1 specifies that long term reference displacement frames may be used for inter prediction of one or more coded displacement frames in the CDS.

**dsps\_num\_ref\_displ\_frame\_lists\_in\_dsps** specifies the number of the displ\_ref\_list\_struct( rlsIdx ) syntax structures included in the displacement sequence parameter set. The value of dsps\_num\_ref\_displ\_frame\_lists\_in\_dsps shall be in the range of 0 to 64, inclusive.

NOTE – A decoder allocates memory for a total number of displ\_ref\_list\_struct( rlsIdx ) syntax structures equal to (dsps\_num\_ref\_displ\_frame\_lists\_in\_dsps + 1) since there can be one displ\_ref\_list\_struct( rlsIdx ) syntax structure directly signalled in the a displacement headers of the current displacement frame.

**dsps\_geometry\_3d\_bit\_depth\_minus1** plus 1 indicates the bit depth of the geometry coordinates of the reconstructed volumetric content. dsps\_geometry\_3d\_bit\_depth\_minus1 shall be in the range of 0 to 31, inclusive.

**dsps\_subdivision\_method** indicates the identifier of the methodto subdivide the meshes associated with the current dsps sequence parameter set. Table 2 describes the list of supported subdivision methods and their relationship with dsps\_subdivision\_method.

**dsps\_subdivision\_iteration\_count** indicates the number of iterations used for the subdivision. When not present the value of dsps\_subdivision\_iteration\_count is inferred to be equal to 0.

**dsps\_displacement\_reference\_qp** specifies the initial value of QuantizationParameter for current frame. When not present dsps\_displacement\_reference\_qp is set to be equal to 49.

**dsps\_extension\_present\_flag** equal to 1 specifies that dsps\_extension\_count is present in the displacement sequence parameter set.

**dsps\_extension\_count** specifies the number of extensions present in the current displacement sequence parameter set. When not present, dsps\_extension\_count is inferred to be equal to 0.

**dsps\_extensions\_length\_minus1** when present, specifies the cumulative length in bytes, DspsExtensionsLength, for all extensions that follow this syntax element. DspsExtensionsLength is computed as follows:

if( dsps\_extension\_count == 0 )  
 DspsExtensionsLength = 0  
 else  
 DspsExtensionsLength = dsps\_extensions\_length\_minus1 + 1

It is a requirement, when dsps\_extension\_count is not equal to 0, that DspsExtensionsLength is equal to 3 \* dsps\_extension\_count plus the sum of all dsps\_extension\_length[ i ].

**dsps\_extension\_type**[ i ] indicates the DSPS extension type for the extension with index i as specified in [CrossReferenceXXX]. Values indicated as reserved are reserved for future use by ISO/IEC and shall not be present in bitstreams conforming to this version of this document. Decoders conforming to this version of this document should ignore such reserved extensions. It is a requirement that a particular dsps\_extension\_type[ i ] value shall only be present once in an entire DSPS, while the order of extensions does not matter.

**dsps\_extension\_length**[ i ] specifies the number of bytes used to represent the payload size of the syntax structure of the associated extension with index i. If dsps\_extension\_length[ i ] is equal to 0, no extension payload is present for the extension with index i. Otherwise, the extension with index i shall have a payload size in bits in the range of 8 \* ( dsps\_extension\_length[ i ] – 1 ) + 1 to 8 \* dsps\_extension\_length[ i ], inclusive.

Displacement SPS extension semantics

**dsps\_extension\_data\_byte** may have any value.

Quantization parameters syntax

**dqp\_lod\_quantization\_flag[** qpIndex ] equal to 1 indicates that the quantization parameter will be sent per level-of-detail using delta coding. dqp\_lod\_quantization\_flag[ qpIndex ] equal to 0 indicates that the quantization parameter will be the same for all level-of-details. qpIndex is the index of the quantization parameter set.

**dqp\_bitdepth\_offset**[ qpIndex ] indicates the bit depth offset value applied to the quantization process of the displacements. qpIndex is the index of the quantization parameter set.

**dqp\_quantization\_parameters**[ qpIndex ][ k ] indicates the quantization parameter to be used for the inverse quantization of the kth-component of the displacements. The value of dqp\_quantization\_parameters[ qpIndex ][ k ] shall be in the range of 0 to 100, inclusive. qpIndex is the index of the quantization parameter set.

**dqp\_log2\_lod\_inverse\_scale**[ qpIndex ][ k ] indicates the scaling factor applied to the kth-component of the displacements for each level of detail. qpIndex is the index of the quantization parameter set.

**dqp\_lod\_delta\_quantization\_parameter\_value**[ qpIndex ][ i ][ k ] specifies the absolute difference of quantization parameter value between the value dsps\_displacement\_reference\_qp and the quantization parameter for the ith-layer and kth-component. When not present, the value of dqp\_lod\_delta\_quantization\_parameter\_value[ qpIndex ][ i ][ k ] is inferred as 0. qpIndex is the index of the quantization parameter set. The value of QuantizationParameter of each LoD layer shall be in the range of 0 to 100.

**dqp\_lod\_delta\_quantization\_parameter\_sign**[ qpIndex ][ i ][ k ] specifies the sign of difference of quantization parameter value between the value dsps\_displacement\_reference\_qp and the quantization parameter for the ith-layer and kth-component. dqp\_lod\_delta\_quantization\_parameter\_sign[ qpIndex ][ i ][ k ] equal to 0 indicate the difference is positive. dqp\_lod\_delta\_quantization\_parameter\_sign[ qpIndex ][ i ][ k ] equal to 1 indicate the difference is negative. When not present, the value of dqp\_lod\_delta\_quantization\_parameter\_sign[ qpIndex ][ i ][ k ] is inferred as 0. qpIndex is the index of the quantization parameter set.

Displacement profile, tier, and level semantics

**dptl\_tier\_flag** specifies the tier context for the interpretation of dptl\_level\_idc as specified in Annex J.10.

**dptl\_profile\_codec\_group\_idc** indicates the codec group profile component to which the CDS conforms as specified in Annex J.10. Bitstreams shall not contain values of dptl\_profile\_codec\_group\_idc other than those specified in Annex J.10. Other values of dptl\_profile\_codec\_group\_idc are reserved for future use by ISO/IEC.

**dptl\_profile\_toolset\_idc** indicates the toolset combination profile component to which the CDS conforms as specified in Annex J.10. Bitstreams shall not contain values of dptl\_profile\_toolset\_idc other than those specified in Annex J.10. Other values of dptl\_profile\_toolset\_idc are reserved for future use by ISO/IEC.

**dptl\_reserved\_zero\_32bits**, when present, shall be equal to 0 in bitstreams conforming to this version of this document. Other values for dptl\_reserved\_zero\_32bits are reserved for future use by ISO/IEC. Decoders shall ignore the value of dptl\_reserved\_zero\_32bits.

**dptl\_level\_idc** indicates a level to which the CDS conforms as specified in Annex J.10. Bitstreams shall not contain values of dptl\_level\_idc other than those specified in Annex J.10. Other values of dptl\_level\_idc are reserved for future use by ISO/IEC.

**dptl\_num\_sub\_profiles** indicates the number of the dptl\_sub\_profile\_idc[ i ] syntax elements.

**dptl\_extended\_sub\_profile\_flag** equal to 1 specifies that the dptl\_sub\_profile\_idc[ i ] syntax elements, if present, should be represented using 64 bits. dptl\_extended\_sub\_profile\_flag equal to 0 specifies that the dptl\_sub\_profile\_idc[ i ] syntax elements, if present, should be represented using 32 bits.

**dptl\_sub\_profile\_idc**[ i ] indicates the i-th interoperability metadata registered as specified by Rec. ITU-T T.35, the content of which is not specified in this document. The number of bits used to represent dptl\_sub\_profile\_idc[ i ] is equal to (dptl\_extended\_sub\_profile\_flag == 0 ? 32 : 64).

**dptl\_toolset\_constraints\_present\_flag** equal to 1 specifies that an additional structure, dptl\_profile\_toolset\_constraints\_information( ), is present in the bitstream. dptl\_toolset\_constraints\_present\_flag equal to 0 specifies that the structure dptl\_profile\_toolset\_constraints\_information( ) is not present.

Displacement profile toolset constraints information semantics

**dptc\_one\_displacemnt\_frame\_only\_flag,** when present, has semantics specified in Annex J.10. where the profile indicated by dptl\_profile\_toolset\_idc is a profile specified in Annex J.10. When not present, dptc\_one\_displacement\_frame\_only\_flag is inferred to be equal to 0.

**dptc\_reserved\_zero\_7bits** shall be equal to 0 in bitstreams conforming to this version of this document. Other values of dptc\_reserved\_zero\_7bits are reserved for future use by ISO/IEC and shall not be present in bitstreams conforming to this version of this document. Decoders conforming to this version of this document shall ignore values of dptc\_reserved\_zero\_7bitsother than 0.

**dptc\_num\_reserved\_constraint\_bytes** specifies the number of the reserved constraint bytes. The value of dptc\_num\_reserved\_constraint\_bytes shall be 0 in bitstreams conforming to this version of this document. Other values of dptc\_num\_reserved\_constraint\_bytes are reserved for future use by ISO/IEC and shall not be present in bitstreams conforming to this version of this document. Decoders conforming to this version of this document shall ignore values of dptc\_num\_reserved\_constraint\_bytes other than 0.

**dptc\_reserved\_constraint\_byte**[ i ] may have any value. Its presence and value do not affect decoder conformance to profiles specified in this version of this document. Decoders conforming to this version of this document shall ignore the values of all the dptc\_reserved\_constraint\_byte[ i ] syntax elements.

* + - * 1. Displacement frame parameter set RBSP semantics

General displacement frame parameter set RBSP semantics

**dfps\_displ\_sequence\_parameter\_set\_id** specifies the value of dsps\_sequence\_parameter\_set\_id for the active displacement sequence parameter set.

**dfps\_displ\_parameter\_set\_id** identifies the displacement frame parameter set for reference by other syntax elements.

**dfps\_output\_flag\_present\_flag** equal to 1 indicates that the displ\_output\_flag syntax element is present in the associated displacement headers. dfps\_output\_flag\_present\_flag equal to 0 indicates that the displ\_output\_flag syntax element is not present in the associated displacement headers.

**dfps\_num\_ref\_idx\_default\_active\_minus1** plus 1 specifies the inferred value of the variable NumRefIdxActive for the tile with displ\_num\_ref\_idx\_active\_override\_flag equal to 0. The value of dfps\_num\_ref\_idx\_default\_active\_minus1 shall be in the range of 0 to 14, inclusive.

**dfps\_additional\_lt\_dfoc\_lsb\_len** specifies the value of the variable MaxLtDisplFrmOrderCntLsb that is used in the decoding process for reference atlas frame lists as follows:

MaxLtDisplFrmOrderCntLsb =  
 2 \* ( Log2MaxDisplFrmOrderCntLsb + dfps\_additional\_lt\_dfoc\_lsb\_len)

The value of dfps\_additional\_lt\_dfoc\_lsb\_len shall be in the range of 0 to 32 – Log2MaxDisplFrmOrderCntLsb, inclusive.

When dsps\_long\_term\_ref\_displ\_frames\_flag is equal to 0, the value of dfps\_additional\_lt\_dfoc\_lsb\_len shall be equal to 0.

**dfps\_overriden\_flag** equal to 1 indicates the parameters dfps\_subdivision\_enable\_flag and dfps\_quantization\_parameters\_enable\_flag are present in the displacement frame parameter set extension.

**dfps\_subdivision\_enable\_flag** equal to 1 indicates dfps\_subdivision\_method and dfps\_subdivision\_iteration\_count are present in the displacement frame parameter set extension. When dfps\_subdivision\_enable\_flag is not present, its value is inferred to be equal to 0.

**dfps\_quantization\_parameters\_enable\_flag** equal to 1 indicates displ\_quantization\_parameters(qpIndex, subdivisionCount ) syntax structure is present in the displacement frame parameter set extension. dfps\_quantization\_parameters\_enable\_flag equal to 0 indicates displ\_quantization\_parameters(qpIndex, subdivisionCount ) syntax structure is not present in the displacement frame parameter set extension. When dfps\_quantization\_parameters\_enable\_flag is not present, its value is inferred to be equal to 0.

**dfps\_subdivision\_method** indicates the identifier of the method to subdivide the meshes associated with the current displacement frame parameter set. When dfps\_subdivision\_method is not present, dfps\_subdivision\_method is equal to dsps\_subdivision\_method. Table 2 describes the list of supported subdivision methods and their relationship with dfps\_subdivision\_method.

**dfps\_subdivision\_iteration\_count** indicates the number of iterations used for the subdivision. When dfps\_subdivision\_method equal to 0 and dfps\_subdivision\_iteration\_count is not present, dfps\_subdivision\_iteration\_count is equal to 0. When dfps\_subdivision\_method is not 0 and dfps\_subdivision\_iteration\_count is not present dfps\_subdivision\_iteration\_count is equal to dsps\_subdivision\_iteration\_count.

**dfps\_extension\_present\_flag** equal to 1 specifies that the syntax element dfps\_extension\_8bits is present in the displacement frame parameter set. dfps\_extension\_present\_flag equal to 0 specifies that the syntax element dfps\_extension\_8bits is not present. The value of dfps\_extension\_present\_flag shall be 0 in this version of this document

**dfps\_extension\_8bits** equal to 0 specifies that no dfps\_extension\_data\_flag syntax elements are present in the DFPS RBSP syntax structure. When present, dfps\_extension\_8bits shall be equal to 0 in bitstreams conforming to this version of this document. Values of dfps\_extension\_8bits not equal to 0 are reserved for future use by ISO/IEC. Decoders shall allow the value of dfps\_extension\_8bits to be not equal to 0 and shall ignore all dfps\_extension\_data\_flag syntax elements in an DFPS NAL unit. When not present, the value of dfps\_extension\_8bits is inferred to be equal to 0.

**dfps\_extension\_data\_flag** may have any value. Its presence and value do not affect decoder conformance to profiles specified in this version of this document. Decoders conforming to this version of this document shall ignore all dfps\_extension\_data\_flag syntax elements.

Displacement information semantics

**di\_use\_single\_displ\_flag** equal to 1 specifies that there is only one displacement in each displacement frame referring to the DFPS. di\_use\_single\_mesh\_falg equal to 0 specifies that there shall be more than one displacement in each displacement frame referring to the DFPS.

**di\_num\_displs\_minus2** plus 2 specifies the number of displacements in each displacement frame referring to the DFPS. The value of di\_num\_displs\_minus1shall be in the range of 0 to 62, inclusive.

**di\_signalled\_displ\_id\_flag** equal to 1 specifies that the displacement ID for each displacement frame is signalled. di\_signalled\_displ\_id\_flag equal to 0 specifies that displacement IDs are not signalled.

**di\_signalled\_displ\_id\_length\_minus1** plus 1 specifies the number of bits used to represent the syntax element di\_submesh\_id[ i ] when present, and the syntax element displ\_id in a displacement header. The value of di\_signalled\_displ\_id\_length\_minus1 shall be in the range of 0 to 15, inclusive. When not present, its value is inferred to be equal to Ceil( Log2( NumDispls ) ) – 1.

**di\_displ\_id**[ i ] specifies the displacement ID of the i-th displacement. The length of the di\_submesh\_id[ i ] syntax element is di\_signalled\_displ\_id\_length\_minus1 + 1 bits. When not present, the value of di\_displ\_id[ i ] is inferred to be equal to i, for each i in the range of 0 to NumDispls - 1, inclusive. It is a requirement of bitstream conformance that di\_displ\_id[ i ] shall not be equal to di\_displ\_id[ j ] for all i != j. The length of the di\_displ\_id[ i ] syntax element is di\_signalled\_displ\_id\_length\_minus1 + 1 bits.

The variable FirstDisplID is computed as follows:

FirstDisplID = di\_displ\_id[ 0 ]  
 for ( i = 1; i < NumDispls; i++ )  
 FirstDispID = Min(FirstDisplID, di\_displ\_id[ i ])

* + - * 1. Displacement reference list structure semantics

**drl\_num\_ref\_entries**[ rlsIdx ] specifies the number of entries in the displ\_ref\_list\_struct( rlsIdx ) syntax structure, where rlsIdx is the index of a displacement frame reference list. For P\_DISPLACEMENT, the value of num\_ref\_entries[ rlsIdx ] shall be in the range of 1 to dsps\_max\_dec\_displ\_frame\_buffering\_minus1 + 1. Otherwise, the value of num\_ref\_entries[ rlsIdx ] shall be in the range of 0 to dsps\_max\_dec\_displ\_frame\_buffering\_minus1 + 1.

**drl\_st\_ref\_displ\_frame\_flag**[ rlsIdx ][ i ] equal to 1 specifies that the i-th entry in the displ\_ref\_list\_struct( rlsIdx ) syntax structure is a short term reference displacement frame entry. st\_ref\_displ\_frame\_flag[ rlsIdx ][ i ] equal to 0 specifies that the i-th entry in the displ\_ref\_list\_struct( rlsIdx ) syntax structure is a long term reference displacement frame entry. When not present, the value of drl\_st\_ref\_displ\_frame\_flag[ rlsIdx ][ i ] is inferred to be equal to 1.

The variable NumLtrDisplFrmEntries[ rlsIdx ] is derived as follows:

NumLtrDisplFrmEntries[ rlsIdx ] = 0  
 for( i = 0; i < drl\_num\_ref\_entries[ rlsIdx ]; i++ )  
 if( !drl\_st\_ref\_displ\_frame\_flag[ rlsIdx ][ i ] )   
 NumLtrDisplFrmEntries[ rlsIdx ]++

**drl\_abs\_delta\_dfoc\_st**[ rlsIdx ][ i ], when the i-th entry is the first short term reference displacement frame entry in displ\_ref\_list\_struct( rlsIdx ) syntax structure, specifies the absolute difference between the displacement frame order count values of the current displacement frame referred to by the i-th entry, or, when the i-th entry is a short term reference displacement frame entry but not the first short term reference displacement frame entry in the displ\_ref\_list\_struct( rlsIdx ) syntax structure, specifies the absolute difference between the displacement frame order count values of the displacement frames referred to by the i-th entry and by the previous short term reference displacement frame entry in the displ\_ref\_list\_struct( rlsIdx ) syntax structure.

The value of drl\_abs\_delta\_dfoc\_st[ rlsIdx ][ i ] shall be in the range of 0 to 215 - 1, inclusive.

**drl\_straf\_entry\_sign\_flag**[ rlsIdx ][ i ] equal to 1 specifies that the i-th entry in the syntax structure displ\_ref\_list\_struct( rlsIdx ) has a value greater than or equal to 0. drl\_straf\_entry\_sign\_flag[ rlsIdx ][ i ] equal to 0 specifies that the i-th entry in the syntax structure displ\_ref\_list\_struct( rlsIdx ) has a value less than 0. When not present, the value of drl\_straf\_entry\_sign\_flag[ rlsIdx ][ i ] is inferred to be equal to 1.

The list DeltaDfocSt[ rlsIdx ][ i ] is derived as follows:

for( i = 0; i < drl\_num\_ref\_entries[ rlsIdx ]; i++ )  
 if( drl\_st\_ref\_displ\_frame\_flag[ rlsIdx ][ i ] )  
 DeltaDfocSt[ rlsIdx ][ i ] =  
 ( 2 \* drl\_straf\_entry\_sign\_flag[ rlsIdx ][ i ] – 1 ) \*  
 drl\_abs\_delta\_dfoc\_st[ rlsIdx ][ i ]  
 else  
 DeltaDfocSt[ rlsIdx ][ i ] = 0

**drl\_dfoc\_lsb\_lt**[ rlsIdx ][ i ] specifies the value of the displacement frame order count modulo MaxDisplFrmOrderCntLsb of the displacement frame referred to by the i-th entry in the displ\_ref\_list\_struct( rlsIdx ) syntax structure. The length of the drl\_dfoc\_lsb\_lt[ rlsIdx ][ i ] syntax element is Log2MaxDisplFrmOrderCntLsb bits.

* + - * 1. Displacement layer RBSP semantics

None

* + - * 1. Displacement header semantics

**dh\_no\_output\_of\_prior\_displ\_frames\_flag** affects the output of previously-decoded displacement frames in the DDB after the decoding of a displacement frame in a CDS AU that is not the first AU in the bitstream as specified in J.11. When no\_output\_of\_prior\_displ\_frames\_flag is not present, its value is inferred to be equal to 0.

It is a requirement of bitstream conformance that the value of no\_output\_of\_prior\_displ\_frames\_flag shall be the same for all displacement frames in an AU.

The value of no\_output\_of\_prior\_displ\_frames\_flag in the displacement headers is also referred to as the output\_of\_prior\_displ\_frames\_flag value of the AU.

**dh\_frame\_parameter\_set\_id** specifies the value of dfps\_displ\_frame\_parameter\_set\_id for the active displacement frame parameter set for the current displacement frame.

**dh\_id** specifies the current displacement frame ID. When not present, the value of dh\_id is inferred to be equal to 0.

**dh\_type** specifies the coding type of the current displacement frame according to Table J-2. The value of dh\_type shall be equal to 0 or 1 in bitstreams conforming to this version of this document. Other values of dh\_type are reserved for future use by ISO/IEC. Decoders conforming to this version of this document shall ignore reserved values of dh\_type.

Table J-2 – Name association to dh\_type

|  |  |
| --- | --- |
| bmsh\_type | Name of bmsh\_type |
| 0 | P\_DISPLACEMENT |
| 1 | I\_DISPLACEMENT |
| 2 - .. | RESERVED |

**dh\_frm\_order\_cnt\_lsb** specifies the displacement frame order count modulo MaxDisplFrmOrderCntLsb for the current displacement frame. The length of the dh\_frm\_order\_cnt\_lsb syntax element is equal to Log2MaxDisplFrmOrderCntLsb bits. The value of the dh\_frm\_order\_cnt\_lsb shall be in the range of 0 to MaxDisplFrmOrderCntLsb - 1, inclusive.

**dh\_ref\_displ\_frame\_list\_dsps\_flag** equal to 1 specifies that the reference displacement frame list of the current displacement frame is derived based on one of the displ\_ref\_list\_struct( rlsIdx ) syntax structures in the active DSPS. dh\_ref\_displ\_frame\_list\_dsps\_flag equal to 0 specifies that the reference displacement frame list of the current displacement frame is derived based on the displ\_ref\_list\_struct( rlsIdx ) syntax structure that is directly included in the displacement frame header of the current displacement frame. When dsps\_num\_ref\_displ\_frame\_lists\_in\_dsps is equal to 0, the value of dh\_ref\_displ\_frame\_list\_dsps\_flag is inferred to be equal to 0.

**dh\_ref\_displ\_frame\_list\_idx** specifies the index, into the list of the displ\_ref\_list\_struct( rlsIdx ) syntax structures included in the active DSPS, of the displ\_ref\_list\_struct( rlsIdx ) syntax structure that is used for derivation of the reference displacement frame list for the current displacement frame. The syntax element dh\_ref\_displ\_frame\_list\_idx is represented by Ceil( Log2( dsps\_num\_ref\_displ\_frame\_lists\_in\_dsps ) ) bits. When not present, the value of dh\_ref\_displ\_frame\_list\_idx is inferred to be equal to 0. The value of dh\_ref\_displ\_frame\_list\_idx shall be in the range of 0 to dsps\_num\_ref\_displ\_frame\_lists\_in\_dsps - 1, inclusive. When dh\_ref\_displ\_frame\_list\_dsps\_flag is equal to 1 and dsps\_num\_ref\_displ\_frame\_lists\_in\_dsps is equal to 1, the value of ref\_displ\_frame\_list\_idx is inferred to be equal to 0.

The variable RlsIdx for the current atlas tile is derived as follows:

RlsIdx = dh\_ref\_displ\_frame\_list\_dsps\_flag ?  
 ref\_displ\_frame\_list\_idx : dsps\_num\_ref\_displ\_frame\_lists\_in\_dsps

**dh\_additional\_dfoc\_lsb\_present\_flag**[ j ] equal to 1 specifies that dh\_additional\_dfoc\_lsb\_val[ j ] is present for the current displacement frame. Dh\_additional\_dfoc\_lsb\_present\_flag[ j ] equal to 0 specifies that dh\_additional\_dfoc\_lsb\_val[ j ] is not present.

**dh\_additional\_dfoc\_lsb\_val**[ j ] specifies the value of FullFrmOrderCntLsbLt[ RlsIdx ][ j ] for the current atlas tile as follows:

FullDisplFrmOrderCntLsbLt[ RlsIdx ][ j ] = dh\_additional\_dfoc\_lsb\_val[ j ] \*  
 MaxDisplFrmOrderCntLsb +dfoc\_lsb\_lt[ RlsIdx ][ j ]

The syntax element dh\_additional\_dfoc\_lsb\_val[ j ] is represented by dfps\_additional\_lt\_dfoc\_lsb\_len bits. When not present, the value of dh\_additional\_dfoc\_lsb\_val[ j ] is inferred to be equal to 0.

**dh\_parameters\_override\_flag** equal to 1 indicates the parameters dh\_subdivision\_override\_flag and dh\_quantization\_override\_flag are present in the displacement header of the current displacement frame.

**dh\_subdivision\_override\_flag** equal to 1 indicates dh\_subdivision\_method and dh\_subdivision\_iteration\_count are present in the displacement header of the current displacement frame. When dh\_subdivision\_override\_flag[ tileID ][ patchIdx ] is not present, its value is inferred to be equal to 0.

**dh\_quantization\_override\_flag** equal to 1 indicates displ\_quantization\_parameters(qpIndex, subdivisionCount) syntax structure is present in the displacement header of the current displacement frame. When dh\_quantization\_override\_flag is not present, its value is inferred to be equal to 0.

The variable QpIndex for the current patch is derived as follows:

QpIndex = dh\_quantization\_override\_flag ? 2 :  
 dfps\_quantization\_parameters\_enable\_flag ? 1: 0

**dh\_subdivision\_method** indicates the identifier of the method to subdivide the subparts of the mesh associated with the displacement header of the current displacement frame. When dh\_subdivision\_method is not present, its value is inferred to be equal to dfps\_subdivision\_method. Table 2 describes the list of supported subdivision methods and their relationship with dh\_subdivision\_method.

**dh\_subdivision\_iteration\_count** indicates the number of iterations used for the subdivision in the displacement header of the current displacement frame. When dh\_subdivision\_iteration\_count is not present, if dh\_subdivision\_method equal to 0, its values is inferred to be equal to 0, otherwise its values is inferred to be equal to dfps\_subdivision\_iteration\_count.

**dh\_num\_ref\_idx\_active\_override\_flag** equal to 1 specifies that the syntax element num\_ref\_idx\_active\_minus1 is present for the current displacement frame. dh\_num\_ref\_idx\_active\_override\_flag equal to 0 specifies that the syntax element num\_ref\_idx\_active\_minus1 is not present. If dh\_num\_ref\_idx\_active\_override\_flag is not present, its value shall be inferred to be equal to 0.

**dh\_num\_ref\_idx\_active\_minus1** is used for the derivation of the variable NumRefIdxActive for the current displacement frame. The value of dh\_num\_ref\_idx\_active\_minus1 shall be in the range of 0 to 14, inclusive.

When the current displacement frame is a P\_DISPLACEMENT displacement frame, dh\_num\_ref\_idx\_active\_override\_flag is equal to 1, and dh\_num\_ref\_idx\_active\_minus1 is not present, dh\_num\_ref\_idx\_active\_minus1 is inferred to be equal to 0.

The variable NumRefIdxActive is derived as follows:

if( dh\_type == P\_DISPLACEMENT ) {  
 if( dh\_num\_ref\_idx\_active\_override\_flag == 1 )  
 NumRefIdxActive = dh\_num\_ref\_idx\_active\_minus1 + 1  
 else {  
 if( num\_ref\_entries[ RlsIdx ] >= dfps\_num\_ref\_idx\_default\_active\_minus1 + 1 )  
 NumRefIdxActive = dfps\_num\_ref\_idx\_default\_active\_minus1 + 1  
 else  
 NumRefIdxActive = num\_ref\_entries[ RlsIdx ]  
 }  
 }  
 else   
 NumRefIdxActive = 0

NumRefIdxActive minus 1 specifies the maximum value of the displacement reference frame index that may be used to decode the current displacement frame.

**dh\_layer\_iner\_flag** specifies whether (when 1) or not (when 0) the dequantized displacement wavelet coefficient of all LoD layer is predicted from the one of the reference displacement frames.

**dh\_lod\_inter\_flag**[ i ] specifies whether (when 1) or not (when 0) the dequantized displacement wavelet coefficient of LoD ith layer is predicted from the one of the reference displacement frames. If dh\_lod\_inter\_flag[ i ] is not present it shall be inferred to be equal to 1.

* + - * 1. Displacement data unit semantics

**ddu\_lod\_count**[ displID ] indicates the number of the subdivision levels(?) used for the displacement signalled in the data unit associated with displId displID.

**ddu\_vertex\_count\_lod**[ displID ][ i ] indicates the number of displacements for the i-th level of wavelet transform for the data unit associated with displId displID.

**ddu\_lod\_split\_flag**[ displID ][ i ] when equal to 1 indicates the i-th level of wavelet transform for the data unit associated with displId displID is split into blocks. When ddu\_lod\_split\_flag[ displID ][ i ] is equal to 0 the i-th level of wavelet transform for the data unit associated with displId displID is not split into blocks. If ddu\_lod\_split\_flag[ displID ][ i ] is not present it shall be inferred to be equal to 0.

**ddu\_num\_subblock\_lod**[ displID ][ i ] indicates the number of subblocks for the i-th level of wavelet transform for the data unit associated with displId displID. If ddu\_num\_subblock\_lod[ displID ][ i ] is not present it shall be inferred to be equal to 1.

**ddu\_nz\_subblock**[ displID ][ k ][ block ] indicates block-th subblock has at least one nonzero value for the k-th component of the displacement coefficient.

**ddu\_coeff\_abs\_level\_gt0**[ displID ][ v ][ k ] indicates whether the v-th coefficient of the k-th component of the displacement coefficient has an absolute value higher than zero (when 1), or not (when 0). If ddu\_coeff\_abs\_level\_gt0[ displID ][ v ][ k ] is not present it shall be inferred to be equal to 0.

**ddu\_coeff\_sign**[ displID ][ v ][ k ] equals 1 indicates that the value of the v-th coefficient of the k-th component of the displacement coefficient has a positive sign. ddu\_coeff\_sign[ displID ][ k ][ v ] equals 0 indicates that the value of the v-th coefficient of the the k-th component of the displacement coefficient has a negative sign.

**ddu\_coeff\_abs\_level\_gt1**[ displID ][ v ][ k ] indicates whether the v-th coefficient of the k-th component of the displacement coefficient has an absolute value higher than one (when 1), or not (when 0). If ddu\_coeff\_abs\_level\_gt1[ displID ][ v ][ k ] is not present it shall be inferred to be equal to 0

**ddu\_coeff\_abs\_level\_gt2**[ displID ][ v ][ k ] indicates whether the v-th coefficient of the k-th component of the displacement coefficient has an absolute value higher than two (when 1), or not (when 0). If ddu\_coeff\_abs\_level\_gt2[ displID ][ v ][ k ] is not present it shall be inferred to be equal to 0

**ddu\_coeff\_abs\_level\_gt3**[ displID ][ v ][ k ] indicates whether the v-th coefficient of the k-th component of the displacement coefficient has an absolute value higher than three (when 1), or not (when 0). If ddu\_coeff\_abs\_level\_gt3[ displID ][ v ][ k ] is not present it shall be inferred to be equal to 0.

**ddu\_coeff\_abs\_level\_rem**[ displID ][ v ][ k ] plus 3 indicates the absolute value of the v-th coefficient of the k-th component of the displacement coefficient.

* 1. Decoding process
     1. General Decoding Process

Input to this process is a displacement bitstream.

Outputs of this process are:

* The variable NumDecDispFrames, indicating the number of decoded displacement frames,
* The variable DispDimension, indicating the dimension of displacement,
* A 3D array DecDispFrames, the decoded displacement frames, where the dimension corresponds to the decoded displacement frame index, the displacement index, and the displacement component index.
* The following 1D arrays
  + DispCountPerFrame, indicating the number of displacement values in a frame,
  + DecDispBitDepth, indicating the displacement bit depth,
  + DecDispOutOrdIdx, indicating the displacement output order index, and
  + DecDispCompTime, indicating the displacement composition time.

where the dimension corresponds to the decoded displacement frame index.

Subclause J.8.2 is repeatedly invoked for each coded displacement frame in the current displacement bitstream to be decoded, in decoding order.

* + 1. Decoding process for a coded displacement frame

The decoding processes specified in this subclause apply to each coded displacement frame.

The decoding process for the current displacement frame takes as inputs the syntax elements and upper-case variables from clause J.7.

The decoding process operates as follows for the current displacement frame:

* The decoding of displacement NAL units is specified in subclause J.8.3.
* The processes in subclause J.8.4 specify the decoding processes for displacement header.
* The processes in subclause J.8.5 specify the decoding processes for displacement data units.
  + 1. Displacement NAL unit decoding process

Inputs to this process are displacement NAL units of the current displacement frame and their associated non-ACL NAL units.

Outputs of this process are the parsed RBSP syntax structures encapsulated within the displacement NAL units.

The decoding process for each displacement NAL unit extracts the RBSP syntax structure from the displacement NAL unit and then parses the RBSP syntax structure.

* + 1. Displacement header decoding process
       1. Displacement frame order count derivation process

Output of this process is DisplFrmOrderCntVal, the displacement frame order count of the current displacement.

Displacement frame order counts are used to identify the output order of displacement frames, as well as for decoder conformance checking.

Each coded displacement frame is associated with a displacement frame order count variable, denoted as DisplFrmOrderCntVal.

When the current displacement frame is not an IRAP coded displacement with NoOutputBeforeRecoveryFlag equal to 1, the variables prevDisplFrmOrderCntLsb and prevDisplFrmOrderCntMsb are derived as follows:

* Let prevDisplFrm be the previous displacement frame in decoding order that has DisplTemporalID equal to 0 and that is not a RASL, RADL, or SLNR coded displacement.
* The variable prevDisplFrmOrderCntLsb is set equal to the displacement frame order count LSB value, dh\_frm\_order\_cnt\_lsb, of prevDisplFrm.
* The variable prevDisplFrmOrderCntMsb is set equal to DisplFrmOrderCntMsb of prevDisplFrm.

The variable DisplFrmOrderCntMsb of the current displacement frame is derived as follows:

* If the current displacement is an IRAP coded displacement with NoOutputBeforeRecoveryFlag equal to 1, DisplFrmOrderCntMsb is set equal to 0.
* Otherwise, DisplFrmOrderCntMsb is derived as follows:

if( ( dh\_frm\_order\_cnt\_lsb < prevDisplFrmOrderCntLsb ) &&  
 ( ( prevDisplFrmOrderCntLsb − dh\_frm\_order\_cnt\_lsb) >=   
 ( MaxDisplFrmOrderCntLsb / 2 ) ) )  
 DisplFrmOrderCntMsb = prevDisplFrmOrderCntMsb + MaxDisplFrmOrderCntLsb (57)  
 else if( ( dh\_frm\_order\_cnt\_lsb > prevDisplFrmOrderCntLsb ) &&  
 ( ( dh\_frm\_order\_cnt\_lsb– prevDisplFrmOrderCntLsb ) >  
 ( MaxDisplFrmOrderCntLsb / 2 ) ) )  
 DisplFrmOrderCntMsb = prevDisplFrmOrderCntMsb – MaxDisplFrmOrderCntLsb (58)  
 else  
 DisplFrmOrderCntMsb = prevDisplFrmOrderCntMsb (59)

DisplFrmOrderCntVal is derived as follows:

DisplFrmOrderCntVal = DisplFrmOrderCntMsb + dh\_frm\_order\_cnt\_lsb (60)

The value of DisplFrmOrderCntVal shall be in the range of −231 to 231 − 1, inclusive. In one CAS, the DisplFrmOrderCntVal values for any two coded displacement frames with the same value of displ\_nal\_layer\_id shall not be the same.

The function DisplFrmOrderCnt( aFrmX ) is specified as follows:

DisplFrmOrderCnt( aFrmX ) = DisplFrmOrderCntVal of the displacement frame aFrmX (61)

The function DiffDisplFrmOrderCnt( aFrmA, aFrmB ) is specified as follows:

DiffDisplFrmOrderCnt( aFrmA, aFrmB ) =  
 DisplFrmOrderCnt( aFrmA ) – DisplFrmOrderCnt( aFrmB ) (62)

The bitstream shall not contain data that result in values of DiffDisplFrmOrderCnt( aFrmA, aFrmB) used in the decoding process that are not in the range of −215 to 215 − 1, inclusive.

NOTE – Let X be the current displacement frame and Y and Z be two other displacement frames in the same CAS, Y and Z are considered to be in the same output order direction from X when both DiffDisplFrmOrderCnt( X, Y ) and DiffDisplFrmOrderCnt( X, Z ) are positive or both are negative.

* + - 1. Decoding process for generating unavailable reference displacement frames
         1. General

This process is invoked once per coded displacement frame when the current displacement frame is a BLA displacement frame or a CRA displacement frame with NoOutputBeforeRecoveryFlag equal to 1.

NOTE – This process is primarily specified only for the specification of syntax constraints for RASL displacement frames. The entire specification of the decoding process for RASL displacement frames associated with an IRAP coded displacement frame that has NoOutputBeforeRecoveryFlag equal to 1 is included herein only for purposes of specifying constraints on the allowed syntax content of such RASL displacement frames. During the decoding process, any RASL displacement frames associated with an IRAP coded displacement frame that has NoOutputBeforeRecoveryFlag equal to 1 can be ignored, as these displacement frames are not specified for output and have no effect on the decoding process of any other displacement frames that are specified for output. However, in HRD operations as specified in Annex E, RASL access units could be taken into consideration in derivation of coded displacement frame buffer (CAB) arrival and removal times.

When this process is invoked, the following applies:

* For each RefDisplFrmList[ j ], with j in the range of 0 to drl\_num\_ref\_entries[ RlsIdx ] − 1, inclusive, that is equal to "no reference displacement frame", a displacement frame is generated as specified in subclause J.8.4.2.2 and the following applies:
  + The value of displ\_nal\_layer\_id for the generated displacement frame is set equal to the displ\_nal\_layer\_id value of the current displacement frame.
  + If drl\_st\_ref\_displ\_frame\_flag[ RlsIdx ][ j ] is equal to 1 the value of DisplFrmOrderCntVal for the generated displacement frame is set equal to RefDisplFrmList[ j ] and the generated displacement frame is marked as "used for short-term reference".
  + Otherwise, when drl\_st\_ref\_displ\_frame\_flag[ RlsIdx ][ j ] is equal to 0 the value of DisplFrmOrderCntVal for the generated displacement frame is set equal to RefDisplFrmList[ j ], the value of dh\_frm\_order\_cnt\_lsb for all displacements in the generated displacement frame is inferred to be equal to ( RefDisplFrmList[ j ] & ( MaxDisplFrmOrderCntLsb – 1 ) ), and the generated displacement frame is marked as "used for long-term reference".
  + The value of DispFrameOutputFlag for the generated reference displacement frame is set equal to 0.
  + RefDisplFrmList[ j ] is set to be the generated reference displacement frame.
  + The value of DisplTemporalID for the generated displacement frame is set equal to the DisplTemporalID value of the current displacement frame.
  + The value of dfps\_displ\_frame\_parameter\_set\_id for the generated displacement frame is set equal to dfps\_displ\_frame\_parameter\_set\_id of the current displacement frame.

The value of displ\_nal\_layer\_id for the generated displacement frame is set equal to displ\_nal\_layer\_id of the current displacement frame.

* + - * 1. Generation of one unavailable displacement frame

When this process is invoked, an unavailable displacement frame is generated as follows:

* For all displacements that are associated with this displacement frame the following applies:

for( t = 0; t< NumDispls; t++ ) {  
 displID = DisplIndexToID( t )  
 xxx[ displID ] = 0  
 …  
 }

* + - 1. Reference displacement frame list construction process

This process is invoked at the beginning of the decoding process for each displacement of a displacement frame.

Reference displacement frames are addressed through reference indices. A reference index is an index into a reference displacement frame list (RDFL). When decoding an I\_DISPLACEMENT displacement, no RDFL is used in decoding of the displacement data. When decoding a P\_DISPLACEMENT displacement, a single reference displacement frame list, RefDisplFrmList, is used in decoding of the displacement data.

At the beginning of the decoding process for each displacement, the RDFL RefDisplFrmList is derived. The RDFL is used in marking of reference displacement frames as specified in subclause J.8.4.4 or in decoding of the displacement data.

NOTE – For an I\_DISPLACEMENT displacement of a displacement frame, RefDisplFrmList could be derived for bitstream conformance checking purposes, but its derivation is not necessary for the decoding of the current displacement frame or for displacement frames that follow the current displacement frame in decoding order.

The reference displacement frame list RefDisplFrmList is constructed as follows:

for( j = 0, dfocBase = DisplFrmOrderCntVal; j < drl\_num\_ref\_entries[ RlsIdx ]; j++) {  
 if( drl\_st\_ref\_displ\_frame\_flag[ RlsIdx ][ j ] ) {  
 RefDisplFrmMfocList[ j ] = dfocBase − DeltaDfocSt[ RlsIdx ][ j ]  
 if( there is a reference displacement frame afA in the DDB with   
 DisplFrmOrderCntVal equal to RefDisplFrmMfocList[ j ] )  
 RefDisplFrmList[ j ] = afA  
 else  
 RefDisplFrmList[ j ] = "no reference displacement frame"  
 dfocBase = RefDisplFrmDfocList[ j ]  
 } else {  
 if( there is a reference displacement frame afA in the DDB with DisplFrmOrderCntVal & ( MaxLtDisplFrmOrderCntLsb − 1 )  
 equal to FullDisplFrmOrderCntLsbLt[ RlsIdx ][ j ] )  
 RefDisplFrmList[ j ] = afA  
 else  
 RefDisplFrmList[ j ] = "no reference displacement frame"  
 }  
 }

The first NumRefIdxActive entries in RefDisplFrmList are referred to as the active entries in RefDisplFrmList and the other entries in RefDisplFrmList are referred to as the inactive entries in RefDisplFrmList.

It is a requirement of bitstream conformance that the following constraints apply:

* drl\_num\_ref\_entries[ RlsIdx ] shall not be less than NumRefIdxActive.
* The displacement frame referred to by each active entry in RefDisplFrmList shall be present in the DDB and shall have DisplTemporalID less than or equal to that of the current displacement frame.
* The displacement frame referred to by each entry in RefDisplFrmList shall not be the current displacement frame.
* A short term reference displacement frame entry and a long term reference displacement frame entry in RefDisplFrmList of an displacement shall not refer to the same displacement frame.
* There shall be no long term reference displacement frame entry in RefDisplFrmList for which the difference between the DisplFrmOrderCntVal of the current displacement and the DisplFrmOrderCntVal of the displacement frame referred to by the entry is greater than or equal to 224.
* Let setOfRefDisplFrms be the set of unique displacement frames referred to by all entries in RefDisplFrmList that have the same displ\_nal\_layer\_id as the current displacement frame. The number of displacement frames in setOfRefDisplFrms shall be less than or equal to dsps\_max\_dec\_displ\_frame\_buffering\_minus1, and setOfRefDisplFrms shall be the same for all displacements of a displacement frame.
* The displacement frames referred to by each active entry in RefDisplFrmList shall have exactly the same number of displacements as the current displacement frame.
* The RefDisplFrmList of all displacements in the current displacement frame shall contain the same unique displacement frames, without, however, any restrictions in ordering of the reference displacement frames.
* When the current displacement frame, with displ\_nal\_layer\_id equal to a particular value layerID, is an IRAP coded displacement, there shall be no displacement referred to by an entry in RefDisplFrmList that precedes, in output order or decoding order, any preceding IRAP coded displacement with displ\_nal\_layer\_id equal to layerID in decoding order (when present).
* When the current displacement frame is not a RASL coded displacement associated with a CRA coded displacement with NoOutputBeforeRecoveryFlag equal to 1, there shall be no displacement referred to by an active entry in RefDisplFrmList that was generated by the decoding process for generating unavailable reference displacement frames for the CRA coded displacement associated with the current displacement.
* When the current displacement frame follows an IRAP coded displacement having the same value of displ\_nal\_layer\_id in both decoding order and output order, there shall be no displacement frame referred to by an active entry in RefDisplFrmList that precedes that IRAP coded displacement in output order or decoding order.
* When the current displacement frame follows an IRAP coded displacement having the same value of displ\_nal\_layer\_id and all the leading displacement frames, if any, associated with that IRAP coded displacement in both decoding order and output order, there shall be no displacement referred to by an entry in RefDisplFrmList that precedes that IRAP coded displacement in output order or decoding order.

When the current displacement frame is a RADL coded displacement, there shall be no active entry in RefDisplFrmList that is a displacement frame that precedes the associated IRAP coded displacement of the RADL coded displacement in decoding order.

* + - 1. Reference displacement frame marking process

This process is invoked once per displacement frame, after decoding of a displacement header and the decoding process for RDFL construction for the displacement frame as specified in subclause J.8.4.3, but prior to the decoding of the displacement data. This process might result in one or more reference displacement frames in the DDB being marked as "unused for reference" or "used for long-term reference".

A decoded displacement frame in the DDB can be marked as "unused for reference", "used for short-term reference", or "used for long-term reference", but only one among these three at any given moment during the operation of the decoding process. Assigning one of these markings to a displacement frame implicitly removes another of these markings when applicable. When a displacement frame is referred to as being marked as "used for reference", this collectively refers to the displacement frame being marked as "used for short-term reference" or "used for long-term reference" (but not both).

Short term reference displacement frames are identified by their displ\_nal\_layer\_id and DisplFrmOrderCntVal values. Long term reference displacement frames are identified by their displ\_nal\_layer\_id values and by the Log2( MaxLtDisplFrmOrderCntLsb ) least significant bits of their DisplFrmOrderCntVal values.

For all cases, the following applies:

* For each long term reference displacement frame entry in RefDisplFrmList, when the displacement frame is marked as "used for short-term reference" and has the same displ\_nal\_layer\_id as the current displacement frame, the displacement frame is marked as "used for long-term reference".

Each reference displacement frame in the same displ\_nal\_layer\_id as the current displacement frame in the DDB that is not referred to by any entry in RefDisplFrmList is marked as "unused for reference"

* + 1. Decoding process for displacement data units
       1. General Decoding process for displacement data units

The decoding process of a displacement data units is performed as specified in subclause J.8.5.2.

The outputs of this process are:

* A 2D array, DecDispFrame, indicating a decoded displacement.
  + - 1. Default Decoding process for displacement data units

First, dispQuantCoeffArray, which is a 2D array of size totalVertCount × MaxDimension indicating the quantized displacement wavelet coefficients, is computed.

dispQuantCoeffArray[ v ][ k ] indicating v-th coefficient of the k-th component of dispQuantCoeffArray is computed as follows:

for( k = 0; k < MaxDimension; k++ ) {  
 for( l = 0; l < displ\_lod\_count; l++ ) {  
 for( m = 0; m < numSubblock[ l ]; m++ ) {  
 if( ddu\_nz\_subBlock[ l ][ m ] ) {  
 for( v = vBlockStart[ l ][ m ]; v < vBlockEnd[ l ][ m ]; v++ ){  
 dispQuantCoeffArray[ v ][ k ] =  
 ddu\_coeff\_sign[ k ][ v ] ? 1 : -1) \*  
 (ddu\_coeff\_abs\_level\_gt0[ k ][ v ] +   
 ddu\_coeff\_abs\_level\_gt1[ k ][ v ] +  
 ddu\_coeff\_abs\_level\_gt2[ k ][ v ] +  
 ddu\_coeff\_abs\_level\_gt3[ k ][ v ] +   
 ddu\_coeff\_abs\_level\_rem[ k ][ v ])  
 }  
 }  
 }  
 }  
 }

Then, verCoordCount is computed as follows:

verCoordCount = totalVertCount

Next, the fixed-point inverse quantization process described in subclause J.8.5.3 is invoked with the parameters verCoordCount, subdivisionIterationCount, dispQuantCoeffArray, , levelOfDetailCounts, as inputs, and the parameter dispCoeffArray as output.

If the coding type of the current displacement frame, dh\_type is equal to P\_DISPLACEMENT, then the following steps are applied:

* refIdx is set to 0
* The reference displacement frame refDisp, which is a 2D array of size totalVertCount × MaxDimension indicating the quantized displacement wavelet coefficients in the reference displacement frame, is selected as the displacement frame corresponding to the ( refIdx + 1 )-th entry in the reference displacement frame list RefDispFrmList, RefDispFrmList[ refIdx ]
* The inter prediction process for the dequantized wavelet coefficients described in subclause J.8.5.4 is invoked with the parameters dispCoeffArray, subdivisionIterationCount and levelOfDetailCounts as inputs, and the parameter dispCoeffArray as output.

Then, DecDispFrame is computed as follows:

DecDispFrame = dispCoeffArray

* + - 1. Fixed-point inverse quantization

Inputs to this process are the variables verCoordCount, subdivisionIterationCount, the 2D array dispQuantCoeffArray, of size verCoordCount × MaxDimension, and the 1D array levelOfDetailCounts, of size (subdivisionIterationCount + 1).

The output of this process is a 2D array dispCoeffArray, of size verCoordCount × MaxDimension.

First, the inverse scale factor shall be computed as follows:

bitDepthPosition = dsps\_geometry\_3d\_bit\_depth\_minus1 + 1  
 bitDepthOffset = dqp\_bitdepth\_offset[ QpIdx ]  
 rTable = { 2048, 2299, 2580, 2896, 3251, 3649 }  
 for( lodIdx = 0; lodIdx < subdivisionIterationCount + 1; lodIdx++ ) {  
 for( dimIdx = 0; dimIdx < MaxDimension; dimIdx++ ) {  
 qp = QuantizationParameter[ QpIdx ][ lodIdx ][ dimIdx ]  
 q = ( qp – 4 ) / 6 Note: integer division is used  
 r = ( qp – 4 ) – 6 \* q  
 InverseScale[ QpIdx ][ lodIdx ][ dimIdx ] = qp >= 4 ?  
 (rTable[r] << (bitDepthPosition – bitDepthOffset + q - 4)) : 0  
 }  
 }

QpIdx is the index of the quantization parameter set.

Second, a 2D array iscale, of size (subdivisionIterationCount + 1)× 3, indicating inverse scale factor is derived as follows:

lodQuantizationFlag = dqp\_lod\_quantization\_flag[ QpIdx ]  
 if ( lodQuantizationFlag ) {  
 for( lodIdx = 0; lodIdx < subdivisionIterationCount + 1; lodIdx++ ) {  
 for( dimIdx = 0; dimIdx < MaxDimension; dimIdx ++ ) {  
 iscale[ lodIdx ][ dimIdx ] = InverseScale[ QpIdx ][ lodIdx ][ dimIdx ]  
 }  
 }  
 } else {  
 for( dimIdx = 0; dimIdx < MaxDimension; dimIdx++ ) {  
 iscale[ 0 ][ dimIdx ] = InverseScale[ QpIdx ][ 0 ][ dimIdx ]  
 levelOfDetailInverseScale[ dimIdx ] =   
 1 << vqp\_log2\_lod\_inverse\_scale[ QpIdx ][ dimIdx ]  
 }  
 for( lodIdx = 1; lodIdx < lodCount; lodIdx++ ) {  
 for( dimIdx = 0; dimIdx < MaxDimension; dimIdx++ ) {  
 iscale[ lodIdx ][ dimIdx ] = iscale[ lodIdx - 1 ][ dimIdx ] \*   
 levelOfDetailInverseScale[ dimIdx ]  
 }  
 }  
 }

Then, all the elements of the 2D array dispCoeffArray, of size verCoordCount × MaxDimension, are initialized as 0. The fixed-point inverse quantization process proceeds as follows:

Vcount0 = 0  
 for( i = 0; i < subdivisionIterationCount; i++ ) {  
 vcount1 = levelOfDetailCounts[ i ]  
 for( v = vcount0; v < vcount1; v++ ) {  
 for( d = 0; d < MaxDimension; d++ ) {  
 dispQuantCoeffArray [ v ][ d ] = dispQuantCoeffArray[ v ][ d ] \* (1<<6)  
 Note: dispQuantCoeffArray contains integer values after AD  
 val = dispQuantCoeffArray[ v ][ d ] \* iscale[ i ][ d ]  
 dispCoeffArray [ v ][ d ] =  
 val < 0 ? –((-val + (1 << 22)) >> 23) : ((val + (1 << 22)) >> 23)  
 }  
 }  
 vcount0 = vcount1  
 }

* + - 1. Inter prediction for dequantized wavelet coefficients

Inputs to this process are:

* dispCoeffArray, which is a 2D array of size verCoordCount × MaxDimension indicating the dequantized displacement wavelet coefficients.
* subdivisionIterationCount, which is a variable indicating the number of subdivision iterations.
* levelOfDetailCounts, a 1D array of size (subdivisionIterationCount + 1) indicating the number of vertex coordinates associated with each subdivision iteration.

The output of this process are:

* dispCoeffArray, which is a 2D array of size verCoordCount × MaxDimension indicating the dequantized displacements wavelet coefficients.

The inter prediction process proceeds as follows:

for( i = 0,vcount0=0; i <= subdivisionIterationCount; i++ ) {  
 vcount1 = levelOfDetailCounts[ i ]  
 if( dh\_lod\_inter\_flag[ i ] == 1 ) {  
 for( v = vcount0; v < vcount1; v++ ) {  
 for( d = 0; d < MaxDimension; d++ ) {  
 dispCoeffArray[ v ][ d ] += refDisp[ v ][ d ]  
 }  
 }  
 }  
 vcount0 = vcount1  
}

* 1. Parsing process

The specifications in Annex K apply.

* 1. Profiles and levels

[Ed. Note (LK)] Input required.

* 1. Displacement hypothetical reference decoder

[Ed. Note (LK)] Input required.

* 1. Supplemental enhancement information

[Ed. Note (LK)] Input required.

1. (Normative)  
   Inverse Binarization Process
   1. General

Arithmetic encoded (ae) syntax elements are parsed according to the processes corresponding to the syntax element’s descriptor and the name as specified in Table K-1, Table K-2, and Table K-3.

Table K-1 – Basemesh sub-bitstream syntax element specific parsing processes (ae(v))

|  |  |  |
| --- | --- | --- |
| **Syntax element** | **Parsing** | **Parameters** |
| bmidu\_mv\_signalled\_flag[ ] | K.2.1 (FL) | numBins = 1 |
| bmidu\_mv\_pred\_mode\_group[ ][ ][ ] | K.2.6(TU) | maxVal = bmsps\_inter\_mesh\_max\_num\_mvp\_cand\_minus1 |
| bmidu\_mv\_\_residual\_abs\_gt0[ ][ ] | K.2.1 (FL) | numBins = 1 |
| bmidu\_mv\_\_residual\_sign[ ][ ] | K.2.1 (FL) | numBins = 1 |
| bmidu\_mv\_\_residual\_abs\_gt1[ ][ ] | K.2.1 (FL) | numBins = 1 |
| bmidu\_mv\_\_residual\_abs\_rem[ ][ ][ ] | K.2.3 (EGk) | k = 0 |
| bmidu\_skip\_group\_flag[ ] | K.2.1 (FL) | numBins = 1 |

Table K-2 – MPEG EdgeBreaker syntax element specific parsing processes (ae(v))

|  |  |  |
| --- | --- | --- |
| **Syntax element** | **Parsing** | **Parameters** |
| mesh\_position\_fine\_residual[][] | K.2.5 (TU + EGk + S) | maxOffset = 7, k = 2 |
| mesh\_position\_coarse\_residual[][] | K.2.5 (TU + EGk + S) | maxOffset = 7, k = 2 |
| mesh\_attribute\_fine\_residual[][][] | /\* TEXCOORD \*/  K.2.5 (TU + EGk + S)  /\* NORMAL \*/  K.2.5 (TU + EGk + S)/\* MATERIAL\_ID \*/  K.2.3 (EGk) | /\* TEXCOORD \*/  maxOffset = 7, k = 2  /\* NORMAL \*/  maxOffset = 7, k = 2/\* MATERIAL\_ID \*/  k = 2 |
| mesh\_attribute\_coarse\_residual[][][] | /\* TEXCOORD \*/  K.2.5 (TU + EGk + S) | /\* TEXCOORD \*/  maxOffset = 7, k = 2 |
| mesh\_normal\_octahedral\_second\_residual[][][] | K.2.5 (TU + EGk + S) | maxOffset = 7, k = 2 |
| mesh\_clers\_symbol[] | K.2.7 |  |
| mesh\_attribute\_seam[][] | K.2.1 (FL) | numBins =1 |
| mesh\_texcoord\_stretch\_orientation[][] | K.2.1 (FL) | numBins = 1 |
| mesh\_handle\_first\_sign[] | K.2.1 (FL) | numBins = 1 |
| mesh\_handle\_second\_shift[] | K.2.1 (FL) | numBins = 1 |
| mesh\_handle\_first\_variable\_delta\_length4\_minus1[i] | K.2.6 (TU) | maxVal = 8 |
| mesh\_handle\_first\_variable\_delta[i] | K.2.1 (FL) | numBins = 4\*(D1L + 1) |
| mesh\_handle\_second\_variable\_delta\_length4\_minus1[i] | K.2.6 (TU) | maxVal = 8 |
| mesh\_handle\_second\_variable\_delta[i] | K.2.1 (FL) | numBins = 4\*(D2L + 1) |
| mesh\_position\_is\_duplicate\_flag[] | K.2.1 (FL) | numBins =1 |
| mesh\_attribute\_is\_duplicate\_flag[][] | K.2.1 (FL) | numBins =1 |
| mesh\_materialid\_default\_not\_equal\_flag[][] | K.2.1 (FL) | numBins =1 |
| mesh\_materialid\_default\_left\_flag[][] | K.2.1 (FL) | numBins =1 |
| mesh\_materialid\_default\_right\_flag[][] | K.2.1 (FL) | numBins =1 |
| mesh\_materialid\_default\_facing\_flag[][] | K.2.1 (FL) | numBins =1 |

Table K-3 – Arithmetic coded displacement syntax element specific parsing processes (ae(v))

|  |  |  |
| --- | --- | --- |
| **Syntax element** | **Parsing** | **Parameters** |
| dsps\_single\_dimension\_flag[ ] | K.2.1 (FL) | numBins = 1 |
| ddu\_nz\_subBlock[ ][ ][ ] | K.2.1 (FL) | numBins = 1 |
| ddu\_coeff\_abs\_level\_gt0[ ][ ][ ] | K.2.1 (FL) | numBins = 1 |
| ddu\_coeff\_sign[ ][ ][ ] | K.2.1 (FL) | numBins = 1 |
| ddu\_coeff\_abs\_level\_gt1[ ][ ][ ] | K.2.1 (FL) | numBins = 1 |
| ddu\_coeff\_abs\_level\_gt2[ ][ ][ ] | K.2.1 (FL) | numBins = 1 |
| ddu\_coeff\_abs\_level\_gt3[ ][ ][ ] | K.2.1 (FL) | numBins = 1 |
| ddu\_coeff\_abs\_level\_rem[ ][ ][ ] | K.2.3 (EGk) | K = 0 |



* 1. Parsing processes
     1. Parsing unsigned fixed-length codes (FL)

Parsing is parameterized by numBins, the number of bins that represent the syntax element.

The result is the unsigned syntax element value parsedVal, parsed and constructed as:

parsedVal  =  0  
 for(BinIdx  =  0;  BinIdx  <  numBins;  BinIdx++)  
 parsedVal  =  (parsedVal  <<  1)  +  dec\_aebin()

where dec\_aebin() is the process described in K.3.2 for the current syntax element.

* + 1. Parsing signed fixed-length codes (FL+S)

Parsing is parameterized by numBins, the number of bins that represent the absolute syntax element value.

The unsigned syntax element magnitude is parsed:

PartVal  =  0  
 for(BinIdx  =  0; BinIdx  <  numBins;  BinIdx++)  
 PartVal  =  (PartVal  <<  1)  +  dec\_aebin()

The result is the signed syntax element value val, parsed and constructed as:

sign  =  dec\_aebin( )  
 val  =  sign  ?  − PartVal  :  PartVal

* + 1. Parsing k-th order exp-Golomb codes (EGk)

Parsing is parameterized by k, the order of the exp-Golomb code.

First, a unary encoded prefix is parsed as:

prefix = 0  
 for(BinIdxPfx = 0; dec\_aebin() != 0; BinIdxPfx++)  
 prefix++

Then, a suffix comprising 𝑘 + prefix bins is parsed:

Suffix = 0  
for(BinIdxSfx = 0; BinIdxSfx < k + prefix; BinIdxSfx++)  
 suffix = (suffix << 1) + dec\_aebin()

The result is the unsigned syntax element value val, constructed as:

val = (1 << (prefix + k)) + suffix – (1 << k)

* + 1. Parsing concatenated truncated unary and k-th order exp-Golomb codes (TU+EGk)

Parsing is parameterized by maxOffset, the limit for the truncated unary offset encoding and 𝑘, the order of the exp-Golomb code;

First, a truncated unary encoded offset is parsed:

offset  =  0  
 for(BinIdxTu  =  0; (offset  <  maxOffset)  &&  (dec\_aebin()  ==  1); BinIdxTu++)  
 offset++

Second, if the value of offset is equal to maxOffset, a unary encoded prefix is parsed:

Prefix  =  0  
 if(offset  ==  maxOffset)  
 for(BinIdxPfx  =  0;  dec\_aebin()  !=  0;   BinIdxPfx++)  
 prefix++

Then, if the value of offset is equal to maxOffset, a suffix comprising 𝑘 + prefix bins is parsed:

suffix  =  0  
 if(offset  ==  maxOffset)  
 for(BinIdxSfx  =  0; BinIdxSfx  <  k  +  prefix;  BinIdxSfx++)  
 suffix  =  (suffix  <<  1) +  dec\_aebin()

The result is the unsigned syntax element value val, constructed as:

val  =  offset  +  (1 << (prefix  +  k)) +  suffix  −  (1 << k)

* + 1. Parsing signed concatenated truncated unary and k-th order exp-Golomb codes (TU+EGk+S)

Parsing is parameterized by maxOffset, the limit for the truncated unary offset encoding and 𝑘, the order of the exp-Golomb code;

First, a truncated unary encoded offset is parsed:

offset  =  0  
 for(BinIdxTu  =  0; offset  <  maxOffset  &&  dec\_aebin()  ==  1;  BinIdxTu++)  
 offset++

Second, if the value of offset is equal to maxOffset, a unary encoded prefix is parsed:

prefix  =  0  
 if(offset  ==  maxOffset)  
 for(BinIdxPfx = 0; dec\_aebin() != 0; BinIdxPfx++)  
 prefix++

Then, if the value of offset is equal to maxOffset, a suffix comprising 𝑘 + prefix bins is parsed:

suffix  =  0  
 if(  offset  ==  maxOffset)  
 for(BinIdxSfx  =  0; BinIdxSfx  <  k  +  prefix;  BinIdxSfx++)  
 suffix  =  (suffix  <<  1) +  dec\_aebin()

The result is the signed syntax element value val, parsed and constructed as:

if(offset  >  0)  
 sign = dec\_aebin()  
 absVal = offset + (1 << (prefix + k)) + suffix – (1 << k)  
 val = sign ? – absVal : absVal  
 else  
 val  =  0

* + 1. Parsing truncated unary codes (TU)

Parsing is parameterized by maxVal the limit for the encoding.

The result is the unsigned syntax element value PartVal parsed and constructed as:

PartVal  =  0  
 for(BinIdxTu  =  0;  PartVal  <  maxVal  &&  dec\_aebin()  ==  1;  BinIdxTu++)  
 PartVal++

* + 1. Parsing Mesh CLERS symbols

Parsing is performed for symbol in mesh\_clers\_symbol syntax element with index i.

The result is the unsigned syntax element value val parsed and constructed as:

val = 0  
 nbBins = 4  
 for( BinIdxClers = 0; BinIdxClers < nbBins; BinIdxClers++ ) {  
 bitClers = dec\_aebin( )  
 if ((bitClers == 0) || ( (BinIdxClers == 1) && (ClersSymbol0 == CLERS\_C)))  
 nbBins = BinIdxClers + 1  
 val += bitClers << BinIdxClers  
 }

Table K-4 – Mesh CLERS symbol inverse binarization when ClersSymbol0 is not CLERS\_C

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **mesh\_clers\_symbol[i]** | **bin string** | | | |
| C | 0 |  |  |  |
| R | 1 | 0 |  |  |
| S | 1 | 1 | 0 |  |
| L | 1 | 1 | 1 | 0 |
| E | 1 | 1 | 1 | 1 |

Table K-5 – Mesh CLERS symbol inverse binarization when ClersSymbol0 is CLERS\_C

|  |  |  |
| --- | --- | --- |
| **mesh\_clers\_symbol[i]** | **bin string** | |
| C | 0 |  |
| R | 1 | 0 |
| S | 1 | 1 |

* 1. CABAC parsing process
     1. Initialization

The arithmetic decoding engine and CPMs shall be initialized according to K.3.3.2 and K.4.3 at the start of the parsing process.

* + 1. Reading a single arithmetic-coded bin

This subclause specifies the reading of a single arithmetic-coded bin. Each evaluation reads a single bin, parameterized by the name of the coded syntax element.

A CPM identified by the global variable Ctx shall be selected according toK.3.3.4.

If the value of Ctx is neither equal to 'bypass' nor 'terminate':

The value of the decoded bin shall be determined in accordance with K.4.4.1 for a single arithmetic-coded bin with Ctx as the argument prob0.

The selected CPM shall then be updated in accordance with K.3.3.3 using the decoded bin value as the argument binVal.

If the value of Ctx is 'bypass', the value of the decoded bin shall be determined in accordance with K.4.4.2 for an arithmetic-coded bypass bin.

If the value of Ctx is 'terminate', the arithmetic decoder shall be flushed in accordance with K.4.4.4.

* + 1. Contextual probability models
       1. General

A CPM is a 16-bit unsigned integer value that models the probability of a zero bin.

NOTE – The values 0, and represent the probability of a zero bin as impossible, equiprobable and certain respectively. The values 0 and can never be attained due to the operation of the context update process.

The array Contexts, with elements Contexts[ ctxTbl ][ ctxIdx ], represents individual adaptive CPMs used by the CABAC parsing process.

* + - 1. Initialization

All CPMs shall be initialized to .

* + - 1. Update after each coded bin

After each bin coded using an adaptive CPM, the modelled probability shall be updated.

The parameter binVal is the value of the coded bin. The parameters ctxTbl and ctxIdx identify the CPM used to arithmetically code it.

The update shall increase or decrease the modelled probability of a zero-valued bin according to the known value of the coded bin, the upper eight bits of the modelled probability and the channel model specified by Table K-6.

prob0 = Contexts[  ctxTbl  ][  ctxIdx  ]  
 if( binVal )  
 Contexts[  ctxTbl  ][   ctxIdx   ]  −=  CtxUpdateDelta[  prob0  >>  8 ]  
 else  
 Contexts[   ctxTbl   ][  ctxIdx   ]  +=  CtxUpdateDelta[  255  −  (prob0  >>  8  )   ]

The lookup table used for updating context probabilities is defined in Table K-6. below. The lookup table entries are arranged in raster scan order with rows of twelve entries.

Table K-6 – Values of CtxUpdateDelta[ i + j]

| 𝑗 | 𝑖 | | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| **0** | 0 | 2 | 5 | 8 | 11 | 15 | 20 | 24 | 29 | 35 | 41 | 47 |
| **12** | 53 | 60 | 67 | 74 | 82 | 89 | 97 | 106 | 114 | 123 | 132 | 141 |
| **24** | 150 | 160 | 170 | 180 | 190 | 201 | 211 | 222 | 233 | 244 | 256 | 267 |
| **36** | 279 | 291 | 303 | 315 | 327 | 340 | 353 | 366 | 379 | 392 | 405 | 419 |
| **48** | 433 | 447 | 461 | 475 | 489 | 504 | 518 | 533 | 548 | 563 | 578 | 593 |
| **60** | 609 | 624 | 640 | 656 | 672 | 688 | 705 | 721 | 738 | 754 | 771 | 788 |
| **72** | 805 | 822 | 840 | 857 | 875 | 892 | 910 | 928 | 946 | 964 | 983 | 1 001 |
| **84** | 1 020 | 1 038 | 1 057 | 1 076 | 1 095 | 1 114 | 1 133 | 1 153 | 1 172 | 1 192 | 1 211 | 1 231 |
| **96** | 1 251 | 1 271 | 1 291 | 1 311 | 1 332 | 1 352 | 1 373 | 1 393 | 1 414 | 1 435 | 1 456 | 1 477 |
| **108** | 1 498 | 1 520 | 1 541 | 1 562 | 1 584 | 1 606 | 1 628 | 1 649 | 1 671 | 1 694 | 1 716 | 1 738 |
| **120** | 1 760 | 1 783 | 1 806 | 1 828 | 1 851 | 1 874 | 1 897 | 1 920 | 1 935 | 1 942 | 1 949 | 1 955 |
| **132** | 1 961 | 1 968 | 1 974 | 1 980 | 1 985 | 1 991 | 1 996 | 2 001 | 2 006 | 2 011 | 2 016 | 2 021 |
| **144** | 2 025 | 2 029 | 2 033 | 2 037 | 2 040 | 2 044 | 2 047 | 2 050 | 2 053 | 2 056 | 2 058 | 2 061 |
| **156** | 2 063 | 2 065 | 2 066 | 2 068 | 2 069 | 2 070 | 2 071 | 2 072 | 2 072 | 2 072 | 2 072 | 2 072 |
| **168** | 2 072 | 2 071 | 2 070 | 2 069 | 2 068 | 2 066 | 2 065 | 2 063 | 2 060 | 2 058 | 2 055 | 2 052 |
| **180** | 2 049 | 2 045 | 2 042 | 2 038 | 2 033 | 2 029 | 2 024 | 2 019 | 2 013 | 2 008 | 2 002 | 1 996 |
| **192** | 1 989 | 1 982 | 1 975 | 1 968 | 1 960 | 1 952 | 1 943 | 1 934 | 1 925 | 1 916 | 1 906 | 1 896 |
| **204** | 1 885 | 1 874 | 1 863 | 1 851 | 1 839 | 1 827 | 1 814 | 1 800 | 1 786 | 1 772 | 1 757 | 1 742 |
| **216** | 1 727 | 1 710 | 1 694 | 1 676 | 1 659 | 1 640 | 1 622 | 1 602 | 1 582 | 1 561 | 1 540 | 1 518 |
| **228** | 1 495 | 1 471 | 1 447 | 1 422 | 1 396 | 1 369 | 1 341 | 1 312 | 1 282 | 1 251 | 1 219 | 1 186 |
| **240** | 1 151 | 1 114 | 1 077 | 1 037 | 995 | 952 | 906 | 857 | 805 | 750 | 690 | 625 |
| **252** | 553 | 471 | 376 | 255 |  | | | | | | | |

* + - 1. Context selection

A CPM shall be selected for each bin of the coded syntax element as specified by the expression Ctx. The values CtxTbl and CtxIdx shall be determined according to the entries for the syntax element.

Table K-7, Table K-8, and Table K-9 entries qualified by Offset, Prefix or Suffix individually apply when selecting a CPM for a bin of that part of the binarized syntax element.

Table K-7 – Values of CtxTbl and CtxIdx for binarized ae(v) coded syntax elements

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Syntax element** | **CtxTbl** | **CtxIdx** | | **Count** |
| bmidu\_mv\_signalled\_flag | 1 | 0 | | 1 |
| bmidu\_mv\_pred\_mode\_group | 2 | Min(1, *BinIdx*) | | 2 |
| bmidu\_mv\_residual\_abs\_gt0 | 3 | 0, 1, 2 | | 3 |
| bmidu\_mv\_residual\_sign | 4 | 0, 1, 2 | | 3 |
| bmidu\_mv\_residual\_abs\_gt1 | 5 | 0, 1, 2 | | 3 |
| bmidu\_mv\_residual\_abs\_rem[ ][ ] | 6 | Prefix  *(BinIdxPfx <= 1)* | Min( 1, *BinIdx* *Pfx* ) | 2 |
| Prefix  *(BinIdxPfx > 1)* | bypass | 0 |
| Suffix | bypass | 0 |
| bmidu\_skip\_group\_flag[ ] | 7 | 0 | | 1 |
| bmidu\_skip\_group\_comp\_flag[ ] | 8 | 0 | | 1 |

Table K-8 – Values of CtxTbl and CtxIdx for MPEG Edge Breaker binarized ae(v) coded syntax elements

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Syntax element** | **CtxTbl** | **CtxIdx** | | **Count** |
| mesh\_position\_fine\_residual[ ][ ] | 1 | Offset | Min( 1, BinIdxTu) | 2 |
| Prefix  (BinIdxPfx <= 4) | 2 + Min( 4, BinIdx Pfx ) | 5 |
| Prefix  (BinIdxPfx > 4) | bypass | 0 |
| Suffix | 7 + Min( 11, BinIdxSfx ) | 12 |
| Sign | bypass | 0 |
| mesh\_position\_coarse\_residual[ ][ ] | 2 | Offset | Min( 2, BinIdxTu) | 3 |
| Prefix  (BinIdxPfx <= 4) | 3 + Min( 4, BinIdx Pfx ) | 5 |
| Prefix  (BinIdxPfx > 4) | bypass | 0 |
| Suffix | 8 + Min( 11, BinIdxSfx ) | 12 |
| Sign | bypass | 0 |
| mesh\_attribute\_fine\_residual[ ][ ][ ]  /\* TEXCOORD \*/  nbPfxCtx = 5, nbSfxCtx = 12  /\* NORMAL \*/  nbPfxCtx = nbSfxCtx = 12  /\* MATERIAL\_ID \*/  nbPfxCtx = nbSfxCtx = 8 | 3 | Offset | Min( 1, BinIdxTu) | 2 |
| Prefix  (BinIdxPfx <= nbPfxCtx -1) | 2 + Min( nbPfxCtx -1, BinIdx Pfx ) | 12 |
| Prefix  (BinIdxPfx > nbPfxCtx -1) | bypass | 0 |
| Suffix | 14 + Min( nbSfxCtx-1, BinIdxSfx ) | 12 |
| Sign | bypass | 0 |
| mesh\_attribute\_coarse\_residual[ ][ ][ ]  /\* TEXCOORD \*/  nbPfxCtx = 5, nbSfxCtx = 12  /\* NORMAL \*/  nbPfxCtx = nbSfxCtx = 12 | 4 | Offset | Min( 2, BinIdxTu) | 3 |
| Prefix  (BinIdxPfx <= nbPfxCtx -1) | 3 + Min( nbPfxCtx -1, BinIdx Pfx ) | 12 |
| Prefix  (BinIdxPfx > nbPfxCtx -1) | bypass | 0 |
| Suffix | 15 + Min( nbSfxCtx-1, BinIdxSfx ) | 12 |
| Sign | bypass | 0 |
| mesh\_clers\_symbol[ ] | 5 | CtxClers ( subclause I.10.3.4.1) | | 30 |
| mesh\_attribute\_seam[ ][ ] | 6 | 0 | | 1 |
| mesh\_texcoord\_stretch\_orientation[ ][ ] | 7 | 0 | | 1 |
| mesh\_handle\_first\_sign[ ] | 8 | 0 | | 1 |
| mesh\_handle\_second\_shift[ ] | 9 | 0 | | 1 |
| mesh\_handle\_first\_variable\_delta\_length4\_minus1[ ] | 10 | Min(3, BinIdxTu) | | 4 |
| mesh\_handle\_first\_variable\_delta[ ] | 11 | bypass | | 0 |
| mesh\_handle\_second\_variable\_delta\_length4\_minus1[ ] | 12 | Min(3, BinIdxTu) | | 4 |
| mesh\_handle\_second\_variable\_delta[ ] | 13 | bypass | | 0 |
| mesh\_position\_is\_duplicate\_flag[ ] | 14 | 0 | | 1 |
| mesh\_attribute\_is\_duplicate\_flag[ ][ ] | 15 | 0 | | 1 |
| mesh\_materialid\_default\_not\_equal\_flag[ ][ ] | 16 | 0 | | 1 |
| mesh\_materialid\_default\_left\_flag[ ][ ] | 17 | 0 | | 1 |
| mesh\_materialid\_default\_right\_flag[ ][ ] | 18 | 0 | | 1 |
| mesh\_materialid\_default\_facing\_flag[ ][ ] | 19 | 0 | | 1 |
| mesh\_normal\_octahedral\_second\_residual[ ][ ][ ] | 20 | Offset | Min( 1, BinIdxTu) | 2 |
| Prefix | 2 + Min( 11, BinIdx Pfx ) | 12 |
| Suffix | 14 + Min( 11, BinIdxSfx ) | 12 |
| Sign | 26 | 1 |

Table K-9 – Values of CtxTbl and CtxIdx for AC displacement binarized ae(v) coded syntax elements

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Syntax element** | **CtxTbl** | **CtxIdx** | | **Count** |
| dsps\_single\_dimension\_flag[ ] | 19 | 0, 1, 2 | | 3 |
| 20 | 0, 1, 2 | | 3 |
| ddu\_nz\_subBlock[ ][ ][ ] | 21 | 0..17 | | 2x3x3 |
| ddu\_coeff\_abs\_level\_gt0[ ][ ][ ] | 22 | 0..17 | | 2x3x3 |
| ddu\_coeff\_sign[ ][ ][ ] | 23 | bypass | | 0 |
| ddu\_coeff\_abs\_level\_gt1[ ][ ][ ] | 24 | 0..17 | | 2x3x3 |
| ddu\_coeff\_abs\_level\_gt2[ ][ ][ ] | 25 | 0..17 | | 2x3x3 |
| ddu\_coeff\_abs\_level\_gt3[ ][ ][ ] | 26 | 0..17 | | 2x3x3 |
| ddu\_coeff\_abs\_level\_rem[ ][ ][ ][ ] | 27 | Offset | ctxIdx \*3\*3\*4 + level \* 3\*4 + k\*4 + BinIdxTu | 0 | |
| Prefix  (BinIdxPfx <= 2) | 72 + Min( 2, BinIdx Pfx ) | 3 | |
| Prefix  (BinIdxPfx <= 2) | bypass | 0 | |
| Suffix | bypass | 0 | |
| Sign | bypass | 0 | |



* 1. Arithmetic decoding engine
     1. General

The arithmetic decoding engine is a context-adaptive, binary arithmetic decoder, performing binary

renormalization and producing binary outputs.

NOTE 1 – An arithmetic encoding engine that complements this decoding engine is described in Annex I.

NOTE 2 – The arithmetic decoding engine is related to that of SMPTE VC-2.

* + 1. State variables

The arithmetic decoder is specified in terms of the following state variables:

* IvlLow, representing the beginning of the 16-bit coding interval.
* IvlRange, representing the size of the 16-bit coding interval.
* IvlCode, a codeword within the interval [ IvlLow, IvlLow + IvlRange − 1 ], updated from the arithmetic-coded bitstream.
  + 1. Initial state

The arithmetic decoding state variables shall be initialized as follows; and 16 bits shall be read from

the arithmetic-coded bitstream:

IvlLow = 0  
 IvlRange = 0xFFFF  
 IvlCode = 0  
 for(i = 0; i < 16 ; i++) {  
 IvlCode <<= 1  
 IvlCode += read\_bits(1)   
 }

* + 1. Arithmetic decoding
       1. Decoding a single binary symbol

Decoding is parameterized by the probability prob0 that the decoded binary symbol is zero-valued.

The decoded binary value binVal is determined and the state variables IvlRange and IvlCode are updated:

rangeTimesProb = IvlRange \* prob0 >> 16  
 binVal = ( rangeTimeProb <= ( IvlCode – IvlLow ) )  
 if (binVal == 0)  
 IvlRange = rangeTimesProb  
 else {  
 IvlLow += rangeTimesProb  
 IvlRange −= rangeTimesProb  
 }

* + - 1. Decoding a single binary bypass symbol

The decoded binary value binVal is determined and the state variables IvlRange and IvlCode are updated:

rangeTimesProb = IvlRange >> 1  
 binVal = rangeTimeProb ≤ IvlCode – IvlLow  
 if( binVal )  
 vlRange = rangeTimesProb  
 else{ IvlLow += rangeTimesProb  
 IvlRange −= rangeTimesProb  
 }

* + - 1. Arithmetic decoder state renormalization

Renormalization stops the arithmetic decoding engine from losing accuracy. Renormalization shall be applied while the size of the coding interval is less than or equal to a quarter of the total available 16-bit range. Each renormalization doubles the interval and reads a bit into the codeword.

If IvlRange is less than or equal to , the state variables IvlRange, IvlLow and IvlCode are updated:

if((IvlLow + IvlRange − 1) ^ IvlLow >= 0x8000) {  
 IvlCode ^= 0x4000  
 IvlLow ^= 0x4000  
 }  
 IvlRange <<= 1  
 IvlLow = (IvlLow << 1) & 0xFFFF  
 IvlCode = ((IvlCode << 1) | read\_bits(1)) & 0xFFFF

If IvlRange remains less than or equal to , the process shall be repeated until it is not.

* + - 1. Arithmetic decoder flushing process

Flushing shall repeatedly perform state renormalization until IvlRange is greater than , and then discard bits from the arithmetic-coded bitstream until it is byte aligned.

while (IvlRange <= 0x4000) {  
 read\_bits(1)  
 IvlRange <<= 1  
 }

/\* byte−align \*/  
 byte\_alignment()

References

[1] SMPTE VC-2 Society of Motion Picture and Television Engineers ST 2042-1 (2017), VC-2 Video Compression.

[2] ISO/IEC WD 23090-9:2022(E), Coded representation of immersive media — Part 9: Geometry-based point cloud compression, FDIS stage.